Markt&Technik

Peter Wollschlaeger

Profi-Tips und Power-Tricks für den Amiga

Lösungen – aus der Praxis für die Praxis

Profi-Tips
und Power-Tricks
für den Amiga

Peter Wollschlaeger

Profi-Tips und Power-Tricks für den Amiga

Lösungen aus der Praxis für die Praxis

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Wollschlaeger, Peter:
Profi-Tips und Power-Tricks für den Amiga :
Lösungen – aus der Praxis für die Praxis / Peter Wollschlaeger. –
Haar bei München : Markt-und-Technik-Verl., 1990
ISBN 3-89090-960-4

15 14 13 12 11 10 9 8 7 6 5 4 3 2
93 92 91

ISBN 3-89090-960-4

Einbandgestaltung: Grafikdesign Heinz Rauner
Dieses Produkt wurde mit Desktop-Publishing-Programmen erstellt
und auf der Linotronik 300 belichtet
Druck und Bindung: Wiener Verlag, Himberg bei Wien
Printed in Austria

Inhaltsverzeichnis

Vorwort und Einleitung 15

Start- und Workbench-Tips 17

1.1 Richtig einschalten 17
1.2 Vorsicht Bildröhre! 17
1.3 Magnetfelder stören den Monitor 18
1.4 Kickstart 1.2 oder 1.3? 18
1.5 Diskette meldet sich 18
1.6 Große Disk-Icons 19
1.7 Diskette nicht auf der Workbench zu öffnen 19
1.8 Icons in anderen Typ wandeln 20
1.8.1 Die Standardlösung 20
1.8.2 Die schnelle Lösung 20
1.9 ».info«: mehr als nur das Icon 21
1.9.1 Snapshot-Tip 22
1.10 Tastatur maßgeschneidert 22
1.11 Fenster zu spart RAM 23
1.12 Blitzstart mit resetfester RAM-Disk 24
1.12.1 Unterschied, Vor- und Nachteile 24
1.12.2 Die ganze Workbench in »RAD:« halten 25
1.12.3 Schnellstart mit nicht Autoboot-fähiger Festplatte 26
1.12.4 Eine kleine RAD: anstatt RAM: 28
1.13 Tips zur Startup-Sequence 30
1.13.1 Startup-Sequence und RAM: 30
1.13.2 Nur ein Guru nach LoadWB? 31
1.14 Tips zu Preferences 31
1.14.1 Preferences werden nicht gespeichert? 31
1.14.2 Preferences-Farben werden nicht übernommen? 31
1.14.3 Es geht auch ohne Preferences und anders 33

1.15 CLI-Programme von der Workbench starten 33
1.15.1 Die Kommando-Datei 33
1.15.2 Das Icon 33
1.15.3 ICONX einsetzen 34
1.15.4 Die Feinheiten 34
1.16 Ein CLI in Reserve ist immer gut 34

CLI-Tips 35

2.1 CLI oder Shell, was nehme ich? 35
2.1.1 Damit die Shell läuft 36
2.2 Der Shell-Editor 36
2.3 Eigenes Prompt 38
2.4 Update auf Shell 39
2.5 RESIDENT ist besser als RAM-Disk? 40
2.5.1 Prüfen, ob residentfähig 41
2.5.2 Residentfähig mit Trick 41
2.6 RUN nachholen 41
2.7 Type anstatt Copy 42
2.7.1 Type in hex 42
2.8 Sprache im CLI 43
2.9 Pfadfinder 43
2.9.1 Güterabwägung 44
2.10 ASSIGN spart Tipparbeit 45
2.11 ALIAS spart Tipparbeit 46
2.12 PC-DOS-Befehle im AmigaDOS 46
2.13 Blau auf weiß 47
2.13.1 Escape-Sequenzen und Control-Codes 47
2.14 Vorsicht Leerzeichen! 48
2.15 CLI-Befehle und nur ein Laufwerk 48
2.16 Mehr mit More 49
2.17 ENV, die Umgebungsvariablen 50
2.18 Es geht auch ohne Execute 51
2.19 Startup-Menü 52
2.20 Schnellere Texte 54

Drucker- und Schnittstellen-Tips 55

3.1 Nehmen Sie die neueste Drucker-Software 55
3.2 Drucker-Update 55
3.3 Der Unterschied zwischen PRT:, SER: und PAR: 56

3.4	Das »File already open«-Problem	56
3.5	Die einfachste Schreibmaschine	56
3.6	Vorsicht CMD (oder Absicht)	58
3.7	Sammeldruck	58
3.8	Einfacher File-Transfer zum PC	58
3.9	Schneller PC-Druck	60
3.10	Immer auf den Seitenanfang	60
3.11	Guru im »Music Construction Set«	61
3.12	Schnelle und gute Hardcopies	61
3.12.1	Nehmen Sie die Workbench 1.3	61
3.12.2	Wählen Sie den richtigen Antrieb	61
3.12.3	Wählen Sie die richtige Dichte	62
3.12.4	Wählen Sie das richtige Threshold	63
3.12.5	Wählen Sie das richtige Dithering	63
3.12.6	So nutzen Sie GraphicDump	63
3.12.7	So werden Sie schneller	64
3.13	Was macht InitPrinter?	64
3.14	Universelle Escape-Sequenzen	65
3.15	Drucker installieren mit InstallPrinter	67
3.16	Billig und gut: Apple-ImageWriter am Amiga	67

Disketten und Festplatte 71

4.1	Programm will partout DF1: oder gar DF2:	71
4.2	Schnellkopie	73
4.3	Einfacher kopieren	73
4.4	Ein Directory zuviel?	73
4.5	Der »DiskDoctor« hilft oft	74
4.5.1	»DiskDoctor« kann auch »undelete«	74
4.5.2	Manchmal hilft noch TYPE	75
4.6	Disk aufräumen	75
4.7	Dateien finden	76
4.8	Disketten-Wechsel vermeiden	77
4.9	Schützen Sie wichtige Dateien	77
4.10	Schnell, aber gezielt löschen	78
4.11	Diskette ganz schnell löschen	79

(Überlebens-)Regeln und Tips für Programmierer 81

5.1	Grundlagen	81
5.1.1	Totale Software-Orientierung	82

5.1.2	Hardware-unabhängig programmieren!	82
5.1.3	Multitasking macht Exec	82
5.1.4	Der Zugriff auf System-Routinen	83
5.1.5	Libraries: Schlüssel zum Amiga	83
5.2	Regel 1: Es gibt nur eine absolute Adresse	84
5.3	Regel 2: Funktions-Ergebnisse testen	84
5.4	Regel 3: Speicher wieder freigeben	85
5.5	Regel 4: Frühzeitig testen	85
5.6	Regel 5: Der Resource-Manager sind Sie	86
5.7	Regel 6: Chip-Daten ins Chip-RAM	86
5.7.1	Daten ins Chip-Memory legen per Compiler-Option	87
5.7.2	Daten ins Chip-Memory legen mit AllocRemember	87
5.8	Regel 7: Das Rad nur einmal erfinden	89
5.9	Extra-Regeln für Assembler-Programmierer	91
5.9.1	D0 ist nicht getestet	91
5.9.2	Vier Register sind Scratch?	91
5.9.3	Programmieren Sie »32-Bit-clean«	92
5.9.4	Denken Sie an die Nachfolger des 68000	92
5.9.5	Vergessen Sie den Return-Code nicht!	92
5.10	Beachten Sie die »Postvorschriften«	93
5.11	Ein Debugger ist schon eingebaut	95
5.11.1	ROM-Wack starten	96
5.11.2	Das ROM-Wack-Display	97
5.11.3	Positionierung	98
5.11.4	Suchen und Finden	98
5.11.5	Speicherinhalt ändern	99
5.11.6	Programm testen	99
5.11.7	ROM-Wack verlassen	100
5.12	Und immer gilt die FGO-Regel	100

Tips für Assembler-Programmierer 103

6.1	Schneller und kürzer	103
6.1.1	Quick bevorzugen	103
6.1.2	Unterschiede zwischen MOVEQ und den anderen Q-Befehlen	103
6.1.3	CLR vermeiden	104
6.1.4	Direktadressierung vermeiden	104
6.1.5	32-Bit-Konstanten vermeiden	104
6.1.6	Stack schneller abbauen	105
6.1.7	Schneller multiplizieren	105
6.1.8	Adreß- anstatt Datenregister	106

6.1.9	Worte anstatt langer Worte	106
6.1.10	Am besten A mit A	106
6.1.11	Schneller schieben	106
6.2	Positionsunabhängig schreiben	107
6.3	MOVEM kann mehr	108
6.4	Richtig springen	109
6.5	Der richtige Startup-Code	109
6.6	C-Funktionen im ROM nutzen	112
6.6.1	Das Prinzip	113
6.6.2	Astartup.obj und Amiga.lib	115
6.6.3	Der Trick mit der Ausgabe-Umleitung	115
6.6.4	»printf()« unter Intuition	118
6.7	Die Kommando-Zeile nutzen	122
6.8	CLI-Befehle in Assembler sparen viel	122
6.9	Verwenden Sie Makros	124
6.10	Menüs mit Makros	126
6.11	I/O mit AmigaDOS in Assembler	130
6.12	Top Down Bottom Up	135

Tips für Basic-Programmierer 137

7.1	Starten und Beenden von Basic	137
7.1.1	Programm und Basic gleichzeitig starten	137
7.1.2	Programm mit einem Doppelklick starten	138
7.1.3	AmigaBasic vom CLI aus starten	138
7.1.4	Basic auf der Start-Diskette	138
7.1.5	Beenden von AmigaBasic	139
7.2	Editor-Tips (und -Tricks)	140
7.2.1	Erst das List-Fenster	141
7.2.2	Inhaltsverzeichnis ansehen	141
7.2.3	Schneller mit FF	142
7.2.4	Undo oder Kommando zurück	142
7.2.5	Schneller zum Listing	142
7.3	Libraries in Basic	143
7.3.1	FDs und BMAPs	143
7.3.2	Der Einsatz von ConvertFD	144
7.3.3	ConvertFD-Tip	144
7.3.4	Wohin mit den BMAPs?	145
7.3.5	Spartip	145
7.3.6	Der Einsatz von Libraries	145
7.3.7	Deklarieren Sie richtig	148

7.3.8	Welche Libraries gibt es?	148
7.4	Out of Memory, was nun?	149
7.4.1	Optimierung von Basic-Programmen	149
7.4.1.1	Speicher messen und einteilen	149
7.4.1.2	Arrays dimensionieren	151
7.4.1.3	OPTION BASE nutzen	151
7.4.1.4	Richtigen Daten-Typ nehmen	151
7.4.1.5	Bis 65535 als »Unsigned Integer«	151
7.4.1.6	Zahlen bis 255 in einem Byte	152
7.4.1.7	' ist länger als REM	153
7.4.1.8	Nichts kostet auch Speicher	153
7.4.1.9	Brauchen Sie eigene Screens und Windows?	153
7.4.2	Entfernen allgemeiner Speicher-Schlucker	153
7.4.2.1	Ohne RAM-Disk	154
7.4.2.2	Ohne RESIDENT	154
7.4.2.3	Ohne extra Puffer	154
7.4.2.4	Ohne Workbench	154
7.4.2.5	Start im Direktmodus	155
7.5	Basic-Menü ist weg	155
7.6	Basic-Menü soll weg sein	155
7.7	Notfalls mit <Ctrl>+<C>	156
7.8	Wo sind die Locate-Koordinaten?	156
7.9	Ausgabe-Window gleich aktiv	157
7.10	Dateien sparsamer speichern	157
7.11	Maus-Tips	158
7.12	Mini-Paint	160
7.13	Schneller scrollen	160
7.14	CLI-Befehle in Basic nutzen	161
7.14.1	Das Prinzip	161
7.14.2	Gibt es DF1: oder andere Devices?	164
7.15	Schleifen anderer Basic-Dialekte	165
7.16	Call ohne CALL	166
7.17	Umlaute und Blanks vermeiden	167
7.18	Listing mit Zeilennummern	167
7.18.1	Auch ohne CLI	168
7.19	Guru-Meldung in Basic	169
7.20	BasicClip nutzen	170
7.21	MAKEDIR in Basic	171
7.22	Daten blitzschnell kopieren	172
7.23	Consol-Fenster in reinem Basic	173
7.24	Laufschrift ruckfrei	174

7.25	Gadget-Abfrage mit einer Zeile	175
7.26	PEEK und POKE langer Adressen	176
7.27	Basic schneller machen	177
7.27.1	In der Kürze liegt die Würze	177
7.27.2	Schleifen freihalten	177
7.27.3	Keine Schleife mit GOTO	178
7.27.4	GOSUB ist schneller als GOTO	179
7.27.5	Benutzen Sie Ganzzahlen	179
7.27.6	Die Ganzzahl-Division ist sehr viel schneller	180
7.27.7	Bestellen Sie die Müllabfuhr rechtzeitig	180
7.27.8	Räumen Sie auf	181
7.27.9	Arbeiten Sie mit Tabellen	181
7.27.10	Nehmen Sie die »graphics.library«	182
7.28	Tips zu einigen Basic-Befehlen und -Funktionen	182
7.29	ACBM- und IFF-Bilder in Basic	190
7.30	Fehler richtig abfangen	192
7.31	Die Window-Struktur	194
7.31.1	PEEK und POKE und Typen	194
7.32	Die RastPort-Struktur	200
7.33	Anwendung der Window- und RastPort-Strukturen	201
7.33.1	Stilarten 1	201
7.33.2	Stilarten 2	202
7.33.3	Zeilenabstand ändern	202
7.33.4	Zeichenabstand ändern	203
7.33.5	Größenänderung limitieren	203
7.33.6	Window-Titel ändern	203
7.33.7	Screen-Titel ändern	204

Einbindung von Assembler-Routinen in Basic 205

8.1	Anforderungen an die Routinen	205
8.1.1	Lageunabhängig	205
8.1.2	Nur ein Segment	206
8.1.3	Unterprogramm muß alle Register retten	206
8.1.4	Nur Code und Daten speichern	206
8.2	Raum für die Routinen	206
8.3	Laden und Aufrufen von Assembler-Routinen	207
8.4	Maschinenprogramm im Systemspeicher	210
8.5	Ein Assembler-Basic-Konverter	211
8.6	Die Parameterübergabe	217

Tips für C-Programmierer 221

9.1 Der Aztec-C-Compiler 221
9.2 Der Lattice-C-Compiler 222
9.2.1 Lattice Version 3 223
9.2.2 BLINK einsetzen 223
9.2.3 Lattice ab Version 4 224
9.3 Wenn das Fenster nicht schließt 224
9.4 Zuerst »amiga.lib«? 224
9.5 Das richtige Start-Modul 225
9.6 Warnmeldungen doch beachten 226
9.7 Gefährliche Fehler 227
9.8 Nehmen Sie Makros 228
9.9 So schreiben Sie kompakte Programme 228
9.9.1 »_main()« anstatt »main()« 228
9.9.2 Amiga.lib zuerst 229
9.9.3 Den ROM und die »Protos« nutzen 229
9.10 Compiler/Linker-Aufruf im Quelltext 230
9.11 Grafiken speichern – ganz einfach 231
9.11.1 Grafik-Tricks mit dem Complement-Modus 231
9.11.2 Zugriff auf den Workbench-Screen 233
9.11.3 Zugriff auf die Bit-Planes 233
9.11.4 Zugriff auf die Farb-Register 234
9.11.5 Das Bild wieder laden 235
9.12 Programm muß Preferences kontrollieren 235
9.12.1 Die Preference-Struktur 236
9.12.2 Der Zugriff auf Preferences 237
9.13 Ausgabe von Datum und Zeit 240
9.14 Richtiges Refresh für Windows 243
9.15 I/O mit AmigaDOS in C 243

Nützliche Programme auf der Extras-Diskette 247

10.1 MEMACS – Ein Tip für Programmierer 247
10.1.1 Puffer 247
10.1.2 Zwei Betriebsarten 248
10.1.3 »Read-file« und »Visit-File« 248
10.1.4 »Dot« und »Mark« 248
10.1.5 »Yank« 248
10.1.6 Anmerkungen zu den Menüs 249
10.1.7 Die Tastenkürzel 255

10.1.8	Die Start-Datei	256
10.2	FED, der Font-Editor	256
10.2.1	Was FED nicht kann	256
10.2.2	Der Start	256
10.2.3	Die Bedienelemente	258
10.2.4	Die Menüs	259
10.2.5	Starten Sie FixFonts	260
10.3	Der Icon-Editor	260
10.3.1	Der Start	261
10.3.2	Das Editieren	262
10.3.3	Die Extras	262
10.4	IconMerge	263
	Stichwortverzeichnis	265

V

Vorwort, Einleitung und Dankeschön

Die Marktforschung sagt, daß Basic die beliebteste Programmiersprache ist, und dann bereits Assembler folgt. C liegt auf Platz drei, holt aber auf. Eine Ergänzung aus der Praxis: In Basic zu programmieren und immer dann, wenn es eng wird, eine Assembler-Routine einzubauen, ist auch sehr beliebt.

Wenn Sie dieses Buch zerlegen, werden Sie einen ähnlichen Trend feststellen. Das »Werk« umfaßt 264 Seiten. Davon wenden sich 67 an den Basic-Programmierer, 32 sind für die Assembler-Freaks gedacht, und 22 Seiten sind für die C-Gemeinde reserviert. Nochmals 12 Seiten berücksichtigen die »mixed language«-Freaks mit Basic/Assembler-Kombination.

Das ist aber erst gut die Hälfte. Für wen sind die restlichen Seiten? Die 19 Seiten (Überlebens-) Regeln für alle Programmierer darf jeder seiner Lieblingssprache selbst hinzuaddieren. Der große Rest ist für alle da, für die User, besonders für die Power-User, für die Einsteiger, die Profis und alle, die es werden wollen.

Um wieder auf die Marktforschungsergebnisse zurückzukommen. Genau daraufhin habe ich das Buch ausgerichtet, ich bin schließlich ein cleverer Profi, könnte ich jetzt behaupten. Um ehrlich zu sein, ich bin nur ein Glückspilz und Programmierer (und manchmal Autor).

Als solcher habe ich Bücher zu allen drei Sprachen geschrieben, und das hatte Folgen. Amiga-Freaks sind nämlich kontaktfreudig und clever. Haben sie ein Problem, schnappen sie sich einen Buchautor per Brief oder Telefon. Mit »Sind Sie der Peter Wollschlaeger, der ...« beginnen die meisten meiner Anrufe.

Manche Frage konnte ich nicht aus dem Stegreif beantworten. Oft wurde etwas ausprobiert und natürlich abgespeichert. Jede Antwort ging auch auf die Disk, und plötzlich war da etwas, was man eine Datensammlung nennt. Die wurde sortiert, katalogisiert, indiziert, optimiert und... ich weiß kein Wort mehr mit »iert«. Doch Sie wissen jetzt, wie das Buch entstand, nämlich aus der Praxis und für die Praxis.

Eine sehr häufige Frage hat das Buch bereits im Entstehungsstadium provoziert. Wie viele Tips und Tricks stehen drin? Hiermit wird diese Frage erstmals beantwortet und auch einmalig, sprich, nach dem Radio-Eriwan-Prinzip, das da lautet: es kommt drauf an. Wenn Sie die Titel und Zwischentitel zählen, sind es rund 350. Wenn Sie die Tips innerhalb der Tips addieren, kommen

Sie auf über 500, und wenn Sie die Anregungen, die in vielen Themen stecken, ausreizen, könnten es 1000 sein. Kurz gesagt: das Buch bringt 350 bis 1000 Tips und Tricks je nach Zählweise, die kleine Differenz ist Ansichtssache.

Und nun muß ich leider persönlich werden. Meine berüchtigte Schreibe – auch »locker vom Hocker« genannt – stammt aus der amerikanischen Schule, die ich – sorry, manche Kollegen – immer noch für die bessere halte. Wie einst beim Schreiben von Artikeln für amerikanische Computer-Zeitschriften gelernt, texte ich noch heute, auch in Büchern. In diesem Sinne möchte ich jetzt mit Scott Knaster einen amerikanischen Star-Autor zitieren:

»An diesem Buch habe ich alles selber gemacht. Ich habe es geschrieben, ich habe es korrigiert, ich habe alle Abbildungen eigenhändig gezeichnet, die Druckerschwärze habe ich persönlich angerührt, ich habe die Bäume für das Papier gefällt und jedes Exemplar selber gebunden, ich habe alle Bücher selber verkauft und ausgeliefert. Ich danke niemandem.«

»Quatsch!«

Ich danke dem Lektor Peter Mayer, den netten Damen der Assistenz, den Duden- und Fachlektoren, dem Produktionsteam, dem Boß Dr. Hempelmann und nicht zuletzt meiner Frau Helga für die Erstkorrektur und den vielen Kaffee.

Und weil der Spruch in jedem meiner Bücher steht:

»Happy Programming«
Ihr
Peter Wollschlaeger

1 Start- und Workbench-Tips

1.1 Richtig einschalten

Es ist kaum zu glauben, aber schon beim Einschalten des Amiga kann man etwas falsch machen, sprich, den Computer beschädigen. Das passiert dann, wenn man den Amiga ausschaltet und ihn gleich darauf wieder einschaltet. Sollte das nötig sein – bei manchem Absturz ist es die einzige Lösung – lassen Sie zwischen den beiden Aktionen wenigstens 30 Sekunden vergehen.

Warum ist das so?

Beim Einschalten entstehen Spannungsspitzen, die die empfindlichen Bauteile beschädigen können. Da nicht sein kann, was nicht sein darf, haben die Entwickler unter anderem Kondensatoren eingebaut. So ein Kondensator hat die Eigenschaft, sich beim Anlegen einer Spannung erst einmal selbst aufzuladen, bevor er die Spannung an die anderen Bauteile weitergibt. Auf diese Art »schluckt« der Kondensator die Einschaltspitzen. Wenn Sie den Amiga ausschalten, sind die Kondensatoren noch geladen, sie entladen sich nur langsam innerhalb einiger Sekunden. Schaltet man nun gleich wieder ein, sind die Kondensatoren sozusagen noch voll und können nichts mehr – also auch nicht die Spannungsspitzen – aufnehmen.

1.2 Vorsicht Bildröhre!

Auch die Bildröhre im Monitor wirkt wie ein riesiger Kondensator, aber einer mit einer Entladezeit von Wochen, und die Röhre hält Hochspannung! Wenn Sie also den Monitor öffnen, ziehen Sie vorher nicht nur den Netzstecker ab, sondern halten sich auch von der Bildröhre fern.

Zur Verdeutlichung ein Spruch aus der Branche: Was ist der Unterschied zwischen einem Schwarzweiß- und einem Farbmonitor? Kommt der Techniker bei einem Schwarzweiß-Monitor gegen die Hochspannung, bricht die Spannung zusammen. Bei einem Farbmonitor bricht der Techniker zusammen.

1.3 Magnetfelder stören den Monitor

Da wir gerade bei den Eigenschaften von Monitoren sind: Diese Geräte haben etwas gegen magnetische Felder bzw. zeigen diese in Form von störenden Mustern an. Stellen Sie also andere Geräte nicht zu dicht an den Bildschirm. Amiga-500-Besitzer sollten das Netzteil nicht neben den Monitor stellen, so rund ein Meter Abstand ist gut. Daß Disketten mit Magnetfeldern gelöscht werden, soll bei der Gelegenheit nur noch einmal in Erinnerung gebracht werden.

1.4 Kickstart 1.2 oder 1.3?

Kickstart heißt beim Amiga der Betriebssystem-Kern. Der Name rührt daher, daß der Amiga nach dem Einschalten bestimmte System-Routinen ausführt, die ihn in einen definierten Zustand, sozusagen in die Startposition bringen. Ausgenommen den Amiga 1000, befindet sich das Kickstart im ROM (Read Only Memory), also in einem permanenten Speicher. Die frühere Kickstart-Version hieß 1.2, die aktuelle Version heißt 1.3. Beim Amiga 1000 ist anstatt ROM-RAM-bestückt. In dieses RAM wird Kickstart nach dem Einschalten von der Diskette geladen, für eine neue Kickstart-Version braucht man also nur eine andere Diskette.

Für die übrigen Modelle gilt: Generell sollte man seinen Amiga auf den neusten Stand bringen, weil es doch inzwischen sehr viel Software gibt, die Kickstart 1.3 mit der zugehörigen Workbench 1.3 voraussetzt. Dazu reicht es, den ROM auszutauschen bzw. vom Händler austauschen zu lassen. Sollten Sie jedoch noch viele Programme der ersten Stunde haben, kann es Ihnen passieren, daß die nicht mehr unter der neuen Version laufen. Daher dieser Tip:

Es gibt kleine Platinen, auf die man mehrere ROMs oder EPROMs mit verschiedenen Betriebssystemen installieren und dann mittels eines Schalters dazwischen umschalten kann. Diese Lösung ist recht preiswert und auf jeden Fall die flexiblere.

1.5 Diskette meldet sich

Sie haben wahrscheinlich verschiedene Workbench-Disketten, die beispielsweise für Basic, C oder Spiele maßgeschneidert sind. Als Kopien der Original-WB-Diskette melden sie sich beim Booten alle mit demselben Text. Nach dem Motto »was hatte ich da eigentlich eingelegt?« nur mal so die Diskette auszuwerfen, geht nun auch nicht mehr. Was tun? Ändern Sie mit einem Editor – zum Beispiel MEMACS – (siehe Kapitel 10) die Startup-Sequence im s-Verzeichnis. Dort können Sie vorhandene Echo-Befehle ändern oder neue einfügen. Der Befehl

```
echo "Das kann jeder Text sein"
```

gibt genau diesen Text auf dem Schirm aus.

Für mehrere Zeilen brauchen Sie nicht unbedingt auch mehrere Echo-Befehle. Die Zeichenfolge »*N« bewirkt, daß der folgende Text auf der nächsten Zeile weitergeschrieben wird.

Längere Texte mit vielen Echo-Kommandos zu erzeugen, ist mühsam. In einem solchen Fall schreiben Sie den Text besser mit einem Text-Editor, speichern ihn in einer Datei und geben ihn dann mit dem TYPE-Befehl aus. Gerade in einer Startup-Sequence sollten Sie aber besser deshalb nicht den Betrieb aufhalten. Schreiben Sie also

```
run type NameDerDatei
```

und die folgenden Befehle laufen weiter, während der Text auf den Schirm geschrieben wird.

1.6 Große Disk-Icons

Einige Disketten kommen mit riesigen Icons. Ob das schön ist – sie nehmen ja einen großen Teil der Workbench ein – ist eine andere Frage. Auf jeden Fall kann man solche Disketten kaum übersehen. Wollen Sie das nachmachen, kein Problem. Sichern Sie zuerst das Original-Icon der Diskette, indem Sie die Datei »disk.info« unter einem anderen Namen kopieren. Gibt es die Datei »disk.info« nicht, zeigt der Amiga trotzdem ein Icon an, nämlich das Standard-Icon. In diesem Fall ist es das einfachste, ein anderes »disk.info« auf die Diskette zu kopieren. Zum Thema Typwechsel lesen Sie bitte im Kapitel 1.7 und über IconEd im Kapitel 10.3 nach.

Starten Sie IconEd (siehe Kapitel 10), und laden Sie das Disk-Icon. Hierzu wählen Sie im Menü »Disk« den Punkt »Load Data« und ändern dann »Name« zum Beispiel in »df0:disk«, wenn die Diskette im eingebauten Laufwerk 0 steckt. Nun können Sie in aller Ruhe ein Riesen-Icon malen und nicht nur so ein kleines Bild, wie es normalerweise links oben im Rahmen hängt. Sichern Sie Ihr Kunstwerk via »Disk/Save Data« mit »Save Full Image«.

Wie funktioniert das?

Beim Einlegen einer Diskette sucht der Amiga nach einer Datei mit dem Namen »disk.info«. Findet er diese nicht, nimmt er das Standard-Icon. Findet der Amiga die Datei, zeichnet er das in »disk.info« abgelegte Bild und schreibt darunter den Namen der Diskette. Voraussetzung dafür ist, daß die Info vom Typ Disk ist. Wenn das nicht der Fall ist, können Sie die Diskette nicht mit der Maus öffnen.

1.7 Diskette nicht auf der Workbench zu öffnen

Aus dem vorigen Trick folgt gleich der nächste: Wenn Sie verhindern wollen, daß eine Diskette von der Workbench aus geöffnet werden kann, dann müssen Sie ihr nur ein falsches Icon geben. »Kein Icon« geht leider nicht. Starten Sie IconED und malen ein Icon (siehe auch Kapitel 10).

Dafür reicht es an sich, mit der Textfunktion etwas wie »Nur im CLI zu bearbeiten« einzutragen. Dann speichern Sie das auf dieser Diskette unter dem Namen »disk.info«, und schon ist alles klar. Ein Mausklick hilft nicht mehr, im CLI oder in der Shell können Sie aber wie üblich »cd df0:« eingeben und mit den Dateien auf der Disk arbeiten. Mehr dazu im nächsten Tip.

1.8 Icons in anderen Typ wandeln

Es gibt überall viele schöne Icons (Piktogramme), die man gerne für seine Programme oder Disketten einsetzen möchte. Doch leider gibt es dabei ein Problem. Es gibt bestimmte Typen von Icons, die zu beachten sind. Tun Sie das nicht, gibt es Ärger. Zum Beispiel können Sie keine Diskette mehr per Mausklick öffnen, wenn Sie die Icon-Datei »disk.info« mit einem Tools-Icon überschrieben haben.

1.8.1 Die Standardlösung

Starten Sie IconEd, und Sie sehen eine Liste der Typen mitsamt der Aufforderung, doch bitteschön den richtigen Typ zu verwenden. Doch genau das wollen wir nicht, sondern ein Icon vom (symbolischen) Typ A in den Typ B umwandeln, und das geht so:

1. Starten Sie IconEd neu.
2. Laden Sie das Icon A.
3. Klicken Sie das zweite Rechteck (Frame) an, und laden Sie ein beliebiges Icon vom Typ B.
4. Löschen Sie das Icon vom Typ B mit »Clear this Frame«.
5. Rufen Sie »Merge this Frame« auf und zeigen Sie dabei auf den Frame 1 (wo Ihr Icon vom Typ A ist).
6. Speichern Sie das Icon im zweiten Frame.

Wie funktioniert das?

Wenn man in einen Frame ein Icon lädt, legt man damit gleichzeitig den Typ fest (der steht auch in der Info-Datei). Der Typ bleibt natürlich auch bestehen, wenn Sie das Icon ändern. Dabei ist es egal, ob Sie malen oder wie wir hier, das ganze Bild löschen und ein anderes hineinkopieren.

1.8.2 Die schnelle Lösung

Wenn Ihnen die vorherige Lösung zu umständlich war, nehmen Sie doch diese: Mit einem File-Editor laden Sie die Info-Datei und ändern das Byte auf der Position $30 (dezimal 48). Es gelten:

01 = Disk
02 = Drawer
03 = Tool
04 = Project
05 = Garbage

1.9 ».info«: mehr als nur das Icon

Daß die Icons (die Daten für Ihre Grafik) in einer Info-Datei stehen, hatten wir gerade bei »disk.info« kennengelernt, doch in einer Info-Datei kann noch mehr stecken.

Zuerst wäre da der Typ zu nennen, wovon es folgende gibt:

```
DISK            jedes Disketten-Icon
DRAW            für Drawer, also Schublade
GARBAGE         der Mülleimer
PROJEKT         Datei, die ein Programm erzeugt
TOOL            ein ausführbares Programm
```

Im Normalfall lädt man ein Programm und dann in diesem eine Datei, die das Programm bearbeitet. Zum Beispiel starten Sie zuerst einen Editor und laden dann eine Textdatei. Diese triviale Methode der MS-DOS-Computer bietet der Amiga auch, doch er kann mehr. Sie klicken auf der Workbench eine Datei namens »Lies_mich« an, und schon startet das NotePad und lädt die Datei.

Das liegt ganz einfach daran, daß in der Info-Datei von »Lies_mich« notiert ist, daß »Lies_mich« zum Programm NotePad gehört. Das Programm ist das Tool (das Werkzeug), eine von ihm erzeugte Datei ist ein Projekt.

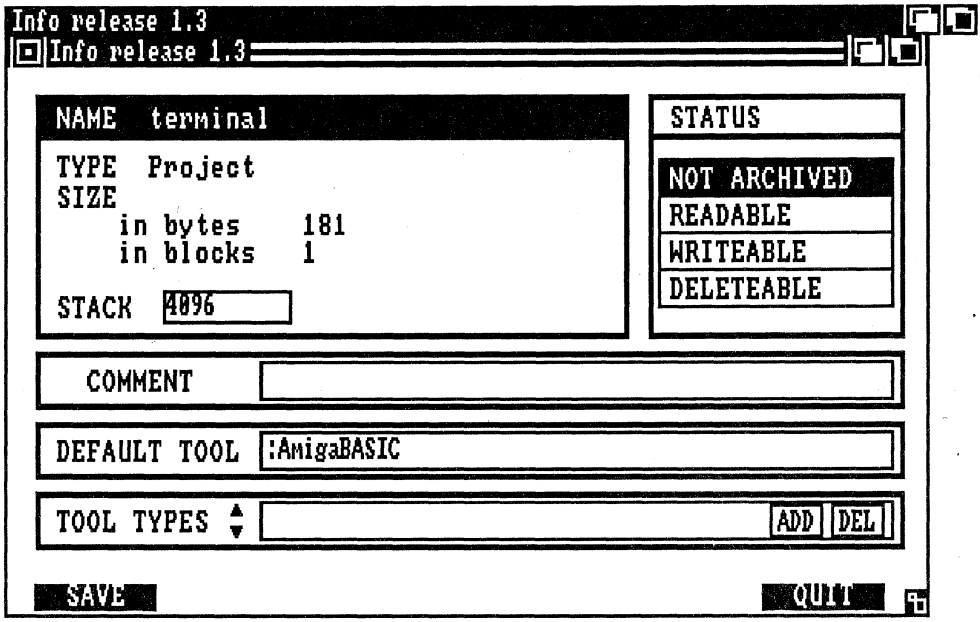

Abb. 1.1: Der Info-Requester bietet viele Möglichkeiten

Generell kann man jedes Dokument einem Programm zuordnen. Dazu klickt man es einmal an, so daß es selektiert wird und wählt dann den Menüpunkt Info aus dem Workbench-Menü. Machen Sie das beispielsweise mit einem Basic-Programm – in diesem Sinne ein Projekt –, erscheint ein Requester wie in Abb. 1.1.

Das »Default Tool« ist AmigaBasic. AmigaBasic war so clever, beim Anlegen der Info-Datei hier auch schon »:AmigaBasic« einzutragen. Wenn trotzdem das Doppelklicken auf ein Basicprogramm-Icon nur eine Fehlermeldung ergibt, liegt das daran, daß Basic nicht auf der obersten Ebene der Diskette ist. Steckt es zum Beispiel im Ordner Basic, müssen Sie den Eintrag unter »Default Tool« in »:Basic/AmigaBasic« ändern. Unter »TOOL TYPES« werden Parameter eingetragen, die man einem Programm beim Aufruf mitgeben will.

1.9.1 Snapshot-Tip

Bliebe noch die Lage des Icons. Die wird notiert, wenn Sie a) ein Icon auswählen (einmal anklicken) und b) Sie dann »Snapshot« (Schnappschuß) im »Special«-Menü aufrufen. Damit ist auch klar, daß diese Arbeit für jedes Icon erforderlich ist, doch Sie können da rationalisieren. Halten Sie die Shift-Taste gedrückt, während Sie ein Icon nach dem anderen anklicken, sie werden damit alle ausgewählt, und es reicht ein Schnappschuß.

1.10 Tastatur maßgeschneidert

Die Tastaturen einzelner Länder unterscheiden sich. Beispielsweise ist auf dem deutschen Keyboard aus amerikanischer Sicht Y und Z vertauscht. Daher gibt es die sogenannte Keymap, praktisch eine Tabelle, mit deren Hilfe der Amiga Tastendrücke in Tasten-Codes umsetzt. Diese Tabelle können Sie durchaus ändern, legen Sie aber vorher eine Kopie an oder arbeiten Sie mit einer Kopie der Workbench-Disk. Beliebt ist die folgende Änderung: Der Hochstrich »'« (CHR$(39)) wird häufig benötigt, zum Beispiel als Kürzel für REM in Basic oder als Text-Delimeter in Assembler. Dieses Zeichen ist aber nicht das links von der Rückschritt-Taste, sondern muß über <Alt><ä> erzeugt werden. Wollen Sie dieses Zeichen bequemer erreichen, können Sie es auf die Taste für den »'« legen. Das ist auf der Tilde-Taste (die Welle) neben der 1-Taste ganz links in der zweiten Reihe. Kopieren Sie in der Schublade »:devs/keymaps« die Datei d nach dx. Nun lassen Sie das Basic-Programm von Abb. 1.2 laufen.

```
OPEN ":devs/keymaps/dx" AS 1 LEN=1
FIELD #1, 1 AS d$
LSET d$=CHR$(39)
PUT #1,232
CLOSE
```

Abb. 1.2: Ein Programm zum Ändern der Keymap

Anschließend wechseln Sie in die Shell und geben ein:

```
setmap dx
```

Das neue Zeichen müßte wirken. Leider kommen Sie mit »setmap d« nicht auf den alten Stand zurück, sondern müssen dafür den Amiga neu starten. Sie können natürlich das Programm gleich mit d anstatt mit dx laufen lassen.

Wenn Sie etwas mit anderen Tastaturbelegungen experimentieren wollen, dann können Sie mit dem Programm von Abb. 1.3 erst einmal die Map (immer einen Teil davon) sichtbar machen:

```
OPEN ":devs/keymaps/d" AS 1 LEN=1
FIELD #1, 1 AS d$
FOR i = 229 TO 270 STEP 4
  FOR j= 0 TO 3
    GET #1,i+j
    PRINT d$; " ";
  NEXT j
  PRINT
NEXT i
CLOSE
```

Abb. 1.3: Ausdruck der Keymap

Das ergibt die Zeichen für die Tasten 1, 2, 3 usw. bis 0 auf der obersten Reihe. Für die Taste 0 wird das Listing als letzte Zeile

```
) » = 0
```

ausdrucken. Nun geben Sie ein:

```
<Shift>+<Alt>+<0>   <Alt>+<0>  <Shift>+<0>   <0>
```

und Sie erhalten die obigen vier Zeichen. Sie sehen, daß jede Taste vierfach belegt ist, genauer: sie so belegt sein kann. Das Listing soll nur eine Anregung sein. Sie werden es sicherlich leicht zu einem Keymap-Editor ausbauen können.

1.11 Fenster zu spart RAM

Daß man im Winter Heizkosten sparen kann, wenn man nicht alle Fenster aufreißt, wissen Sie ja schon. Aber so ein ähnliches Gesetz gilt – auch im Sommer – für den Amiga. Jedes offene Window kostet RAM. Abhilfe schafft ein simpler Trick.

Öffnen Sie das jeweilige Window, und schieben Sie die Icons, die Sie tatsächlich brauchen, auf die Workbench. Dann schließen Sie das Window wieder. Beachten Sie dabei die Speicheranzeige (free memory) in der Menüleiste, und Sie sehen, was Sie gewinnen.

1.12 Blitzstart mit resetfester RAM-Disk

Ab der Workbench 1.3 gibt es neben der bekannten RAM-Disk »RAM:« noch einen zweiten Vertreter dieser Gattung namens »RAD:«, was für »resetfeste RAM-Disk« steht. Der Vorteil dieser RAM-Disk ist, daß ihr Inhalt nach einem Reset oder Absturz erhalten bleibt. Im Falle von »RAM:« müssen Sie nach jedem Reset wieder alle Dateien auf die RAM-Disk kopieren. Daraus könnte man also folgern, daß man nur noch »RAD:« verwenden sollte, doch die Sache hat einen Haken.

1.12.1 Unterschied, Vor- und Nachteile

»RAM:« ist eine dynamische RAM-Disk, das heißt, ihre Größe ändert sich nach Bedarf. Löschen Sie in »RAM:« Dateien, steht der Platz wieder Ihren Programmen zur Verfügung. »RAD:« hingegen ist statisch, d.h., ihre Größe ist konstant, ganz unabhängig davon, wieweit »RAD:« tatsächlich mit Daten gefüllt ist. Folglich kommt es auf den speziellen Anwendungsfall an. Praktisch müssen wir drei Fälle unterscheiden:

1. Sie haben so viel RAM (wenigstens 2 Mbyte), daß Sie es sich leisten können, die ganze Workbench-Diskette auf einer 880 Kbyte großen »RAD:« zu halten. In diesem Fall ist nach einem Reset ein Blitzstart möglich. Voraussetzung dafür ist, daß Sie Kickstart 1.3 haben. Ansonsten können Sie aber auch blitzschnell alle CLI- und Workbench-Befehle von der RAM-Disk ausführen und sich bei nur einem Laufwerk den häufigen Diskettenwechsel ersparen. Wenn Sie schon eine Festplatte mit Autoboot-Fähigkeit besitzen, lohnt sich das allerdings nicht.
2. Sie haben eine Festplatte ohne Autoboot-Fähigkeit. Dann lohnt es sich, in einer kleinen »RAD:« die Dateien anzulegen, die man braucht, um die Kontrolle an die Festplatte zu übergeben. Damit ist nach einem Reset ein schneller Start ohne Diskette möglich.
3. Sie haben keine Festplatte und nicht soviel RAM übrig, kopieren aber nach jedem Start bestimmte CLI-Befehle in die RAM-Disk. In diesem Fall sollten Sie eine kleine »RAD:« anlegen, in die genau diese Programme und praktischerweise noch das S-Directory passen. Dann ersparen Sie sich nach einem Reset das Kopieren von der Diskette.

Die Methoden 2 und 3 lassen sich natürlich kombinieren, doch gehen wir nun diese drei Möglichkeiten der Reihe nach durch.

1.12.2 Die ganze Workbench in »RAD:« halten

Um die ganze Workbench-Diskette nach »RAD:« zu kopieren, brauchen Sie eine RAM-Disk von 880 Kbyte Größe und dafür einen Amiga mit wenigstens 2 Mbyte RAM und Kickstart 1.3. Auf »RAD:« wird nach einem Kaltstart die ganze Workbench-Diskette kopiert, bei weiteren Warmstarts (Resets) geht dann alles blitzschnell. Das Kopieren auf die RAM-Disk mit DISKCOPY geht übrigens viermal schneller als eine Disk-zu-Disk-Kopie. Damit das läuft, sind folgende Vorbereitungen notwendig:

1. Legen Sie eine Kopie der Workbench-Diskette an, und arbeiten Sie damit weiter.
2. Laden Sie die Datei »MountList« im Verzeichnis »Devs« der Workbench-Diskette in Ihren Editor, und suchen Sie dort den Eintrag »RAD:«. Hier ändern Sie die Zeile »HighCyl=21« in

   ```
   HighCyl=79
   ```

   Damit hat die »RAD:« 80 Zylinder (0..79) und somit die gleiche Größe wie eine Diskette. Speichern Sie die geänderte Datei ab.
3. Laden Sie die Datei »Startup-Sequence« im Verzeichnis »s« in Ihren Editor, und suchen Sie dort die Zeilen

   ```
   FastMemFirst
   BinDrivers
   ```

   Genau zwischen diese beiden Zeilen fügen Sie die von Abb. 1.4 ein:

```
assign >NIL: RAD: exists
if warn
  echo "RAD: wird montiert"
  mount RAD:
  if not exists RAD:c
    echo "Die Workbench wird auf RAD: kopiert"
    SYS:System/diskcopy <NIL: df0: to RAD: name "RAMWB"
  endif
endif
RAD:c/failat 10
RAD:c/cd RAD:c
RAD:c/echo "Steuerung wird an RAD: übergeben"
assign c: RAD:c
assign s: RAD:s
assign l: RAD:l
assign libs: RAD:libs
assign devs: RAD:devs
assign fonts: RAD:fonts
assign sys: RAD:
echo "Die ganze Workbench ist auf RAD:"
```

Abb. 1.4: Änderungen für »RAD: in der Workbench«

Damit die Änderung wirksam wird, müssen Sie nicht nur die geänderten Dateien abspeichern, sondern danach auch einen Kaltstart ausführen. Schalten Sie also den Amiga aus, und warten Sie mit dem Wiedereinschalten wenigstens 30 Sekunden.

1.12.3 Schnellstart mit nicht Autoboot-fähiger Festplatte

Wenn Sie eine Festplatte ohne Autoboot-Fähigkeit haben, zum Beispiel die A2090, müssen Sie nach jedem Reset Ihre Workbench-Diskette einlegen. Mit einer kleinen »RAD:« (77 Kbyte) können Sie das vermeiden und den ganzen Vorgang erheblich beschleunigen. Hier das Kochrezept:

1. Legen Sie eine Kopie der Workbench-Diskette an, und arbeiten Sie damit weiter.
2. Kopieren Sie – wenn noch nicht geschehen – die ganze Workbench-Diskette auf die Start-Partition der Festplatte, typisch ist das DH0:.
3. Kopieren Sie den Treiber »HdDisk« und sein Icon »HdDisk.info« in das Verzeichnis »Expansion« auf der Workbench-Diskette.
4. Kopieren Sie nochmals den Treiber »HdDisk« und sein Icon »HdDisk.info«, jetzt aber in das Verzeichnis »Expansion« auf der Festplatte (DH0:).
5. Laden Sie die Datei »MountList« im Verzeichnis »Devs« der Workbench-Diskette in Ihren Editor, und suchen Sie dort den Eintrag »RAD:«. Hier ändern Sie die Zeile »HighCyl=21« in

   ```
   HighCyl=7
   ```

   Hat Ihre Festplatte mehrere Partitions in der Mountlist, kopieren Sie diesen Eintrag von RAD: auch dahin. Speichern Sie die geänderte Mountlist auf der Workbench-Diskette und auf der Festplatte (je im Devs-Directory).
6. Laden Sie von der Workbench-Diskette die Datei »Startup-Sequence« im Verzeichnis »s« in Ihren Editor. Fügen Sie nach der Zeile »BinDrivers« die Zeilen von Abb. 1.5 ein:

```
failat 30
assign >NIL: INIT: DH0:
if not warn
  cd INIT:
  execute INIT:s/hd-startup
endif
failat 10

assign >NIL: RAD: exists
if warn
```

```
  echo "RAD: wird montiert"
  mount RAD:
  if not exists RAD:c
    echo "RAD: wird vorbereitet"
    relabel drive RAD: name RamDrive
    makedir RAD:c
    copy c:copy RAD:c quiet
    RAD:c/copy c:AddBuffers|assign|Bindrivers RAD:c quiet
    RAD:c/copy c:cd|echo|endif|execute RAD:c quiet
    RAD:c/copy c:failat|makedir|SetPatch RAD:c quiet
    RAD:c/makedir RAD:Devs
    RAD:c/makedir RAD:L
    RAD:c/makedir RAD:libs
    RAD:c/makedir RAD:System
    RAD:c/makedir RAD:s
    RAD:c/makedir RAD:expansion
    RAD:c/copy devs:syste-configuration RAD:Devs quiet
    RAD:c/copy L:disk-validator RAD:L quiet
    RAD:c/copy libs:icon.library RAD:libs quiet
    RAD:c/copy SYS:System/FastMemFirst RAD:SYstem quiet
    RAD:c/copy s:startup-sequence RAD:s quiet
    RAD:c/copy SYS:expansion RAD:expansion quiet
  endif
  echo "RAD: für HD-Start eingerichtet"
endif
```

Abb. 1.5: »RAD:« für den Festplattenstart

Speichern Sie die geänderte »Startup-Sequence« im Verzeichnis »s« der Workbench-Diskette und auch der Festplatte.

7. Jetzt brauchen wir nur noch die zu Beginn des Listings von Punkt 6 erwähnte Kommandodatei »hd-startup«, die in der Startpartition der Festplatte (DH0: bzw. oben mit dem logischen Namen INIT:) im s-Directory sein muß. Nun weiß ich leider nicht, wieviel Partitionen Ihre Platte hat. Ist es nur eine, ist die Sache ganz einfach. In diesem Fall lassen Sie eine vorhandene »hd-startup« wie Sie ist, bzw. erstellen Sie eine leere Datei dieses Namens, sie muß also nur existieren.

   Andernfalls müssen Sie die weiteren Partitionen anmelden und einige Pfade und Zuweisungen dafür definieren. Nehmen wir an, Sie haben noch zwei weitere Partitionen, die für das Fast-File-System eingerichtet sind und »FF0:« und »FF1:« heißen. »FF0:« enthält die komplette Workbench-Diskette mit allen Directories, weshalb Sie den logischen Namen »FASTWB:« bekommt. »FF1:« steht stellvertretend für noch eine oder weitere Partitionen, falls vorhanden. Dann könnte Ihre »HD-Startup« so wie Abb. 1.6 aussehen:

```
INIT:c/cd INIT:c
assign DEVS: INIT:Devs
assign L: INIT:L
mount FF0:
mount FF1: ;wenn vorhanden
assign FASTWB: FF0:
assign c: FASTWB:c
assign SYS: FASTWB:
assign DEVS: FASTWB:Devs
assign LIBS: FASTWB:Libs
assign L: FASTWB:L
assign S: FASTWB:S
assign FONTS: FASTWB:Fonts
```

Abb. 1.6: Eine mögliche »HD-Startup«-Datei

Damit die Änderung wirksam wird, müssen Sie nicht nur die geänderten Dateien abspeichern, sondern danach auch einen Kaltstart ausführen. Schalten Sie also den Amiga aus, und warten Sie mit dem Wiedereinschalten wenigstens 30 Sekunden.

1.12.4 Eine kleine RAD: anstatt RAM:

Wenn Sie nach jedem Start bestimmte CLI-Befehle und das S-Directory in die RAM-Disk (RAM:) kopieren, sollten Sie besser eine kleine »RAD:« anlegen. Dann ersparen Sie sich nach einem Reset das Kopieren von der Diskette. Die RAM-Disk »RAM:« bleibt dabei bestehen, nur ist sie dann leer. Ganz entfernen dürfen Sie »RAM:« nicht, weil einige Programme sie als temporären Speicher benötigen (siehe auch Abschnitt 1.12.1) . Nun wieder das Kochrezept:

1. Legen Sie eine Kopie der Workbench-Diskette an, und arbeiten Sie damit weiter.
2. Laden Sie die Datei »MountList« im Verzeichnis »Devs« der Workbench-Diskette in Ihren Editor, und suchen Sie dort den Eintrag »RAD:«. Hier ändern Sie die Zeile »HighCyl=21« in

   ```
   HighCyl=14
   ```

   Die Anzahl der Zylinder hängt natürlich von der Größe und Anzahl der Dateien und Directories ab, die Sie auf die »RAD:« kopieren wollen. Jeder Zylinder belegt 11 Kbyte, gezählt wird ab 0. Hier hätten wir also 15 Zylinder oder 165 Kbyte verbraucht. Mit dem Befehl »INFO« können Sie sich die Belegung aller Laufwerke (»RAM:« und »RAD:« inklusive) anzeigen lassen. Fügen Sie – immer noch im Eintrag »RAD:« der MountList – diese Zeile hinzu:

   ```
   BootPri=-129
   ```

Gehen Sie in den Eintrag »NEWCON:« der MountList und bringen dort diese Zeile hinzu:

```
MOUNT=1
```

Speichern Sie die geänderte MountList auf der Workbench-Diskette.

3. In der Schublade »Prefs« wählen Sie »Pointer« aus (das Icon einmal anklicken) und dann Info aus dem Workbench-Menü. Im dann erscheinenden Dialog ändern Sie den Eintrag in »Default Tool« von

   ```
   SYS:Prefs/Preferences
   ```

   in

   ```
   Preferences
   ```

   Schließen Sie die Eingabe mit <Return> ab, und speichern Sie die Änderung (»Save« anklicken). Wiederholen Sie diese Schritte für »Printer« und »Serial«.

4. Laden Sie die Datei »Startup-Sequence« im Verzeichnis »s« in Ihren Editor, und suchen Sie dort die Zeilen

   ```
   FastMemFirst
   BinDrivers
   ```

   Genau zwischen diese beiden Zeilen fügen Sie die von Abb. 1.7 ein. Die sind allerdings nur beispielhaft. Sie werden vielleicht ganz andere Copy-Befehle schreiben. Einige Befehle wie »newcli«, »endcli« oder »newshell« sind aber immer erforderlich.

```
failat 30
assign >NIL: RAD: exists
if warn
  echo "RAD: wird montiert"
  mount RAD:
  if not exists RAD:c
    echo "RAD: wird vorbereitet"
    relabel drive RAD: name RamDrive
    makedir RAD:c
    copy c:copy RAD:c quiet
    RAD:c/copy c:assign|delete|dir|makedir RAD:c quiet
    RAD:c/copy c:cd|echo|ed|if|endif|execute RAD:c quiet
    RAD:c/copy c:newcli|newshell|endcli RAD:c quiet
    RAD:c/makedir RAD:s
    RAD:c/copy s: RAD:s all quiet
    RAD:c/makedir RAD:System
    RAD:c/copy SYS:System/CLI|Shell|Shell.info
  endif
endif
```

```
failat 10
assign s: RAD:s
SYS:System/SetMap d
path RAD:c RAD:System sys:utilities sys:sytem sys:prefs sys: s:
add
assign SYS: RAD:
echo "RAD: ist fertig"
```

Abb. 1.7: Die Zeilen für eine kleine »RAD:«

Sie haben eben zwei Anweisungen geschrieben, die weiter unten nochmals vorkommen. Also entfernen Sie ziemlich am Ende (kurz vor »LoadWB«) diese Zeilen:

```
SYS:System/SetMap d
path ram: c: sys:utilities sys:System u.s.w.
```

Fügen Sie direkt vor dem Befehl »LoadWB« diese Zeilen ein:

```
path c: ram: add
assign c: RAD:c
```

Damit die Änderung wirksam wird, müssen Sie nicht nur die geänderten Dateien abspeichern, sondern danach auch einen Kaltstart ausführen. Schalten Sie also den Amiga aus, und warten Sie mit dem Wiedereinschalten wenigstens 30 Sekunden.

1.13 Tips zur Startup-Sequence

Daß die Startup-Sequence eine Kommando-Datei (auch Stapel- oder Batch-Datei genannt) ist, die bei jedem Start des Amiga automatisch ausgeführt wird, wissen Sie ja schon, doch vielleicht sind Ihnen einige der folgenden Tips neu.

1.13.1 Startup-Sequence und RAM:

Wenn Execute eine Kommando-Datei aufruft, die ihrerseits wieder Execute aufruft, speichert AmigaDos die »Return-Adresse« im logischen Gerät »T:«. Damit das schneller geht und nicht ein unnötiger Diskettenzugriff erfolgt, sollte »T:« immer im RAM liegen. Die original Startup-Sequence enthält auch deshalb folgende Befehle:

```
makedir ram:t
assign t: ram:t
```

Falls Ihre Startup-Sequence diese Befehle nicht enthält, sollten Sie sie unbedingt nachtragen.

1.13.2 Nur ein Guru nach LoadWB?

Sie haben sich eine Sparversion der Workbench-Diskette erstellt und erleben damit leider nur einen bildschönen Absturz? Sie unterbrechen den Start mit <Ctrl>+<D>, geben alle Befehle einzeln ein – immer eine gute Idee in solchen Fällen – und stellen fest, daß der Guru nach »LoadWb« auftaucht?

Dann ist die Lösung ganz einfach. Auf jeder Workbench-Diskette muß es ein Verzeichnis »libs« geben und in diesem die Datei »icon.library«.

1.14 Tips zu Preferences

Mit dem Programm Preferences kann man bekanntlich festlegen, wie sich der Amiga nach dem Start oder nach einer Änderung der Voreinstellung verhält. Doch einiges klappt scheinbar nicht, so wie man sich das vielleicht denkt. Dazu und auch sonst ein paar Tips.

1.14.1 Preferences werden nicht gespeichert?

Passiert es Ihnen, daß Sie in den Preferences etwas geändert haben, und nach dem nächsten Kaltstart ist wieder der alte Stand da? Dann machen Sie doch einmal folgendes:

Nach der Änderung von Preferences und dem Abspeichern mit »Save« legen Sie die Workbench-Diskette, mit der Sie immer starten wollen, in das interne Laufwerk (df0:) ein. Dann starten Sie per Doppelklick auf sein Icon das Programm »CopyPrefs«, und schon ist das Problem gelöst.

Wie funktioniert das?

Die Einstellungen in Preferences speichert der Amiga in einer Datei namens »system-configuration« im Verzeichnis mit dem logischen Namen »DEVS:«. Nun gibt es aber Konfigurationen, wo nach dem Start der ganze Betrieb auf einer Festplatte oder RAM-Disk läuft. Zeigt nun »DEVS:« darauf, wird die »system-configuration« auch da gespeichert. Beim Neustart hingegen wird sie von der Workbench-Diskette gelesen. »CopyPrefs« macht nun nichts weiter, als die aktuelle Preferences-Einstellung auf die Diskette im Laufwerk DF0: zu kopieren. Da muß dann natürlich auch die für den nächsten Start vorgesehene Workbench-Diskette eingelegt sein.

1.14.2 Preference-Farben werden nicht übernommen?

Als ich die Hardcopies für dieses Buch erstellen mußte, hatte ich eine tolle Idee. Man schiebe in Preferences alle drei Farbregler auf den linken Anschlag, damit wird das Bild schwarzweiß, und schon sieht man, wie es später auf dem Drucker aussehen wird. Die Idee war nicht so gut, denn die Prefences-Farben gelten nur für den Workbench-Screen (Bildschirm), viele Programme arbeiten aber mit ihrem eigenen Screen. Doch genau hier hilft das Programm »Palette«.

Mit diesem Programm können Sie die Farben des aktuellen Screens ändern, und das geht so:

1. Ordnen Sie das Palette-Iocn ziemlich weit oben auf dem Workbench-Screen an.
2. Starten Sie das Programm, das seinen eigenen Screen erzeugt, bzw. bringen sie diesen in den Vordergrund.
3. Schieben Sie den Screen so weit nach unten, bis das Palette-Icon sichtbar ist.
4. Starten Sie »Palette«.

Jetzt brauchen Sie nur noch die Farben zu regeln, bis Ihnen das Bild gefällt. Einiges ist aber noch erklärungsbedürftig.

Die Farben können Werte von 0 bis 15 annehmen, aufgrund der hexadezimalen Darstellung werden aber die Zahlen 10 bis 15 als A bis F geschrieben.

Palette läßt sich auch gut vom CLI aus fahren, weil dann ein passender Screen erscheint. Der Aufruf muß dann lauten

```
:tools/palette <planes> <res>
```

Probieren Sie einmal

```
:tools/palette 5 0
```

Für <planes> setzen Sie die Anzahl der Bit-Planes ein, wobei diese Werte von 1 bis 5 erlaubt sind:

1: 2 Farben (schwarzweiß)
2: 4 Farben
3: 8 Farben
4: 16 Farben
5: 32 Farben

Mit »res« (Resolution), einer Zahl zwischen 0 und 3, bestimmen Sie die Auslösung wie folgt:

0: 320 x 200
1: 320 x 400
2: 640 x 200
3: 640 x 400

Beachten Sie, daß bei einer X-Auflösung von 640 Pixels nur höchstens 16 Farben möglich sind.

1.14.3 Es geht auch ohne Preferences und anders

Das Programm Preferences, genau: alle Programme im Ordner »Prefs« müssen durchaus nicht auf einer (abgespeckten) Workbench-Diskette vorhanden sein. Wichtig ist die Datei »system-configuration« im Verzeichnis »devs:«. Da werden nämlich die Werte von Preferences gespeichert.

Folglich können Sie auch eine beliebige Workbench-Diskette ganz einfach auf Ihren »Stand der Technik« bringen, indem Sie Ihre »system-configuration« in das Verzeichnis »devs:« der anderen Diskette kopieren.

1.15 CLI-Programme von der Workbench starten

Zur Workbench 1.3 gehört ein Programm namens ICONX, das zwei interessante Möglichkeiten bietet. Zum einen können Sie damit typische CLI-Programme, wie zum Beispiel DIR durch einen Doppelklick auf ein Icon von der Workbench aus starten. Zum anderen können Sie sich auf diese Art den etwas aufwendigen Startup-Code in eigenen Programmen (siehe Kapitel 6.5 und 9.5) ersparen. Nur noch einmal zur Verdeutlichung: Programme, die unter der Workbench laufen sollen, müssen speziell dafür geschrieben sein.

1.15.1 Die Kommando-Datei

Zuerst brauchen Sie eine Kommando-Datei (einfacher ASCII-Text), die Sie mit einem Editor, zum Beispiel mit ED, erstellen. Nehmen wir das einfache Beispiel »DIR«, so muß in der Kommando-Datei nur dieses eine Wort stehen. Dazu müssen Sie natürlich nicht Ihren Editor anwerfen. Tippen Sie statt dessen im CLI (oder der Shell) ein:

```
type * to wbDIR <Return>
DIR             <Return>
<Ctrl>+<\>
```

(Mit dem Befehl »type * to FileName« wird die Tastatureingabe direkt in eine Datei umgeleitet, und zwar so lange, bis Sie <Ctrl>+<\> eingeben.)

1.15.2 Das Icon

Als nächstes brauchen Sie ein Icon vom Typ Projekt. Das können Sie ganz neu aufbauen (siehe Kapitel 10), aber fürs erste leihen wir uns eines, zum Beispiel das eines Basic-Programms.

```
copy :Basic/BasicDemos/music.info to wbDIR.info
```

1.15.3 ICONX einsetzen

Nun müssen Sie zur Workbench zurückkehren und das Icon »wbDIR« suchen. Schließen und öffnen Sie das Workbench-Window. Beim »Wiederaufbau« sehen Sie, wo oder worunter sich das Icon versteckt. Ziehen Sie das Icon auf eine freie Stelle, und nehmen Sie einen »Snapshot« aus dem Special-Menüs. Wählen Sie das Icon aus (einmal anklicken), und ziehen Sie dann »Info« aus dem Workbench-Menü. Im jetzt erscheinenden Dialog ändern Sie den Eintrag in »Default Tool« in

```
c:ICONX
```

Geben Sie nun »Save« ein. Wieder auf der Workbench, sollte jetzt ein Doppelklick auf das »wbDIR«-Icon reichen, um folgendes ablaufen zu lassen: Ein CLI-Fenster macht auf, das aktuelle Directory wird gelistet, eine kurze Wartezeit läuft ab, das CLI-Fenster geht wieder zu.

1.15.4 Die Feinheiten

Das war der grobe Überblick, doch etwas Feintuning ist noch möglich. Zuerst können Sie im vorherigen Punkt unter »Tool Types« noch die Parameter WINDOW und DELAY eintragen. Das Window wird wie jedes Consol-Window definiert, zum Beispiel als

```
WINDOW=CON:20/20/600/200/Mein Dir-Window
```

Der zweite Parameter heißt »DELAY=« gefolgt von einer Zahl, welche die Wartezeit bestimmt. Setzen Sie dafür null ein, sollte laut Commodore-Handbuch so lange gewartet werden, bis Sie <Ctrl>+<C> eingeben, nur bei mir funktioniert das nicht.

1.16 Ein CLI in Reserve ist immer gut

Sie kennen sicherlich die häßliche Meldung »Software Error, Task Held«, auf deutsch: Es ist ein Fehler aufgetreten, die Task wurde angehalten. Jetzt in dieses Window zu klicken, ist nicht so gut, denn dann bootet der Amiga neu und alle evtl. noch nicht gesicherten Daten sind verloren.

Doch keine Panik. Wir sind in einem Multitasking-System, mehrere Programme (Tasks) laufen quasi gleichzeitig. Eine dieser Tasks hatte den Fehler, nur sie wurde angehalten, alle anderen laufen weiter und sind OK. Man kann also weiterarbeiten, indem man einfach auf eine andere Task schaltet, sprich, deren Window anklickt. Auf diese Art kann man zum Beispiel vom CLI auf die Workbench gehen, doch was ist, wenn man im CLI gestartet war, und das die einzige Task ist? Da kann man ja noch nicht einmal seinen Debugger anwerfen, um vielleicht dem Fehler noch auf den Grund zu kommen.

Die Abhilfe ist ganz einfach. Öffnen Sie immer ein zweites CLI (mit NEWCLI) oder eine zweite Shell. Deren Window können Sie ja verkleinern und in irgendeine Ecke schieben. Da bleibt es dann als Reserve für den Fall der Fälle.

2 CLI-Tips

2.1 CLI oder Shell, was nehme ich?

Klären wir doch erst einmal, was was ist. CLI steht für »Command Line Interpreter«, also ein Programm, das eine Kommandozeile (eine Zeile, die Sie eingeben) interpretiert und ausführt. Damit erhalten Sie eine tastaturorientierte Bedienoberfläche ähnlich wie die der PCs (MS-DOS).

Im Gegensatz dazu steht die Workbench (Werkbank) als eine grafikorientierte Bedienoberfläche, die sie ja schon bestens kennen. Im Normalfall – Sie arbeiten mit der zum Amiga gelieferten Workbench-Diskette – startet der Amiga auch mit der Workbench. In das CLI gelangen Sie erst durch Anklicken des CLI-Piktogramms in der Schublade »System« oder mittels speziell präparierter Start-Disketten.

Auf der Workbench-Diskette befindet sich aber noch ein Icon mit dem Titel »Shell«. Als Shell (Schale) bezeichnet man in vielen Betriebssystemen die (in aller Regel) tastaturorientierte Bedienoberfläche. Die Schale umhüllt den Betriebssystem-Kern und ist gleichzeitig das Interface zwischen dem Anwender und dem Betriebssystem. Es gibt aber auch Shells für Entwicklungssysteme, und damit kommen wir dem Unterschied zum CLI schon etwas näher.

Eine gute Shell ist mehr als ein Kommando-Interpreter. Sie ist frei konfigurierbar (an die speziellen Bedürfnisse eines Anwenders anpaßbar), sie bietet einen Editor und eine – wenn manchmal auch nur einfache – Programmiersprache.

Eine Super-Shell hat UNIX – übrigens in vielen Teilen Vorbild für das Amiga-Betriebssystem – aber die wäre für den Amiga zu groß. So komfortabel und kompliziert wie die UNIX-Shell ist die Amiga-Shell nicht, aber sie bietet doch schon allerhand. Auf jeden Fall kann sie eine Menge mehr als das CLI und braucht dafür kaum mehr Speicher (so rund 3 Kbyte).

Damit ist die Entscheidung eigentlich klar. Wenn Sie mit einer tastaturorientierten Bedienoberfläche arbeiten wollen – und das ist oft ganz vorteilhaft und manchmal sogar ein Muß – dann nehmen Sie die Shell.

Wenn Sie hingegen von eigenen Programmen aus andere Programme aufrufen, zum Beispiel AmigaDOS in Basic nutzen wollen, dann nehmen Sie das CLI. Das ist einfacher, und der Bedienkomfort spielt dabei keine Rolle, den bietet ja Ihr Basic-Programm.

2.1.1 Damit die Shell läuft

Wenn Sie den Amiga mit der Original-Workbench-Diskette 1.3 oder einer Kopie davon starten, steht Ihnen die Shell zur Verfügung. Ist das nicht der Fall oder läuft sie nicht richtig, dann fehlt etwas auf dieser Diskette. Das können Sie aber leicht nachtragen.

In der Startup-Sequence müssen (nicht unbedingt zusammen) diese Zeilen stehen:

```
resident CLI L:Shell-Seg SYSTEM pure add
mount newcon:
```

In der Datei »devs/mountlist« muß folgender Eintrag vorhanden sein:

```
NEWCON:
  Handler = L:Newcon-Handler
  Priority = 5
  Stacksize = 1000
#
```

Im Verzeichnis »l« müssen diese Dateien vorhanden sein:

```
Newcon-Handler
Shell-Seg
```

Im c-Directory sollte »NewShell« stehen, wenn Sie eine neue Shell mit diesem Befehl öffnen wollen. Für den Start per Doppelklick von der Workbench aus benötigen Sie »Shell« und »Shell.info« von Ihrem Workbench-Original.

Schließlich sollten Sie noch das Shell-Icon anklicken, Info aus dem Workbench-Menü wählen und dann folgende Einträge sehen:

```
Stack: 4000
Default Tool: SYS:System/CLI
Tool Types: WINDOW=NEWCON:0/20/640/200/AmigaShell
```

ALIAS funktioniert nur, wenn die Shell aktiviert und nach dem obigen Regeln vollständig ist.

2.2 Der Shell-Editor

Im allgemeinen werden Sie die Shell nutzen, um Programme oder Kommando-Dateien aufzurufen. Dazu geben Sie ein Kommando ein, zum Beispiel

```
copy antom to fritz
```

und Sie erhalten dann vielleicht die Meldung »can't open antom for input – object not found«. Die Datei »antom« gibt es also nicht, ist auch klar, Sie wollten ja eigentlich auch »anton« tippen. Im CLI hätten Sie jetzt Pech gehabt und müßten das ganze Kommando noch einmal eingeben. In der Shell drücken Sie die Taste <cursor auf> und schon ist der Befehl wieder da. Sie fahren den Cursor auf das »m«, ändern das in »n«, drücken <Return>, und der Befehl wird ausgeführt.

Die Shell verfügt über einen sogenannten History-Buffer, zu deutsch Befehlsspeicher, in dem sie sich alle Ihre Eingaben – auch die fehlerhaften – merkt. Mit den beiden Tasten <Cursor auf> und <Cursor ab> können Sie durch diesen Puffer fahren und den aktuell gezeigten Befehl mit <Return> wieder ausführen. Sie können den Befehl auch ändern und ihn dann ausführen. Dazu müssen Sie nicht einmal den Cursor an das Ende der Zeile stellen. Es reicht, irgendwo <Return> zu drücken.

Hier alle Tasten des Shell-Editors im Überblick:

`<Cursor links/rechts>`

bewegt den Cursor innerhalb der Zeile.

`<Shift>+<Cursor links/rechts>`

bewegt den Cursor an den Anfang/das Ende der Zeile.

`<Cursor auf/ab>`

Bewegt den Cursor durch den Befehlsspeicher, läßt vorher eingegebene Zeilen erscheinen.

`<Zeichen>+<Shift>+<Cursor auf>`

sucht im Befehlsspeicher einen Befehl der mit »Zeichen« beginnt. Zum Beispiel wird man mit <co Shift Cursor-auf> den zuletzt eingegebenen Copy-Befehl erreichen.

`<Backspace>`

löscht das Zeichen links vom Cursor.

`<Del>`

löscht das Zeichen unter dem Cursor.

`<Ctrl>+<k>`

löscht die Zeichen vom Cursor bis Zeilenende.

`<Ctrl>+<u>`

löscht die Zeichen vom Zeilenanfang bis zum Cursor.

`<Ctrl>+<w>`

setzt den Cursor auf den nächsten Tabulator oder an das Zeilenende.

`<Ctrl>+<x>`

löscht die gesamte Zeile.

2.3 Eigenes Prompt

Wenn Sie das CLI oder die Shell starten, führt der Amiga zuerst die Datei »CLI-Startup« bzw. »Shell-Startup« aus, jedenfalls dann, wenn diese Dateien im s-Directory vorhanden sind. Die wahrscheinlich einzige Zeile im »CLI-Startup« bzw. die erste Zeile »Shell-Startup« lautet

```
prompt "%N>"
```

Dieser Befehl erzeugt das sogenannte Prompt, also die Eingabeaufforderung. Sie können die Startup-Datei mit einem Editor (z.B. ED) ändern und ein neues Prompt erfinden. Ab dem nächsten Aufruf von CLI oder Shell ist die Änderung wirksam. Sie können aber auch den Prompt-Befehl direkt eintippen. Dann gilt dieses Prompt so lange wie Sie kein anderes eingeben oder die Shell/ das CLI nicht verlassen.

Für das Prompt selbst gilt zuerst, daß zwischen den Anführungszeichen jeder beliebige Text stehen darf, zum Beispiel:

```
prompt "Dein nächster Befehl bitte:"
```

Ansonsten sind aber noch einige Sonderzeichen zulässig, die folgendes bewirken:

%N gibt die aktuelle Task-Nummer aus (CLI 1, CLI 2 usw.)
%S gibt den aktuellen Pfad aus.

Das läßt sich auch kombinieren. Zum Beispiel als

```
prompt "%N%S>"
```

Ist man dann gerade im CLI 1 (oder der Shell 1), und zwar auf der Disk »DH0(20MB)« im s-Directory, so lautet das Prompt

```
1.DH0(20MB):s>
```

2.4 Update auf Shell

Sie haben sich an die Vorteile der Shell gewöhnt und landen über einige Programme doch wieder im alten CLI. Der Grund ist, daß diese Programme anstatt NEWCON, was für die Shell steht, noch CON anziehen, und das ergibt das alte CLI-Fenster.

Deshalb wollen wir mit einem Disk- oder File-Editor – notfalls auch mit einem Editor, der alles schluckt (z.B. DevPac) – so ein Programm ändern, auch »patchen« genannt. Das Problem dabei ist, daß man beim Patchen nur ändern, aber keine neuen Zeichen hinzufügen kann. Deshalb verwenden wir diese Taktik:

- Im zu patchenden Programm nach jedem Auftreten von »CON:« suchen und das durch »COX:« ersetzen.
- Eine »mountlist« für »COX:« erstellen.

Nun der Reihe nach:

1. Legen Sie eine Kopie der Workbench-Diskette an, und kopieren Sie auf diese das zu patchende Programm.
2. Ersetzen Sie im zu patchenden Programm jedes Auftreten von »CON:« durch »COX:«.
3. Laden Sie die Datei »mountlist« (ist im Ordner »devs«) in Ihren Editor und ändern Sie den Eintrag »NewCon:« in »COX:«. Besser ist es, diesen Eintrag zu kopieren und in der Kopie den Namen zu ändern. Dann können alte und neue Programme mit ihrem jeweiligen Con-Handler arbeiten. Das sähe dann so aus:

```
/* Das ist schon da: */
NEWCON:
  Handler = L:Newcon-Handler
  Priority = 5
  StackSize = 1000
#

/* Das fügen Sie ein: */
COX:
  Handler = L:Newcon-Handler
  Priority = 5
  StackSize = 1000
#
```

4. Ergänzen Sie die Startup-Sequence um die Zeile
   `mount COX:`

Nun müssen Sie den Amiga neu booten und können Ihren Patch austesten.

Zwei Hinweise: Im Ordner »l« muß sich immer der NewCon-Handler befinden. Wenn Sie mit der erweiterten »mountlist« arbeiten (siehe Punkt 3), sind Sie flexibler, weil dann für andere Workbench-Disketten nur noch diese Schritte erforderlich sind:

1. Erweiterte »mountlist« kopieren und dabei das Original überschreiben.
2. Das Programm mit dem alten »CON:« in »COX:« patchen.
3. »mount COX:« in die Startup-Sequence eintragen.

2.5 RESIDENT ist besser als RAM-Disk?

Aus früheren Amiga-Tagen taucht immer wieder der tolle Tip auf, häufig benötigte Programme in die RAM-Disk zu kopieren. Die Begründung ist ganz simpel. Eine RAM-Disk ist tausendmal schneller als eine Diskette und unterliegt keinerlei mechanischer Abnutzung.

Doch die Sache hat einen gewaltigen Haken. Eine RAM-Disk emuliert eine normale Diskette im Hauptspeicher. Rufen Sie ein Programm auf, wird es von der Diskette in den Hauptspeicher geladen. Rufen Sie ein Programm auf, das auf der RAM-Disk steht, wird es von dort geladen. Praktisch wird ein RAM-Bereich in einen anderen kopiert. Ergebnis: Das Programm steht zweimal im RAM!

Genau das verhindert ein neues Feature der Version 1.3 namens RESIDENT. Die Syntax ist ganz einfach diese:

```
resident NameDesProgramms
```

Das klingt doch gut oder? Was soll da noch das dicke Fragezeichen? Nun, die Sache hat leider einen Haken. Nicht jedes Programm ist für RESIDENT geeignet. Schuld hat das Multitasking. Das heißt nun aber nicht, daß die ungeeigneten Programme nicht multitaskingfähig sind, das Problem ist ein anderes.

Rufen zwei Tasks dasselbe Programm auf, wird es von beiden Tasks in verschiedene RAM-Bereiche geladen, jede Task hat ihre eigene Kopie des Programms. RESIDENT bewirkt hingegen, daß ein Programm mit diesem Attribut nicht mehr in den RAM geladen wird, wenn es da schon ist. Es wird in diesem Fall einfach gestartet.

Nun kommen wir auf den Knackpunkt. Nehmen wir an, Task A hat das Programm Snooper gestartet, und es läuft jetzt. Währenddessen kommt Task B auf die Idee, Snooper aufzurufen. Da es resident ist, wird das Programm nochmals gestartet. Nun kommt die Task-Umschaltung zum Tragen. Folglich hat Snooper immer abwechselnd eine Zeitlang für A und dann für B zu arbeiten. Beide Tasks wollen natürlich ganz verschiedene Dinge von Snooper, wie soll er das schaffen? Nun, Snooper muß über zwei Eigenschaften verfügen. Es muß – wie eben geschildert – durch zwei

oder mehr Programme gleichzeitig ausführbar sein, »shareable« heißt das auf neudeutsch, und es muß reentrantfähig sein, auf deutsch, man muß es an beliebiger Stelle unterbrechen und immer wieder aufrufen können.

Wie man so ein Programm schreibt, kann ich auf die Schnelle nicht schildern, aber dafür ein paar Tips, die zeigen, wie Sie RESIDENT auch so nutzen können.

2.5.1 Prüfen, ob residentfähig

Zuerst können Sie bei neueren Programmen sehr einfach feststellen, ob sie residentfähig sind oder nicht. Ist ein Programm residentfähig, ist das P-Bit (Protect-Bit = Schutz-Bit) gesetzt. Wechseln Sie einmal ins C-Directory, und geben Sie LIST ein. Programme mit dem Buchstaben p sind residentfähig.

2.5.2 Residentfähig mit Trick

Ist ein Programm nicht residentfähig, kann man es trotzdem dazu machen. Dafür reicht die schlichte Eingabe von

```
resident NameDesProgramms pure add
```

In diesem Fall sollten Sie dann selbst darauf achten, daß es nicht zu Problemen kommt. Rufen Sie also das Programm nicht nochmals auf, während es noch läuft und – das ist schon schwieriger – lassen Sie auch keine Programme laufen, die ihrerseits das residente Programm aufrufen könnten.

2.6 RUN nachholen

Sie wissen, daß Sie mit »RUN Befehl« eine weitere CLI- oder Shell-Task aufmachen können, so daß der Befehl dann »von alleine« läuft, während Sie so lange mit dem Amiga etwas anderes machen können. Gelobt sei das Multitasking!.

Nun haben Sie aber beispielsweise den Ausdruck eines langen Listings gestartet und den Drucker auch noch auf Schönschrift geschaltet. Doch anstatt »run type snooper.bas to prt:« hatten Sie nur »type snooper.bas to prt:« getippt. Die nächste Stunde ist der Drucker beschäftigt, und der Amiga ist so lange blockiert oder?

Er ist es nicht, nur das eine CLI- oder Shell-Fenster nimmt so lange keine Eingaben mehr an, bis der Drucker fertig ist. Sie können aber durchaus noch das Shell-Symbol doppelklicken – dafür ggf. die Workbench-Diskette öffnen – und damit ein zweites Shell-Fenster aufmachen. In diesem können Sie wieder arbeiten und natürlich auch mit anderen Programmen, die Sie auch noch starten können.

2.7 Type anstatt Copy

Fiel Ihnen auf, daß im Abschnitt 2.6 anstatt dem üblichen »copy to prt:« hier »type« eingesetzt wurde? Der Type-Befehl ist für die Ausgabe von Textdateien gedacht, doch mit Zusatz »to Ziel« kann man ihn von der standardmäßigen Ausgabe auf dem Schirm auf ein Gerät oder eine Datei umleiten. Es ist also zulässig:

```
type snooper.bas to prt:
```

oder

```
type snooper.bas to df1:snooper.bak
```

Und was ist der Vorteil? Nun, »type« belegt rund 2 Kbyte während »copy« schon fast 10 Kbyte schluckt. Damit wird »type« zuerst einmal schneller von der Diskette geladen, und das Laufwerk wird geschont. Viele Anwender halten aber auch den Copy-Befehl in ihrer RAM-Disk. Doch wenn der hauptsächlich nur gebraucht wird, um Textdateien zu kopieren oder auszudrucken, dann kann man 8 Kbyte auf der RAM-Disk sparen.

2.7.1 Type in hex

Programm-Code oder Daten ergeben auf dem Bildschirm nur unsinnige und kaum zu überblickende Zeichenfolgen, wenn man sie mit TYPE ausgibt. Manchmal will man sich solche Dateien aber ansehen, zum Beispiel, um nach einer Versionsbezeichnung oder dem Datum zu suchen. In diesem Fall nehmen Sie die Option h wie hexadezimal. Für die Datei mit dem Namen xxx tippen Sie

```
type xxx opt h
```

Es werden immer je Zeile 16 Byte in hex und dann nochmals in ASCII dargestellt, zum Beispiel so:

```
0000: 000003E7 00000003 414E4F4E 5F4D4F44    ........ANON_MOD
0010: 554C4500 000003E8 00000001 54455854    ULE.........TEXT
0020: 000003E9 00000014 48790000 014D4879    ........Hy...MHy
0030: 000000DE 48790000 006F4879 00000024    ....Hy...oHy...$
0040: 4EB90000 00004FEF 00104E75 5A31203D    N.....O...NuZ1 =
0050: 20256C64 20205A32 203D2025 6C642020     %ld  Z2 = %ld
0060: 205A3320 3D20256C 640A0000 00000000     Z3 = %ld.......
0070: 00000000 00000000 000003EC 00000001    ................
0080: 00000000 00000014 00000000 000003EF    ................
0090: 81000002 5F707269 6E746600 00000001    ...._printf.....
00A0: 0000001A 01000002 5F6D6169 6E000000    ........_main...
00B0: 00000000 00000000 000003F2             ............
```

Die Zahlen vor dem dem Doppelpunkt sind der Zähler oder die relativen Adressen (in hex). Was als Text darstellbar ist, erscheint auch so, für die nicht druckbaren Zeichen werden Punkte eingesetzt.

Sie können die Ausgabe aber auch in eine andere Datei umleiten, um sie dann mit einem Editor besser betrachten zu können. So können Sie dann auch rückwärts und wieder vorwärts blättern. Für die Umleitung von xxx in die Datei yyy tippen Sie:

```
type xxx to yyy opt h
```

2.8 Sprache im CLI

Ab der Workbench 1.3 gibt es einen neuen Handler namens »SPEAK:« und der ist natürlich »input-fähig«. Somit kann man jede Textdatei auf »SPEAK:« umleiten und damit auch im CLI (oder der Shell) Sprache ausgeben, zum Beispiel so:

copy ansage to speak:

Der Text wird nach den Regeln der englischen Sprache gesprochen, doch lassen Sie sich dadurch nicht beeindrucken. Schreiben Sie Text hin, hören Sie sich ihn an, und ändern Sie dann nur die Wörter, die falsch klingen.

Wenn Sie nur mal so einige lustige Sprüche hören wollen, geben Sie doch einfach mal ein:

```
dir > speak:
```

2.9 Pfadfinder

Sie haben es schon gemerkt: Bestimmte Programme werden ausgeführt, egal in welchem Verzeichnis Sie gerade sind, andere sind plötzlich unbekannt, die Eingabe ihres Namens bringt nur eine Fehlermeldung. Der Grund ist folgender:

Geben Sie im CLI oder in der Shell irgendeinen Text ein, zum Beispiel »Anton«, nimmt AmigaDOS an, daß dies ein Programm-Name ist. Dieses Programm versucht es dann zu laden, die Frage ist nur, wo es auf der Disk zu finden ist. Da es zu lange dauern würde, bei jedem Aufruf alle angeschlossenen Disketten und Festplatten zu durchsuchen, gilt diese Folge:

1. Aktuelles Verzeichnis.
2. Verzeichnis »:c«.
3. Verzeichnisse, auf die ein Pfad gesetzt ist.

Ist das Programm in keinem dieser Verzeichnisse, gibt es eine Fehlermeldung, natürlich auch dann, wenn sich hinter dem Namen gar kein Programm, sondern nur ein Datenfile verbirgt. Wollen Sie ein Programm von überall her laden, haben Sie zwei Möglichkeiten:

1. Kopieren Sie das Programm in das c-Directory der Workbench-Diskette, löschen Sie es im Quellverzeichnis, um es nicht doppelt auf einer Diskette zu haben.
2. Setzen Sie einen Pfad auf das Directory, in dem sich das Programm befindet.

Letzteres ist ganz einfach. Wollen Sie beispielsweise MORE, das sich im Utilities-Verzeichnis befindet, auf diese Art ansprechen, tippen Sie ein:

```
path :utilities add
```

Damit wird dieser Pfad der aktuellen Pfadliste hinzugefügt. (add). Würden Sie »path reset Pfadnamen« eingeben, würden erst die aktuellen Pfade gelöscht. Sie können durchaus mehrere Pfade (bis zu 10) in einem Befehl aufführen, sie müssen nur durch wenigstens eine Leerstelle getrennt sein. Beispiel:

```
path :utilities :basic :lc/progs add
```

2.9.1 Güterabwägung

Bliebe nur die Frage zu klären »wie, wo, was?«. Zuerst können Sie den Path-Befehl

1. direkt eintippen,
2. in die Startup-Sequence eintragen,
3. in die Datei CLI-Startup oder Shell-Startup eintragen.

Alternativ können Sie die Programme auch in das C-Directory kopieren. Für letzteres gilt, daß nur wirklich häufig genutzte Programme nach »:c« sollten. Mit der Länge des Verzeichnisses wächst nämlich auch die Suchzeit.

Das gleiche Problem entsteht mit der wachsenden Länge der Pfadliste. Es wirkt hier sogar noch schlimmer, weil ggf. zwischen den Laufwerken gewechselt werden muß.

Für ganz seltene Fälle, wo Sie nur keine Lust haben, mehrmals lange Pfadnamen zu tippen, geben Sie den Befehl direkt ein. Er wirkt dann nur für die aktuelle Session (bis Sie den Amiga ausschalten oder »resetten«).

Für Programme, die Sie nur im CLI oder der Shell nutzen, nehmen Sie Punkt 3. Die Pfade werden dann erst gesetzt, wenn Sie die Shell oder das CLI aufrufen. Da »CLI-Startup« und »Shell-Startup« unabhängig voneinander sind, können Sie so unterschiedliche Pfadlisten realisieren.

2.10 ASSIGN spart Tipparbeit

Der Befehl ASSIGN hat zwei Aufgaben. Zuerst kann er eine Menge Tipparbeit ersparen, weil sich damit sehr lange Pfadnamen durch ein sogenanntes logisches Gerät ersetzen lassen. Geben Sie zum Beispiel

```
assign b: :basic
```

ein, so müssen Sie ab jetzt nur noch »b:« anstatt Basic tippen, zum Beispiel in »run b:AmigaBasic«. Hier lohnt sich das kaum, doch auch längere Zuweisungen wie

```
assign bc: dh0:Compiler/Basic/HB.Compiler
```

sind möglich. Die zweite Anwendung von ASSIGN ist Pflicht bei den meisten Entwicklungssystemen. Zum Beispiel gilt für Lattice-C:

```
assign LC: dh0:lc/c
assign LIB: dh0:lc/lib
assign INCLUDE: dh0:lc/include
assign QUAD: RAM:
```

Der Compiler und der Linker arbeiten mit den logischen Geräten LC:, LIB:, INCLUDE: und QUAD:. Wo sich die zugehörigen Directories befinden, müssen Sie nicht wissen. AmigaDOS und ASSIGN machen das schon. Vorteil der Geschichte: Sie können das Entwicklungssystem installieren wie Sie wollen, auf einer Festplatte oder drei Disketten-Laufwerken oder in einer riesigen RAM-Disk oder wie auch immer. Wenn Sie dann mit ASSIGN diesen vier logischen Geräten die Pfade zuweisen, läuft alles.

Auch hier haben Sie die freie Auswahl. Sie können ASSIGN

1. direkt eintippen,
2. in die Startup-Sequence eintragen,
3. in die Datei CLI-Startup oder Shell-Startup eintragen.

Wenn Sie nur ASSIGN tippen, erhalten Sie eine Liste der aktuellen Zuweisungen. Schreiben Sie »assign name:«, also ohne den Pfad, wird die Zuweisung gelöscht. Geben Sie einen neuen Pfad an, wird der alte überschrieben.

2.11 ALIAS spart Tipparbeit

Vorab: ALIAS funktioniert nur, wenn die Shell aktiviert und gemäß Kapitel 2.1 vollständig ist.

Ab der Workbench 1.3 verfügt der Amiga über eine eigene Makrosprache. Ein Makro ist ein Kürzel für einen langen Befehl oder eine Befehlsfolge. Erzeugt werden diese Makros mittels des Befehls ALIAS. Ein einfaches Beispiel zeigt schon das Prinzip. Tippen Sie in der Shell

```
alias d dir
```

ein, und ab sofort brauchen Sie anstatt »dir« nur noch »d« zu tippen. Das bringt natürlich noch nicht viel, aber was halten Sie von diesem Beispiel:

```
alias t run :basic/AmigaBasic :basic/terminal
```

Hier wurde der Buchstabe t zum Kommando »run :basic/AmigaBasic :basic/terminal«. Ich brauche das, um – wo ich auch bin – ganz schnell mein Terminalprogramm aufrufen zu können.

Es kommt aber noch schöner. Ab und zu will ich natürlich auch andere Basic-Programme starten, nicht aber immer den ganzen Text tippen. Dafür habe ich dieses Makro:

```
alias b run :basic/AmigaBasic :basic/[]
```

In diesem Fall stehen die rechteckigen Klammern als Platzhalter für einen Text, den man noch eingeben muß. Um zum Beispiel das Programm »Snooper« im Verzeichnis »:basic« zu starten, reicht jetzt

```
b snooper
```

Wenn Sie ALIAS öfter brauchen, sollten Sie die entsprechenden Anweisungen in die Datei »Shell-Startup« (im s-Directory) eintragen. In der Startup-Sequence lohnt sich das nicht, weil ALIAS nur mit der Shell läuft.

2.12 PC-DOS-Befehle im AmigaDOS

Wer vom PC kommt oder in seinem Amiga eine PC-Karte hat, will nicht dauernd umdenken. Leider kann man dem PC nicht die AmigaDOS-Befehle beibringen, doch umgekehrt geht das dank ALIAS (siehe oben) ohne weiteres.

```
alias del delete
alias md makedir
alias cls echo "*E[0;0H*E[J"
```

Der letzte Befehl ist etwas tricky. Er nutzt die Fähigigkeit des CLI oder der Shell, sogenannte Escape-Sequenzen ausführen zu können.

2.13 Blau auf weiß

Im vorherigen Trick wurde das Thema Escape-Sequenzen angesprochen. Hier sind noch zwei nützliche:

```
alias reverse echo "*E[0;0H*E[41;30m*E[J"
alias normal echo "*E[0;0H*E[40;31m*E[J"
```

»reverse« schaltet das Window so um, daß blaue Schrift auf weißem Hintergrund erscheint. »normal« stellt den alten Zustand wieder her.

2.13.1 Escape-Sequenzen und Control-Codes

Für das CLI (oder die Shell) gelten die folgenden Escape-Sequencen:

```
<Esc>+<[>0M  Normalschrift
         1M  Fettschrift
         2M  Schriftfarbe 2 (schwarz)
         3M  Kursivschrift
         4M  Unterstrichene Schrift
         7M  Invers ein
         30M  Schriftfarbe 0 (blau)
         31M  Schriftfarbe 1 (weiß)
         32M  Schriftfarbe 2 (schwarz)
         33M  Schriftfarbe 3 (orange)
         40M  Hintergrundfarbe 0
         41M  Hintergrundfarbe 1
         42M  Hintergrundfarbe 2
         43M  Hintergrundfarbe 3
         nt   n = Anzahl der Zeilen
         nu   n = Anzahl der Spalten
         nx   n = Abstand zum linken Rand
         ny   n = Abstand zum rechten Rand
```

Ganz nützlich sind auch die folgenden Control-Codes:

```
<Ctrl>+<N>  alternativer Zeichensatz
<Ctrl>+<O>  originaler Zeichensatz
<Ctrl>+<L>  Fenster löschen
<Ctrl>+<G>  System-Beep
<Ctrl>+<J>  Cursor abwärts
<Ctrl>+<K>  Cursor aufwärts
<Ctrl>+<I>  Tabulator
```

Wie die Beispiele zeigen, müssen Sie in einem Text, der mit TYPE oder ECHO ausgegeben werden soll, anstatt der Tastenkombination <Esc>+<[> »*E[« einsetzen.

2.14 Vorsicht Leerzeichen!

Ein Leerzeichen in einem Datei- oder Verzeichnis-Namen ist zulässig, kann aber auch zur Falle werden. Nehmen wir an, auf der Disk gibt es Verzeichnis namens »Alles Spiele«. Dann führt

```
dir Alles Spiele
```

zu einer Fehlermeldung, weil AmigaDOS ein Directory names »Alles« nicht kennt. Weiter liest es den Namen nicht, denn Leerstellen gelten als Trennzeichen zwischen den Parametern. Folglich ist »Spiele« auch ein Parameter, doch so eine Option kennt der Dir-Befehl (leider) nicht.

Doch auch das Problem ist lösbar. Setzen Sie den Namen einfach in Anführungszeichen.

```
dir "Alles Spiele"
```

funktioniert, muß es auch, weil der Erfinder dieses Namens mit »makedir „Alles Spiele"« das Ding so angelegt hatte.

2.15 CLI-Befehle und nur ein Laufwerk

Das Problem kennen Sie sicherlich. Sie wechseln die Diskette und geben einen CLI-Befehl ein, zum Beispiel

```
dir df0:
```

Daraufhin fordert der Amiga wieder Ihre Boot-Diskette an und dann die andere. Wollen Sie nicht Diskjockey spielen, hilft folgender Trick. Geben Sie ein:

```
dir df0: ?
```

daraufhin erscheint die Meldung

```
DIR, OPT/K,ALL/S,DIRS/S,FILES/S,INTER/S:
```

Sie sehen, daß dies eine Kurzerklärung des DIR-Befehls ist. Der Amiga wartet jetzt auf die Eingabe einer Option. Geben Sie nur <Return>, wird der normale Dir-Befehl ausgeführt. Würden

Sie zum Beispiel »all«+<Return> eingeben, würden auch die Dateien in allen Unterverzeichnissen gelistet.

Wichtig ist jedoch die Tatsache, daß der Amiga an dieser Stelle wartet. Sie können jetzt in aller Ruhe die andere Diskette einlegen und dann <Return> geben. Seien Sie dabei nicht zu schnell, sondern warten Sie, bis der Drive wieder ruhig ist. Andernfalls erscheint die Meldung »no disk present in unit 0«.

Wenn Sie die Mini-Hilfe nicht sehen wollen, können Sie auch »+« anstatt des Fragezeichens eingeben. Doch so ganz kann ich das nicht empfehlen, weil man dann nur noch den einsamen Cursor in einer Zeile sieht, also sozusagen das Prompt (die Eingabeaufforderung) fehlt.

Wie funktioniert das?

Wie alle CLI-Befehle ist auch »dir« ein ganz normales Programm und steht im C-Directory auf der Boot-Diskette. Es wird also erst durch den Aufruf in den Speicher geladen und läuft im Normalfall auch sofort los. Dann wirkt es natürlich auf die eingelegte Diskette. Durch den Zusatz »?« (oder »+«) wartet es auf Ihre Eingabe, Sie haben dann die Möglichkeit, die Diskette zu wechseln.

2.16 Mehr mit More

Die Ausgabe von Texten mit TYPE hat bei längeren Scripts den Nachteil, daß das Bild rollt. Es an der richtigen Stelle anzuhalten, ist gar nicht so einfach. Die Lösung heißt MORE. Doch MORE kann viel mehr, als alle x Zeilen auf einen Tastendruck zu warten, lassen Sie sich überraschen.

```
more NameDerDatei
```

wirkt wie TYPE, hält aber die Eingabe an, wenn die Window-Grenze erreicht ist und wartet auf einen Tastendruck. MORE befindet sich allerdings im Verzeichnis Utilities der Workbench-Diskette. Die Eingabe lautet also korrekt

```
:utilities/more NameDerDatei
```

Wenn Ihnen das zu umständlich ist, setzen Sie den Pfadfinder (siehe 2.18) ein. Für den Dateinamen müssen Sie leider immer den kompletten Pfadnamen eintippen.

Sie können MORE auch von der Workbench aus starten. Dazu wählen Sie zuerst das MORE-Icon aus (einmal anklicken) und doppelklicken dann auf die Datei. Sie können aber MORE auch ohne die Angabe eines Dateinamens starten. In diesem Fall fragt Sie MORE danach.

MORE wartet – wie gesagt – nach jeder Seite auf einen Tastendruck, doch das kann mehr als nur die Leertaste sein. Es gilt:

Blättern:

```
<Leertaste>     nächste Seite
<Backspace>     1 Seite zurück
<Return>        1 Zeile weiter
<               auf die erste Seite
>               auf die letzte Seite
<%><Zahl>       geht auf %-Zahl der Dateilänge
<Esc>           MORE hält an, dann fortsetzen mit <h>
                oder beenden mit <q>
```

Suchen:

```
</><Suchtext>   sucht den auf »/« folgenden Text,
                unterscheidet Groß/Kleinbuchstaben
<.><Suchtext>   sucht den auf ».« folgenden Text,
                unterscheidet nicht Groß/Kleinbuchstaben
<n>             sucht das nächste Auftreten von »Suchtext«
```

Extras:

```
<h>             Hilfe, bringt sinngemäß diese Tabelle
<q>             Quit, beendet MORE, dito <Ctrl>+<C>
<Shift><e>      Wechsel zum Editor, der in ENV:EDITOR
                festgelegt ist (siehe Abschnitt 2.17).
```

2.17 ENV, die Umgebungsvariablen

Ab der Workbench 1.3 gibt es die sogenannten Umgebungsvariablen, neudeutsch »Environment«. Über dieses Feature verfügen die meisten Betriebssysteme, und besonders aufregend ist es nicht. Das Environment ist ein Speicherbereich, der allgemein bekannt ist. Er hat die Funktion einer Pin-Wand. Sie hinterlegen dort Nachrichten. Programme können dann (müssen nicht) dort nachsehen, ob es eine Nachricht für sie gibt.

Dazu müssen Sie und das Programm bestimmte Vereinbarungen treffen. Haben Sie das Programm nicht geschrieben, steht in seinem Handbuch, wie die Vereinbarung aussieht bzw. was Sie in »Sachen Environment« tun müssen.

Nehmen wir das Beispiel von MORE. Dieses Programm wechselt nach dem Kommando <Shift>+<e> zu Ihrem Lieblings-Editor. Dazu schaut es im Environment nach, ob und wie Sie eine

Umgebungsvariable namens EDITOR definiert haben. Sie müssen dazu eingeben oder (besser) in der Startup-Sequence schreiben:

```
setenv EDITOR ":tools/MEMACS"
```

Wie funktioniert das?

Commodore arbeitet – gemessen an anderen Betriebssystemen – wenigstens vorerst mit einem ganz billigen Trick. In der Startup-Sequence oder in der von ihr aufgerufenen Kommandodatei »StartupII stehen unter anderem diese Kommandos:

```
makedir ram:env
assign ENV: ram:env
```

Damit wird in der RAM-Disk das Verzeichnis »env« angelegt. Zusätzlich wird des einfacheren Zugriffs wegen dafür auch der logische Gerätename »ENV:« vergeben.

Der Befehl »setenv EDITOR „:tools/MEMACS“« macht nun folgendes: Er legt im Verzeichnis »ENV:« eine Datei namens EDITOR an und schreibt dann in diese Datei den Text »:tools/MEMACS«. Folglich muß MORE nur diese Datei öffnen, den Text auslesen und ihn mittels der Execute-Funktion als Befehl ausführen lassen. Der neue Befehl GETENV, zum Beispiel als

```
getenv EDITOR
```

hat dieselbe Wirkung wie

```
type env:EDITOR
```

Ob das immer so bleibt, ist eine andere Frage. Commodore könnte sich eines Tages durchaus entscheiden, die Umgebungsvariablen in einem RAM-Bereich zu halten, der nicht die RAM-Disk ist. Deshalb sollten Sie schon jetzt nur SETENV und GETENV anwenden.

2.18 Es geht auch ohne Execute

Sie wissen, daß man sehr einfach eine Kommando-Datei aufbauen kann, weil das letztlich nur ein Text-File ist, das gültige CLI-Befehle enthält. Sie wissen ferner, daß man eine solche Datei mit

```
execute NameDerDatei [Parameter]
```

ausführen kann. Doch das »execute« kann man auch weglassen. Das im Kapitel 4.9 geschilderte s-Bit macht's möglich. Erstellen Sie doch einmal ein Text-File folgenden Inhalts:

```
echo "Hallo Welt"
echo "Ich bin ein Batch"
```

Diese Datei speichern Sie im s-Directory unter dem Namen »batch«. Dann geben Sie ein

```
protect :s/batch s add
```

Nun können Sie »batch« tippen, und die Datei wird ausgeführt. Erkenntnis: Ist das s-Bit gesetzt, kann man sich die Eingabe von »Execute« sparen.

Wie funktioniert das?

Wenn Sie nur einen Namen eintippen, nimmt AmigaDOS an, daß dies ein Programm ist, das es laden und starten soll. Stellt es fest, daß es sich nicht um ein ausführbares Programm handelt, kommt die Meldung »unable to load <Name>...«. So war es bisher.

Nun ist AmigaDOS etwas schlauer. Bevor es meckert, schaut es nach, ob das s-Bit gesetzt ist. Wenn ja, führt es <Name> als »execute <Name>« aus.

Damit ist auch klar, daß Sie das s-Bit nur bei wirklichen Kommando-Dateien setzen sollten. Anderfalls kommt die Meldung »Unknown command <Name>«.

2.19 Startup-Menü

Nachdem ich eine Weile mit dem Amiga gearbeitet hatte, fiel mir auf, daß ich nach dem Start fast immer die gleichen Kommandos tippte. Basic, Assembler und C benutze ich häufig, und dafür ist jedesmal ein »CD« und ein Aufruf fällig.

Daraus entstand die Kommando-Datei »menu«. Erstellen Sie zuerst ein Textfile für die Anzeige, zum Beispiel dieses:

```
Mein Menü-System

B =  Basic
D =  DevPac
L =  Lattice-C
```

Sie gestalten das natürlich viel besser und speichern es dann unter dem Namen »MenuText« im s-Directory.

Nun schreiben Sie eine Kommando-Datei mit diesem Inhalt:

```
type :s/MenuText
skip >NIL: ?
LAB B
  cd :basic
  AmigaBasic
  skip done
LAB D
  cd :devpac
  genam2
  skip done
LAB L
  cd :lat
  g
LAB done
```

Speichern Sie diesen Text auch im s-Directory. Nun können Sie in die Startup-Sequence oder in das Shell-Startup ein »execute menu« einbauen. Sie können das Kommando aber auch direkt aufrufen, wenn Sie den Tip aus Kapitel 2.18 anwenden.

Wie funktioniert das?

Der Skip-Befehl hat normalerweise die Syntax »skip marke«. Er springt dann zur Marke, die mit »LAB Marke« definiert ist. Das nutzen wir hier auch kräftig aus. Der Skip-Befehl mit der Syntax

```
skip ?
```

oder

```
skip >NIL: ?
```

wartet jedoch auf die Eingabe einer Marke, um dann dorthin zu springen. Die zweite Form unterdrückt lediglich die Anzeige des Hilfetextes. Folglich warten wir hier auf die Eingabe von B, D oder L, um dann zu einer dieser Marken zu springen. Da die Kommando-Datei ja immer weiter abgearbeitet wird, und wir nicht beispielsweise nach der Beendigung von Basic automatisch im DevPac landen wollen, erfolgt nach jeder Gruppe ein »skip done«. Man kann natürlich auch wieder an den Anfang springen, also in einer Schleife fahren, die dann einen Menü-Punkt »done« anbieten sollte.

2.20 Schnellere Texte

Der Amiga hat ab der Workbench 1.3 einen neuen Befehl im c-Verzeichnis, und der heißt schlicht FF. Das steht für »Fast Fonts«. Tatsächlich wird damit die Textausgabe deutlich schneller. Mit

```
FF -0
```

aktivieren Sie dieses Programm, mit »FF -N« können Sie es wieder ausschalten. Am besten tragen Sie »FF -0« gleich in die Startup-Sequence ein, es lohnt sich.

3 Drucker- und Schnittstellen-Tips

3.1 Nehmen Sie die neueste Drucker-Software

Viele Grafik-Programme befinden sich auf einer Workbench- bzw. CLI-Diskette, mit der Sie Ihren Amiga starten können, um dann gleich im Programm zu sein. Doch manchmal gibt es beim Drucken Probleme, meistens treten Verzerrungen auf (Kreise werden zu Ovalen).

Der Grund dafür liegt vielfach darin, daß die Drucker-Software auf solchen Disketten noch nicht auf dem neusten Stand ist. Dabei handelt es sich um das allgemeine »printer-device« und den speziellen Treiber für Ihren Drucker. Kopieren Sie deshalb von Ihrer neuesten Workbench-Diskette diese Dateien auf die »Grafik-Diskette«, zum Beispiel so:

```
copy df0:devs/printer-device to df1:devs
copy df0:devs/printers/NameDesTreibers to df1:devs/printers
```

Die Druckertreiber befinden sich übrigens auf der Extras-Diskette. Um von dort einen Treiber auf eine Workbench-Diskette zu kopieren, setzen Sie am besten das Programm »InstallPrinter« ein, das sich auf der Workbench-Diskette im Verzeichnis »Utilities« befindet. Dieses Programm müssen Sie nur per Doppelklick starten und dann dem Dialog folgen.

3.2 Drucker-Update

Auch die Drucker bestehen mehr und mehr aus Software, die in ROM- oder EPROM-Bausteinen gespeichert ist. Das ändert aber nichts am Grundprinzip, wonach Software immer Bugs (Fehler) hat. Das ändert erst recht nichts am Hauptprinzip, wonach es von jeder Software immer neue, verbesserte und erweiterte Versionen gibt.

Sie sollten sich deshalb ab und zu bei einem autorisierten Fachhändler nach »Updates« für Ihren Drucker erkundigen. Oft genug stellt sich dann heraus, daß Sie mit einem neuen EPROM eine höhere Auflösung, mehr Schriften oder sonstige Extras einbauen können.

3.3 Der Unterschied zwischen PRT:, SER: und PAR:

Der Amiga hat eine ganz tolle Einrichtung, nämlich das logische Gerät »PRT:« für Printer. Das Adjektiv logisch sagt schon, daß dieses Gerät nicht als Hardware existiert, Sie können es aber wie ein Gerät ansprechen, also Daten an diesen »Drucker« senden.

Von diesem Printer besteht allerdings eine Verbindung zu einer physikalischen Schnittstelle, nämlich entweder zum parallelen oder zum seriellen Anschluß auf der Rückseite. Welche von beiden Schnittstellen zum Drucken benutzt werden soll, können Sie in Preferences einstellen.

Zusätzlich liegt zwischen »PRT:« und dem Drucker noch eine Art Filter, Druckertreiber genannt, das jeden Drucker an den Amiga anpaßt. Auch der Druckertreiber läßt sich in Preferences wählen. Folge des Ganzen: Alle Programme, die nur auf den Drucker »PRT:« drucken, sind völlig unabhängig von der Hardware.

Wenn Sie genau das nicht wollen, sondern unter Umgehung des Druckertreibers die Hardware direkt ansprechen möchten, können Sie anstatt »PRT:« für die parallele Schnittstelle »PAR:« einsetzen und für die serielle »SER:«. Dieses Feature braucht man speziell, wenn man Steuerzeichen zum Drucker senden will.

3.4 Das »File already open«-Problem

Um Steuerzeichen an den Drucker zu senden, darf man nicht LPRINT einsetzen, denn das läuft über »PRT:«. Wie im Abschnitt 3.2 geschildert, werden damit die Zeichen gefiltert, und genau das wollen wir nicht. Versuchen Sie jetzt aber »PAR:« zu öffnen, meldet Basic nur »File already open«. Da hilft es auch nicht, vorher »CLOSE« zu geben. Abhilfe schafft dieser Trick:

```
OPEN "LPT1:" FOR OUTPUT AS 1
CLOSE 1
OPEN "PAR:" FOR OUTPUT AS 1
PRINT #1, CHR$(27)CHR$(15);
PRINT #1, "Das ist Kleinschrift auf dem NEC P6"
CLOSE 1
```

Sie sehen, daß mit dem Öffnen und sofortigem wieder Schließen von »LPT1:« das Problem umgangen wird. Anmerkung am Rande: Zwischen den CHR$-Funktionen kann, aber muß kein Semikolon stehen.

3.5 Die einfachste Schreibmaschine

Sie wollen nur eine oder wenige Zeilen ausdrucken oder unter einen schon fertigen Brief im Drucker noch ein P.S. setzen? Dafür sein Textverarbeitungsprogramm anzuwerfen, lohnt sich

nicht, und außerdem hat das vielleicht den Fehler, vor jedem Ausdruck auf eine neue Seite vorzuschieben. Die Lösung ist ganz einfach. Tippen Sie im CLI oder in der Shell

type * to prt:

ein, und ab sofort geht alles, was Sie tippen, auf den Drucker und Gott sei Dank aber erst, wenn Sie die Return-Taste drücken, also zeilenweise. Deshalb arbeiten Sie besser immer mit der Shell, weil Sie dann die Zeilen vor dem Abschicken auch noch korrigieren können. Mit der Tastenkombination <Ctrl>+<\> (Control- und Backslash-Taste gleichzeitig) schalten Sie die Ausgabe wieder auf das aktuelle Shell-Window um.

Wie funktioniert das?

Der Stern (*) ist der Name des Ausgabefensters. Mit »to Ziel« wird die Ausgabe des Type-Befehls auf ein anderes Gerät umgeleitet. Sie können aber auch als Ziel eine Datei angeben, und somit auf die Schnelle kurze Files erstellen.

Sollten Sie auf Ihrer Spar-Disk oder Ihrer RAM-Disk TYPE nicht parat haben, können Sie dafür auch COPY einsetzen.

Und noch andere Stilarten

Sie können übrigens auch während des Tippens/Druckens die Stilart wechseln, wobei im allgemeinen diese Escape-Sequenzen gelten:

<Esc> [1m **fett**
<Esc> [2m *kursiv*
<Esc> [3m <u>unterstrichen</u>
<Esc> [4m normal

Ganz genau stimmen die Zahlen nicht für jeden Drucker, probieren Sie es aus, nehmen Sie auch »[[5m«. Beachten Sie, daß die Escape-Taste als linke rechteckige Klammer auf dem Schirm dargestellt wird, der folgt dann die zweite Klammer. Die Eingabe für »fett« sieht also so aus:

```
[[1m
```

Dabei ist die erste Klammer tatsächlich invers (blau auf weiß), aber das können wir schlecht drucken.

Voraussetzung ist natürlich, daß Ihr Drucker das mitmacht, sein Treiber im Verzeichnis »:devs/printers« vorhanden ist, und der Drucker in Preferences eingestellt wurde.

3.6 Vorsicht CMD (oder Absicht)

In der Utilities-Schublade befindet sich ein Programm namens CMD. Wenn Sie das »nur mal so« doppelklicken, passiert erst einmal gar nichts. Doch wenn Sie das nächste Mal drucken, passiert scheinbar auch nichts. Der Befehl wird ausgeführt, und der Drucker rührt sich nicht.

Der Grund: CMD leitet die Ausgabe vom Paralleldrucker auf eine Datei in der RAM-Disk um, sie hat den Namen »ram:CMD_file«. Danach schaltet CMD die Umleitung zurück, so daß der nächste Ausdruck wieder zum Drucker geht. Es sei denn, Sie rufen vorher wieder CMD auf, das kann ja auch Absicht sein. Klicken Sie CMD nur einmal an und wählen dann »Info« aus dem Workbench-Menü, finden Sie unter Tool-Types den Eintrag »device=parallel«. Laut Handbuch können Sie das in »seriell« ändern (nicht »ser:«), doch das führt zumindest auf meinem A2000B mit Kickstart 34.5 und Workbench 34.20 zum totalen Absturz.

3.7 Sammeldruck

Man hat ja oft eine ganze Latte von Dateien zu drucken, aber dafür dann zigmal »copy xxx to prt:« zu tippen, ist ja auch nicht gerade die optimale Lösung.

Doch Sie können durchaus mehrere Dateien mit nur einem Befehl drucken. Diese Liste muß man in Klammern setzen und die einzelnen Namen durch den senkrechten Strich trennen. Nehmen wir an, die Dateien abc, def und ghi sollen gedruckt werden. Dann lautet der Befehl:

```
copy (abc|def|ghi) to prt:
```

Schneller ist die Methode auch, denn würden Sie statt dessen drei Copy-Befehle schreiben, würde jedesmal das Copy-Programm von der Diskette geladen werden.

Alternativ können Sie auch das Programm »PrintFiles« im Verzeichnis »Utilities« der Workbench-Diskette einsetzen – wenn es gerade greifbar bzw. auf der eingelegten Diskette vorhanden ist. Die Syntax ist ganz einfach:

```
printfiles datei1 datei2 datei-usw
```

Vor oder nach jedem Namen können Sie noch »-f« setzen, womit ein Form Feed (Seitenvorschub) erzeugt wird.

3.8 Einfacher File-Transfer zum PC

Der Datenaustausch mit dem PC läßt sich manchmal nicht umgehen. Ist der Amiga mit einer PC-Karte ausgerüstet, ist die Sache einfach. Der Datenaustausch kann über die Diskette stattfinden.

Ansonsten haben aber beide Computer auch serielle Schnittstellen, die einfach über ein sogenanntes Nullmodem-Kabel zu verbinden sind. Das Kabel ist sehr einfach herzustellen. Sie verbinden die Kontakte

```
2 -> 3
3 -> 2
7 -> 7
```

Die Kreuzung der Leitungen 2 und 3 ist das ganze Nullmodem, Leitung 7 ist die Masseleitung.

Jetzt brauchen wir nur noch etwas Software. Da in aller Regel nur Textfiles zu übertragen sind (compilierte Programme laufen nicht), ist die Sache recht einfach.

Nehmen wir den Normalfall an, nämlich daß der PC vom Amiga etwas lernen will, dann braucht der Amiga dieses Sende-Programm:

```
INPUT "Zu sendender File: ",a$
OPEN a$ FOR INPUT AS 2
OPEN "com1:9600,n,8,1" AS 1
WHILE NOT EOF(2)
 LINE INPUT #2,a$
 PRINT a$
 PRINT #1, a$
WEND
PRINT #1,"ENDE_DER_DATEI"
CLOSE
```

Sie sehen, das Prinzip ist ganz einfach. Die zu übertragende Datei wird zeilenweise gelesen und mit einem einfachen »PRINT #1, a$« zur Schnittstelle geschickt. »LINE INPUT« ist hier erforderlich, weil bei einem einfachen »INPUT« die Zeile nur bis zum nächsten Komma oder den anderen Basic-Trennzeichen gelesen werden würde. Zum Schluß wird der String »ENDE_DER_DATEI« gesendet. Das ist sicherer als ein EOF-Zeichen, das evtl. auch im Text vorkommen könnte.

Das folgende Empfangsprogramm auf der PC-Seite wurde dort absichtlich in Quick-Basic geschrieben, um einige Unterschiede aufzeigen zu können.

```
CLS
OPEN "COM1:9600,N,8,1,cs,ds" FOR RANDOM AS 1
INPUT "Empfangenen File speichern unter: ", a$
OPEN a$ FOR OUTPUT AS 2
DO
  LINE INPUT #1, a$
  IF a$ = "ENDE_DER_DATEI" THEN EXIT DO
  PRINT a$
  PRINT #2, a$
LOOP
CLOSE
```

Die Parameter »cs« und »ds« in der COM-Anweisung sorgen dafür, daß das Programm – wenn es sein muß – ewig auf Zeichen von der seriellen Schnittstelle wartet. In AmigaBasic müssen Sie das weglassen. Auf jeden Fall muß aber das Empfangsprogramm zuerst gestartet werden. Die Schleife »DO ... LOOP« können Sie in AmigaBasic als »WHILE 1 ... WEND« schreiben. Anstatt »THEN EXIT DO« können Sie hier einfach »THEN END« schreiben. In anderen Fällen müßten Sie zu einer Marke außerhalb der Schleife springen. Mehr zu Schleifen finden Sie im Kapitel 7.15.

3.9 Schneller PC-Druck

Ich habe schon einige Bücher über den Amiga geschrieben, wobei sich ein Problem immer besonders elegant löst. Der Text wird auf dem PC geschrieben, die Programme entstehen und laufen natürlich auf dem Amiga. Doch zum Wechseln muß man nicht von einem Computer zum anderen rasen, sondern nur schlicht die Maus klicken, die PC-Karte macht's möglich. Doch einen Haken hat die Sache. Das Buch muß mindestens zweimal ausgedruckt werden (vor und nach der Korrektur) und das dauert und dauert und dauert... Der Grund ist, daß die PC-Parallelschnittstelle (LPT1:) nur per Software emuliert wird. Druckt man nur reine Texte, fällt das nicht besonders auf, doch wenn man DTP-Seiten zu drucken hat, werden aus einigen Kbytes plötzlich Megabytes. Abhilfe schafft folgender Trick:

Man überträgt die Datei auf die Amiga-Seite (das geht schnell) und druckt dann dort über »PAR:«. Diese Parallelschnittstelle ist echte Hardware und allemal schneller als der schnellste Drucker. Hier nun das Kochrezept:

Zuerst müssen Sie auf der Amiga-Seite das Programm »PCDisk« (im Verzeichnis PC) aktivieren, sonst arbeitet »awrite« nicht.

Nun nehmen wir an, der Text hieße »kap2.txt« und ist unter diesem Namen schon auf der PC-Seite im Directory »c:\word« gespeichert. Dann kopieren sie ihn mit

```
awrite c:\word\kap2.txt dh0:kap2.txt /b
```

auf die Amiga-Seite. Wichtig ist der Parameter »/b« für »binär«, was garantiert, daß alle Zeichen 1 zu 1 kopiert werden. Geben Sie auch immer die vollständigen Pfadnamen an. Nun gehen Sie auf die Amiga-Seite und drucken den Text mit

```
copy dh0:kap2.txt to par:
```

Das Druck-Ergebnis ist »beein-druckend«. Der Vorgang läuft gut dreimal schneller als über das »weiche« LPT1.

3.10 Immer auf den Seitenanfang

In aller Regel ist zumindest das letzte Blatt eines Ausdrucks nicht voll, und damit hätten wir ein Problem. Endlospapier läßt sich nicht am Falz abreißen, Drucker mit Trennautomatik stehen

falsch. Die Lösung ist ganz einfach. Senden Sie zum Schluß ein »Form Feed« (Formular-Vorschub) an den Drucker. In Basic heißt das schlicht

```
LPRINT CHR$(12)
```

Das in anderen Sprachen nachzuvollziehen, dürfte zu den leichtesten Übungen zählen.

3.11 Guru im »Music Construction Set«

Geheimnisvolle Abstürze beim Umgang mit der MIDI-Schnittstelle könnten an einer falschen Einstellung in Preferences liegen. Probieren Sie diese Werte:

```
Baud Rate    : 31250
Buffer Size : 2048
Read Bits    : 8
Write Bits   : 8
Stop Bits    : 1
Parity       : None
Handshaking : None
```

3.12 Schnelle und gute Hardcopies

Was nützt die schönste Grafik, wenn man sie nicht auch zu Papier bringen kann. Dabei soll sie natürlich möglichst wenig verlieren, und schnell soll es auch gehen. Daß der Drucker dabei eine entscheidende Rolle spielt, versteht sich von selbst, doch durch den geschickten Einsatz der Software können Sie selbst sehr viel zum guten Gelingen beitragen.

3.12.1 Nehmen Sie die Workbench 1.3

Sowohl das für die Hardcopy zuständige Programm »GraphicDump« als auch die Druckertreiber sind deutlich verbessert worden. Das bringt nicht nur neue Features, sondern je nach Drucker wurden die Ausdrucke auch 5- bis 30mal schneller.

3.12.2 Wählen Sie den richtigen Antrieb

Einmal abgesehen von Laserdruckern stellt man bei Hardcopies häufig feine waagerechte Streifen im Ausdruck fest. Hier hilft es oft, vom Traktor-Antrieb auf den Friktions-Antrieb zu wechseln, also auf Einzelblatt-Einzug zu gehen.

Legen Sie dann aber das Blatt sehr sorgfältig ein, bzw. richten Sie das Endlospapier präzise aus. Wenn das Papier schief läuft, wird das Druckbild häßlich.

3.12.3 Wählen Sie die richtige Dichte

Die Dichte wird in »Preferences« unter »Graphic 2« eingestellt. Dort heißt sie »Density«.

Eigentlich gilt, daß das Druckbild um so schärfer wird, je höher die Dichte ist, doch Vorsicht. Viele Drucker erreichen bestimmte Dichten nur mit zwei oder drei Durchläufen, und einen Reproduzierbarkeitsfehler haben sie alle. Sie sollten deshalb diese Druckdichte höchstens bei reinen Schwarzweiß-Drucken wählen. Schon bei Grau-Druck führt dieser Effekt zu verwaschenen Bildern, und in Farbe wird es ganz schlimm, wenn der Drucker mit einem Farbband arbeitet. Hier passiert es nämlich, daß im ersten Durchlauf das Band verschmutzt, was dann besonders bei Gelb auffällt.

Die Tabelle von Abb. 3.1 soll Ihnen helfen, die richtige Dichte zu wählen. Beachten Sie, daß dies die Zahlen sind, die Sie in Preferences einstellen, um die maximale Auflösung des jeweiligen Druckers zu erreichen. Die Zahlen sagen nichts über die Qualität des Druckers. So bringt der HP-LaserJet in Dichte 4 300x300 dpi, und der CBM-MPS1000 in Stufe 6 nur 240x216 dpi.

Bleibt nur noch anzumerken, daß mit der Dichte auch die Druckzeit steigt.

Drucker	Maximale Dichte	2. Lauf ab Dichte	3. Lauf ab Dichte
Apple ImageWriter	7	5	-
Calcomp ColorMaster	1	-	-
Canon PJ 1080A	1	-	-
CBM MPS 1000	6	3	4
CBM MPS 1250	6	2	4
Diabolo C150	1	-	-
Epson Q (24 Nadeln)	4	-	-
Epson X (9 Nadeln)	6	2	4
Epson X alt (8/9 Nadeln)	6	-	-
HP DeskJet	4	-	-
HP LaserJet	4	-	-
HP PaintJet	1	-	-
HP ThinkJet	2	-	-
NEC P6/6/7/9/2000	7	4	-
Okidata 92	1	1	-
Okidata 2931	4	3	-
Okimate 20	1	-	-
Seiko 5300/a	3	-	-

Abb. 3.1: Druckerdichten in Theorie und Praxis

3.12.4 Wählen Sie das richtiges Threshold

Wenn Sie ein Farbbild schwarzweiß drucken wollen (müssen), dann sollten Sie in Preferences das »Threshold« richtig einstellen. Gemeint ist damit die Schwelle, die zwischen schwarz und weiß trennt. Die Regel:

Je höher der Wert ist, desto heller müssen die Farben sein, die als weiß gedruckt werden sollen. Oder: Je niedriger der Wert ist, desto dunkler müssen die Farben sein, damit sie als schwarz gedruckt werden.

3.12.5 Wählen Sie das richtige Dithering

Dither nennt man die Nachbildung von Graustufen bzw. Farben durch verschiedene Muster. In Preferences bedeuten:

Ordered

Das Punktmuster, das eine Graustufe bildet, ist geordnet, zum Beispiel immer in der Form 3 Punkte, 2 Lücken, 3 Punkte. Es hat Vorteile bei niedrig auflösenden Druckern.

Halftone

Beim Halbton wird mit variablen Dichten gearbeitet. Typisch ist das für Bilder in Zeitungen. Dafür brauchen Sie einen Drucker von wenigstens 150 dpi.

F-S

Eine Dither-Methode, die von Floyd-Steinberg erfunden wurde. Macht sich gut auf 300-dpi-Druckern (Laser).

Fraction oder Integer

Im Integer-Modus wird Punkt für Punkt 1:1 vom Schirm auf den Drucker umgesetzt. Das macht sich gut bei Bildern mit vielen dünnen horizontalen und waagerechten Linien (Gittern). Dabei wird aber der Aspekt (Seiten/Höhenverhältnis) außer acht gelassen, womit dann Kreise zu Ovalen werden. Bei solchen kurvenreichen Bildern sollten Sie »Fraction« vorziehen.

3.12.6 So nutzen Sie GraphicDump

Scheinbar gibt es ein Problem. Das für die Hardcopy zuständige Programm »GraphicDump« befindet sich im Ordner »Utilities«. Sie wollen aber beispielsweise ein Basic-Programm starten, das eine Grafik malen lassen und dann eine Hardcopy ziehen kann. Sie haben aber nach dem Start von »GraphicDump« nur wenige Sekunden, das Bild auf den Schirm zu bringen.

Die Lösung ist ganz einfach. Zuerst sollten Sie »GraphicDump« direkt auf die Worbench legen, damit Sie nicht erst lange danach suchen müssen bzw. es einfach erreichen können. Nun starten Sie das Programm, das die Grafik erzeugt, bringen Sie auf den Schirm und klicken das Fenster in den Hintergrund (auf das schwarze Rechteck rechts oben). Damit sind Sie auf der Workbench, wo Sie evtl. noch offene Fenster schließen und dann »GraphicDump« anklicken. Jetzt läuft die Uhr, aber Sie brauchen nur noch einen Mausklick: Entweder klicken Sie jetzt das Workbench-Window weg (auf das schwarze Rechteck rechts oben) oder Sie klicken Ihr Grafik-Window nach vorne (auf das weiße Rechteck rechts oben).

3.12.7 So werden Sie schneller

Es gibt verschiedene Möglichkeiten, den Grafikausdruck zu beschleunigen, mehrere davon auf einmal wirken natürlich besonders gut.

»Black and White« ist am schnellsten

Am schnellsten ist der Druck von nur einer Bit-Plane, und das erreichen Sie, indem Sie in Preferences auf »Black and White« schalten.

Horizontal ist schneller

Horizontale Hardcopies sind sehr viel schneller als vertikale, weil Sie den Blitter einsetzen.

»Smoothing« kostet Zeit

Das »Smoothing« (Versuch des Vermeidens von Treppen in diagonalen Linien) verdoppelt die Zeit.

»F-S« ist langsam

Das »F-S dithering« verdoppelt die Zeit. Vierfach langsamer können Sie übrigens nicht werden, weil »F-S« das »Smoothing« immer ausschaltet.

Hohe Dichte dauert länger

Je niedriger die Dichte (Density) eingestellt ist, desto schneller wird gedruckt.

3.13 Was macht InitPrinter?

Nach dem Einschalten des Amiga geht auch Ihr Drucker in einen bestimmten Anfangszustand, nämlich in den, den Sie in Preferences voreingestellt haben. Nun kann aber ein Programm den Drucker in einem ganz anderen Zustand belassen haben; die Schriftart, die Seitenlänge, einfach nichts stimmt mehr.

In diesem Fall starten Sie »InitPrinter« (in der System-Schublade), und der Amiga sendet die Escape-Sequenzen an den Drucker, die ihn wieder auf die Werte von Preferences bringen. Wenn es Sie allerdings stört, daß der Drucker dann eine Seite vorschiebt – einige tun das – dann können Sie vielleicht auch den Drucker aus- und wieder einschalten.

3.14 Universelle Escape-Sequenzen

Ihr Drucker-Handbuch führt zwar über viele Seiten seine Escape-Sequenzen auf, aber am besten sollten Sie die gleich vergessen, wenn Sie ein vom Drucker unabhängiges Programm einsetzen oder selbst schreiben wollen. Diese Escape-Sequenzen sind nämlich bei jedem Drucker anders und gelten für Ihren Ducker auch nur, wenn Sie direkt über »PAR:« ausgeben. Das tun aber viele Programme nicht. Arbeiten Sie hingegen mit »PRT:« gelten die Einstellungen in Preferences, wobei die wichtigste die Angabe Ihres Druckers ist. Damit wird nämlich ein Treiber verwendet, der die Standard-Escape-Sequenzen des Amiga in die speziellen eines Druckers übersetzt. Wenn Sie selbst programmieren, müssen Sie diese Standard-Escape-Sequenzen natürlich kennen. Abb. 3.2 bringt die komplette Tabelle.

Esc-Sequenz	Wirkung auf Drucker
<ESC> c	Reset
<ESC> #1	Initialisierung
<ESC> D	Zeilenvorschub
<ESC> E	Wagenrücklauf + Zeilenvorschub
<ESC> M	Zeilenrücklauf
<ESC> [0m	Standard-Zeichensatz
<ESC> [3m	kursiv ein
<ESC> [23m	kursiv aus
<ESC> [4m	unterstrichen ein
<ESC> [24m	unterstrichen aus
<ESC> [1	Fettschrift ein
<ESC> [2m	Fettschrift aus
<ESC> [nm	Vordergrundfarbe (nn=31-33>
<ESC> [nm	Hintergrundfarbe (nn=41-43>
<ESC> [0w	normale Zeichenbreite
<ESC> [2w	Elite ein (12 cpi)
<ESC> [1w	Elite aus
<ESC> [4w	condensed ein (Kompaktschrift)
<ESC> (3w	condensed aus
<ESC> [5w	enlarged ein (Breitschrift)
<ESC> [Sw	enlarged aus
<ESC> [6"z	Schattenschrift ein
<ESC> [5"z	Schattenschrift aus
<ESC> [4"z	doublestrike ein (Doppelanschlag>
<ESC> [3"z	doublestrike aus
<ESC> [2"z	NLQ ein
<ESC> [1"z	NLQ aus

```
<ESC> [2v       Hochstellen ein
<ESC> [1v       Hochstellen aus
<ESC> [4v       Tiefstellen ein
<ESC> [3v       Tiefstellen aus
<ESC> [0v       Zeile normal
<ESC> L         Zeile teilweise hochstellen
<ESC> K         Zeile teilweise tiefstellen
<ESC> (B        amerikanischer Zeichensatz
<ESC> (R        französischer Zeichensatz
<ESC> (K        deutscher Zeichensatz
<ESC> (A        englischer Zeichensatz
<ESC> (E        dänischer Zeichensatz 1
<ESC> (H        schwedischer Zeichensatz
<ESC> (Y        italienischer Zeichensatz
<ESC> (Z        spanischer Zeichensatz
(ESC> (J        japanischer Zeichensatz
<ESC> (6        norwegischer Zeichensatz
<ESC> (C        dänischer Zeichensatz 2
<ESC> [2p       Proportionalschrift ein
<ESC> [1p       Proportionalschrift aus
<ESC> [0p       Proportionalschrift löschen
<ESC> [n E      proportionales Offset n setzen
<ESC> (5 F      linksbündig justieren
<ESC> [7 F      rechtsbündig justieren
<ESC> [6 F      Blocksatz
<ESC> [1 F      zentriert justieren
<ESC> [0 F      keine Justierung
<ESC> [3 F      nach Zeichenbreite justieren
<ESC> [0z       8 Zeilen / Zoll
<ESC> [1z       6 Zeilen / Zoll
<ESC> [nt       n Zeilen / Seite
<ESC> [nq       n Zeilen bei Perforation überspringen
<ESC> [0q       kein Perforationssprung
<ESC> #9        Kopf an linken Rand
<ESC> #0        Kopf an rechten Rand
<ESC> #8        Kopf an oberen Rand
<ESC> #2        Kopf an unteren Rand
<ESC> [n1;n2r   oberen und unteren Rand einstellen
<ESC> [n1;n2s   linken und rechten Rand einstellen
<ESC> #3        Randeinstellung löschen
<ESC> H         horizontaler Tabulator
<ESC> J         vertikaler Tabulator
<ESC> [0g       horizontalen Tabulator löschen
<ESC> [3g       alle horizontalen Tabulatoren löschen
<ESC> [1g       vertikalen Tabulator löschen
<ESC> [4g       alle vertikalen Tabulatoren löschen
<ESC> #4        alle Tabulatoren löschen
<ESC> #5        Standard-Tabulatoren setzen
<ESC> [Pn"x     Erweiterter Befehl n
```

Abb. 3.2: Die universellen Escape-Sequenzen

3.15 Drucker installieren mit InstallPrinter

Das Programm »InstallPrinter« im Utilities-Verzeichnis ist recht praktisch, wenn es um die Installation eines neuen Druckertreibers geht.

Sie starten es einfach per Doppelklick und werden dann aufgefordert, die Extras-Diskette in ein beliebiges Laufwerk einzulegen. Dann erscheint eine Liste aller Treiber bzw. Druckernamen, sie tippen einen davon ein, und das war's schon.

Das klappt aber leider nur mit dieser Extras-Diskette. Haben Sie einen Treiber aus anderer Quelle, müssen Sie ihn selbst auf Ihre Workbench-Diskette kopieren, und zwar in das Verzeichnis »devs/printers«.

3.16 Billig und gut: Apple-ImageWriter am Amiga

Der Apple-Macintosh ist bekanntlich der Mercedes unter den Personalcomputern (Preis inklusive), und das gilt auch für sein Zubehör, wenigstens wenn es von Apple kommt. Deshalb ist der Nadeldrucker von Apple mit dem Namen »ImageWriter I« eine ganz tolle Maschine. Fast alle Macintosh der ersten Jahre waren mit diesem Drucker ausgerüstet, doch dann brachte Apple erst Laser-Drucker und später noch den ImageWriter II heraus.

Was ein wahrer Macintosh-Freak ist, hat a) Geld und kauft b) immer das Neuste, so daß plötzlich viele ImageWriter übrig wurden. Kaum ein Apple-Händler nimmt oder nahm die Drucker noch in Zahlung, womit der Preis in den Keller ging.

Kennen Sie keinen Macianer, der noch einen ImageWriter herumzustehen hat, fragen Sie bei Apple-Händlern nach oder schauen Sie einmal in die Kleinanzeigen von Macintosh-Magazinen. Andere Computer-Anwender kaufen den Drucker selten, weil sie dem Gerücht erlegen sind, daß er eine Spezialanfertigung nur für den Macintosh ist.

Doch dem ist nicht so. Der ImageWriter funktioniert sehr schön am Amiga, Commodore liefert sogar serienmäßig einen Treiber dafür, IBM aber nicht. Diesen Treiber müssen Sie zuerst auf Ihre Workbench-Diskette bringen, das Programm InstallPrinter (siehe Kapitel 3.15) hilft Ihnen dabei. Der Treiber heißt zwar »ImageWriterII«, aber das macht nichts, auch der ImageWriter I spielt sehr schön damit, und wenn Sie einen »II« abstauben können, noch besser... Dann müssen Sie nur noch den ImageWriter in Preferences auswählen, den Port auf »Serial« und die serielle Schnittstelle auf

```
9600 Baud
8 Write-Bits
1 Stop-Bit
No Parity
Protokoll XON/XOFF
```

stellen.

Genau der letzte Punkt ist der Trick. Mit diesem Software-Handshake funktioniert der Drucker sehr schön, mit dem voreingestellten DTR hat er bzw. der Amiga Probleme, die sich hauptsächlich derart auswirken, daß Hardcopies vorzeitig abgebrochen werden.

Dafür brauchen Sie dann auch nur ein einfaches Kabel, und das muß – Trick 2 – ein Nullmodem-Kabel sein. Offensichtlich glaubt nämlich der ImageWriter nicht, daß er ein Endgerät ist, sondern hält sich für einen Computer. Also verbinden Sie für Ihr serielles Kabel lediglich die Pins

```
2 -> 3
3 -> 2
7 -> 7
```

Nun müssen Sie den ImageWriter nur noch auf XON/XOFF einstellen – serienmäßig macht er nämlich »DTR«. Da die Möglichkeit besteht, daß schon jemand auf dem Mäuseklavier wild gespielt hat, schildere ich aber besser alle DIP-Schalter. Diese sind sehr einfach zugänglich. Wenn Sie den vorderen Deckel abnehmen, finden Sie die Schalter unter einer wegklappbaren Folie auf der rechten Seite vor der Kopfführung. Den Druckkopf schieben sie ggf. etwas nach links.

Die Schalterblöcke heißen SW 1 und SW 2 und sind auch so auf der Folie beschriftet (vorne liegt SW 2), dito sind die einzelnen Schalternummern lesbar. Die Schalterstellungen heißen Open und Closed, auf den Schalterblöcken steht deutlich für die eine Richtung OPEN. In der folgenden Tabelle bedeutet beispielsweise »SW 1–3« soviel wie »Block 1, Schalter 3«.

Beachten Sie, daß alle Einstellungen von SW 1 mit Escape-Sequenzen überschrieben werden können, SW 2 muß aber stimmen.

SW 1-1	SW 1-2	SW 1-3	Zeichensatz
Open	Open	Open	USA 1
Open	Open	Closed	Deutsch
Open	Closed	Open	USA 2
Open	Closed	Closed	Französisch
Closed	Open	Open	Italienisch
Closed	Open	Closed	Schwedisch
Closed	Closed	Open	Britisch
Closed	Closed	Closed	Spanisch

SW 1-4	Seitenlänge
Open	66 Zeilen
Closed	72 Zeilen

SW 1-5	Daten-Bit 8
Open	Erkannt
Closed	Ignoriert

SW 1-6	SW 1-7	Zeichen-Abstand
Open	Open	Pica
Closed	Opem	Elite
Open	Closed	Komprimiert
Closed	Closed	Elite proportional

SW 1-8	Zeilenvorschub nach Return
Open	Nein
Closed	Ja

SW 2-1	SW 2-2	Baudrate
Open	Open	300
Closed	Open	1200
Open	Closed	2400
Closed	Closed	9600

Protokoll

SW 2-3	
Open	DTR
Closed	XON/XOFF

4 Disketten und Festplatte

4.1 Programm will partout DF1: oder gar DF2:

Es gibt Programme, die einfach unterstellen, daß Sie zwei Diskettenlaufwerke haben. Startet man solche Programme, verlangen sie plötzlich, daß man ihre zweite Diskette in das Laufwerk »DF1:« packt.

Doch Sie müssen sich deshalb nicht gleich ein zweites Laufwerk kaufen. Kopieren Sie statt dessen diese Diskette – falls Sie sowieso nicht schon mit einer Arbeitskopie arbeiten – und nennen Sie sie um in »DF1« (ohne Doppelpunkt!).

```
relabel df0: df1
```

Testen Sie das.

```
cd df1:
```

müßte jetzt funktionieren, auch wenn Sie gar kein zweites Laufwerk haben. Nun können Sie immer, wenn das Programm seine zweite Diskette anfordert, einfach die Scheibe in das Laufwerk »DF0:« legen.

- Wie funktioniert das?

Der Amiga kann wahlweise mit dem Disketten-Namen oder der Laufwerkbezeichnung arbeiten. »Freiwillig« nimmt er den Namen, muß er auch, weil er nur auf diese Art mehr Disketten verwalten kann, als Laufwerke vorhanden sind. Nach dem Namen setzt er aber immer den Doppelpunkt, und somit wird aus dem Namen »DF1« dann »DF1:«. Genau damit wird das nach »DF1:« verlangende Programm überlistet.

Für den wohl seltenen Fall, daß ein Programm »DF2:« anfordert, oder nur weil es Ihnen Spaß macht, können Sie ein drittes, viertes oder beliebig viele Laufwerke erzeugen, vorausgesetzt Sie haben schon zwei, nämlich »DF0:« und »DF1:«. Letzteres, das externe Laufwerk läßt sich nämlich wie eine Festplatte in Partitionen einteilen. Eine Partition ist ein Stück einer Disk. Zum Beispiel kann man die Zylinder 0–20 zur Partition 1 und die Zylinder 21–79 zur Partition 2 ernennen.

Die Partitionen bekommen Namen wie die anderen Laufwerke auch, in unserem Beispiel heißen sie »DF2:« und »DF3:«. Einen kleinen Schönheitsfehler hat die Aktion: Da wir »DF1:« partitionieren, ist dieses Laufwerk verschwunden. Und das müssen Sie dafür tun:

Legen Sie eine Kopie Ihrer Workbench-Diskette an, und arbeiten Sie damit weiter. Editieren Sie nun die »MountList« im Verzeichnis »Devs«, so daß folgende Einträge hinzukommen:

```
DF2:  Device = trackdisk.device
      Unit = 1
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      PreAlloc = 11
      Interleave = 0
      LowCyl = 0; HighCyl = 20
      Buffers = 20
      BufMemType = 3
#
DF3:  Device = trackdisk.device
      Unit = 1
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      PreAlloc = 11
      Interleave = 0
      LowCyl = 21; HighCyl = 79
      Buffers = 20
      BufMemType = 3

#
```

Wichtig sind hier »LowCyl« und »HighCyl«, womit Sie den Start- und den Endzylinder für jede Partition angeben. Jede Zahl im Bereich von 0 bis 79 ist erlaubt. Logisch, daß sich Partitionen nicht überlappen dürfen und mit Lücken Platz verschenkt wird.

Nachdem Sie die geänderte »MoutList« gespeichert haben, müssen Sie Ihren Amiga neu booten und jetzt im CLI oder gleich in der Startup-Sequence die neuen Laufwerke noch anmelden. Das geschieht hier mit

```
mount df2:
mount df3:
```

Jetzt müssen Sie nur noch darauf achten, daß Sie jede Diskette auch in zwei Teilen formatieren müssen, zum Beispiel so:

```
format drive df2: name Teil1
format drive df3: name Teil2
```

4.2 Schnellkopie

Sie können durchaus mehrere Dateien mit nur einem Befehl kopieren. Diese Liste muß man in Klammern setzen und die einzelnen Namen durch den senkrechten Strich trennen. Nehmen wir an, die Dateien abc, def und ghi sollen kopiert werden. Dann lautet der Befehl:

```
copy (abc|def|gh) to MyDir
```

Schneller ist die Methode auch, speziell wenn das Ziel die RAM-Disk ist, die ja häufig mit vielen Files zu laden ist. Sie kopieren zum Beispiel mit

```
copy (abc|def|gh) to ram:
```

alle drei Dateien in die RAM-Disk. Würden Sie statt dessen drei Copy-Befehle schreiben, würde jedesmal das Copy-Programm von der Diskette geladen werden. Dazu liest man zwar manchmal auch den Tip, erst das Copy-Programm in die RAM-Disk zu bringen und dann damit weiterzuarbeiten, aber: Nun läuft zwar die Diskettte nicht mehr jedesmal an, doch wozu soll ich das Copy-Programm in die RAM-Disk kopieren, wenn ich es da eigentlich gar nicht brauche?

4.3 Einfacher kopieren

Es ist sicherlich kein Vergnügen, des öfteren solche Kommandos wie

```
copy df1:devs/keymaps/d to df0:devs/keymaps/d
```

einzutippen. Doch die Hälfte davon können Sie sparen, wenn Sie schon im Ziel-Directory sind. Nehmen wir an, Sie hatten vorher

```
cd df0:devs/keymaps
```

getippt. Dann reicht es, wenn Sie den Copy-Befehl reduzieren auf:

```
copy df1:devs/keymaps/d ""
```

Das »Nichts« der beiden Anführungszeichen hat zur Folge, daß AmigaDOS das aktuelle Verzeichnis als Ziel einsetzt.

4.4 Ein Directory zuviel?

Eine Ergänzung zum vorherigen Trick: Wenn Sie ihn nicht anwenden und doch ein Ziel-Directory angeben, das aber nicht existiert, wird es vom AmigaDOS jetzt (ab Workbench 1.3) automatisch angelegt. Das klingt sehr edel, hat aber leider auch zur Folge, daß bei einem Tippfehler die Daten

dahin gehen, wo Sie sie gar nicht erwarten. Oft denkt man dann, man hätte etwas falsch gemacht und wiederholt den Befehl, diesmal natürlich richtig. Ergebnis: Es gibt noch ein Directory mitsamt Inhalt, das nur unnötig Platz auf der Disk blockiert. Um solchen Übeltätern auf die Spur zu kommen, sollten Sie auf der obersten Ebene (cd :) ab und zu mal

```
dir all dirs
```

eintippen. Das zeigt alle Directories an, während »dir all« oder »dir opt a« auch alle Files auflistet.

4.5 Der »DiskDoctor« hilft oft

Wenn Sie so eine hübsch häßliche Meldung der Art

```
Error validating disk - disk structure corrupt
```

sehen, dann, ja dann hat die Diskette leider einen Schaden. Doch keine Panik auf der Amiganic, Hilfe ist in Sicht. Das Programm »DiskDoctor« (üblicherweise im c-Verzeichnis der Workbench-Diskette) ist dafür gedacht, solche »korrupten« Disketten zu reparieren, indem es die Zuordnung von Datei-Namen und ihren Datenblöcken wieder herstellt. Sie müssen in das CLI oder in die Shell gehen und tippen

```
diskdoctor df0:
```

oder die Laufwerk-Kennung ein, mit der Sie arbeiten wollen. Sie werden dann aufgefordert, die zu reparierende Diskette einzulegen.

Der Vorgang kann übrigens eine ganze Weile dauern. Wenn Sie nicht solange Däumchen drehen wollen und zwei Laufwerke haben, machen Sie ein zweites CLI-Fenster auf, starten den »DiskDoctor« darin und machen dann selbst im ersten CLI-Window etwas anderes. Der Umstand hat nur den Grund, daß »run DiskDoctor« nicht funktioniert.

Wenn Sie so oder so mit der Reparatur der Diskette fertig sind, mißtrauen Sie dieser Scheibe. Kopieren Sie alle Dateien auf eine andere Diskette, formatieren dann die ehemals defekte und markieren Sie sie mit einem Zeichen. Sollte diese Diskette später nochmals den »DiskDoctor« benötigen, retten Sie nur noch die Daten und verschrotten dann besser diese Diskette.

4.5.1 »DiskDoctor« kann auch »undelete«

Wenn Sie eine Datei versehentlich gelöscht haben, können Sie sie auch mit dem »DiskDoctor« zurückholen. Tatsächlich wird nämlich durch das Löschen nur der Eintrag im Inhaltsverzeichnis

der Diskette als gelöscht markiert. Die Daten selbst bleiben unverändert, aber das nur so lange, wie eine neue Datei diesen Platz nicht benötigt. Deshalb sollten Sie nach einem versehentlichen Löschen keine neuen Dateien mehr anlegen, sondern sofort den »DiskDoctor« rufen.

4.5.2 Manchmal hilft noch TYPE

Wenn eine einzelne Datei beschädigt ist, versagt der COPY-Befehl, weil er bei einem Fehler die Zieldatei nicht anlegt. Nehmen Sie statt dessen TYPE – natürlich nur für Textdateien – zum Beispiel als

```
type kaputteDatei to neueDatei
```

So wird wenigstens so lange Zeile für Zeile in die neue Datei getippt, wie noch etwas lesbar ist. Wenn Sie Glück haben, ist die Defektstelle erst kurz vor dem Ende, so daß Sie den größten Teil noch retten können.

4.6 Disk aufräumen

Je länger Sie eine Diskette aktiv benutzen, sprich, Dateien auf sie schreiben und andere löschen, desto langsamer wird die Diskette. Der Grund ist folgender:

Wenn Sie auf eine leere Diskette Dateien kopieren, werden diese der Reihe nach abgelegt. Löschen Sie einige dieser Dateien, entstehen vorerst Lücken. Kommen nun neue Dateien hinzu, die im Stück in keine Lücke passen, werden sie auf mehrere dieser »Gaps« verteilt. Die Dateien sind damit gestreut oder englisch »scattered«. Da der Schreib-/Lese-Kopf jetzt ständig hin- und herfahren muß, um die Brocken (oder Krümel) einer Datei zusammenzusuchen, wird das System natürlich langsamer.

Die Abhilfe ist ganz simpel. Man muß nur alle Dateien der Diskette auf eine neu formatierte kopieren. Soll dies eine Boot-Diskette werden, geben Sie nach dem Formatieren noch

```
install drive df0:
```

ein. Übrigens: Wenn beim Formatieren ein Fehler auftritt, sollten Sie die Diskette besser gleich wegwerfen. Daß beim nächsten Versuch (Retry) kein Fehler auftritt, ist nur Glückssache. Ob das Glück lange anhält, ist mehr als fraglich.

Mit zwei Laufwerken ist dann das Kopieren sehr einfach:

```
copy df0: to df1: all
```

Haben Sie eine Festplatte, könnte die Befehlsfolge so aussehen:

```
makedir dh0:crunch
copy df0: to dh0:crunch all
;Hier neue Diskette einlegen
cd df0:
copy dh0:crunch to "" all
delete dh0:crunch all
```

Haben Sie keine Festplatte, aber genügend RAM (eine Diskette belegt 880 Kbyte), können Sie im obigen Beispiel »ram:« anstatt »dh0:« schreiben.

Die alte Diskette können Sie übrigens so lassen, wie sie ist. Sie wird nämlich ihr weiteres Leben als Sicherheitskopie im Schreibtisch verbringen, und dafür ist sie auch unaufgeräumt allemal schnell genug.

4.7 Dateien finden

Auf einer richtig vollen Diskette oder gar einer Festplatte eine Datei zu finden, ist gar nicht so einfach. Doch mit dem erweiterten Search-Befehl der Workbench 1.3 ist das kein Problem mehr. Geben Sie ein:

```
search from df0: NameDerDatei all file
```

Das ist Ihnen zuviel der Tipperei? OK, dann fügen Sie doch in die Datei »shell-Startup« im s-Directory diese Zeile ein (unterstellt, Sie arbeiten nur mit der Shell):

```
alias finde search from [] [] all file
```

Jetzt müssen Sie nur noch tippen:

```
finde df0: NamederDatei
```

Haben Sie nur ein Laufwerk, können Sie für das erste Klammerpaar »df0:« einsetzen, womit sich der ganze Suchbefehl auf »finde NameDerDatei« reduziert.

Leider hat die Lösung noch einen Haken. Wird der File-Name gefunden, so wird er angezeigt, gibt es ihn mehrfach, wird er entsprechend oft gelistet. Sie wissen somit, daß es die Datei gibt, doch Sie wissen nicht, in welchem Verzeichnis sie steht. Deshalb bietet sich diese Lösung an:

```
dir > ram:xxx opt a
ed ram:xxx
<Esc>uc
<Esc>f "NameDerDatei"
```

Wollen Sie auf einem bestimmten Laufwerk suchen lassen, lautet die erste Zeile

```
dir > ram:xxx df0: all
```

»all« ist die modernere Schreibweise von »opt a«. Wie auch immer, in beiden Fällen wird die gesamte Directory-Struktur mit allen Files in die Datei »xxx« auf der RAM-Disk geschrieben. Dann wird der Editor »ED« aufgerufen und mit dessen Suchbefehl (<Esc>f) der Begriff gefunden oder auch nicht. Damit die Chancen besser stehen, wurde vorher mit »<Esc>uc« auf »upper case« geschaltet, womit es egal ist, ob der Filename in Klein- oder Großbuchstaben existiert. Im Erfolgsfall können Sie dann direkt ablesen, wo die Datei liegt. Ggf. müssen Sie den Text noch etwas mit den Cursor-Tasten schieben. Verlassen Sie ED mit »<Esc>x«.

4.8 Disketten-Wechsel vermeiden

Der Amiga erwartet leider, daß sich die CLI-Befehle im c-Verzeichnis der Start-Diskette befinden. Dummerweise merkt er sich auch, welche das ist. Folge: Immer wenn Sie einen CLI-Befehl eintippen, verlangt der Amiga die Start-Diskette. Da stört ihn auch herzlich wenig, daß es auf der eingelegten Diskette schon ein c-Verzeichnis mit dem aufgerufenen CLI-Befehl gibt.

Die simpelste Abhilfe ist die Angabe eines Pfades. Schreiben Sie also anstatt »dir«

```
df0:c/dir
```

So läuft das schon. Wenn Sie nun keine Lust haben, immer diese sechs Zeichen extra zu tippen, haben Sie zwei Möglichkeiten. Die erste ist, die Zuweisung umzulenken, und zwar so:

```
df0:c/assign c: df0:c
```

Beachten Sie, daß der Assign-Befehl selbst noch mit dem Pfadnamen aufgerufen werden muß. Leider wirkt diese Maßnahme nur bis zum nächsten Diskettenwechsel. Sie könnten also danach in der Shell auf den Befehl zurückgehen und ihn wieder ausführen.

Sie können aber auch gleich – ohne Assign – auf den letzten CLI-Befehl »mit Pfad« gehen, zum Beispiel auf das »df0:c/dir«, und dort das »dir« ändern (wenn Sie es nicht wieder brauchen). Siehe hierzu auch die Kapitel 2.2 und 2.10.

4.9 Schützen Sie wichtige Dateien

Bevor Sie den nächsten Tip anwenden, der zeigt, wie man eine Datei mit einem einzigen »d» einfach »killt«, aber auch überhaupt, sollten Sie Ihre wichtigen Dateien schützen. Dafür gibt es den CLI-Befehl »protect«, der so anzuwenden ist:

```
protect Datei1 r
protect Datei2 rw
protect Datei3 rwd
protect Prog1 E
```

»Datei1« kann nur gelesen werden, »Datei2« darf auch beschrieben und »Datei3« kann zusätzlich noch gelöscht werden. »Prog1« kann nur ausgeführt werden, somit ist nicht einmal das Lesen möglich. Die vier Buchstaben können beliebig kombiniert werden. Das CLI setzt dafür einzelne Bits im Schutzwort der Datei.

Ab der Workbench 1.3 ist der Protect-Befehl um einiges leistungsfähiger geworden. Es gibt folgende neue Attribute:

```
s : Script
p : Pure
a : Archive
```

Das s-Bit kennzeichnet Scripts (mehr dazu siehe Kapitel 2.18).

Das p-Bit sollten Sie nur unter Beachtung der Regel aus Kapitel 2.5.2 setzen.

Das Archive-Bit sollten Sie eigentlich nicht setzen, das tun nämlich schon spezielle Backup-Programme. Im Prinzip läuft das so:

AmigaDOS löscht das Bit, wenn es auf eine Datei schreibt.

Das Backup-Programm sichert die Datei und setzt das Bit. Das Backup-Programm sichert eine Datei nicht, wenn ihr Archive-Bit gesetzt ist. Endergebnis: Nur seit dem letzten Backup geänderte Dateien werden gesichert.

Neu ist auch, daß man jetzt mit den Zusätzen ADD ein Attribut hinzufügen und mit SUB eines wegnehmen kann. Folglich heben sich auf:

```
protect NameDerDatei rwd add
protect NameDerDatei rwd sub
```

Die letzte Zeile läßt sich auch schreiben als

```
protect NameDerDatei -r-w-d
```

4.10 Schnell, aber gezielt löschen

Sie möchten in einem Verzeichnis einige Dateien löschen, die leider nicht mit einem gemeinsamen Muster anzusprechen sind. In diesem Fall geben Sie ein:

```
dir opt i
```

In diesem interaktiven Modus werden die Dateien nacheinander mit einem Fragezeichen angezeigt. Drücken Sie dann die Del-Taste, wird die angezeigte Datei gelöscht. Hier alle Tasten für den I-Modus im Überblick:

```
<Return>        : Weiter
<E>+<Return>    : Wechsel in das angezeigte Directory
<B>+<Return>    : Aus einem Directory eine Ebene höher
<T>+<Return>    : Inhalt der Datei zeigen (Abbruch mit <Ctrl>+<C>).
<Del>+<Return>  : Datei löschen, bei einem Directory nur
                  wenn es leer ist.
<Q>+<Return>    : Ende
```

4.11 Diskette ganz schnell löschen

Mit dem Befehl FORMAT vom CLI oder der Shell aus aufgerufen, können Sie eine Diskette wesentlich schneller löschen als mit DELETE, besonders dann, wenn die Diskette recht voll ist. Geben Sie ein:

```
FORMAT DRIVE df0: NAME NameDerDisk QUICK
```

Wichtig ist hier die Option QUICK. Damit werden nämlich tatsächlich nur die beiden Spuren mit dem Boot- bzw. Root-Block (Inhaltsverzeichnis) formatiert. Außerdem wird die Sektor-Belegungstabelle als leer markiert. Die anderen Spuren sind ja noch formatiert. Daß sie Daten enthalten, weiß aber das System nicht, weil die Sektor-Belegungstabelle aussagt, daß sie frei sind. Folglich werden sie einfach durch die neuen Daten überschrieben.

5 (Überlebens-)Regeln und Tips für Programmierer

5.1 Grundlagen

Es folgen gleich viele (Überlebens-)Regeln und Tips für Programmierer und alle, die es werden wollen. Doch halt! Ich meine, bevor ich die Regeln so einfach aufzähle, sollte ich wenigstens andeuten, warum man sie braucht, sprich, ein paar Grundlagen bringen.

Beim Amiga wird die ohnehin schon sehr leistungsfähige CPU noch durch weitere spezielle CPUs unterstützt. Diese sogenannten Co-Prozessoren sind hochspezialisierte Rechner, die zwar nur ganz bestimmte Aufgaben erledigen können, das aber mit einem Tempo, das auch ein 68000er selbst dann nicht erreichen würde, wenn er sich nur um diese eine Aufgabe kümmern müßte. Diese »Zusatz-Rechner« im Amiga sind

```
der Copper,
der Blitter,
und der Sprite-Prozessor.
```

Der Copper kontrolliert das Grafik-System. Einige Aufgaben erledigt er selbst, andere gibt er weiter.

Der Blitter hat die Aufgabe, Daten von einem Speicherbereich in den anderen zu kopieren und dabei noch einige Randbedingungen (Verknüpfung mit im Zielbereich vorhandenen Daten) zu erfüllen.

In Spielen rasen Rennwagen, Ufos, Raketen oder was auch immer sagenhaft schnell über den Schirm. Auch das heißt zuerst Transfer von Daten, aber das macht nicht mehr der Blitter. Diese so wahnsinnig schnellen Figuren nennt man Sprites. Beim Amiga gibt es einen speziellen Sprite-Prozessor, der 8 dieser Kobolde (engl. sprites) verwaltet.

Außer diesen Co-Prozessoren verfügt der Amiga noch über eine Sound-Maschine, die aus vier programmierbaren DA-Wandlern besteht. Damit lassen sich Musik in Vollendung, aber auch natürlich klingende Sprache generieren.

Seine Farbvielfalt gewinnt der Amiga aus 32 Farbregistern (speziellen Speichern). Jedem dieser »Color Register), kann eine von 4096 möglichen Farben zugewiesen werden.

Der Amiga hat auch einen Datenbus, und »Bus« heißt, Daten werden über den Bus geschickt, und wenn ein Gerät sendet, haben die anderen Pause. So der Normalfall. Der Amiga umgeht diesen Flaschenhals, unter dem alle Standard-Systeme leiden, auf zweierlei Art. Zum einen verfügen die Spezial-Prozessoren über sogenanntes DMA (Direct Memory Access), womit sie Daten ohne den Umweg über die CPU in den RAM transferieren können. Zum anderen läuft der Amiga intern mit 16 MHz. Immer ein Clock-Zyklus gehört dem 68000 und der nächste den Co-Prozessoren. Praktisch heißt das, der 68000 läuft immer mit seiner vollen Geschwindigkeit von 8 MHz und die Co-Prozessoren auch. Das bringt sagenhaft Tempo.

5.1.1 Totale Software-Orientierung

So verlockend es ist, direkt auf diese superschnelle Hardware zuzugreifen und damit »rasende« Programme zu schreiben, Sie dürfen es nicht! Zum Trost: Die Programme werden auch bei vorschriftsmäßiger Programmierung extrem schnell.

Der Amiga ist ein System, das total Software-orientiert ist. Es gibt nur eine einzige feste Adresse in diesem System, und das ist die Adresse 4, die Basis von Exec (siehe Regel 1). Natürlich haben findige Hacker schon Adressen im ROM gefunden, worüber sie direkt auf die Hardware zugreifen können. Ein darauf basierendes Programm ist aber per Prinzip schon zum Tode verurteilt. Niemand garantiert Ihnen, daß diese ROM-Adressen oder auch die Adressen von Systemvariablen im RAM konstant bleiben. Commodore kann jederzeit neue Amigas produzieren, in denen sich die Adressen ändern.

5.1.2 Hardware-unabhängig programmieren!

Diese absolute Hardware-Unabhägigkeit hat ihren tiefen Sinn schon in einer speziellen Eigenschaft des Amiga, nämlich seiner Fähigkeit zum Multitasking. Das heißt nicht nur, daß mehrere Programme quasi gleichzeitig laufen, sondern auch, daß jedes neu hinzukommende Programm irgendwo im Speicher abgelegt wird, wo gerade Platz ist. Wenn dieses Programm selbst Speicher für seine Daten belegt, wird ihm ein gerade irgendwo freier Block zugewiesen. Damit sind also schon absolute Variablen, wie Sie zum Beispiel die Sprache Modula erlaubt (eine Variable kann so deklariert werden, daß sie auf einer bestimmten Adresse liegt), beim Amiga nicht zulässig. Außerdem verstößt der Zugriff auf eine absolute Adresse gegen eine Grundregel des Multitasking. Sie werden sich sicherlich das Drama vorstellen können, das aufgeführt wird, wenn zwei Programme versuchen, gleichzeitig auf dieselbe Adresse zuzugreifen oder wenn ein Programm glaubt, es hätte da die richtigen Daten abgelegt und ein anderes verändert Sie dann.

5.1.3 Multitasking macht Exec

Zuständig für das Multitasking ist Exec. Exec ist die Abkürzung von Executive, was im amerikanischen soviel wie Chef (leitender Angestellter) bedeutet. Exec ist an sich auch nur ein Prozeß wie zum Beispiel das CLI, der aber immer (solange der Amiga eingeschaltet ist) läuft.

Exec selbst ist das Multitasking-System des Amiga. So ein System hat die Aufgabe, die Resourcen eines Systems auf verschiedene Tasks (Programme) zu verteilen. Resourcen sind die CPU (der 68000), der Hauptspeicher und die Peripherie-Geräte. Unteilbare Resourcen, wie die CPU, werden beim Amiga nach einem Prioritäten-Verfahren vergeben. Zuerst wird der Task mit der höheren Priorität abgearbeitet, dann der nächstniedere. Haben mehrere Tasks die gleiche Priorität, werden sie nach einem Zeitscheibenverfahren (Time-Sharing) in Intervallen bearbeitet. Durch Hardware-Interrupts bekommt Exec die Sache immer wieder in den Griff, auch wenn ein Task seine Tätigkeit nicht beenden möchte.

Ein Task hat drei Zustände, nämlich laufend, nicht laufend und wartend. Mit Rücksicht auf andere Tasks sollte ein Task möglichst oft in den Zustand wartend (auf Eingabe) gesetzt werden, das aber via Wait() und nicht etwa nach der unfeinen Polling-Methode (siehe Regel 5.9).

5.1.4 Der Zugriff auf System-Routinen

Um von absoluten Adressen unabhängig zu sein, arbeiten die meisten OS nach diesem Schema:

Alle Unterprogramme erhalten eine Nummer, Funktionsnummer genannt. Im OS steht eine Tabelle, in der notiert ist, welche Adresse zu jeder Funktionsnummer gehört. Das OS hat nun eine Routine, deren Adresse sich nie ändert. Das ist der Dispatcher. Um ein Unterprogramm aufzurufen, übergibt man dem Dispatcher die Funktionsnummer. Dieser berechnet danach (und mit Hilfe der Tabelle) die Adresse der Routine und ruft sie auf.

Der Amiga macht das etwas raffinierter und damit zukunftssicherer. Der Nachteil der Standard-Methode ist nämlich, daß man sehr schlecht neue Routinen hinzufügen kann (Tabelle steht im ROM). Beim Amiga stehen die Tabellen im ROM oder RAM oder auf der Diskette. Die Tabellen sind ein Teil der sogenannten Libraries (Bibliotheken).

5.1.5 Libraries: Schlüssel zum Amiga

Eine Library ist vereinfacht ausgedrückt eine Sammlung von Unterprogrammen mit einer zugehörigen Tabelle (je Unterprogramm ein Eintrag). Für jeden Zweck (zum Beispiel DOS, Intuition, Grafik) gibt es eine eigene Library. Will man eine Funktion einer Library benutzen, muß man die Library mit »OpenLibrary« öffnen. Diese Funktion gibt einen Zeiger auf den Beginn (die Startadresse) der Tabelle zurück. Um nun ein Unterprogramm aufrufen zu können, muß man die Startadresse der Tabelle angeben und ein sogenanntes Offset, das die Differenz zwischen Startadresse und zugehörigem Tabellenplatz ist. In dieser Konsequenz gilt das aber nur für Assembler-Programme. In C ist die Sache etwas bequemer.

»Gemanagt« wird das Ganze vom sogenannten Library-Manager (ein Teil von Exec). Der Manager weiß, ob sich eine Library schon im ROM oder RAM befindet. Wenn nicht, versucht er, die Library von der Diskette zu laden. Klappt das nicht (Library ist nicht auf der Diskette oder Speicher ist schon voll), gibt er Null als Adresse zurück.

Der Umstand hat noch einen Grund: Wir haben ein Multitasking-System, was auch heißt, daß quasi gleichzeitig verschiedene Tasks (Programme) eine Library benutzen können. Der erste Task wird die Library notfalls von der Diskette in den RAM laden (genau: das Laden veranlassen). Öffnen weitere Tasks dieselbe Library, wird der Manager nur noch die Adresse an diese Tasks melden und sich merken, daß diese Tasks die Library (auch) braucht. Daraus folgt: Eine Library darf erst wieder aus dem Speicher gelöscht werden, wenn der letzte Task gesagt hat, daß er sie nicht mehr braucht. Dafür gibt es die Funktion »CloseLibrary«. Jeder Task (also jedes Programm, das Sie schreiben) muß deshalb alle Libraries, die er geöffnet hat, auch wieder schließen. Andernfalls könnte bald der Speicher knapp werden.

5.2 Regel 1: Es gibt nur eine absolute Adresse

Man sieht immer wieder Listings, die auf absolute Adressen zugreifen, doch nur eine einzige, nämlich die 4, ist erlaubt. Das ist die »_SysBase«, und dort liegt der Zeiger auf die Basis von »Exec«. Alle weiteren Adressen sind über definierte Offsets und System-Funktionen zu ermitteln. Ggf. erhält man damit die Zeiger auf Datenstrukturen, die wiederum Zeiger auf andere Datenstrukturen halten. Doppelte Indirektion mit Offset ist zwar oft die Folge solcher Wege, aber diese Methode ist die einzig zulässige. Gehen Sie davon aus, daß Programme mit absoluten Adressen nur auf Ihrem Amiga und nur mit der einen Kickstart- und Workbench-Version laufen.

Ansonsten muß man sich nur einmal merken, wie man mit der doppelten Indirektion umgehen muß, und bald wird das zur Routine. Ein Beispiel: Wir haben ein Window geöffnet und das Ergebnis (den Zeiger auf die Window-Struktur) in der Variablen »windowptr« gesichert. Nun müssen wir auf die Signal-Bits zugreifen. In der Window-Struktur gibt es aber nur einen Zeiger (beim Offset wd_UserPort), der auf die Message-Port-Struktur zeigt, wo beim Offset »MP_SIGBIT« das gesuchte Datum steht. Das sieht dann so aus:

```
move.l windowptr,a0          ;zeige auf Window-Struktur
move.l  wd_UserPort(a0),a0 ;nun auf Message-Port
move.b  MP_SIGBIT(a0),d1    ;das Ergebnis
```

Wenn Ihnen das zu mühsam ist, programmieren Sie in C, wo sich das Ganze reduziert auf:

```
windowptr->UserPort->mp_SigBit
```

5.3 Regel 2: Funktions-Ergebnisse testen

Testen Sie generell die Ergebnisse aller Funktionen, die nicht ausdrücklich als »VOID« deklariert sind. Sie müssen davon ausgehen, daß Ihr Programm in Konkurrenz zu vielen anderen läuft (Multitasking), und somit durchaus nicht klar ist, ob noch der Speicher für beispielsweise eine neue Window- oder Screen-Struktur frei ist. Gehen Sie davon aus, daß fast jede Aktion Speicher

kostet. Im Extremfall müssen Sie sogar unterstellen, daß die paar Bytes fehlen, die für das Öffnen einer ROM-Library benötigt werden.

5.4 Regel 3: Speicher wieder freigeben

Geben Sie auf jeden Fall allokierten Speicher wieder frei und nicht nur den mit »alloc()« beschafften. Schließen Sie auch alle Screens, Windows und die Libraries. Beachten Sie diese Regel besonders im Fehlerfall. Hier passiert es häufig, daß man Speicher für einige Datenstrukturen beschafft hat, dann etwas schiefgeht, und vor dem Abbruch des Programms eben nicht der Speicher für die bisher allokierten Strukturen freigegeben wird.

Natürlich kann es vorkommen, daß Ihr Programm lt. Regel 4 festgestellt hatte, daß der Speicher ausreicht, nun loslegt, und es dann »mitten drin« doch nicht reicht. Der Grund ist schlicht, daß sich ein anderer Task den Speicher inzwischen »geschnappt« hat. Nun haben Sie aber derweil eine unbekannte Anzahl von Speicherblöcken für Images, Sprites und sonstiges reserviert. Wie geben Sie diese wieder frei?

Die Lösung bietet Intuition mit seiner Funktion »AllocRemember«. Ein Beispiel dazu finden Sie im Abschnitt 5.6.2. Diese Funktion notiert alle erfolgreichen Allokierungen in einer gemeinsamen Liste, einer Struktur vom Typ »Remember«, die Sie zur Verfügung stellen müssen, genauer: einen Zeiger darauf (siehe Beispiel). Nehmen wir an, die Variable heißt »*rem«, dann können Sie mit diesem einen Befehl

```
FreeRemember(&rem, TRUE);
```

alle Speicherblöcke auf einmal freigeben. Beachten Sie unbedingt das »TRUE«, denn – es ist kaum zu glauben – setzen Sie hier »FALSE« ein, werden nicht die Speicherblöcke, sondern nur die Liste gelöscht.

5.5 Regel 4: Frühzeitig testen

Kein Anwender findet es schön, wenn ein Programm erst anläuft und dann »mittendrin« aussteigt. Es gilt zwar auch für diesen Fall die Regel »informieren Sie den Anwender über den Grund des Abbruchs«, doch am besten prüfen Sie gleich zu Programmbeginn, ob noch genügend Speicher frei ist und die sonstigen Voraussetzungen für diese Applikation erfüllt sind. Wenn nicht, nennen Sie dem User die Gründe und die möglichen Abhilfemaßnahmen.

Wenn dennoch im Programmlauf Fehler auftreten und alle Stricke reißen, müssen Sie sogar mit einem Dead-End-Alert aussteigen, also nach einer Guru-Meldung neu booten lassen. Still aussteigen und den Anwender mit einem anderen Programm abstürzen zu lassen, ist nicht die feine englische Art.

5.6 Regel 5: Der Resource-Manager sind Sie

Im Gegensatz zu den ganz großen Multitasking-Systemen müssen Sie beim Amiga selbst etwas – wenn auch sehr wenig – tun, wenn es um den Zugriff auf System-Resourcen geht. Wenn zwei Tasks auf Resourcen wie ein Laufwerk gleichzeitig zugreifen wollen, gibt es Konflikte. Um diese zu vermeiden, müssen Sie diese Aktion einkleiden in

```
forbid();      /* Multitasking aus */
/* Aktion */
permit();      /* Multitasking an  */
```

Natürlich sollte das »forbid()« nur so kurz wie möglich wirken, weil in der Zeit alle anderen Tasks keine Chance haben, doch viel wichtiger ist, daß man es nicht vergißt. Es gibt nämlich durchaus Gelegenheiten, wo man nicht unbedingt an Regel 4 denkt, zum Beispiel diese:

Ein Workbench-Programm darf nicht einfach loslaufen, sondern muß auf die Starterlaubnis, sprich, eine Message warten. Diese Nachricht muß es sich merken und am Programm-Ende wieder zurücksenden. Das kann dann in Assembler so aussehen:

```
CALLEXEC    Forbid
move.l      Message(pc),a1
CALLEXEC    ReplyMessage
rts
end
```

Das ergibt gleich zwei Fragen, nämlich warum braucht man hier »Forbid«, und warum fehlt das »Permit«? Der Veranlasser ist die Funktion »ReplyMsg«. Das Betriebssystem hatte unserem Programm nämlich nicht nur die Starterlaubnis gegeben, sondern sich auch im PLB (Prozeß-Leit-Block) notiert, daß unser Prozeß läuft. Mit »ReplyMsg« melden wir uns wieder ab, und das muß natürlich auch im PLB eingetragen werden. Folglich impliziert die Funktion »ReplyMsg« einen Zugriff auf den PLB. Kommt gleichzeitig eine anderer Prozeß auf die Idee, auch auf den PLB zuzugreifen, knallt es leider.

Das »Permit« fehlt hier, weil einem Task automatisch das »Forbid-Recht« entzogen wird, wenn er endet (logisch), aber auch, wenn er »Wait« aufruft.

5.7 Regel 6: Chip-Daten ins Chip-RAM

Die Custom-Chips des Amiga (Blitter, Copper usw.) können hardwarebedingt nur auf die ersten 512 Kbyte – das sogenannte Chip-Memory – zugreifen. Folglich müssen auch alle Datenstrukturen, die von diesen Bausteinen genutzt werden, in diesem Bereich liegen. Hat der Amiga aber mehr als 512 Kbyte RAM, wird ein Programm in aller Regel in den Erweiterungsspeicher –

das sogenannte »Fast Memory« – geladen. »Fast« (schnell) heißt das übrigens, weil »da oben« die CPU nicht mehr Taktzyklen an die Custom-Chips abgeben muß.

Mit dem Programm wandern auch die Daten in das »Fast Memory«, sind also für die Custom-Chips unerreichbar. Günstigenfalls führt das zu einem verschwundenen Mauszeiger und ähnlichen Effekten, doch auch Abstürze sind drin. Zwei Lösungen sind möglich.

1. Per Compiler-Option Daten ins Chip-RAM
2. Im Programm gezielt Chip-RAM allokieren

5.7.1 Daten ins Chip-Memory legen per Compiler-Option

Die erste Möglichkeit ist bequem und sicher, hat aber den Nachteil, daß alle Daten ins Chip-Memory gelangen, und da ist es oft genug schon eng. Dennoch sollten Sie wissen, wie das geht. Im Compiler-Handbuch steht es, doch richtet man sich danach, ergibt das bei Lattice-C in der Version 4.0 nur den Error 510. Der Trick dabei ist folgender: In Lattice-C sind Variablen wie »IntuitionBase« systemgobal, weshalb man sie als extern erklären kann. Nun geht z.B. der »Data-Hunk« ins »Chip-Memory« mit

```
lc -L -ad NameDerSource
```

Sie sehen dann zwar noch die Meldung »b ignored«, die dürfen Sie aber auch ignorieren.

5.7.2 Daten ins Chip-Memory legen mit AllocRemember

Die zweite Möglichkeit macht etwas mehr Arbeit, hat aber auch einen zweiten Vorteil. Werden nämlich Datenstrukturen nur temporär benötigt, kann man den dynamisch allokierten Speicher auch wieder freigeben, was bei statischen Strukturen bekanntlich nicht möglich ist. Wie so ein Programm aussieht, zeigt Abb. 5.1.

Das Prinzip ist ganz einfach. Man darf halt nur die Daten in das »Chip-Memory« legen, die dort unbedingt sein müssen, zum Beispiel Pointer-Strukturen.

```
/* Demo SetPointer() mit AllocRemember() */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>

#define MEMF_CHIP 2

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
```

```
struct Window *Window;
USHORT Maus[] =
        {
          0x0000, 0x0000,
          0x0830, 0x0C10,
          0x0460, 0x0620,
          0x02C0, 0x0340,
          0x0180, 0x0180,
          0x0380, 0x01C0,
          0x0640, 0x0260,
          0x0C20, 0x0430,
          0x1810, 0x0818,
          0x0000, 0x0000
       };

main()
{
   struct Remember *RememberKey;
   UBYTE i;

   struct
    {
      USHORT m[20];
    } *MP, *AllocRemember();

   open_libs();

   Window = (struct Window *) open_window(
            20,20,400,170,"Titel",
            WINDOWCLOSE | ACTIVATE, CLOSEWINDOW);
   if (Window == NULL) exit(FALSE);

   RememberKey = NULL;
   MP = AllocRemember(&RememberKey,sizeof(Maus),MEMF_CHIP );

   for (i=0; i<20; i++)
     MP->m[i] =  Maus[i];

   SetPointer(Window, &MP->m[0], 8, 8, -4, -4);

   Wait(1L << Window->UserPort->mp_SigBit);

   FreeRemember(&RememberKey, TRUE);

   close_all();

}
#include "stdwindow.h" /* siehe Regel 7 */
```

Abb. 5.1: So bringt man Daten in das Chip-Memory

Das Programm generiert einen neuen Maus-Zeiger. Kern der Übung ist diese Zeile:

```
MP = AllocRemember (& RememberKey, sizeof(Maus), MEMF_CHIP );
```

Es handelt sich dabei um eine erweiterte Malloc-Funktion. Der »RememberKey« zeigt auf die Remember-Struktur, in der Intuition die Speicherreservierungen verwaltet. Vor dem ersten Aufruf muß dieser Zeiger mit NULL geladen werden. Der zweite Parameter ist die gewünschte Größe (wie bei malloc()), der dritte aber ist das was wir wollen, nämlich eine Vorgabe für den Zielbereich. Wenn Sie hier MEMF_CHIP eintragen, wird der Speicher im Chip-Memory allokiert. Die Konstante habe ich hier mit »#define« definiert, weil es mir widerstrebte, nur wegen einer Konstanten einen ganzen Include-File (hier memory.h) zu laden.

Normalerweise beschafft man sich Speicher für ganze Strukturen, weshalb ich in diesem Beispiel auch eine »struct« angelegt habe, nur um einmal das »Wie« aufzuzeigen. Da steht zwar jetzt nur der Platz für den Array m[20] drin, aber Sie können ja auch noch die übrigen Sprite-Daten dazupacken, so wie es sich eigentlich auch gehört.

MP (wie Mouse Pointer) ist nun der Zeiger auf diese Struktur im Chip-Memory. Der Einfachheit halber erspare ich mir die 20 Zuweisungen und kopiere einfach die Daten aus der vorher definierten Struktur »Maus« mittels der For-Schleife.

Zum Schluß muß man gemäß Regel 3 den Speicher wieder freigeben, und das geschieht hier mit

```
FreeRemember (&RememberKey, TRUE);
```

Der Parameter TRUE besagt, daß sowohl die Listen-Knoten freigegeben werden sollen, die auf die einzelnen Speicherblöcke zeigen (vorausgesetzt AllocRemember wurde mehrfach aufgerufen), als auch die Speicherblöcke selbst. Setzen Sie diesen Parameter auf FALSE, wird nur die Liste gelöscht, eine gewagte Aktion, wie ich meine, also bleiben Sie »true«.

5.8 Regel 7: Das Rad nur einmal erfinden

Daß man häufig benutzte Routinen in Include-Files hält und die dann immer wieder einzieht, ist eine uralte Maßnahme, doch beim Amiga ist sie besonders wichtig. Wenn man sich nämlich bei den vielen komplexen Datenstrukturen nur einmal ein wenig verhaut, reicht das oft schon für einen bildschönen und schwer zu findenden Absturz. Besonders kritisch sind übrigens als Konstanten definierte Datenstrukturen. Vergißt man da einmal ein Element oder nimmt den falschen Typ, »gurut« es garantiert. Deshalb: Alles was läuft, gleich in einen Include-File packen und den im nächsten Programm einsetzen. Ein ganz nützliches Exemplar dieser Gattung – welche Anwendung macht kein Window auf? – zeigt Abb. 5.2.

```
/****************** stdwindow.h *****************/
open_libs()
{
   IntuitionBase =
   (struct IntuitionBase *)
            OpenLibrary("intuition.library",0L);
   if (IntuitionBase == NULL)  exit(FALSE);

   GfxBase =
   (struct GfxBase *) OpenLibrary("graphics.library",0L);
   if (GfxBase == NULL)
   {
      CloseLibrary(IntuitionBase);
      exit(FALSE);
   }
}
open_window(x, y, w, h, name, flags,i_flags )
short x,y,w,h;
char *name;
{
   struct NewWindow nw;

   nw.LeftEdge = x;
   nw.TopEdge = y;
   nw.Width = w;
   nw.Height = h;
   nw.DetailPen = -1;
   nw.BlockPen = -1;
   nw.Title = name;
   nw.Flags = flags;
   nw.IDCMPFlags = i_flags;
   nw.Screen = NULL;
   nw.Type = WBENCHSCREEN;
   nw.FirstGadget = NULL;
   nw.CheckMark = NULL;
   nw.BitMap =  0;     nw.MinWidth = 50;
   nw.MinHeight =20;
   nw.MaxWidth = 640;
   nw.MaxHeight =200;

   return( OpenWindow(&nw) );
}

close_all()
{
   CloseWindow(Window);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);
   exit(TRUE);
}
```

Abb. 5.2: Ein Include-File, der uns viel Arbeit abnimmt

5.9 Extra-Regeln für Assembler-Programmierer

Assembler-Programmierer müssen noch einige weitere Dinge beachten, denn die große Freiheit dieser Sprache heißt leider auch große Verantwortung.

5.9.1 D0 ist nicht getestet

Gemäß Regel 3 müssen Sie Funktionsergebnisse testen, im allgemeinen das Register D0. Häufig hat die aufgerufene Routine die Flags schon aktualisiert, doch das ist nicht garantiert. Folglich schreibt man meistens so etwas:

```
lea      intname,a1          ;Intuition öffnen
moveq    #0,d0
CALLEXEC OpenLibrary
tst.l    d0                  ;Fehler
beq      Fehler2             ;wenn ja
move.l   d0,_IntuitionBase   ;BasisZeiger sichern
```

Das ist die sicherste Lösung, doch nicht die optimale. Wenn Sie nämlich die Variable »_IntuitionBase« nicht noch für einen weiteren Test einsetzen wollen, können Sie auch schreiben:

```
lea      intname,a1          ;Intuition öffnen
moveq    #0,d0
CALLEXEC OpenLibrary
move.l   d0,_IntuitionBase   ;BasisZeiger sichern
beq      Fehler2             ;wenn Fehler beim Öffnen
```

Da auch der Move-Befehl die Flags aktualisiert, kann man sich auf diese Art den TST-Befehl sparen.

5.9.2 Vier Register sind Scratch?

Offiziell sind die Register D0, D1 und A0, A1 Scratch (engl. Schmierpapier), da müssen Sie also davon ausgehen, daß diese Register nach einem Funktionsaufruf andere Werte haben. Die übrigen Register sollten von den System-Routinen nicht geändert werden, doch das ist eher als eine schöne Absichtserklärung zu bewerten. Sie sollten sich deshalb generell selbst um die von Ihnen eingesetzten Register kümmern, bzw. folgende Taktiken anwenden:

1. Gezielt sichern, z.B. mit
   `move.l d1,-(sp)`
2. Global sichern mit
   `movem.l d0-d7/a0-a7,-(sp)`
3. Mit Variablen arbeiten, z.B.
   `move.l d0,windowptr`

5.9.3 Programmieren Sie »32-Bit-clean«

Der Mercedes unter den PCs (namens Macintosh) hat zur Zeit ein Riesenproblem. Mit Einführung des 32-Bit-QuickDraw, das 16,7 Millionen Farben erlaubt, laufen zahlreiche Programme nicht mehr. Das Peinliche an der Geschichte ist, daß bei den billigeren Modellen der Anwender noch die Wahl hat, das neue QuickDraw von der Diskette zu laden oder auch nicht. Doch bei den teueren Top-Geräten steckt diese Software schon im ROM.

Ganz klar haben die Entwickler dieser Programme gegen die Programmier-Richtlinien des Herstellers verstoßen. Sie machen das natürlich nicht und betrachten und behandeln Adressen immer als 32 Bit breit, auch wenn der 68000 nur einen 24-Bit-Adreßbus hat.

5.9.4 Denken Sie an die Nachfolger des 68000

Das Schöne am 68000 ist, daß nur ein einziger Befehl auf den Nachfolgern nicht läuft, nämlich

```
move SR,<ea>
```

womit man sich beim 68000 die Condition Codes des Status-Registers ansehen kann, ohne in den Supervisor-Modus zu gehen. Genau das geht ab dem 68010 nicht mehr, weil dort »move SR,<ea>« ein privilegierter Befehl ist, der nur im Supervisor-Modus ausgeführt werden kann. Sie müssen aber nicht selbst dahin und wieder zurückschalten, sondern können (sollten) statt dessen schon jetzt die Exec-Funktion »GetCC« einsetzen.

5.9.5 Vergessen Sie den Return-Code nicht!

Jedes Programm gibt den Wert des Registers D0 als Return-Code zurück, und meistens hat D0 einen Wert, landen doch fast alle Funktionsergebnisse in diesem Register. Nun heißt aber per Konvention jeder Wert von D0 ungleich Null »Fehler«.

Normalerweise macht das wenig, jedenfalls solange das Programm nicht von einem anderen aufgerufen wird, das den Return-Code auswertet. Genau das ist aber häufig der Fall, wenn das Programm in einer Kommando-Datei, zum Beispiel der Startup-Sequence, ausgeführt wird. Steht hier ein »if warn«, kann es damit zu unerwünschten Ergebnissen kommen. Also setzen Sie besser immer D0 auf Null, typisch so:

```
moveq #0,d0
rts
```

Anders sieht es aus, wenn Ihr Programm ein anderes aufruft. Hier retten Sie zuerst mit

```
move.l  d0,-(sp)
```

Ihren Return-Wert und rufen dann die andere Routine auf. Wenn da ein Fehler auftritt, müssen Sie den behandeln. Ist das erledigt, kann Ihr Programm enden mit

```
move.l   (sp)+,d0
rts
```

Ein Beispiel für diese Technik finden Sie im Kapitel 6.5.

5.10 Beachten Sie die »Postvorschriften«

Bei der Deutschen Bundespost gibt es sehr genaue Vorschrift über den Transport und die Behandlung von Nachrichten. Beim Amiga gilt ähnliches, kommunizieren doch alle Tasks – die eigenen und die des Betriebssystems – mittels Messages (elektronischer Briefe), die sie sich gegenseitig schicken. Dazu noch einige Details:

Jede Eingabe beim Amiga läuft über das sogenannte Input-Device in den Input-Stream. Letzterer ist ein Strom von Daten, genauer ein Puffer-Bereich. Wenn der Anwender eine Taste drückt, die Maus bewegt oder eine Maus-Taste betätigt, generiert das Input-Device daraus ein Input-Event (Input-Ereignis). Ein Input-Event wiederum ist eine Message (Nachricht), die das Ereignis beschreibt.

Der Input-Stream kann nun von jedem Task abgefragt werden. Diese Abfrage ist nicht ganz einfach, na sagen wir, ziemlich umständlich zu programmieren. Deshalb sollten wir auch diesen Job Intuition überlassen, da sind nämlich die passenden Routinen schon eingebaut. Dabei funktioniert Intuition als eine Art Filter, das nur die uns interessierenden Events durchläßt. Intuition sorgt automatisch dafür, daß die Events immer an den derzeit aktiven Task (an das aktive Window) weitergeleitet werden.

Die Events gelangen (neben dem Console-Device) in den IDCMP (Intuition Direct Communication Message Port). Der IDCMP ermöglicht den Empfang aller Ereignisse im »Rohformat«. Für viele Ereignisse stört das nicht, zum Beispiel das Window-Close-Event oder die Mauskoordinaten können gar nicht besser vorliegen.

Die Abfrage des IDCMP in einem Programm ist an sich ganz einfach. Wir erhalten eine Message, wenn ein Event eingetreten ist. Im Class-Feld des Message-Ports steht dann ein Langwort, in dem ein Bit gesetzt ist, das dem Ereignis-Typ entspricht. Dieses Bit entspricht genau dem des IDCMP-Flags, das Sie vorher gesetzt hatten.

Haben wir also beim Öffnen des Windows CLOSEWINDOW gesetzt, könnten wir theoretisch nun folgendes tun:

```
msg = GetMsg(Window->UserPort); /* Lese  Message */
  if( msg->Class == CLOSEWINDOW ) .....
```

Praktisch dürfen wir aber so nicht verfahren, denn zwei Regeln sollten Sie immer im Auge behalten, nämlich:

- Nach GetMsg() muß ReplyMsg() folgen
- Nach ReplyMsg() sind die Daten im Message-Port ungültig.

Daraus folgt mindestens dieser Ablauf:

1. Message mit GetMsg() holen
2. Daten aus Message-Port in andere Variable kopieren
3. Mit ReplyMsg() Erhalt der Message quittieren
4. Auf Message reagieren
5. Weiter bei 1.

Das »mindest« möchte ich sehr betonen, weil ein so geschriebenes Programm gegen eine Grundregel des Multitasking verstößt. Jedes Programm sollte, wenn es auf ein Ereignis wartet, das auch sagen. Praktisch geschieht dies mit

```
Wait (1L << Window->UserPort->mp_SigBit);
```

Damit meldet sich der Task sozusagen ab mit der Meldung »ich gehe jetzt schlafen, weckt mich, wenn ein Ereignis für mich anliegt«. Praktisch wird damit der Task von Exec aus der Liste der aktiven Tasks auf die Liste der wartenden Tasks versetzt, womit anderen Tasks mehr Zeit zugeteilt werden kann. Tritt das Event ein, wird der Task automatisch wieder aktiviert.

Die falsche Methode wäre das sogenannte Polling. Hierzu fragt das Programm in einer Schleife ständig den Message-Port ab, ob eine Message anliegt. Natürlich vergehen bis dahin aus CPU-Sicht Ewigkeiten, die Antwort kann also durchaus 9999mal nein und dann einmal ja lauten. Leider blockiert diese Abfragerei CPU-Zeit, die den anderen Tasks fehlt. Solche Programm erkennt man an einer Programmzeile dieser Art

```
while( msg = GetMsg(Window->UserPort) == 0 )
  ;
```

Beachten Sie das Semikolon. Manchmal steht es noch auf derselben Zeile wie »while«, was dann auch noch schlechter Programmierstil ist. Ein korrektes C-Programm sieht in diesem Teil so aus:

```
Wait(1L << Window->UserPort->mp_SigBit);
while( msg = GetMsg(Window->UserPort) )
{
class = msg->Class;
code = msg->Code;
ReplyMsg( msg );

switch( class)
...
}
```

In Assembler sollte man die Event-Loop sinngemäß wie folgt schreiben:

```
move.l   windowptr,a0        ;zeige auf Window-Struktur
move.l   wd_UserPort(a0),a0  ;nun auf Message-Port
move.l   a0,a5               ;rette Port-Adresse
move.b   MP_SIGBIT(a0),d1    ;Signal Bit holen
moveq    #0,d0               ;Nummer in
bset     d1,d0               ;Maske wandeln
CALLEXEC Wait                ;Schlaf gut!

move.l   a5,a0               ;hole Port-Adresse
CALLEXEC GetMsg              ;hole Message
move.l   d0,a1               ;muss nach a1
move.l   im_Class(a1),d4     ;Msg-Typ
move.w   im_Code(a1),d5      ;Untergruppe
move.l   im_IAddress(a1),a4  ;Adr. f. Gadgets
CALLEXEC ReplyMsg            ;quittiere Msg in a1
```

5.11 Ein Debugger ist schon eingebaut

Im Amiga-ROM ist bereits ein Debugger eingebaut, dem Commodore den schönen Namen ROM-Wack verpaßt hat. Das Wichtigste zuerst: Um ROM-Wack einsetzen zu können, brauchen Sie ein Terminal, das an die serielle Schnittstelle des Amiga anzuschließen ist. Das kann auch ein zweiter Amiga, ein C64, ein PC oder was auch immer mit Terminal-Software sein, ROM-Wack ist da sehr anspruchslos. Ich habe meinen Amiga sogar an die RS-422 eines Macintosh geklemmt und MacTerminal als Software genutzt. Auf einem PC kann man MS-Windows und dessen Terminal-Emulator nehmen, und sogar solche Exoten wie VT220, der VAX-Terminal-Emulator für den PC, akzeptiert ROM-Wack.

Als Verbindung reicht ein Nullmodem-Kabel, also

```
2 -> 3
3 -> 2
7 -> 7
```

Das Terminal muß arbeiten mit

```
9600 Baud
8 Bit
1 Stop-Bit
keine Parity
kein Protokoll
```

5.11.1 ROM-Wack starten

Sozusagen zum Kennenlernen können Sie beim Booten <Ctrl>+<D> drücken, und das kurz bevor die Workbench geladen wird. Nun tippen Sie ein

```
loadwb -debug
```

Ganz im Gegensatz zum DOS-Handbuch muß »debug« klein geschrieben werden, sonst funktioniert die Sache überhaupt nicht. Wenn Sie nun in die Menüleiste gehen und über das Spezial-Menü nach rechts hinausgehen, taucht ein neues Menü mit dem Punkt »Debug« auf. Natürlich können Sie das auch in die Startup-Sequence eintragen, doch dann müssen Sie immer höllisch aufpassen, nicht diesen Punkt anzuwählen, wenn kein Terminal angeschlossen ist.

Um ROM-Wack etwas praxisgerechter zu starten, können Sie in C einfach die Exec-Funktion »Debug(0)« aufrufen, in Assembler schreiben Sie schlicht »jsr _LVODebug(a6)« (mit a6 auf ExecBase zeigend). Ein Mini-Programm für das DevPac sähe so aus:

```
opt l-
incdir    ":devpac/include/"
include   exec/exec_lib.i
CALLEXEC  Debug
rts
```

Die zweite Möglichkeit ist wahrscheinlich die nützlichste, denn ROM-Wack wird automatisch dann aufgerufen, wenn der Amiga normalerweise mit einem fatalen Fehler abstürzen würde.

Noch wichtiger ist der dritte Weg, und der geht so: Es gibt Fehler, wo der Amiga automatisch bootet. Jetzt müssen Sie nur sehr, sehr schnell sein, und solange die Power-LED noch blinkt, die Del-Taste (Code $7F) drücken. Dann wird der Boot-Vorgang abgebrochen und ROM-Wack gestartet.

In solchen Fällen können andere Debugger nur noch versagen. Sie können jedoch Ihren Debugger vorerst ruhig aktiv lassen. Er wird dann in den harmlosen Fällen anstatt ROM-Wack aufgerufen. Deshalb führt auch der oben geschilderte Aufruf »Debug(0)« ggf. in Ihren Debugger. Erst wenn alle Stricke reißen, starten Sie Ihren Amiga ohne dieses Tool und lassen ROM-Wack wirken.

Wenn ROM-Wack anspricht, wird der Status des Amiga eingefroren. Außer einem kleinen Supervisor-Stack ($200-$400) und der seriellen Schnittstelle läßt ROM-Wack alle anderen Bereiche unberührt. Das Multitasking wird ausgeschaltet, doch die Interrupts laufen weiter. Der Amiga läßt sich also noch via Tastatur und Maus bedienen (wenn Sie durch das Programm steppen). Sie sollten da allerdings bescheiden sein, denn es passiert folgendes:

Die Interrupts deponieren Daten in Puffern, die normalerweise von den zuständigen Device-Tasks »geleert« werden. Das kann hier nicht mehr der Fall sein, weshalb Sie auf dem Amiga (nur jetzt) nicht allzu aktiv werden sollten.

5.11.2 Das ROM-Wack-Display

ROM-Wack startet mit dem Display von Abb. 5.3.

```
rom-wack
PC: FC08F4  SR: 0004  USP: C5FDB8  SSP: C7FFFA  XCPT: 0000  TASK: C36D10
DR: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
AR: 00C3EA3C 00C6A8D8 00000000 00000000 00000000 00000000 00C00276
SF: 00C6 A898 00C3 D0D6 0000 0FA0 0000 0000 00C0 041E 00C0 4FDE 0D05 00C6 0970
-
```

Abb. 5.3: Das Register- und Status-Display von ROM-Wack

Am interessantesten ist hier das Feld »XCPT« (in älteren ROM-Versionen TRAP), weil man über die Exception-Nummer der Fehlerquelle schon oft sehr nahe kommt: Es bedeuten:

```
0:  Normaler Eingang über Debug().
1:  Bus-Error
3:  Adreß-Error
4:  Illegaler Befehl
5:  Division durch 0
6:  CHK-Befehl (sollte nicht vorkommen)
7:  TRAPV-Befehl (sollte nicht vorkommen)
8:  Privilegverletzung
9:  Trace-Bit gesetzt (Single Step)
A:  Line-A-Emulator
B:  Line-F-Emulator
2n: Trap-Befehl n
```

Das Task-Feld zeigt die Task-Nummer an, von der ROM-Wack aufgerufen wurde. Wenn das Feld 0 ist, heißt das Supervisor-Modus.

Die Zeile »SF« für Stack-Frame zeigt das letzte Stück Stack. Das ist besonders nützlich, wenn man im testenden Programm den Funktionsaufrufen signifikante Zahlen mitgibt, die man dann leicht im Stack-Frame finden kann.

Der Rest müßte klar sein. Etwas ungünstig ist, daß man in den Zeilen DR und AR selbst von D0–D7 bzw. A0–A6 zählen muß.

Der Speicherinhalt kann in Blöcken – Frames genannt – von 0–64 Kbyte angezeigt werden. Default ist eine Zeile mit 16 Byte (dargestellt als 8 Worte) und den zugehörigen 16 ASCII-Zeichen. Durch das Kommando »:n« mit n als eine Hexzahl kann die Frame-Größe in Bytes eingestellt werden. Beispielsweise ergibt

```
:20
```

zwei Zeilen mit je $10 Byte. Geben Sie nie den Doppelpunkt alleine ein. Das ergibt dann eine Blockgröße von 64 Kbyte und bis die durchgelaufen ist, dauert es bei 9600 Baud fast ewig.

5.11.3 Positionierung

Es gibt zwei Arten von Kommandos, nämlich einzelne Sonderzeichen, die sofort ausgeführt werden, und Strings, die mit <Return> abgeschlossen werden müssen. Die Regel ist ganz einfach. Die Buchstaben A–Z/a–z und die Ziffern 0–9 gehören immer zu Strings. Die Strings können nur mit der Backspace-Taste editiert werden, <Ctrl>+<X> löscht die ganze Zeile.

Folgende Einzelkommandos ändern die aktuelle Adresse:

```
.       ein Frame weiter
,       ein Frame zurück
>       ein Wort weiter (oder Leertaste)
<       ein Wort zurück
```

Zu den obigen Positionierungs-Kommandos gehören noch diese Strings:

nnnn (hex) auf Adresse nnnn
+nn nn Byte weiter
–nn nn Byte zurück

Liegt das Display ungünstig, können Sie mit <Return> den aktuellen Frame nochmals anzeigen. Ganz toll sind die Einzel-Kommandos »[« und »]«, weil damit folgendes geht

```
4       ; gehe auf Adresse 000004 (_SysBase)
[       ; gehe zur Adresse auf die das Langwort ab 00004 zeigt
]       ; Return zur vorherigen Indirekt-Adresse
```

Das klappt allerdings nur auf Terminals, wo diese Zeichen direkt erreichbar sind. Unter MS-Windows zum Beispiel, das die Alt-Taste gesondert behandelt, kommen Sie an diese Zeichen auf einer MF-II-Tastatur nicht heran.

5.11.4 Suchen und Finden

Mit »find« kann man ein Muster suchen lassen, wobei die Suche per Default bis $1000000 läuft. Will man das limitieren, kann man ein anderes Limit setzen. Zuerst sollten Sie vielleicht das Limit setzen und zwar mit

```
nnnn  <Return>
limit <Return>
```

Dann tippen Sie

```
find  <Return>
```

und lesen daraufhin

```
find pattern?
```

Jetzt tippen Sie das Muster (in hex) gefolgt von <Return> ein.

5.11.5 Speicherinhalt ändern

Um ein Wort im Speicher zu ändern, gehen Sie auf das Wort und tippen dann »=«. Nun geben Sie das neue Wort ein oder nur <Return>, wenn Sie doch nicht ändern wollen. Eine bereits begonnene Eingabe können Sie mit <Ctrl>+<X> abbrechen.

Wenn Sie mehrere Wörter hintereinander ändern wollen, geben Sie besser »alter« ein. Das könnte dann so aussehen:

```
alter
001400 0280 = 123
001402 00C8 = <
001400 0123 = 124
001402 00C8 = 333
001404 0000 = >
001406 3700 = <Return>
```

Sie sehen, daß es sich hier a) um ein wiederholtes »=«-Kommando handelt und b), daß man mit den Kommandos »>« und »<« auch weiter- bzw. zurückblättern kann, anstatt einen Wert einzugeben.

Sie können vom aktuellen Wort bis zum eingestellten Limit einen Speicherbereich mit Bytes, Worten oder Langworten füllen. Es kommt dabei auf die Länge des Eingabe-Strings an. Stellen Sie unbedingt das Limit ein. Dann tippen Sie

```
fill  <Return>
```

und werden dann wie bei »find« nach dem »pattern« gefragt.

5.11.6 Programm testen

Das Programm wird ab dem aktuellen PC fortgesetzt, wenn Sie »go« oder »resume« eintippen. Dafür reicht aber auch <Ctrl>+<D>. Sie können mit der Tab-Taste oder <Ctrl>+<I> steppen, und wenn Ihnen das zu langweilig wird, wieder »go« geben.

Breakpoints

Sie können auch Breakpoints setzen, wobei gilt:

```
set   : Setzt einen Haltepunkt auf die aktuelle Adresse
clear : löscht einen Haltepunkt auf der aktuellen Adresse
show  : zeigt alle Breakpoints
reset : löscht alle Breakpoints
```

Wenn ein Breakpoint erreicht wird, zeigt ROM-Wack den Register-Frame an. Der Haltepunkt wird dann automatisch gelöscht. Sie können dann mit »go« oder »resume« weitermachen. Ich ziehe es allerdings vor, in meinen Assemblerprogrammen die »Breakpoints« sozusagen in den Quelltext einzubauen, weil ich dann immer weiß, wo ich bin. Dazu füge ich einfach nur ein paar

```
CALLEXEC Debug
```

ein und hangele mich dann im ROM-Wack mit <Ctrl>+<D> von einem Aufruf zum nächsten.

5.11.7 ROM-Wack verlassen

Wenn das zu testende Programm normal abläuft und sein »rts« erreicht, landet man automatisch wieder im CLI. Sie können aber auch – keinen zu harten Crash vorausgesetzt – mit dem Kommando »user« wieder in den Multitasking-Modus zurück. Geben Sie dann Ihren Disks noch die Chance, evtl. Pufferinhalte wegzuschreiben, und booten Sie danach Ihren Amiga besser neu.

Die härteren Methoden sind diese Kommandos:

boot: Bringt manchmal noch den schönen Alarm »Software Error, Task held« oder wirkt wie »ig«.

ig: Bewirkt einen Kaltstart.

5.12 Und immer gilt die FGO-Regel

Die drei Buchstaben stehen für Funktionalität, Geschwindigkeit und Oberfläche.

Die Regel stammt aus der Macintosh-Welt und woher auch sonst? Mit diesem Computer begann der Siegeszug der grafisch orientierten Bedienoberflächen und auch der Lernprozeß der Programmierer. Damit wären wir beim Thema.

Wer von einem mauslosen Computer auf den Amiga umsteigt, ist natürlich fasziniert von den Möglichkeiten dieser Grafikmaschine und legt für sein neues Programm erst einmal eine Benutzerobfläche hin, die so intuitiv bedienbar ist, daß jedes Handbuch überflüssig wird. Das erste

Ziel eines GUI (Graphic User Interface), nämlich die leichte Erlernbarkeit eines Programms wurde also erreicht.

Doch die Anwender sind unverschämte Leute. Kaum, daß sie mit dem Programm umgehen können und seine Funktionen beherrschen, verlangen sie auch schon, daß diese Funktionen möglichst schnell ausgeführt werden. Ein Testbericht der Art, daß Ihr Super-Wordprozessor zum Ersetzen aller e durch xyz in 10 Seiten Text vier Sekunden braucht und die Konkurrenz nur derer zwei, kann schon das große Aus sein.

Schauen Sie sich um (und auf sich selbst), und sie werden merken, daß die schnellsten Programme einer Klasse immer auch die beliebtesten sind. Häufig werden sogar zugunsten der hohen Arbeitsgeschwindigkeit Mängel im GUI in Kauf genommen.

Woraus folgt: Die leichte Erlernbarkeit ist zwar ein edles Ziel, sie darf aber nicht dazu führen, daß das Programm für den fortgeschrittenen Anwender zu langsam wird. Langsam ist ein Programm in diesem Sinne auch dann, wenn keine Abkürzungen vorhanden sind. Sie müssen sich das vor Augen führen. Anstatt

Drücken der rechten Maustaste
- Anfahren des Menüs
- Wahl von »Sichern«
- Warten auf File-Dialog
- Scrollen durch die Listbox
- Wahl des File-Namens aus Listbox
- Klick auf OK

tippt der »advanced user« doch lieber <Ctrl>+<s> <name> <return>. Kleine Anmerkung dazu: Auch wenn wissenschaftlich erwiesen ist, daß der kurze Wechsel von der Tastatur zur Maus und zurück schneller ist als so manches Tastenkürzel, hilft Ihnen das gar nichts. Wenn der User den subjektiven Eindruck hat, auf die falsche Art schneller zu sein, dann geben Sie ihm seine Tasten für alle Lebenslagen.

Um wieder auf das FGO zu kommen. Kümmern Sie sich zuerst um das F, also um alle nötigen und – noch besser – um möglichst viele Funktionen. Mit der Funktionsvielfalt steigt nämlich die Zahl der potentiellen User (Käufer). Dann sehen Sie zu, daß diese Funktionen mit der höchstmöglichen Geschwindigkeit ausgeführt werden. Erst an dritter Stelle kommt das O. Das heißt nun nicht, daß Ihr GUI den Charme einer Schreibmaschine haben darf, doch die Oberfläche ist OK, wenn sie dem Intuition-Standard entspricht. Opfern Sie keine Minute für Gags im GUI, solange F und G nicht stimmen. Gerade die kommerziellen Anwender und die Power-User sehen über manchen Mangel in der Oberfläche hinweg, wenn sie von der Funktionsvielfalt und dem Tempo begeistert sind.

6 Tips für Assembler-Programmierer

In diesem Kapitel geht es um Assembler-Tricks und -Tips. Der Tenor liegt auf »schneller und kürzer«, denn ich meine, man setzt Assembler hauptsächlich deshalb ein. Dann gilt aber auch: wenn schon, denn schon. Einige grundlegende Assembler- und Amiga-Techniken sind jedoch auch dabei.

6.1 Schneller und kürzer

Assembler ist zwar die schnellste aller Programmiersprachen, doch manchmal ist ein C-Programm schneller. Wie ist das möglich? Nun, das C-Programm hat wenig eigenen Code und ruft hauptsächlich ROM- bzw. Library-Routinen auf, die in Assembler geschrieben sind. Deren Autoren aber sind Profis, die so einige Tricks auf der Pfanne haben.

6.1.1 Quick bevorzugen

Die Quick-Befehle, wie zum Beispiel »moveq« heißen nicht umsonst so. Verwenden Sie die Quick-Form wenn immer möglich, denn es gilt:

```
move.l    #1,d0     ; 6 Byte,     12 Zyklen
moveq     #1,d0     ; 2 Byte,     4 Zyklen
```

In ähnlichen Größenordnungen liegen die Zeit- und Raumgewinne bei Anwendung der anderen Q-Befehle.

Die von manchen Assemblern tolerierte Schreibweise »moveq.l« ist übrigens Nonsens. »moveq« addressiert immer lang.

6.1.2 Unterschied zwischen MOVEQ und den anderen Q-Befehlen

Beim Befehl »moveq #x,Ziel« ist die Konstante eine 8-Bit-Zahl im Zweier-Komplement, kann also Werte zwischen –128 und +127 annehmen. Diese Zahl wird dann automatisch auf 32 Bit erweitert, der Operand ist also immer lang. Deshalb ist auch die Schreibweise »moveq.l« Nonsens.

Hingegen darf bei den Befehlen »addq« und »subq» sehr wohl ein ».B«, ».W« oder ».L« angehängt werden. Dafür ist aber die Operandengröße auf Zahlen zwischen 1 und 8 begrenzt.

6.1.3 CLR vermeiden

Man liest zwar manchmal den tollen Tip, nie den CLR-Befehl zu gebrauchen, doch das ist nur die halbe Wahrheit. Es stimmt, daß »moveq #0,d0« um zwei Zyklen schneller ist als »clr.l d0«. Doch wenn Sie nur ein Byte oder Wort löschen dürfen, müssen Sie wohl oder übel »clr.b« oder »clr.w« nehmen, weil – wie oben geschildert – moveq wie »moveq.l« wirkt.

6.1.4 Direktadressierung vermeiden

```
move.l  #1,Var
```

klingt zwar viel kürzer als

```
moveq    #1,d0
move.l   d0,Var
```

doch der Einzeiler kostet 10 Byte und 28 Zyklen, während die zwei Zeilen zusammen nur 8 Byte und 24 Zyklen brauchen. Der Vorteil besteht auch noch bei der ARI-Adressierung (Adreßregister indirekt), wie bei »Var(a0)«.

6.1.5 32-Bit-Konstanten vermeiden

Man sollte immer versuchen, 32-Bit-Operanden in einem Datenregister zu halten, wie die folgenden Beispiele zeigen.

```
and.l   #1,d1     ; 6 Byte, 16 Zyklen
movq    #1,d0     ; zusammen
and.l   d0,d1     ; 4 Byte, 12 Zyklen
;-------------------------------------
or.l    #1,d1     ; 6 Byte, 16 Zyklen
movq    #1,d0     ; zusammen
or.l    d0,d1     ; 4 Byte, 12 Zyklen
;-------------------------------------
sub.l    #9,d1    ; 6 Byte, 16 Zyklen
movq    #9,d0     ; zusammen
sub.l   d0,d1     ; 4 Byte, 12 Zyklen
;-------------------------------------
cmp.l    #9,d1    ; 6 Byte, 14 Zyklen
movq    #9,d0     ; zusammen
cmp.l   d0,d1     ; 4 Byte, 12 Zyklen
;-------------------------------------
```

Natürlich wird die Methode noch effektiver, wenn Sie ein mit »moveq« geladenes Datenregister mehrfach weiterverwenden können.

6.1.6 Stack schneller abbauen

Der folgende Trick stammt zwar vom Atari-ST, aber dabei sehen Sie dann auch gleich, warum das Amiga-Betriebssystem besser ist. Beim Atari erfolgt die Parameterübergabe generell über den Stack, der Amiga nimmt bekanntlich den deutlich schnelleren Weg über Register. Doch auch beim Amiga kann es Ihnen passieren, in die Atari-Situation zu gelangen, nämlich wenn Sie von Assembler aus eine C-Routine aufrufen. In diesem Fall gilt das Motto, wer x Bytes auf den Stack gepackt hat, muß den Stack nach dem Call auch wieder aufräumen. Die Standard-Lösung dafür heißt

```
add.l x,sp
```

Bei Werten von x über 8 ist das OK, doch bis 8 sparen Sie 4 Byte und 8 Zyklen, wenn Sie dafür »addq« einsetzen. Doch darüber gibt es auch noch eine Lösung.

```
add.l     #16,sp      ; 6 Byte,  16 Zyklen
lea       16(sp),sp   ; 4 Byte,  8 Zyklen
```

6.1.7 Schneller multiplizieren

Daß man mit einer Zweierpotenz am besten mittels »lsl« multipliziert, wissen Sie schon, der MUL-Befehl kostet 70 Zyklen und »lsl« nur 12. Doch viel öfter muß man wohl mit 10 multiplizieren, weshalb Profis hierfür folgendes Unterprogramm in petto haben:

```
m10: move.l  d0,d1   ;Multiplikant kopieren
     lsl.l   #2,d1   ;mal 4
     add.l   d1,d0   ;
     add.l   d1,d0   ;
     rts
```

Die Routine braucht 4 Byte mehr als MUL, aber nur noch 32 anstatt 70 Zyklen. Das Prinzip ist folgendes:

Der Multiplikant wird in d0 übergeben und dann gleich nach d1 kopiert. Nehmen wir an, wir übergeben 30, dann gilt:

```
m10: move.l  d0,d1   ;d0 = 30, d1 = 30
     lsl.l   #2,d1   ;d1 = 30 * 4 = 120
     add.l   d1,d0   ;d0 = 30 + 120 = 150
     add.l   d0,d0   ;d0 = 150 + 150 = 300
```

Die Eselsbrücke für diese Methode lautet »2 mal 4 ist 8 plus 1 ist 9 plus 1 ist 10«.

6.1.8 Adreß- anstatt Datenregister

Es klingt zwar richtig ordentlich, daß Daten in die Datenregister und Adressen in die Adreßregister gehören, aber Sie müssen sich nicht daran halten, die meisten Amiga-Libraries tun das auch nicht. Nur Intuition ist da sehr eigen. Wie auch immer, wenn die Datenregister nicht reichen, nehmen Sie die A-Register. Der wesentliche Unterschied ist, daß bei Operationen mit Adreßregistern die Flags nicht aktualisiert werden. Einige Befehle, zum Beispiel das Schieben, laufen allerdings nur in Datenregistern. Keine Sorge: Der Assembler wird's schon anmeckern.

6.1.9 Worte anstatt langer Worte

Etwas widersinnig aber Tatsache ist, daß Wort-Operationen mit den langen Adreßregistern schneller sind als Langwort-Operationen (Bytes sind nicht zulässig). Ernsthaft: beim Zieloperanden gibt es keine Unterschiede, doch beim Quelloperanden sollten Sie die Wortlänge vorziehen, wenn immer das möglich ist.

6.1.10 Am besten A mit A

Die schnellste und kürzeste Methode ein Adreßregister mit null zu laden, ist die Form

```
sub.l   a0,a0   ; 2 Byte, 8 Zyklen
```

Dies folgt der Regel, daß A-A-Operationen immer schneller sind. Eine Konstante kann man so oder so in ein Adreßregister laden:

```
move.w  #4713,a0    ;  6 Byte, 12 Zyklen
lea     4713.w,a0   ;  4 Byte,  8 Zyklen
```

Die beim Amiga ja nicht zu vermeidende Wandlung eines BPTR (in a0) in einen APTR geht am schnellsten so:

```
add.l   a0,a0
add.l   a0,a0
```

6.1.11 Schneller schieben

Der Befehl LSL (als ein Beispiel) kann bekanntlich nur um 8 Bit mit einem Konstanten-Operanden schieben, darüber muß man den »Shift-Count« erst in ein Register laden. Doch da sind zwei LSL-Befehle schneller. Zum Beispiel kann man um 9 Bit mit diesen Befehlen schieben:

```
lsl.l   #8,d0
lsl.l   #1,d0
```

Der Code ist korrekt, aber nicht optimal. »lsl.l #1,d0« heißt auch »multipliziere mit 2«, doch hier kommt eine andere Regel zum Tragen, die da heißt, daß ein Computer schneller addieren als multiplizieren kann. Folglich schreibt man für »lsl.l #1,d0« besser »add.l d0,d0«.

Beim Schieben um 16 Bit muß man sich vor Augen führen, daß das rechte Wort auf den Platz des linken wandert. Das geschieht aber schneller mit »swap d0«, das beide Worte tauscht. Oft reicht das schon. Will man den zweiten Effekt des »lsl«, nämlich daß Null-Bits nachgeschoben werden, auch haben, muß man noch ein »clr.w d0« folgen lassen. Damit ist man aber immer noch schneller als bei den beiden anderen möglichen Lösungen.

Die folgende Tabelle bringt die schnellsten Lösungen für verschiedene »Shift-Counts«.

```
Bits zu      Schnellste
schieben     Lösung
-------------------------------
       1     add.l  d0,d0
x =  2-8     lsl.l  #x,d0

       9     lsl.l  #8,d0
             add.l  d0,d0

      10     lsl.l  #8,d0
             lsl.l  #2,d0

      15     lsl.l  #8,d0
             lsl.l  #7,d0

      16     swap   d0
             clr.w  d0

      24     swap   d0
             clr.w  d0
             lsl.l  #8,d0

      25     swap   d0
             clr.w  d0
             lsl.l  #8,d0
             add.l  d0,d0
```

6.2 Positionsunabhängig schreiben

Sie können durchaus positionsabhängigen Code schreiben, der DOS-Lader sorgt für das Relokieren, doch einen Haken hat die Sache. Dafür braucht AmigaDOS nämlich eine Relokatiertabelle, die a) Raum auf der Disk kostet und b) das Laden verlangsamt. Sie sollten deshalb nun nicht die größten Anstrengungen unternehmen, doch schon wenn die Tabelle kürzer wird, hilft das viel.

Die erste Maßnahme ist natürlich die PC-relative Adressierung, doch eines geht leider nicht:

```
move.w   #1,dahin(pc)   ;sorry
```

Ein pc-relatives Ziel ist leider nicht erlaubt. Abhilfe bringt diese Form:

```
lea      dahin(pc),a0
move.w   #1,(a0)
```

Häufiger wird man allerdings mit einem Basis-Register arbeiten, das auf eine der beim Amiga ja so zahlreichen Datenstrukturen zeigt, zum Beispiel so:

```
lea       NewWindow(pc), a0
move.w    #20,nw_LeftEdge(a0)
move.w    #20,nw_TopEdge(a0)
```

Auf diese Art können Sie eine ganze Window-Struktur laden oder modifizieren und alle Befehle sind positionsunabhängig.

Die Offsets »nw_LeftEdge« und »nw_TopEdge« sind in Include-Files – wie sie zum Beispiel mit dem DevPac geliefert werden – vordefinierte Konstanten. Haben Sie diese Files nicht, sollten Sie sie einmal aufbauen, und Sie sparen sich zukünftig das mühsame Abzählen.

6.3 MOVEM kann mehr

Mit einem einzigen MOVEM-Befehl kann man diverse Register retten, zum Beispiel so:

```
movem.w  d0-d3/a0-a3,-(sp)   ; retten
;ändern
movem.w (sp)+,d0-d3/a0-a3    ; und zurück
```

Doch MOVEM läßt sich auch für andere Zwecke einsetzen. Nehmen wir an, beim Offset o1 zu a0 stehen die vier Werte für ein Rechteck der Form »links, oben, Breite, Höhe« (x0, y0, b, h), und das wollen Sie in ein Rechteck der Form »links, oben, rechts, unten« (x0, y0, x1, y1) umrechnen. Diese vier Werte sollen ab dem Offset o2 abgelegt werden.

Dann heißt das: x0 und y0 müssen nur kopiert werden, x1 wird x0 + b und y1 wird y0 + h. Was halten Sie von dieser Lösung?

```
movem.w  o1(a0),d0-d3
add.w    d0,d2
add.w    d1,d3
movem.w  d0-d3,o2(a0)
```

Wetten, daß kein C-Compiler einen so effektiven Code erzeugt!

6.4 Richtig springen

Schon von der Programmlogik her ist es korrekt, folgenden Ablauf zu wählen:

```
teste auf Fehler
springe, wenn Fehler
normal hier weiter
```

Der 68000 ist dafür konzipiert, so daß immer bei Nichtnutzung des Sprungbefehls die wenigsten Zyklen verbraucht werden. Ab dem 68010 verdoppelt sich der Verlust sogar, weil diese Prozessoren über eine Pipeline verfügen. Das heißt, während ein Befehl ausgeführt wird, werden schon die Folgebefehle in »die Röhre« geladen. Springt dann das Programm woanders hin, wird das Pipelining unterbrochen, und das kostet Zeit.

Im obigen Beispiel ist die Sache noch recht einfach, denn sicherlich sind Fehler nicht der Normalfall. Ansonsten müssen Sie die Wahrscheinlichkeit abschätzen, so daß das Programm im häufigeren Fall geradeaus läuft und im selteneren Fall springt.

6.5 Der richtige Startup-Code

In C braucht man sich um das Problem »Startup-Code« an sich wenig Sorgen zu machen. Man linkt ein Modul hinzu, und das war's auch schon. Wenn Sie allerdings mit diesem Modul nicht ganz zufrieden sind, zum Beispiel, weil mehr Speicher verbraucht wird als nötig, dann müssen Sie doch tiefer in das Thema einsteigen. Der Startup-Code ist in Assembler geschrieben, beginnen wir also damit.

Das Problem: Workbench-Programme müssen im Gegensatz zu CLI-Programmen eine Zusatzbedingung erfüllen. Sie dürfen nicht einfach loslaufen, wann sie wollen, sondern sie brauchen sozusagen die Starterlaubnis der Workbench. Deshalb müssen solche Programme zu Beginn auf eine Message (den Startbefehl) warten. Wenn sie fertig sind, müssen sie das der Workbench melden, indem sie genau diese Message (die sie sich gut gemerkt haben) an die Workbench zurückschicken.

Wird dasselbe Programm vom CLI aus aufgerufen, entfällt natürlich diese Geschichte. Das heißt auch, daß unser Programm unterscheiden muß, von wo es aufgerufen wurde.

Der Trick hinter der ganzen Geschichte ist der sogenannte Startup-Code. Der Name ist nicht ganz korrekt, hat sich jedoch so eingebürgert. Zum Start-Code gehört nämlich immer auch ein Ende-Code. Um nun beides in ein einziges File packen zu können, das man als Include-File zuerst lädt oder als Modul mit dem Linker vor sein Programm setzt, kommt wieder ein kleiner Trick zum Tragen.

Normalerweise hieße die Folge:

```
Start-Code
Unser  Programmteil
End-Code
```

Praktisch gehen wir aber so vor:

```
Start-Code
jsr _main
End-code

_main       Unser Programmteil
rts
```

Sie wissen jetzt, warum in meinen Assembler-Listings das Label »_main« an den Anfang gesetzt wird. Auch dürfte jetzt klar sein, daß Assembler-Programme immer mit einem schlichten aber wichtigen »rts« enden müssen.

Doch nun zur Praxis: Abb. 6.1 zeigt das Listing des Startups.

Im »ROM Kernel Manual, Libraries and Devices« finden Sie ein recht langes Assembler-Listing, das den Startup-Code für C- Programme bildet. Dieses Listing habe ich etwas umgestaltet und drastisch gekürzt. Wenn Sie noch einige der dort aufgeführten Features, zum Beispiel Alerts für den wohl sehr wahrscheinlichen Fall (???), daß sich die DOS-Lib nicht öffnen läßt, einbauen wollen – OK, da ist die Quelle.

```
*  startup.i

*  Startup-Code fuer Assembler-Programme. Recht frei nach dem
*  Beispiel im ROM-Kernel-Manual Libraries and Devices,
* aber so geschrieben, daß als Include-File brauchbar und auf
* das wirklich notwendige reduziert (und ohne Bug).
* ------------------------------

   incdir     ":devpac/include/"

   include    "exec/exec_lib.i"
   include    "libraries/dosextens.i"

   movem.l    d0/a0,-(sp)      ;rette Kommandozeile
   clr.l      _WBenchMsg       ;sicherheitshalber
*  Teste, von wo wir gestartet wurden
*  ------------------
   sub.l      a1,a1        ;a1=0 = eigener Task
   CALLEXEC   FindTask     ;wo sind wir?
   move.l     d0,a4        ;Adresse retten

   tst.l      pr_CLI(a4)        ;Laufen wir unter WB?
   beq.s      fromWorkbench     ;wenn so
```

```
* Wir wurden vom CLI gestartet
* ----------------
  movem.l   (sp)+,d0/a0  ;Parms Kommandozeile holen
  bra       run          ;und starten

* Wir wurden von  Workbench gestartet
* --------------------
fromWorkbench
  lea       pr_MsgPort(a4),a0
  CALLEXEC  WaitPort           ;Warte auf Start-Message
  lea       pr_MsgPort(a4),a0 ;sie ist da
  CALLEXEC  GetMsg             ;hole sie
  move.l    d0,_WBenchMsg      ;immer Msg sichern!

  movem.l   (sp)+,d0/a0        ;bringe Stack i.O.
 run  bsr.s   _main            ;rufe unser Programm auf

  move.l    d0,-(sp)           ;rette seinen Return-Code

  tst.l     _WBenchMsg         ;gibt's eine WB-Message
  beq.s     _exit              ;nein: dann war's CLI

  CALLEXEC  Forbid             ;keine Unterbrechung jetzt
  move.l    _WBenchMsg(pc),a1 ;hole die Message
  CALLEXEC  ReplyMsg           ;und gib sie zurueck

_exit
  move.l    (sp)+,d0           ;hole Return-Code
  rts                          ;das war's

_WBenchMsg ds.l  1
  cnop      0,2
```

Abb. 6.1: Der Startup-Code

Die Exec-Funktion »FindTask« findet die Adresse einer Task-Kontroll-Struktur. Normalerweise übergibt man der Funktion die Adresse eines Strings mit dem Task-Namen im Register a1. Ist dieser Zeiger Null, erhält man die Adresse des eigenen Tasks. Nun muß ich leider gestehen, daß hier »Task« nicht korrekt ist; die Thematik genau abzuhandeln, würde aber den Rahmen dieses Buches sprengen. Deshalb ganz kurz: Wir laufen unter einem DOS-Prozeß. Ein Prozeß ist so etwas Ähnliches wie ein Task, nur höherwertiger. »FindTask« gibt deshalb die Adresse unseres PLB (Prozeß-Leit-Block) zurück. (Die Struktur steht in »include/libraries/dosextens.i«.)

Innerhalb dieser Struktur gibt es das Feld »pr_CLI«. Das soll heißen: »Pointer (Zeiger) auf den Command Line Interpreter«. Dieser Zeiger ist Null, wenn wir unter der Workbench laufen. Also geht es in diesem Fall zum Label »fromWorkbench«, und zu dem da folgenden Code sollten Sie wissen:

Im PLB beim Offset »pr_MsgPort« steht die Adresse, die Exec-Funktion »Waitport« sehen will. Dieser Aufruf läßt unseren Task warten, bis er an die Reihe kommt. Stellen Sie sich das so vor: Es gibt eine Liste aller laufenden Tasks. Exec sorgt dafür, daß einer nach dem anderen für eine gewisse Zeit an die Reihe kommt, denn praktisch kann ja immer nur ein Task laufen (wir haben nur einen 68000 im Amiga). Seinen Startbefehl erhält der Task über diesen Message-Port, genau: nur die Nachricht, daß eine Message da ist. Deshalb muß man mit »GetMsg« diese Nachricht aus dem Port lesen. Sie steht danach im Register d0. Da wir diese Nachricht noch brauchen, sichern wir sie in der Variablen »_WBenchMsg«.

Wenn unser Task unter der Workbench läuft, und er »geweckt« wurde, sind jetzt praktisch diese Zeilen interessant:

```
run
 bsr.s     _main          ;rufe unser Programm auf
 CALLEXEC  Forbid         ;keine Unterbrechung jetzt
 move.l    _WBenchMsg,a1  ;hole die Message
 CALLEXEC  ReplyMsg       ;und gib sie zurueck
```

Mit »bsr _main« wird nun endlich unser Programmteil aufgerufen. Danach erfolgen die Rückzugsgefechte, das, was ich in der Einleitung den Ende-Code genannt habe. Wir müssen uns bei der Workbench ordnungsgemäß abmelden, was an sich dadurch geschieht, daß wir mit »ReplyMsg« die ursprünglich beim Start erhaltene Message zurückgeben. Da auch andere Tasks theoretisch zur selben Zeit auf den PLB zugreifen können, könnte es sein, daß unsere Nachricht nicht ankommt oder schlimmer, das totale Chaos ausbricht. Hier wird deshalb Regel 5 aus Kapitel 5 angewandt.

Praxistip

Wenn Sie den Quelltext von Abb. 6.1 assemblieren, müßte das mit einer Ausnahme ohne Fehler über die Bühne gehen. Die Ausnahme ist das fehlende Label »_main«. Belassen Sie es aber ruhig beim Quelltext. Wegen der paar Zeilen lohnt sich das Linken nicht. Sie brauchen nämlich nichts weiter zu tun, als ein »include startup.i« vor Ihr Assembler-Programm zu setzen und dem ersten Befehl Ihres Programms das Label »_main« zu geben.

6.6 C-Funktionen im ROM nutzen

Da wir gerade beim Linken sind: Normalerweise bindet man Assembler-Routinen in C-Programme ein. Beim Amiga ist der umgekehrte Weg empfehlenswert, denn einige nützliche C-Funktionen befinden sich da im ROM. Ich habe zwar immer noch den dumpfen Verdacht, daß die C-Funktionen im ROM vergessener Debug-Code sind, aber sie sind offiziell dokumentiert und können deshalb mit Commodores Segen eingesetzt werden.

In Sachen Dokumentation gleich der erste Tip: Wenn man weiß, wie man die »ROM Kernel Reference Manuals« von Commodore lesen muß, entdeckt man interessante Dinge. Die Lesevorschrift: Alles, was groß und fett gedruckt ist, ist trivial; die interessanten Dinge stehen kleingedruckt im Anhang. So findet man auf Seite B 12 von »Exec« ein paar Seiten mit dem Titel »amiga.lib.doc«. Diese Seiten enthalten den Hinweis, daß unter anderem die C-Funktion printf() im ROM vorhanden ist.

Daher empfiehlt Commodore C-Programmierern, »amiga.lib« vor die Laufzeit-Bibliothek des C-Compilers zu linken, damit der Code kürzer wird.

Ich empfehle, die Routinen in Assembler zu nutzen, denn das macht dessen Code auch kürzer und spart so aufwendige Übungen wie die Umwandlung von Binärzahlen in hexadezimale oder dezimale Zeichenketten.

6.6.1 Das Prinzip

Die Routinen sind für den Aufruf von C her vorgesehen, erwarten also ihre Parameter auf dem Stack. Folglich muß man in Assembler seine Parameter auf den Stack bringen, dann die Funktion mit JSR aufrufen (BSR geht nur über 32 Kbyte) und schließlich den Stackpointer wieder auf den alten Wert stellen. Ansonsten muß man noch bedenken, daß C die Parameter einer Funktion von rechts nach links liest, das heißt, aus dem C-Aufruf

```
function (a, b, c);
```

wird in Assembler

```
pea c
pea b
pea a
jsr function
```

Um nun die – in diesem Beispiel – 3 Langworte (= 12 Byte) wieder vom Stack zu holen, ist dann noch der Befehl

```
lea 12(sp),sp
```

erforderlich. Mehr zum Thema »Stack aufräumen« finden Sie im Abschnitt 6.1.6. Als erstes Beispiel soll mit Abb. 6.2 die Anwendung der Funktion »printf()« gezeigt werden. Hier druckt sie einfach die Zahlen 111, 222 und 333, natürlich anständig formatiert.

```
            opt     l+      ;linkbarer Code
* ------------------------------
* Demo ROM-printf() in Assembler.
* Linken Sie diesen File als xxx.o mit
*
* blink AStartup.obj,xxx.o to xxx library lib:amiga.lib
*
* ------------------------------

          XREF.L    _printf         ; .L nur beim DevPac

          XDEF      _main
_main
          pea       333             ; Zahl 3
          pea       222             ; Zahl 2
          pea       111             ; Zahl 1
          pea       fmt             ; Format-String
          jsr       _printf
          lea       16(sp),sp       ; 4 Longs vom Stack
          rts
fmt       dc.b      'Z1 = %ld  Z2 = %ld  Z3 = %ld',10,0
```

Abb. 6.2: So einfach druckt man drei Zahlen formatiert

Nach den drei Daten muß hier noch die Adresse einer Format-Zeichenkette übergeben werden. Beachten Sie ein paar feine Unterschiede zur C-Schreibweise und zum Standard-C der Compiler-Bibliotheks-Funktionen. Das C-Äquivalent des obigen Beispiels hieße:

```
printf ("Z1 = %ld Z2 = %ld Z3 = %ld\n", 111,222,333);
```

In der Format-Zeichenkette wird nur das Escape-Zeichen »%« nebst Zubehör erkannt, der Rest gilt als Text. So ist zum Beispiel »\n« nicht möglich, kann aber einfach durch ein Byte mit dem dezimalen Wert 10 ersetzt werden. Die Zeichenkette kann beliebig zusammengesetzt werden, als Abschluß gilt das Null-Byte.

Das ROM-printf() kann nur ganze Zahlen, Zeichen und Zeichenketten formatieren. Wenn man die Routine von C aus aufruft, ist für Zahlen nur »%l« (= Langwort) erlaubt. Nur in Assembler können auch Worte (jedoch keine Bytes, da mit dem Stack gearbeitet wird) übergeben werden.

Es gibt – hier am Beispiel »%l« – folgende Varianten:

```
%lc  : Das nächste Langwort wird als Zeichen (8 Bit)
       ausgegeben.

%ld  : Ausgabe als Dezimalzahl

%lx  : Ausgabe als Hexadezimalzahl
```

```
%ls : Ausgabe als Zeichenkette (Übergeben wird die Adresse
      einer mit Null abgeschlossenen Zeichenkette).
```

Dem »%« können noch diese Zeichen folgen:

```
-   : (Minus): Innerhalb der Feldweite wird links justiert.
nn  : Eine Zahl für die Feldweite, Auffüllen mit Leerzeichen.
0   : Führende Nullen werden nicht durch Leerzeichen ersetzt.

.nn : Eine Zahl nach dem Punkt für das Maximum an Zeichen.
snn : Eine Zahl nach »%s« für die maximale Anzahl Zeichen,
      die als Zeichenkette ausgegeben werden sollen.
```

Das obige Programm wurde mit dem DevPac-Assembler geschrieben. Beim Amiga-Assembler muß die Opt-Direktive weggelassen und »XREF« anstatt »XREF.L« geschrieben werden. Die Funktion printf() verlangt einige Referenzen zur Ein-/Ausgabe, die im Startup-Modul definiert sind. Deshalb muß man »Astartup.obj« und natürlich auch »amiga.lib« beim Linker-Lauf miteinbinden. Vergessen Sie nicht, anstatt »lib:« ggf. Ihren Pfadnamen für »amiga.lib« einzusetzen.

6.6.2 Astartup.obj und Amiga.lib

»Amiga.lib« brauchen Sie wohl oder übel, doch die finden Sie bei vielen Assemblern und C-Compilern als Zugabe, manchmal gut versteckt. Mit Sicherheit – da habe ich Sie her – gibt es die »Amiga.lib« beim Amiga-Assembler und mit Lattice-C.

Auch die Datei »Astartup.obj« gehört zum Amiga- bzw. Metacomco-Assembler, doch daß ich die angezogen habe, war pure Faulheit. Denn das ausführbare Programm belegt rund 2,5 Kbyte, was zwar schon weniger als die 7,5 Kbyte des C-Pendants ist, doch das ist immer noch viel mehr als nötig. Der größte Teil geht für das Startup-Modul drauf. Deshalb zeigt das folgende Beispiel auch, wie man ohne dieses Modul auskommen kann. Sie können auch meine Kurzfassung des »Startup« nehmen (siehe 6.5) und müssen dann nur noch die beiden Referenzen »befriedigen«, die BLINK dann anmahnt.

Doch nun zum nächsten Beispiel in Abb. 6.3, das ohne »Astartup.obj« auskommt. Hier werden auch noch einige weitere C-Funktionen vorgeführt und – darauf kommt es an – gezeigt, wie man die Ein-/Ausgabe der C-Funktionen auf ein eigenes Konsol-Fenster umleitet.

Die DOS-Library muß aus zwei Gründen geöffnet werden. Zuerst brauchen die C-Funktionen »DOSBase« für die Verwendung der DOS-Funktionen Read() und Write(), zum zweiten braucht man die Open-Funktion des DOS, um das Konsol-Fenster zu öffnen.

6.6.3 Der Trick mit der Ausgabe-Umleitung

Der Trick an der ganzen Geschichte ist nun, daß das Handle des Konsol-Fensters in die Variablen »_stdin« und »_stdout« (definiert in Amiga-Lib) eingetragen wird. Das ist die ganze Ein-/

Ausgabe-Umleitung. Das Programm soll dann zehn Zufallszahlen in der Form »Zufallszahl n = zzz« formatiert ausgeben.

Die Funktion RangeRand(x) erzeugt Pseudo-Zufallszahlen im Bereich 1...x, wobei x eine Konstante im Bereich von 1...65535 sein muß. Nur um die Eingabe zu demonstrieren, wird dann mit »_getchar« auf eine Taste gewartet. Deren Code steht übrigens danach in »D0.B«.

```
      opt     l+        ; linkbarer Code
*-------------------------------
* Demo printf() und weitere in einem Console-Window
* Dieses File xxx.o können Sie ohne Startup.obj linken mit
*        blink xxx.o to xxx library lib:amiga.lib
* -------------------------------

     incdir   ":devpac/include/"
     include  exec/exec_lib.i
     include  libraries/dos_lib.i
     include  libraries/dos.i

     XREF.L   _printf
     XREF.L   _RangeRand
     XREF.L   _getchar

     XDEF     _DOSBase
     XDEF     _stdout
     XDEF     _stdin

     move.l   #dosname,a1          ; Name der DOS-Lib
     moveq    #0,d0                ; Version egal
     move.l   _SysBase,a6
     jsr      _LVOOpenLibrary(a6)
     move.l   d0,_DOSBase          ; Zeiger notieren
     beq      exit                 ; falls Fehler

     move.l   d0,a6                ; Basis DOS-Lib
     move.l   #con,d1              ; Con-Definition
     move.l   #MODE_OLDFILE,d2     ; CON: gibt es
     jsr      _LVOOpen(a6)
     tst.l    d0                   ; alles OK?
     beq      CloseDOS             ; wenn nicht
     move.l   d0,_stdin            ; Handles eintragen
     move.l   d0,_stdout
* 10 Zufallszahlen zwischen 1 und 999 ausgeben
* Anzeige formatiert als "Zufallszahl nn = zzz"
* -------------------------
     moveq    #1,d3                ; Zähler auf 1
loop
     pea      999                  ; Range vorgeben
     jsr      _RangeRand
     addq.l   #4,sp                ; Stack korrigieren
```

```
        move.w   d0,-(sp)              ; Ergebnis zur Ausgabe
        move.w   d3,-(sp)              ; Zähler auch
        pea      fmt(pc)               ; Format-String
        jsr      _printf
        addq.l   #8,sp                 ; Stack korrigieren

        addq.l   #1,d3                 ; Zähler += 1
        cmp      #10,d3                ; fertig?
        ble      loop
* Einen Text ausgeben
* ----------
        pea      msg                   ; Format-String
        jsr      _printf
        addq.l   #4,sp                 ; Stack zurück
* Auf Taste warten
* --------
        jsr      _getchar
* Consol-Window schließen
* ------------

        move.l   _DOSBase,a6
        move.l   _stdout,d1     ;Handle
        jsr      _LVOClose(a6)
* DOS-Lib schließen
* --------
CloseDOS
        move.l   a6,a1          ;DOS-Lib-Basis
        move.l   _SysBase,a6
        jsr      _LVOCloseLibrary(a6)
exit
        rts
* Datenbereich
* -------
fmt     dc.b     'Zufallszahl %2d = %3d',10,0
        cnop 0,4
msg     dc.b     'Tippen Sie Return...',0
        cnop 0,4
_stdin  ds.l  1
_stdout ds.l  1
_DOSBase ds.l 1
dosname dc.b     'dos.library',0
        cnop 0,4
con     dc.b     'CON:0/12/640/200/Console',0
```

Abb. 6.3: Ohne »Astartup« und mit Ausgabeumleitung

Das Beispiel von Abb. 6.4 dient nur der Einstimmung auf das nächste Problem. Nach der schon bekannten Methode soll nun der freie Hauptspeicher angezeigt werden. Dank der Exec-Funktion AvailMem() ist das eine Kleinigkeit.

```
            opt     l+
* Linken Sie dieses File als xxx.o mit
*
* blink AStartup.obj,xxx.o to xxx library lib:amiga.lib
*
            incdir  ":devpac/include/"
            include exec/exec_lib.i
            include exec/memory.i

            XREF.L  _printf
            XDEF    _main
_main
            moveq   #MEMF_PUBLIC,d1
            CALLEXEC AvailMem           ; free RAM -> d0
            move.l  d0,-(sp)            ; auf den Stack
            pea     fmt                 ; Format-String
            jsr     _printf             ; und drucken
            addq.l  #8,sp
            rts
fmt         dc.b    'Freier RAM = %ld Bytes',10,0
```

Abb. 6.4: So einfach kann man den freien RAM anzeigen

6.6.4 »printf()« unter Intuition

So ein kleines CLI-Programm ist zwar ganz nett, aber nicht Amiga-like. Deshalb müssen wir das Programm aufbohren. Der freie Speicherplatz soll jetzt ständig in einem kleinen Fenster angezeigt werden, also auch, wenn andere Tasks laufen. Zusatzforderungen: Das Programm soll auch per Maus-Doppelklick von der Workbench aus gestartet werden können (wenn es eine Info-Datei hat), und das Fenster soll ein Close-Gadget haben, mit dem man das Programm beenden (und das Window verschwinden lassen) kann. Damit bleibt nur eine Lösung: Das Programm muß unter Intuition laufen, »printf()« soll aber genutzt werden. Wie das geht, zeigt das Listing von Abb. 6.5. Bevor Sie die Länge monieren: Der Code belegt 560 Byte, die File-Größe auf der Diskette ist 788 Byte. Nun schreibe mal einer dieses Programm kürzer in C!

```
      opt     l+      ;linkbarer Code
*-------------------------------
* Demo _sprintf unter Intuition.
* Laufende Anzeige von Free Mem in einem Window.
* Programm abgewandelt aus Amiga-Assembler-Buch M&T 90525
* (c) 1987/1990 Peter Wollschlaeger
*
* Linken Sie diesen File als xxx.o mit
*
* blink xxx.o to xxx library lib:amiga.lib
*
* -------------------------------
```

```
      incdir  ":devpac/include/"
      include exec/exec_lib.i
      include exec/memory.i
      include libraries/dos.i
      include libraries/dos_lib.i
      include intuition/intuition.i
      include intuition/intuition_lib.i
      include graphics/graphics_lib.i

GRAFIC macro
       move.l   windowptr,a1     ;Adresse der Window-Struktur
       move.l   wd_RPort(a1),a1  ;Von da auf Rast Port
       CALLGRAF \1               ;Grafik-Funktion
       endm

       XREF.L  _sprintf
       XDEF    _main

_main

* DOS-Library öffnen
* ------------
       lea      dosname,a1
       moveq    #0,d0
       CALLEXEC OpenLibrary
       tst.l    d0
       beq      abbruch
       move.l   d0,_DOSBase       ;Basis-Zeiger sichern

* Intuition-Library öffnen
* ---------------
       lea      intname,a1
       moveq    #0,d0
       CALLEXEC OpenLibrary
       tst.l    d0
       beq      closedos
       move.l   d0,_IntuitionBase  ;Basis-Zeiger sichern

* Graphics Library öffnen
* -------------
      lea       grafname,a1
      moveq     #0,d0
      CALLEXEC  OpenLibrary
      tst.l     d0
      beq       closeint
      move.l    d0,_GfxBase        ;Basis-Zeiger sichern

* Window öffnen
* --------
     lea       windowdef,a0        ; Zeiger auf Window-
                                   ; Struktur
```

```
     CALLINT   OpenWindow          ; öffne Window
     tst.l     d0                  ; ging was schief?
     beq       closegraf           ; wenn ja
     move.l    d0,windowptr        ; Window-Zeiger sichern
     moveq     #1,d0
     GRAFIC    SetAPen             ; Zeichenfarbe weiß

* Hauptschleife
*--------
loop
     moveq     #MEMF_PUBLIC,d1     ; freien RAM
     CALLEXEC  AvailMem            ; -> d0 einlesen
     move.l    d0,-(sp)            ; Free Mem auf Stack
     pea       fmt                 ; Format-String dito
     pea       buffer              ; Ziel auch
     jsr       _sprintf
     lea       12(sp),sp           ; 3 Longs vom Stack

     moveq     #60,d0              ; X-Position für Text
     moveq     #19,d1              ; Y
     GRAFIC    Move                ; Move TO X,Y
     lea       buffer,a0           ; Text-Adresse
     moveq     #12,d0              ; Länge
     GRAFIC    Text                ; Text zeichnen

     move.l    windowptr,a0        ; Von unserem Window
     move.l    wd_UserPort(a0),a0  ; den Empfangs-Port
     CALLEXEC  GetMsg              ; testen
     tst.l     d0                  ; Haben wir Post?
     bne fini                      ; kann nur CLOSEWINDOW
                                   ; sein

     move.l    #25,d1              ; 25/50 = 1/2 Sekunde
     CALLDOS   Delay               ; warten
     bra loop                      ; und von vorn
fini
     move.l    d0,a1               ; Message ist in d0
     CALLEXEC  ReplyMsg            ; antworten

closewindow
     move.l  windowptr,a0          ; Fenster zu
     CALLINT CloseWindow

closegraf
     move.l    _GfxBase,a1         ; Die Libs schließen
     CALLEXEC  CloseLibrary

closeint
    move.l    _IntuitionBase,a1
    CALLEXEC  CloseLibrary
```

```
closedos
    move.l    _DOSBase,a1
    CALLEXEC  CloseLibrary

abbruch
    moveq     #0,d0                ; keinen Fehler melden
    rts

* Datenbereich
* ---------

W_Gadgets equ   WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras  equ   SMART_REFRESH!ACTIVATE

W_Title   dc.b   ' Freier RAM z.Z. ',0
          cnop   0,4

windowdef
          dc.w   200,20              ; links, oben
          dc.w   220,24              ; Breite, Hoehe
          dc.b    -1,-1              ; Pens des Screen
          dc.l    CLOSEWINDOW        ; einziges IDCMP-Flag
          dc.l    W_Gadgets!W_Extras ; Window Flags
          dc.l    0                  ; keine User-Gadgets
          dc.l    0                  ; keine User-Checkmark
          dc.l    W_Title            ; Titel des Window
          dc.l    0                  ; kein eigener Screen
          dc.l    0                  ; keine Super-Bitmap
          dc.w    100,20             ; Min Größe
          dc.w    640,200            ; Max
          dc.w    WBENCHSCREEN       ; useWorkbench Screen

intname   INTNAME                    ; Namen der Libs via
grafname  GRAFNAME                   ; Makros erzeugen
dosname   DOSNAME
          cnop    0,4
fmt       dc.b    '%ld Bytes',0      ; Format-String
          cnop    0,4
buffer    ds.b    40                 ;muß größer sein!
          cnop    0,4

_IntuitionBase  ds.l    1
_GfxBase        ds.l    1
_DOSBase        ds.l    1
windowptr       ds.l    1
```

Abb. 6.5: Es geht auch unter Intuition

Die Lösung funktioniert nach diesem Prinzip: anstatt »printf()« wird »sprintf()« eingesetzt. Der einzige Unterschied: Die Ausgabe erfolgt in einen Puffer, dessen Adresse zusätzlich zu übergeben

ist. Diesen Puffer kann man dann mit der Grafik-Funktion »Text()« ausgeben, wie hier geschehen, oder »IntuiText« einsetzen, wenn man noch interessante Texteffekte erzielen will.

Für Intuition-Kenner dürfte das Programm leicht verständlich sein, daher nur zwei Anmerkungen: Es sieht so aus, als ob das Programm nach der unfeinen Polling-Methode arbeitet. Das ist nicht der Fall. Durch den Aufruf der DOS-Funktion »Delay()« wird es für jeweils eine halbe Sekunde »suspendiert«. Den Puffer für »sprintf()« sollten Sie immer etwas größer wählen als die Byte-Zählerei ergibt (hier 40 anstatt 8), es sei denn, Sie sind ein Fan von Guru-Meldungen.

6.7 Die Kommando-Zeile nutzen

Alles, was Sie beim Aufruf nach den Programm-Namen und einem Blank eingeben, ist die Kommandozeile. Es ist dann Sache des Programms, diese Kommandozeile auszuwerten. Dazu ein paar Tips:

Direkt nach dem Start steht die Adresse des Beginns der Kommandozeile in Register A0, ihre Länge steht in D0. Da beide Register »Scratch« sind, muß das Programm zuerst die Kommandozeile sichern, typisch mit

```
movem.l  a0/d0,-(sp)
```

Der Parser (ein Stück Programm, das die Kommandozeile auswertet) wird dann mit

```
movem.l (sp)+,a0/d0
```

die beiden Register wieder vom Stack holen. Typisch werden Parser so geschrieben, daß Sie so lange »parsen« bis ein Null-Byte gefunden wird (das »move« setzt dann gleich das Z-Flag). Aber auch weil der Text mit einem LF abgeschlossen ist, muß man etwas tun. Alle DOS-Funktionen wollen nämlich C-Strings sehen, weshalb man das LF durch ein 0-Byte ersetzen muß. Typisch löst man das als:

```
move.b #0,-1(a0,d0)
```

oder

```
clr.b -1(a0,d0)
```

6.8 CLI-Befehle in Assembler sparen viel

CLI-Befehle und sonstige externe Programme kann man sehr leicht mittels der Execute-Funktion aufrufen. Gerade in Assembler kann das eine Menge Arbeit sparen. Abb. 6.6 demonstriert das Prinzip am Beispiel von »dir«.

```
* Directory mittels Execute anzeigen

    _SysBase          equ    4
    _LVOOpenLibrary   equ    -552
    _LVOCloseLibrary  equ    -414
    _LVOOutput        equ    -60
    _LVOWrite         equ    -48
    _LVORead          equ    -42
    _LVOInput         equ    -54
    _LVOExecute       equ    -222

*DOS/Lib oeffnen:
_main
    move.l   #dosname,a1          ;Name der DOS-Lib
    moveq    #0,d0                ;Version egal
    move.l   _SysBase,a6          ;Basis Exec
    jsr      _LVOOpenLibrary(a6)  ;DOS-Lib oeffnen
    tst.l    d0                    :Fehler?
    beq      fini                  ;wenn Fehler, Ende
    move.l   d0,a6                 ;Zeiger merken

*Programm aufrufen:
    move.l   #string,d1            ;Adresse CLI-Befehl
    clr.l    d2                    ;kein Input
    clr.l    d3                    ;Output im CLI-Window
    jsr      _LVOExecute(a6)       ;CLI aufrufen

    move.l   a6,a1                 ;DOS-Lib schliessen
    move.l   _SysBase,a6
    jsr      _LVOCloseLibrary(a6)
fini
    rts

* Datenbereich

dosname   dc.b    'dos.library',0
          cnop    0,2
string    dc.b    'dir',0
```

Abb. 6.6: Aufruf von CLI-Befehlen

Die Funktion _LVOExecute verlangt als Parameter zuerst die Adresse (in d1), ab der der Befehl im Klartext steht.

In d2 und d3 müssen Nullen stehen oder File-Handles, wie man sie von _LVOOpen erhält. Steht in d2 die Handle einer Eingabedatei, wird diese Datei gelesen und ihr Inhalt als Befehlssequenz interpretiert.

Nun kommt der Knüller: Zuerst wird natürlich der mit d1 bezeichnete Befehl ausgeführt. Da muß aber keiner stehen, Sie können auch d1 nullen. Das heißt, daß nun nur die Kommandosequenz in dem mit d2 bezeichneten File ausgeführt wird. Wissen Sie jetzt, was der EXECUTE-Befehl des CLI macht? Steht in d3 eine Null, erfolgt die Ausgabe im aktuellen CLI-Fenster, ansonsten im File, dessen Handle hier eingetragen ist.

6.9 Verwenden Sie Makros

Makro ist die Kurzform von Makro-Befehl. Makro selbst heißt so viel wie groß. Prinzipiell ist ein Makro nur eine Zusammenfassung einer Gruppe von Einzelbefehlen, wie wir sie bisher kannten, unter einem neuen Namen. Man kann die Einzelbefehle auch Mikro-Befehle nennen. Leider ist die Makro-Sprache nicht genormt. Jeder Assembler hat da seine eigene Syntax, die des eben gezeigten GST-Assemblers ist sogar ziemlich ausgefallen. Ich bringe deshalb alle folgenden Beispiele in der Makrosprache der Assembler von HiSoft (DevPac und Metacomco (stimmen überein), die den »Makro-Dialekten« der meisten Assembler am ehesten entsprechen.

Hier ein Beispiel:

```
CALLEXEC macro
         move.l  _SysBase,a6
         jsr     _LVO\1(a6)
         endm
```

Dieses Makro realisiert die Funktion CALLEXEC (Funktion der Exec-Library aufrufen). Jedes Makro hat einen Namen, der im Label-Feld stehen muß, gefolgt von dem Schlüsselwort »macro«. Ein Makro endet mit dem Schlüsselwort »endm«. Zwischen »macro« und »endm« kann eine beliebige Anzahl von Befehlen stehen. Ist das Makro einmal definiert, kann es beliebig oft mit seinem Namen aufgerufen werden. Innerhalb eines Makros dürfen auch die Namen anderer, dann schon vorher definierter Makros, stehen. Ob und wie tief Makros so geschachtelt werden dürfen, lesen sie aber besser in Ihrem Handbuch nach.

Abb. 6.7 bringt zwei Makros, so wie Sie sie einfach zu Beginn eines Programms tippen können.

```
CALLLIB  MACRO
         JSR \1(A6)
         ENDM

LINKLIB  MACRO
         MOVE.L \2,A6
         CALLLIB \1
         ENDM
```

Abb. 6.7: Zwei Makros, die man immer braucht

Das Makro »CALLLIB« wie »Call Library« beinhaltet das berühmte »jsr offset(a6)«. Wichtig ist hier, daß wir dem Makro einen Parameter übergeben müssen, nämlich den einzusetzenden String. Für solche Parameter haben Makros Variablen. Meistens üblich dafür sind Ziffern mit einem Schrägstrich oder (bei SEKA) einem Fragezeichen davor. Bei Metacomco und dem neusten Devpac sind auch noch die Buchstaben A bis Z erlaubt. Beachten Sie bitte, daß der Schrägstrich bei Metacomco und HiSoft ein »Backslash« (nach links gekippter Strich) sein muß.

Beim zweiten Makro haben Sie sicherlich schon erkannt, daß ein Makro ein anderes aufrufen darf. Dieses muß aber vorher definiert sein! Doch nun zur Praxis.

Das Programm soll lediglich das bekannte »Hallo Welt« ausgeben. Abb. 6.8 zeigt die Lösung.

```
_SysBase          equ    4     ;Basis von Exec
_LVOOpenLibrary   equ -552     ;Library oeffnen
_LVOCloseLibrary  equ -414     ;Library schliessen
_LVOOutput        equ  -60     ;DOS: Output-Handle holen
_LVOWrite         equ  -48     ; Ausgabe

_main
     move.l   #dosname,a1            ;Name der DOS-Lib
     moveq    #0,d0                  ;Version egal
     LINKLIB  OpenLibrary,_SysBase   ;DOS-Lib öffnen
     tst.l    d0                     ;Fehler?
     beq      fini                   ;wenn Fehler, Ende
     move.l   d0,_DOSBase            ;Zeiger merken
     LINKLIB   Output,_DOSBase       ;Hole Output-Handle
     print     d0,#string,20 ;       ;Text ausgeben

     move.l    _DOSBase,a1           ;Basis der Lib
     LINKLIB   CloseLibrary,_SysBase ;Funktion "Schließen"
fini
    rts                              ;Return zum CLI

_DOSBase    dc.l    0
dosname     dc.b    'dos.library',0
            cnop    0,2
string      dc.b 'Hallo Welt!',10
            cnop 0,2
```

Abb. 6.8: Ein Programm mit Makros

Das sieht doch richtig gut aus. Was ist nun der Haken an der Sache? Es fehlen die Makros, die ich Ihnen mit Abb. 6.9 vorstellen möchte.

```
LINKLIB    MACRO
    IFNE NARG-2
    FAIL — Makro LINKLIB: Nicht 2 Argumente —
    ENDC
```

```
    MOVE.L A6,-(SP)
    MOVE.L \2,A6
    JSR    _LVO\1(A6)
    MOVE.L (SP)+,A6
    ENDM

print MACRO
    IFNE NARG-3
    FAIL -- Makro print: Nicht 3 Argumente --
    ENDC
    MOVE.L    \1,D1            ;Ausgabe-Handle
    MOVE.L    \2,D2            ;Address Text
    MOVEQ    #\3,D3            ;Laenge Text
    LINKLIB    Write,_DOSBase  ;Funktion "Schreiben"
    ENDM
```

Abb. 6.9: Die Makros zu Abb. 6.8

Das Makro LINKLIB finden Sie in ähnlicher Form unter vielen anderen in den Include-Files von Metacomco und Hisoft.

Die Zeilen

```
IFNE NARG-2
FAIL -- Makro LINKLIB: Nicht 2 Argumente --
ENDC
```

können Sie prinzipiell auch weglassen. Da Sie aber in den Include-Files häufig anzutreffen sind (schauen Sie mal rein, lohnt sich), sollte das aber doch erklärt werden. NARG ist eine Assembler-Variable und heißt »Number Arguments«. Diese Varibale ist nur innerhalb eines Makros gültig (sonst Null) und hält die Anzahl der Parameter, mit denen das Makro aufgerufen wurde.

6.10 Menüs mit Makros

Besonders effektiv ist der Einsatz von Makros für die doch recht komplexen Datenstrukturen des Amiga. Hier soll das am Beispiel von Menüs demonstriert werden. Abb. 6.10 zeigt nur zwei Menü-Titel und zwei Items, von denen eines ein Sub-Item hat. Das gibt dann schon diese Liste, in C ist die auch nicht kürzer. Nun stellen Sie sich ein mittelprächtiges Programm vor, zählen einmal die Titel, Items und Sub-Items und »schupdiwups« tippen Sie 100 solcher Strukturen, und jeder Fehler führt zum Absturz.

```
menu1
    dc.l    menu2                ;^Nachfolger
    dc.w    10,0                 ;Position x,y
    dc.w    120,10               ;breit, hoch
    dc.w    MENUENABLED          ;Flags
```

```
    dc.l    MName1                  ;^Titel
    dc.l    item11                  ;^Item-Liste
    dc.w    0,0,0,0                 ;System-Use
MName1      dc.b    ' Projekt ',0   ;Titel 1
            cnop    0,2

menu2
    dc.l    0                       ;^Nachfolger
    dc.w    150,0                   ;x,y
    dc.w    150,10                  ;breit, hoch
    dc.w    MENUENABLED             ;Flags
    dc.l    MName2                  ;^Titel
    dc.l    item21                  ;^Item-Liste
    dc.w    0,0,0,0                 ;System-Use
MName2    dc.b    ' Edit ',0        ;Titel 2
        cnop    0,2

item11                              ;Item von Menu 1
    dc.l    item12                  ;^next Item
    dc.w    -5,0                    ;x,y
    dc.w    120,11                  ;breit, hoch
    dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP
    dc.l    0                       ;keine Excludes
    dc.l    text11                  ;^Text
    dc.l    0                       ;Select Fill
    dc.b    0                       ;Kein Cmd-Key
    dc.b    0                       ;Dummy
    dc.l    0                       ;Kein Sub-Item
    dc.w    0                       ;next Select

item12
    dc.l    0                       ;^next Item
    dc.w    -5,22                   ;x,y
    dc.w    120,11                  ;breit, hoch
    dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP
    dc.l    0                       ;keine Excludes
    dc.l    text12                  ;^Text
    dc.l    0                       ;Select Fill
    dc.b    0                       ;keinCmd-Key
    dc.b    0                       ;Dummy
    dc.l    sub12_1                 ;Sub-Item
    dc.w    0                       ;Keine next Select
sub12_1
    dc.l    0                       ;^next Item
    dc.w    100,10                  ;x,y
    dc.w    150,20                  ;breit, hoch
    dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP!COMMSEQ
    dc.l    0                       ;keine Excludes
    dc.l    text12_1                ;^Text
    dc.l    0                       ;Select Fill
    dc.b    'x'                     ;Cmd-Key
    dc.b    0                       ;Dummy
    dc.l    0                       ;Kein Sub-Item
    dc.w    0                       ;next Select
```

Abb. 6.10: Menüs in Assembler, Lösung 1

Das Listing in Abb. 6.10 ist noch nicht einmal vollständig, es fehlen noch die »dc.b« für die Texte. Diese Texte hat die Lösung in Abb. 6.11 schon, ein paar mehr Menüs führt es auch noch auf, und dennoch ist es deutlich kürzer. Statt im Mittel 10 Zeilen je Menüpunkt, ist es jetzt nur noch eine.

```
menu1    MENU    menu2,10,0,120,10,MName1,item11,Projekt
menu2    MENU    0,150,0,150,10,MName2,item21,Edit

item11   ITEM    item12,-5,0,120,11,text11,0,Lösche
item12   ITEM    0,-5,22,120,11,text12,sub12_1,
sub12_1  ITEM    0,100,10,150,20,text12_1,0,Wirklich?
item21   ITEM    item22,-5,0,120,11,text21,0,Kopieren
item22   ITEM    0,-5,22,120,11,text22,0,Einsetzen
```

Abb. 6.11: So einfach wird die Sache mit Hilfe von Makros

Nun wollen Sie natürlich wissen, wie die Makros aussehen, OK, Abb. 6.12 bringt die Auflösung.

```
MENU  macro
      dc.l    \1              ;^Nachfolger
      dc.w    \2,\3           ;x,y
      dc.w    \4,\5           ;breit, hoch
      dc.w    MENUENABLED     ;Flags
      dc.l    \6              ;^Titel
      dc.l    \7              ;^Item-Liste
      dc.w    0,0,0,0         ;System-Use
\6    dc.b    ' \8 ',0        ;Text-String
      cnop    0,2
      endm

ITEM  macro
      dc.l    \1               ;^next Item
      dc.w    \2,\3            ;x,y
      dc.w    \4,\5            ;breit, hoch
      dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP
      dc.l    0                ;keine Excludes
      dc.l    \6               ;^Text
      dc.l    0                ;Select Fill
      dc.b    0                ;Cmd-Key
      dc.b    0                ;Dummy
      dc.l    \7               ;Sub-Item
      dc.w    0                ;next Select
\6    dc.b    0,1,RP_JAM1,0 ;Text-Struktur
      dc.w    8,2
      dc.l    0,ttt\6,0
ttt\6 dc.b    ' \8 ',0         ;Text selbst
      cnop    0,2
      endm
```

Abb. 6.12: Die Makros zur rationellen Erstellung von Menüs

Etwas fehlt noch. Im Makro ITEM ist das COMMSEQ-Flag nicht gesetzt, »Cmd-Key« ist auch 0. Das Programm soll aber auf »Amiga-x « reagieren können. Ich hätte nun natürlich deshalb »sub12_1« anstatt mit dem Makro einfach wie in Abb. 6.10 diskret hinschreiben können, aber hier sollen ja Tricks verraten werden, zum Beispiel dieser: Die Struktur wird ohne das Feature programmiert und anschließend modifiziert. Dazu werden kurz vor dem Aufruf von MenuStrip() diese beiden Zeilen eingebaut:

```
or.w   #COMMSEQ,sub12_1+12   ;Flag hinzu
move.b #'x',sub12_1+26       ;Taste eintragen
```

Jetzt wollen Sie wahrscheinlich noch wissen, wie man mit Menüs in Assembler umgeht. OK. Sie haben die Intuition- und die Graphics-Library geöffnet und machen jetzt das Window auf, das diese Menüs haben soll. An dieser Stelle steigt Abb. 6.13 ein.

```
* Window öffnen
* ---------
    lea       windowdef,a0     ;zeige auf Window-Struktur
    CALLINT   OpenWindow       ;öffne Window
    tst.l     d0               ;ging was schief?
    beq       closegraf        ;wenn ja
    move.l    d0,windowptr     ;Window-Zeiger sichern
* Menu installieren
* ----------
    move.l    d0,a0            ;^Window
    lea       menu1,a1         ;^Menu
    CALLINT   SetMenuStrip     ;Menü installieren
* Auf Event warten, dann auswerten
* ------------------
event
    move.l    windowptr,a0         ;zeige auf Window-Struktur
    move.l    wd_UserPort(a0),a0   ;nun auf Message-Port
    move.l    a0,a5                ;rette Port-Adresse
    move.b    MP_SIGBIT(a0),d1     ;Signal Bit holen
    moveq     #0,d0                ;Nummer in
    bset      d1,d0                ;Maske wandeln
    CALLEXEC  Wait                 ;Schlaf gut!

    move.l    a5,a0                ;hole Port-Adresse
    CALLEXEC  GetMsg               ;hole Message
    move.l    d0,a1                ;muß nach a1
    move.l    im_Class(a1),d4      ;Msg-Typ
    move.w    im_Code(a1),d5       ;Untergruppe
    CALLEXEC  ReplyMsg             ;quittiere Msg in a1

    cmpi.l    #CLOSEWINDOW,d4      ;Window Closed?
    beq       closewindow          ;wenn so
```

```
    cmpi.l    #MENUPICK,d4        ;Menu angewählt?
    bne       event
do_menu
    cmpi      #MENUNULL,d5        ;Item selektiert?
    beq       event               ;wenn nicht
*Hier folgt die Auswertung des Menu-Codes und die
* Verzweigung zu den Routinen. Für die Abfrage gilt:
* Bit 0...4   : Menü
* Bit 5...1o  : Item
* Bit 11...15 : Sub-Item
* Man kann direkt abfragen, z.B. so:
    cmpi      #$20,d5             ;Exit?
    beq       closewindow
* oder so
    move      d5,d0
    and       #$1F,d0
    cmpi      #1, d0
    beq       menu_1
    cmpi      #2, d0
    beq       menu_2
    bra       event
menu_1
     bsr   get_item   ;Code 5 Bit -> recht, mit 3F unden
```

Abb. 6.13: Der Umgang mit Menüs in Assembler

Ich persönlich ziehe noch einen anderen Trick für die Menü-Abfrage vor. Ich baue zuerst anstatt der Abfrage und der Verzweigung eine kleine Routine ein, die den Menü-Code in Hex anzeigt. Dann notiere ich mir für alle Items und Sub-Items die Zahlen und gehe dann nur noch nach der Methode »cmp...beq« vor. Jede Routine endet dann mit einem »bra event«. Damit kann man sehr schön das Ganze in viele kleine Jobs zerlegen, die alle einzeln ausgetestet werden können. Mehr dazu im Tip 6.12.

6.11 I/O mit AmigaDOS in Assembler

Sicherlich die einfachste Methode für die Ein- und Ausgabe von Texten ist die Anwendung von C-Funktionen, wie printf() oder scanf(). Wie man diese C-Funktionen, die sich auch im ROM befinden, in Assembler einsetzt, zeigt das Kapitel 6.6. Doch leider sind wir damit auf das aktuelle CLI-Fenster angewiesen, eine Methode, die wir Window-Freaks inzwischen gar nicht mehr so mögen.

Eine recht bequeme Lösung mit eigenem Window bietet jedoch AmigaDOS. Das kleine Programm von Abb. 6.14 listet ein Textfile in einem Window. Das hat einen eigenen Titel, Drag-Gadget und Size-Gadgets, und das alles mit diesen wenigen Zeilen. Der Name der zu

listenden Datei wird hier beim Aufruf übergeben (Kommandozeile). Das Listing zeigt aber schon, wie Sie im Programm einen Text auch über die Tastatur einlesen lassen können.

Das I/O läuft im DOS über die Funktionen Read() und Write(). Beide verlangen als Parameter

ein Handle in D1,

die Adresse des Puffers mit den Daten bzw. für die Daten in D2

und die Anzahl der zu übertragenden Bytes in D3.

Beide Funktionen geben die tatsächlich geschriebene bzw. gelesene Anzahl Bytes zurück, eine negative Zahl bedeutet Fehler.

So weit so gut, nur was ist ein Handle (zu deutsch Griff)? Nun, das ist vereinfacht ausgedrückt die File-Nummer, die man von Open() erhält. Sie können mit Open() ein Disketten-File öffnen. Die Open-Funktion erwartet den File-Namen und die Zugriffsart. MODE_OLFILE öffnet ein existierendes File, MODE_NEWFILE würde ein neues anlegen. Das Besondere daran ist, daß auch die Console (Tastatur und Bildschirm) in diesem Sinne ein File ist. Da Sie (wie ich hoffe) eine Console haben, ist der Modus MODE_OLDFILE. Nur hier muß man auch noch einige Parameter mehr übergeben, und das sind (durch Schrägstriche getrennt) die Parameter eines Windows. Mit der Anweisung

```
ziel  dc.b 'CON:0/12/640/200/Console',0
```

wird ein Window mit der linken oberen Ecke 0/12, der Breite 640 und der Höhe 200 geöffnet. Das Window erhält den Titel »Console«. Als Ergebnis erhalten wir ein Handle, und damit können wir auf das Window mit Write() und Read() zugreifen.

Das Window hat zwar eine nicht zu übersehende Ähnlichkeit mit einem CLI-Window, das braucht Sie aber nicht daran zu hindern, das Programm von der Workbench aus zu starten (siehe Kapitel 6.5).

Schön einfach, und da plagen wir uns noch mit den aufwendigen Intuition-Windows? Nun, die Sache hat einen Haken, das I/O ist auf Texte (genauer: jede Art von Bytes) beschränkt.

```
    incdir    ":devpac/include/"
   include exec/exec_lib.i
   include  libraries/dos_lib.i
   include libraries/dos.i

_SysBase equ   4

* Makros zum File-Handling
* ------------
```

```
  OPEN    macro
  move.l  \1,d1     ;Name
  move.l  \2,d2     ;Mode
  jsr     _LVOOpen(a6)
  endm

  READ  macro
  move.l  \1,d1     ;Handle
  move.l  \2,d2     ;Puffer-Adr
  move.l  \3,d3     ;Max.
  jsr  _LVORead(a6)
  endm

WRITE  macro
  move.l  \1,d1     ;Handle
  move.l  \2,d2     ;Puffer-Adr
  move.l  \3,d3     ;Anzahl
  jsr  _LVOWrite(a6)
  endm

 CLOSE  macro
  move.l  \1,d1     ;Handle
  jsr  _LVOClose(a6)
  endm

_main
  movem.l a0/d0,-(sp)     ;Kommandozeile retten

* DOS-Lib öffnen:
*--------
  move.l  #dosname,a1     ;Name der DOS-Lib
  moveq   #0,d0           ;Version egal
  move.l  _SysBase,a6     ;Basis Exec
  jsr     _LVOOpenLibrary(a6)  ;DOS-Lib öffnen
  tst.l   d0              ;Fehler?
  beq     fini            ;wenn Fehler, Ende
  move.l  d0,a6           ;Zeiger merken

* Kommandozeile holen
* -----------
  movem.l (sp)+,a0/d0     ;hole Kommandozeile
  move.b  #0,-1(a0,d0.l) ;terminiere mit 0

* Beide Files öffnen
* ----------
  OPEN    a0,#MODE_OLDFILE
  move.l  d0,d4           ;Handle
  tst.l   d0              ;ging was schief?
  bne     weiter          ;wenn nicht
  jsr     _LVOOutput(a6) ;Hole Output-Handle
```

```
  WRITE   d0,#msg1,#len1 ;und melde Fehler
  bra     cl_lib         ;und Ende
weiter
  OPEN    #ziel,#MODE_OLDFILE ;wie vor nun mit Zielfile
  move.l  d0,d5
  tst.l   d0
  bne     loop
  jsr     _LVOOutput(a6)
  WRITE   d0,#msg2,#len2
  bra     cl_q
* Hier geht's nun los
* ----------

loop
  READ    d4,#buffer,#256  ;Lese 256 Bytes oder weniger
  WRITE   d5,#buffer,d0    ;Ist-Laenge schreiben
  tst.l   d0      ;EOF?
  bne     loop

  READ    d5,#buffer,#1        ;warte auf Taste

cl_z
  CLOSE   d5                   ;Close Zielfile
cl_q
  CLOSE   d4                   ;und Quellfile
cl_lib
  move.l  a6,a1                ;DOS-Lib-Basis
  move.l  _SysBase,a6          ;Basis Exec
  jsr     _LVOCloseLibrary(a6) ;Funktion "Schliessen"
fini
   rts        ;Return zum CLI

* Datenbereich:
* --------
dosname dc.b  'dos.library',0
        cnop  0,2
msg1    dc.b  'Quellfile nicht gefunden',10
len1    equ  *-msg1
        cnop   0,2
msg2    dc.b  'Konnte Console nicht oeffnen',10
len2    equ  *-msg2
        cnop   0,4
quelle  ds.b  40
ziel    dc.b   'CON:0/12/640/200/Console',0
        cnop   0,4
buffer  ds.b   256
        end
```

Abb. 6.14: I/O in einem CON-Window

Beachten Sie, daß ich als allererstes mit

```
movem.l a0/d0,-(sp)   ;Kommandozeile retten
```

zwei Register rette. In A0 steht die Adresse, in D0 die Länge der Kommandozeile. »argc, argv« gibt es in Assembler leider nicht, weshalb man die Kommandozeile auch selbst »parsen« muß (eine einfache Übung). Hier erwarte ich nur ein Argument, nämlich den Namen des Quell-Files. Da das wieder ein C-String werden muß, setze ich mit

```
move.b     #0,-1(a0,d0.l)
```

das Null-Byte anstelle des CR. Ansonsten wäre noch zu erwähnen, daß ich für die Fehlermeldungen nicht »conhandle« benutze, sondern mit Output() das Handle des CLI-Windows ermittle, und darin eventuelle Fehlermeldungen ausgebe (Was soll ich tun? An dieser Stelle weiß ich noch nicht, ob sich CON überhaupt öffnen läßt). Übrigens: »Close(Output-Handle)« ist verboten!

Der Begriff »DOS-Library öffnen« klingt etwas verwirrend, ist aber üblich. Natürlich ist das DOS schon da (es steht im ROM), wir brauchen aber den Zeiger auf die Basis-Adresse der Sprungtabelle zu den DOS-Funktionen.

Praktisch: Pascal-Strings in Assembler

Um einen Text auszugeben, kann man natürlich das Makro WRITE benutzen, was allerdings unpraktisch ist, wenn man es sehr oft aufruft. Ein Unterprogramm macht sich in diesem Fall besser, doch da stört, daß man immer die Länge des Textes übergeben muß. Die Abhilfe ist recht einfach. Man definiere die Texte als Pascal-Strings, sprich, halte im ersten Byte die Länge. Ein Beispiel:

```
msg1 dc.b 10,'Hallo Welt'
```

Nun sieht der Aufruf so aus:

```
lea    msg1,a0
bsr    print
```

Das folgende Unterprogramm holt mit »move.b (a0)+,d3« die Länge in das Register D3 (wo sie »Write« sehen will) und setzt gleichzeitig A0 auf den Textbeginn. A0 muß dann noch in D2 umgeladen werden, weil – grober Mangel – die DOS-Funktionen Adressen in Datenregistern sehen wollen.

```
print
     moveq  #0, d3       ;d3 löschen
     move.b (a0)+,d3     ;Textlänge eintragen
```

```
move.l a0,d2        ;Adresse für Write
move.l d5,d1        ;Output-Handle
jsr  _LVOWrite(a6) ;Funktion "Schreiben"
rts
```

6.12 Top Down Bottom Up

Zwar zum Schluß des Kapitels, doch immerhin überhaupt, sollte noch noch eine Kleinigkeit geklärt werden. Wie schreibt man überhaupt ein Assembler-Programm?

Die Antwort ist ganz einfach: mit System! Das erfolgreichste System heißt »Top Down Bottom Up«. Wie das funktioniert, wird das folgende Beispiel zeigen.

Es soll ein Disk-Editor entwickelt werden. Im Hauptmenü hat der Anwender die Auswahl unter den Kommandos »L)esen«, »E)ditieren«, »S)chreiben« und »E(x)it«. Wie das Menü angeboten wird, lasse ich erst einmal dahingestellt sein. Ein Tastendruck oder Mausklick führt zu einer der folgenden Routinen, und die sehen erst einmal so aus:

```
Lese
     rts
Edit
     rts
Schreib
     rts
Exit
     rts
```

Nun nehme ich mir die erste Routine (Lese) vor und fülle Sie aus, aber wie!

```
Lese
     bsr   read_sec
     bsr   anzeige
     rts
read_sec
     rts
anzeige
     bsr  wandle
     bsr  print
     rts
wandle
     rts
print
     rts
```

Sie sehen sofort die Struktur des Programms. Was in den einzelnen Unterprogrammen passiert, interessiert zuerst gar nicht. Man kann nun hergehen und die einzelnen Unterprogramme nacheinander mit »Fleisch füllen«. Ist ein Unterprogramm fertig, wird es ausgetestet. Erst wenn

es läuft, wird das nächste begonnen. Praktisch geht man sogar noch einen Schritt weiter. Zum Beispiel benötigt das Unterprogramm »Lese« eine Routine, die einen Sektor liest, und eine weitere, die den gelesenen Sektor auf dem Schirm in hex ausgibt. Dazu brauche ich unter anderem ein Unterprogramm »Anzeige«. Anzeige benötigt aber eine Routine, die ein Wort in die entsprechenden ASCII-Strings umwandelt. Damit ergibt sich dieser Ablauf:

Begonnen habe ich ganz oben und bin zum Schluß beim Unterprogramm »print« gelandet. Dies muß ich nun wirklich bearbeiten. Wenn die Routine »print« läuft, kann ich »wandle« beginnen. Denn nun erst kann ich ja die gewandelten Hex-Zahlen ausgeben und somit die Routine »print« auch testen. Jetzt werde ich mir »anzeige« vornehmen, das durch mehrfachen Aufruf von Print einen Pufferinhalt auf dem Schirm ausgibt. Danach werde ich »read_sec« schreiben, was diesen Puffer mit Daten füllt. Nun schließlich kann ich im Hauptmenü »Lese« aufrufen, und wäre damit wieder ganz oben.

Dieses »von oben nach unten und wieder zurück« nennt man »top down bottom up«. Dies ist eine Methode der Programmierung, die gerade in Assembler sehr zu empfehlen ist. Sie beschränken damit die Fehlersuche immer nur auf einen kleinen, überschaubaren Bereich. Scheuen Sie dabei auch nicht den Mehraufwand, einzelne Unterprogramme temporär mit Spieldaten zu versorgen. Der Aufwand ist gering, der Nutzen ist groß. Zum Beispiel soll die Routine »wandle« ein Wort in D0 als Hex-String ausgeben. Schreiben Sie dann einfach

```
move $19AF,d0
```

vorläufig als erste Zeile im Unterprogramm »wandle«. Wenn Sie nun die Routine testen und »19AF« auf dem Schirm sehen, dann können Sie schon ziemlich sicher sein, daß »wandele« funktioniert.

Haben Sie keine Hemmungen, für jede Kleinigkeit ein Unterprogramm zu schaffen. Wenn Sie eine ganze Seite ohne »rts« schreiben, ist das schon zuviel, und wenn das Unterprogramm nur drei Zeilen hat, ist das auch OK. Wenn alles läuft, können Sie immer noch das Unterprogramm in die aufrufende Routine verlegen.

Noch ein Tip:

Werfen Sie nichts weg!

Sichern Sie vor jeder Änderung, die die Struktur des Programms betrifft (nicht die Beseitigung von Fehlern) den aktuellen Stand unter einem eigenen Namen, typisch mit »Name.nnn«, wobei »nnn« eine systematische Zahl ist. Oft stellt sich nämlich heraus, daß der eingeschlagene Weg doch ein Holzweg ist. Dann will man wieder auf den Stand zurück, der noch ging, und einen anderen Weg probieren. Da solche Verzweigungen mehrfach auftreten, spricht man hier von einem Entwicklungsbaum. In echten Profi-Systemen gibt es ein Programm, das solche Entwicklungsbäume automatisch registriert und darstellt (und auch das Löschen verhindert). Für den Hausgebrauch tut es auch Papier und Bleistift.

7 Tips für Basic-Programmierer

In diesem Kapitel finden Sie nahezu zahllose Tips und Tricks für den Basic-Programmierer, allerdings sehr wenig über Basic an sich. Es geht hauptsächlich um die speziellen Eigenschaften des Amiga und die Ausnutzung seiner vielen Möglichkeiten. Einige Tips zu AmigaBasic gibt es natürlich auch, doch häufiger wird gezeigt, wie man Probleme in Basic löst, in dem man Basic nicht nutzt. Ein Widerspruch? Nun, AmigaBasic hat eine sehr schöne Schnittstelle zum Betriebssystem und damit zu vielen hundert neuen Funktionen. Alles, was Sie an so tollen Sachen in anderen Programmen sehen, ist auch in AmigaBasic machbar, man muß nur wissen, wie.

7.1 Starten und Beenden von Basic

Es gibt drei Möglichkeiten, AmigaBasic zu starten und auch drei Methoden, es (ordentlich) zu beenden. Beim Start können Sie noch auf verschiedene Arten den Namen eines Programms mitgeben, das dann gleich ausgeführt wird. Zusätzlich können Sie ein Programm auch so präparieren, daß es mit einem Doppelklick gestartet werden kann (und vorher automatisch AmigaBasic lädt). Die trivialste Methode, nämlich der Start durch einen Doppelklick auf das AmigaBasic-Piktogramm, zählt natürlich nicht als Trick.

7.1.1 Programm und Basic gleichzeitig starten

Wenn Sie wollen, daß ein Programm sofort nach dem Laden von AmigaBasic ausgeführt werden soll, gehen Sie so vor:

1. Öffnen Sie die Schublade, in der sich das Programm befindet. Ordnen Sie die Fenster so an, daß Sie sowohl das Programm-Piktogramm als auch das Piktogramm von AmigaBasic sehen.

2. Klicken Sie *einmal* auf AmigaBasic. Das Piktogramm muß schwarz werden.

3. Drücken Sie die Shift-Taste und halten Sie sie fest. Bei gedrückter Shift-Taste doppelklicken Sie auf das Programm-Piktogramm.

7.1.2 Programm mit einem Doppelklick starten

Sie können ein Basic-Programm so präparieren, daß ein Doppelklick auf sein Piktogramm zum Starten ausreicht.

1. Klicken Sie das Programm-Piktogramm einmal an, so daß es schwarz wird.
2. Wählen Sie aus dem Menü Workbench den Punkt Info an.
3. Im nun erscheinenden Requester steht unter »Default Tool« der Text »:AmigaBasic«. Sind Sie auf der Original-Diskette (oder einer Kopie davon) ist das OK. Steht Ihr AmigaBasic jeoch zum Beispiel auf der Festplatte DH0: in einer Schublade, zum Beispiel mit dem Namen »basic«, so ändern Sie den Text in »DH0:basic/AmigaBASIC«. Das heißt, Sie müssen immer den korrekten Pfadnamen einsetzen.
4. Klicken Sie links unten auf SAVE.

7.1.3 AmigaBasic vom CLI aus starten

Sie haben zwei Möglichkeiten, AmigaBasic vom CLI aus zu starten. Tippen Sie ein:

```
amigabasic
```

oder

```
run amigabasic
```

Im ersten Fall läuft AmigaBasic als ein Unterprogramm von CLI, im zweiten Fall als eigener Task. Wenn Sie von Basic aus zwischendurch (mittels Klick auf das CLI-Fenster) in das CLI gehen wollen, müssen Sie mit RUN starten. Der Trick dabei ist folgender:

Das Ganze funktioniert so nur, wenn Sie in dem Directory sind, in dem sich AmigaBasic befindet. Wenn Sie aber AmigaBasic in das c-Directory kopieren, wird es sozusagen ein CLI-Befehl und ist dann von jedem anderen Directory aus auch aufrufbar.

Soll ein Programm sofort nach dem Laden von AmigaBasic gestartet werden, so tippen Sie nach AmigaBasic ein Leerzeichen und dann den Programm-Namen. Beispiel:

```
run amigabasic music
```

7.1.4 Basic auf der Start-Diskette

Wer hauptsächlich in Basic programmiert oder das zumindest zeitweise tut, weil er an einem größeren Projekt arbeitet, will gleich nach dem Einschalten des Amiga in Basic »landen«. Das geht generell sehr einfach, aber das Thema hat drei Varianten.

Immer müssen Sie AmigaBasic auf die (Kopie Ihrer) Startdiskette kopieren, es sei denn, Sie haben eine Festplatte. Nun nehmen Sie sich die Startup-Sequence im s-Verzeichnis vor, sprich, laden Sie diese Datei in einen Texteditor. Am Ende finden Sie die Zeilen

```
LoadWB delay
endcli >NIL:
```

Damit wird erst die Workbench geladen und dann das CLI, das bisher aktiv war, beendet. Fügen Sie vor das »LoadWB« diese Zeile ein:

```
AmigaBasic
```

Festplattenbesitzer schreiben »:basic/AmigaBasic« oder sonstwie den kompletten Pfad. Speichern Sie die Startup-Sequence, und starten Sie den Amiga neu, Sie werden im Basic landen. Wenn Sie Basic verlassen, wird die Disk kurz anlaufen, denn jetzt wird der Folge-Befehl, nämlich das Laden der Workbench ausgeführt. Ergebnis: Nach dem Beenden von Basic sind Sie wieder auf der Workbench.

Sie können auch schreiben:

```
run AmigaBasic
```

Das hat den Vorteil, daß die Workbench schon geladen wird, während Sie im Basic sind, kostet aber etwas Speicher für ein zweites CLI. Auf diese Art können Sie aber auch den Aufruf von Basic noch weiter vorverlegen, wie weit, hängt von Ihrer Hardware-Konfiguration und Ihren Wünschen ab, und somit die zum Teil recht lange Startup-Sequence abarbeiten lassen, während Sie schon im Basic tätig sind.

Bliebe noch als letzte Möglichkeit, nämlich nur »Amigabasic« zu schreiben und die Zeilen danach wegzulassen. Das ist sinnvoll, wenn Sie gar nicht auf die Workbench wollen.

7.1.5 Beenden von AmigaBasic

Um die Problematik zu erkennen, machen Sie einmal folgenden Versuch:

Starten Sie Basic mit dem mitgelieferten Programm »music«. Öffnen Sie das List-Fenster. Nach einer kurzen Pause wird die Musik weiterspielen. Schließen Sie das Ausgabe-Fenster, die Musik spielt immer noch. Auf der Workbench starten Sie ein anderes Programm, zum Beispiel den Taschenrechner, die Musik spielt immer noch. Achten Sie auf die Speicheranzeige im Schirm-Titel. Schließen Sie nun das List-Fenster. Die Musik stoppt, der freie Speicher wird größer.

Daraus folgt: Basic kann korrekt beendet werden, indem man *alle* seine Fenster schließt oder Quit oder SYSTEM anwendet.

Quit steht im Project-Menü. Dies setzt voraus, daß das Programm nicht läuft. Sie können ein Programm mit Stop im Run-Menü anhalten oder mit Amiga-Punkt (rechte Amiga-Taste und .).

Der Befehl **SYSTEM** kann in das Ausgabe-Fenster eingetippt werden, aber auch im Programmtext stehen.

7.2 Editor-Tips (und -Tricks)

Der Basic-Editor wird zwar oft seiner Langsamkeit wegen geschmäht, aber da tut man ihm unrecht. Wenn man das Pferd richtig zu reiten weiß, kann man damit Rennen gewinnen. Zuerst einige Grundfunktionen:

Falsche Zeichen können sofort mit der Backspace-Taste gelöscht werden.

An beliebiger Cursor-Position gilt:

Backspace löscht das Zeichen links vom Cursor.

Del löscht das Zeichen rechts vom Cursor.

Steht der Cursor vor dem ersten Zeichen einer Zeile, wird durch Backspace diese Zeile an das Ende der vorherigen angefügt.

Eine Auswahl (siehe nächster Absatz) kann durch die Del-Taste gelöscht werden.

Sie können im List-Fenster einen Text auswählen, indem Sie beim Textbeginn die linke Maustaste drücken und dann die Taste festhaltend die Maus bis zum Auswahlende ziehen. Der jeweils ausgewählte Text ist rot unterlegt.

Alternativ können Sie am Auswahlbeginn klicken und dann – die Shift-Taste haltend – am Auswahlende klicken.

Ziehen Sie die Auswahl über die Fenstergrenzen hinaus, rollt der Text automatisch nach links bzw. nach oben.

Ein einzelnes Wort (oder ein alleinstehendes Zeichen) kann durch einen Doppelklick innerhalb des Wortes ausgewählt werden.

Eine Zeile wird am schnellsten so ausgewählt: Ziehen Sie links von der Zeile beginnend eine Zeile nach unten.

Der Cursor kann innerhalb des List-Fensters mit den Cursor-Steuertasten verschoben werden. Bewegen Sie den Cursor über die Fenstergrenzen hinaus, rollt der Text automatisch nach links bzw. nach oben.

Alternativ kann der Cursor durch Klicken mit der Maus positioniert werden.

Cursor-Steuertasten + Shift

Bei gedrückter Shift-Taste können Sie mit den senkrechten Cursor-Steuertasten jeweils eine Fensterseite blättern. Bei gedrückter Shift-Taste können Sie mit den waagerechten Cursor-Steuertasten jeweils eine dreiviertel Zeilenbreite weiterschalten.

Cursor-Steuertasten + Alt

Bei gedrückter Alt-Taste gelangen Sie mit den vier Cursor-Steuertasten an die entsprechenden Textenden.

Viele Zeilen löschen

Wenn Sie viele Zeilen löschen wollen, kann folgende Methode empfohlen werden:

Schreiben Sie vor den Beginn der ersten Zeile oder in eine neue Zeile davor eine Zeilennummer, zum Beispiel 1.

Schreiben Sie vor die letzte Zeile oder in eine neue Zeile dahinter eine höhere Zeilennummer, zum Beispiel 2.

Tippen Sie in das Ausgabefenster

```
delete 1-2
```

7.2.1 Erst das List-Fenster

Das Standard-List-Fenster erweist sich oft als zu schmal. Folglich schiebt man es nach links, um es dann breiter ziehen zu können, und das dauert dann und dauert. Zwei Lösungen gibt es.

1. Starten Sie mit »New«, bringen Sie das Fenster auf die richtige Größe, und laden Sie erst dann das Programm.
2. Tippen Sie am Ende des Listings ein paarmal Return, bis das Fenster leer ist. Das leere Window ist beweglicher.

7.2.2 Inhaltsverzeichnis ansehen

Daß Sie sich mit »files« das aktuelle Inhaltsverzeichnis ansehen können, wissen Sie ja schon, doch was machen Sie, wenn da eine lange Liste durchrast? Nun, Sie können die Ausgabe mit Ctrl-S (Control-Taste + s gleichzeitig) anhalten und mit einer beliebigen Taste wieder fortsetzen (und wieder mit Ctrl-S anhalten). Anstatt Ctrl-S können Sie auch *A*-S (rechte Amiga-Taste + s gleichzeitig) nehmen, aber dafür brauchen Sie beide Hände. Es geht aber auch mit einem Finger. Drücken einfach die rechte Maustaste! Das funktioniert übrigens auch, wenn ein Basic-Programm »ohne anzuhalten« in das Ausgabefenster schreibt.

7.2.3 Schneller mit FF

Auch der Basic-Editor wird mit »FF« (Fast Fonts) schneller. Dieses nützliche und nur 3 Kbyte kleine Programm gehört seit der Workbench 1.3 zum Lieferumfang. Da es im C-Directory steht, können Sie es von überallher aufrufen. Praktischerweise macht man das vor dem Start von Basic, aber dazu müßte man erst in das CLI oder in die Shell gehen, und das vergißt man leicht. Auch in Basic kann man das nachholen, indem man das Basic-Window in den Hintergrund klickt und dann die Shell startet. Ist es da nicht einfacher, in Basic ein Basic-Programm laufen zu lassen? Hier ist es:

```
DECLARE FUNCTION dOpen& LIBRARY
LIBRARY "dos.library"
n$=CHR$(0)
w$ = "CON:010/020/600/200/CLI-Window"+n$
h&=dOpen&(SADD(w$),1005&)
execute& SADD("ff"+n$),0,h&
IF h& THEN dClose h&
LIBRARY CLOSE
```

Wie man solche CLI-Befehle in Basic ausführt, steht im Abschnitt 7.5

7.2.4 Undo oder Kommando zurück

Wenn Sie eine Zeile geändert haben, und das dann doch nichts war, oder wenn Sie zu lange auf die Backspace- oder Del-Taste gedrückt hatten und plötzlich Zeichen verschwunden sind, dann ist noch nichts verloren. Drücken Sie A-L (rechte Amiga-Taste und L), und Basic stellt die Zeile wieder her.

Leider funktioniert das aber nur, solange Sie die Zeile nicht verlassen haben. Drücken Sie also nicht zu eilig die Return-Taste. Auch die Cursor-Tasten oder ein Klick in ein anderes Window haben diese Abschluß-Wirkung, mit einer Ausnahme: Der Menüpunkt »Show List« hat auch diese Undo-Funktion.

Wie funktioniert das?

Basic hält in einem Puffer das Listing und in einem anderen die aktuelle Zeile. Erst durch das Abschlußzeichen, zum Beispiel die Return-Taste, wird die Zeile in das Listing kopiert und überschreibt dabei den alten Stand. »Show List« hingegen zeigt das noch nicht geänderte Listing und vergißt dabei den Zeilenpuffer. A-L ist nichts weiter als das Tastenkürzel von »Show List«.

7.2.5 Schneller zum Listing

Wenn man ein Programm abbrechen muß (Amiga-Punkt oder <Ctrl>+<C>), ist leider oft das List-Fenster verschwunden. Doch wenn Sie die Sache genauer verfolgen, stellt sich heraus, daß Basic (fast) immer zu dem Fenster zurückkehrt, von dem das Programm gestartet wurde. Also aktivieren

Sie zuerst das List-Fenster, wenn Sie nicht eh schon da sind, und starten Sie dann das Programm mit <A>+<R>

7.3 Libraries in Basic

Der Amiga bietet eine Unmenge von fertigen Unterprogrammen, die Sie fast alle in Basic nutzen können. Diese Unterprogramme sind sozusagen nach Themen geordnet, diese Themengruppen nennt man Libraries, zu deutsch Bibliotheken. Einige dieser Libraries stehen im ROM des Amiga, andere befinden sich auf Disketten und werden erst bei Bedarf in das RAM geladen.

7.3.1 FDs und BMAPs

Wenn Sie bisher beim Thema »Basic und Libraries« nicht ganz durchgeblickt haben, dann liegt das bestimmt nicht an Ihnen, sondern an den Erfindern der folgenden etwas seltsamen Regeln. Im Basic-Programm schreibt man beispielsweise

```
LIBRARY "graphics.library"
```

Damit das funktioniert, muß es im selben Verzeichnis (oder in :libs) eine Datei namens »graphics.bmap« geben. Die gibt es jedoch erst einmal gar nicht, dafür existiert aber eine Datei mit dem Namen »graphics_lib.fd«. Dieses FD-File muß mittels des Basic-Programms »ConvertFD« in ein BMAP-File umgewandelt werden, doch wieso?

Fast alle der Library-Unterprogramme benötigen Parameter. So müssen Sie dem imaginären Unterprogramm »SetzePunkt« schon sagen, wo es den Punkt setzen soll. Die Library-Routinen erwarten diese Parameter in bestimmten Registern der CPU, zum Beispiel den X-Wert in D0 und den Y-Wert in D1. Sie schreiben aber nur »CALL SetzePunkt(33,77)«. Folglich muß Basic wissen, daß es die 33 in das Register D0 und die 77 in das Register D1 packen muß.

Diese Informationen und noch einige je Unterprogramm stehen in den FD-Files und zwar im Klartext. Sie können sich so einen FD-File ohne weiteres mit einem Editor ansehen oder ihn ausdrucken. Die FD-Files liefert Commodore in dieser allgemeinen Form, und die Hersteller der verschiedenen Sprachen wie Assembler, Basic oder C nutzen sie. AmigaBasic arbeitet jedoch nicht direkt mit diesen FD-Files, sondern sozusagen mit einer aufbereiteten Form, die speziell an die Besonderheiten von AmigaBasic angepaßt und deshalb auch schneller ist. Dieses Format ist das BMAP-Format und deshalb muß ein FD-File erst – aber nur einmal – mit ConvertFD erzeugt werden.

Sie müssen sich nur immer eines vor Augen führen: Die Library als solche steht im ROM oder im RAM des Amiga. Die BMAP-Datei ist nur das Interface (die Schnittstelle) zwischen AmigaBasic und einer Library.

7.3.2 Der Einsatz von ConvertFD

Bevor Sie loslegen, schauen Sie einmal im Verzeichnis »libs« Ihrer Workbench-Diskette nach, ob es da nicht schon einige BMAP-Files gibt. Für diese können Sie gleich beim Punkt »Wohin mit den BMAPs?« weitermachen, ansonsten, auf geht's. ConvertFD ist ein ganz normales Basic-Programm, das Sie sich ohne weiteres ansehen und auch an Ihre Wünsche anpassen können. Der Trick an der ganzen Geschichte ist nur, daß Sie die richtigen Pfade und Namen eingeben.

Erstellen Sie zuerst eine Arbeitskopie der mit dem Amiga gelieferten Diskette »Extras1.3D« oder »Extras1.2D«, wenn Sie noch mit der Version 1.2 arbeiten sollten. Nun starten Sie Basic von der Arbeitsdiskette. Im Kommando-Fenster wechseln Sie jetzt mit

```
chdir "FD1.3"
```

in das FD-Verzeichnis. Setzen Sie »FD1.2« für die ältere Version ein. Nun geben Sie in das Kommando-Fenster ein:

```
run ":BasicDemos/ConvertFD"
```

Der Pfad »:BasicDemos/« ist erforderlich, weil sich ConverFD im BasicDemo-Verzeichnis befindet. ConvertFD startet jetzt und fragt Sie nach dem Namen der zu lesenden FD-Datei. Geben Sie im Fall der »graphics.library« ein:

```
graphics_lib.fd
```

Jetzt kommt noch die Frage nach der Zieldatei, und da muß die Antwort lauten:

```
graphics.bmap
```

Dieses Namensschema ist durchgängig. So gilt beispielsweise:

```
Basic-Name      FD-Name         BMAP-Name
-----------------------------------------
dos.library     dos_lib.fd      dos.bmap
icon.library    icon_lib.fd     icon.bmap
exec.library    exec_lib.fd     exec.bmap
```

7.3.3 ConvertFD-Tip

Das Programm ConvertFD muckelt so eine Weile vor sich hin, und zwischendurch passiert leider gar nichts. Es wäre doch schön, wenn man sehen könnte, wie das Programm arbeitet. Nun denn:

Geben Sie LIST, um das Programm bearbeiten zu können, und suchen Sie dann das Unterprogramm »SUB GetLine«. Da steht eine Zeile der Art

```
LINE INPUT #1, buf$
```

Damit wird immer eine Zeile aus der FD-Datei in die Variable »buf$« eingelesen. Da es sich um reinen Text handelt, können Sie diese Variable auch drucken. Also fügen Sie nach dieser Zeile hinzu:

```
PRINT buf$
```

und schon sehen Sie (ab dem nächsten RUN), wie ConvertFD arbeitet.

7.3.4 Wohin mit den BMAPs?

Wenn Sie mit ConvertFD eine BMAP-Datei erzeugt haben, wird sie sich im aktuellen Verzeichnis, hier »FD1.3«, befinden. Das ist allerdings der falsche Platz. Wenn Ihr Programm eine Library anzieht, sucht Basic die BMAP-Datei zuerst im aktuellen Verzeichnis und dann im Verzeichnis »libs« auf der Startdiskette. Wird sie auch da nicht gefunden, bricht Basic Ihr Programm mit einer Fehlermeldung ab. Es gibt also zwei Möglichkeiten, die beide ihre Vor- und Nachteile haben.

1. Das Verzeichnis, in dem sich AmigaBasic und Ihr Programm befinden.
2. Das Verzeichnis LIBS.

Methode 1 hat den Vorteil, daß Basic, Ihre Programme und die BMAPSs nicht auf der Startdiskette sein müssen. Der Nachtteil ist, daß Sie die BMAPs immer mitkopieren müssen, wenn Sie das Programm auf eine andere Diskette bringen. Das hat dann auch zur Folge, daß dieselben BMAP-Dateien auf mehreren Disketten sind.

Methode 2 hat den Nachteil, daß Basic die BMAPs nicht findet, wenn Sie von einer anderen Diskette starten. Sie ist aber immer vorzuziehen, wenn Ihr Amiga von einer Festplatte bootet.

7.3.5 Spartip

Daß Sie in beiden der o.g. Fälle die FD-Dateien nicht mehr brauchen, ist klar. Doch Basic erzeugt von den BMAP-Dateien auch Info-Files (...bmap.info), die Sie getrost löschen können.

7.3.6 Der Einsatz von Libraries

Wie schon gesagt, gibt es ROM- und Disk-Libraries. In beiden Fällen muß man die Library zuerst öffnen, was bei ROM-Libraries eine (nicht zu umgehende) Formsache ist, und bei Disk-Libraries das Laden veranlaßt. Hier kann aber unter Umständen Speichermangel beklagt werden. Was dann zu tun ist, schildern die Tips zum Thema »Out of Memory«.

Geöffnet wird eine Library mit der Anweisung »LIBRARY „Name“«, zum Beispiel mit

```
LIBRARY "graphics.library"
```

Nach dem Öffnen oder vorher müssen Sie alle Funktionen, die Sie benutzen wollen und die einen Wert zurückgeben, noch deklarieren, damit Basic den Typ des Rückgabewertes kennt. Der Typ kann Integer (%), LongInteger (&) oder »void« (gar nichts) sein. Letzteren Typ kennt Basic nicht. In diesem Fall dürfen Sie die Funktion auch nicht deklarieren. Zum Beispiel:

```
DECLARE  FUNKTION  ReadPixel%  LIBRARY
```

oder

```
DECLARE  FUNKTION  ReadPixel%()  LIBRARY
```

oder

```
DECLARE  FUNKTION  ReadPixel%(rp,  x,  y)  LIBRARY
```

Wichtig ist der Typ (hier %), der Rest dient der Dekoration oder Dokumentation, wird aber von Basic ignoriert. Nun noch ein kleines Beispiel (Abb. 7.1), das die grundsätzliche Technik zeigt.

Sehr detailliert wird die Thematik in Horst-Rainer Hennigs Buch »Programmierpraxis AmigaBasic« (Markt & Technik Verlag) beschrieben.

```
LIBRARY "graphics.library"
DECLARE FUNCTION ReadPixel%(rp,x,y) LIBRARY
rp& = WINDOW(8) 'RastPort

CLS
RANDOMIZE TIMER

FOR i=1 TO 1000
  x% = FIX(RND * 600)
  y% = FIX(RND * 200)
  WritePixel rp&,x%,y%
NEXT

c=0
PRINT "Ich suche..."

FOR i=1 TO 1000
  x% = FIX(RND * 600)
  y% = FIX(RND * 200)
  f% = readpixel%( rp&,x%,y% )
  IF f% THEN c=c+1
NEXT

PRINT c;" Punkte gefunden"

LIBRARY CLOSE
```

Abb. 7.1: Der Umgang mit Libraries

Das Programm setzt mittels der Library-Funktion »WritePixel« zufällig 1000 Bildpunkte. Anschließend werden wieder zufällig mit »ReadPixel« 1000 Punkte gelesen, genauer: es wird geprüft, ob auf diesen Koordinaten ein Punkt gesetzt ist. Im Ja-Fall wird der Zähler »c« um eins erhöht.

Wichtig ist, daß die Funktion »ReadPixel« deklariert wurde, denn sie gibt einen Wert (die Farbe des Punktes oder null) zurück. »WritePixel« hingegen gibt keinen Wert zurück und darf deshalb auch nicht deklariert werden.

Das Beispiel in Abb. 7.2 soll aufzeigen, daß man auch mehrere Libraries gleichzeitig öffnen kann, mehr als fünf dürfen es aber nicht sein.

```
DECLARE FUNCTION viewportAddress& LIBRARY
LIBRARY "intuition.library"
LIBRARY "graphics.library"
DEFINT x,y,f

SCREEN 2,320,256,3,1
WINDOW 2,,,0,2
PALETTE 0,0,0,0
PALETTE 1,0,0,0

FOR i=1 TO 300
  x = RND*320
  y = FIX(RND*256)
  f = FIX(RND*7)
  PSET (x,y),f
NEXT

x=0: y=0
WHILE NOT MOUSE(0)
  x=x+1 'y=y+1 wenn vertikal
  GOSUB DoScroll
WEND

WINDOW CLOSE 2
SCREEN CLOSE 2
LIBRARY CLOSE
END

DoScroll:
w& = x*(2^16)+y
vp&=viewportAddress&(WINDOW(7))
ri& = PEEKL(vp& +36)
POKEL ri&+8,w& 'ri+10 wenn vertikal
CALL RethinkDisplay& 'Gegen Flimmern
CALL ScrollVport&(vp&)
RETURN
```

Abb. 7.2: Bis zu fünf Libraries können gleichzeitig offen sein

Das Programm malt wieder einen Sternenhimmel, jetzt aber mit PSET. Doch nun rollt das ganze Bild so lange horizontal, bis die linke Maustaste gedrückt wird. Im Unterprogramm »DoScroll« wird aus der ViewPort-Struktur die Adresse der RasInfo-Struktur (ri&) geholt. Da sind beim Offset 8 für x bzw. 10 für y die Scroll-Werte einzutragen. Je nach deren Größe wird der Screen verschoben, wofür »ScrollvPort« zuständig ist. Wenn das Bild flimmert – hier fällt das kaum auf – muß man mit »RethinkDisplay« für Ruhe sorgen. Diese Funktion aus der Intuition-Library überprüft und initialisiert alle Parameter der Bildausgabe.

7.3.7 Deklarieren Sie richtig

Eine Anmerkung zum Titel: Ein Tip zu dem, was Sie dem Zöllner auf dem Flughafen erzählen sollen, wird das nicht. Es geht hier um das »DECLARE FUNCTION«. Wie schon gesagt, sollte man Funktionen, die keinen Wert zurückgeben, auch nicht deklarieren, dito, wenn einen der Wert nicht interessiert. Nun gibt es aber Funktionen, wo das scheinbar nicht geht. So bringt zum Beispiel die Zeile

```
SetSoftStyle&(WINDOW(8),s%,m%)
```

mit konstanter Boshaftigkeit einen Syntaxfehler. Erst, wenn man das ändert in

```
void& = SetSoftStyle&(WINDOW(8),s%,m%)
```

und jetzt natürlich die Funktion deklariert, kehrt Ruhe ein, das Programm läuft korrekt. Die Deklaration können Sie aber sparen und zwei Klammern auch noch, wenn Sie schreiben:

```
SetSoftStyle& WINDOW(8),s%,m%
```

7.3.8 Welche Libraries gibt es?

Die Frage kann ganz einfach mit der folgenden Liste beantwortet werden, allerdings mit einer Einschränkung: sie gilt für die Workbench 1.3. Es gibt ROM- und Disk-Libraries. Für erstere reicht die BMAP-Datei (siehe oben, »Wohin mit den BMPAs«), bei Disk-Libraries müssen Sie zusätzlich dafür sorgen, daß diese im selben Inhaltsverzeichnis (LIBS) sind.

ROM-Libraries

dos.library
exec.library
expansion.library
graphics.library
intuition.library
layers.library
mathffp.library

Disk-Libraries

diskfont.library
icon.library
mathieeedoubbas.library
mathieeedoubtrans.library
translator.library

Beim Einsatz der DOS-Library müssen Sie beachten, daß viele Funktionen die Namen von Basic-Befehlen haben (Open, Close, Input u.s.w). Da das dann zum Syntax-Error führt, müssen Sie diese Funktionen umbenennen. Üblich sind die Präfixe d und x. Beim Einsatz der DOS-Library schreiben Sie zum Beispiel dOpen (für DOS-Open) anstatt Open.

Ansonsten gibt es nur noch einen Namenskonflikt und das ist »translate« in der »translator.library«, wo Sie dann »xtranslate« schreiben sollten.

7.4 Out of Memory, was nun?

Die Fehlermeldung »Out of Memory« gibt AmigaBasic aus, wenn der Hauptspeicher (das RAM) für ein bestimmtes Basic-Programm zu klein ist. Ähnliche Meldungen können aber auch andere Programme erzeugen. Zur Lösung gibt es zwei Ansätze:

1. Optimierung des Basic-Programms
2. Entfernen allgemeiner Speicher-Schlucker

Kommt die Fehlermeldung von einem Basic-Programm, beginnen Sie mit dessen Optimierung. Reicht das nicht, entfernen Sie auch noch die allgemeinen Speicher-Schlucker. Bei Nicht-Basic-Programmen beginnen Sie logischerweise beim Ansatz 2.

7.4.1 Optimierung von Basic-Programmen

Zuerst können Sie davon ausgehen, daß es wenig bringt, nur aus Speichergründen den Programm-Code zu optimieren, das sind nur Bytes. Was die »Kilo-Bytes« kostet, sind die Datenstrukturen.

7.4.1.1 Speicher messen und einteilen

Bevor Sie loslegen, sollten Sie allerdings wissen, wo es kneift. Es gibt nämlich drei verschiedene Speicherbereiche, bei denen Sie ansetzen können.

Der Systemspeicher

ist der gesamte Speicher Ihres Amiga. Alle Programme laufen in diesem Speicher, so auch der AmigaBasic-Interpreter, der erst einmal selbst 200 Kbyte benötigt. Bestimmte Basic-Routinen wie SOUND und WAVE können auch noch Teile des Systemspeichers belegen.

Der Programmspeicher

ist der Bereich, in dem Ihre Basic-Programme abgelegt werden, was – gemessen am Rest – der kleinste Teil ist. In diesem Bereich werden aber auch alle Variablen gespeichert, wobei Arrays entsprechender Dimension (siehe unten) hier die größten »Speicherschlucker« sind.

Der Stapelspeicher

ist am schwierigsten abzuschätzen, weil seine Größe vom aktuellen Programmzustand abhängt. Auf dem Stapel werden alle Rückkehradressen von Unterprogrammen (GOSUB) und SUB-Programmen (CALL) abgelegt. Außerdem werden hier auch alle Schleifen (FOR und WHILE) sowie Funktionen (DEF FN) verwaltet. Je tiefer die Unterprogramme verschachtelt sind (vom UP GOSUB zum nächsten UP) und je mehr Schleifen offen sind, desto größer wird der Stapelspeicherbedarf. Im allgemeinen dürfte aber der voreingestellte Wert von 1024 reichen, kleiner geht auch nicht.

Mit Clear können Sie die Größe der Bereiche vorgeben. Es gilt die Syntax:

```
CLEAR  [,  Programmbereich,  Stapelbereich]  ]
```

Die Argumente geben an, wieviel Speicher in Bytes für den jeweiligen Bereich (siehe oben) reserviert werden sollen. Vorher sollten Sie mittels FRE (siehe unten) prüfen, ob Ihre Anforderungen realistisch sind.

Der Systemspeicher ergibt sich automatisch als der Rest und errechnet sich aus Gesamtspeicher minus andere Programme minus Programmbereich minus Stapelbereich.

Der CLEAR-Befehl löscht zuerst alle Variablen, setzt alle mit DEF FN definierten Funktionen zurück und schließt alle offenen Dateien. Wird er ohne Argumente aufgerufen (nur CLEAR) ist das alles.

```
CLEAR  'alle Variablen löschen
CLEAR ,7000  ' 7000 Bytes für's Programm
CLEAR ,,5000 ' 5000 Bytes für den Stapel
REM alles mit einer Anweisung
CLEAR  ,7000,5000
```

Die volle angezeigte Größe des Systemspeichers (z.B. 600 000) kann nicht mit CLEAR für den Programm- oder Stapelspeicher reserviert werden, sondern höchstens etwa 220 000, meistens

nicht mehr als 150 000. Wenn Basic startet, sind 25 000 Byte für den Programmspeicher und 1024 Byte für den Stapelspeicher voreingestellt.

Die Größe der Speicherbereiche kann abgefragt werden, wobei ergeben:

```
PRINT FRE( 1) ' Programmspeicher
PRINT FRE(-2) ' Stapelspeicher
PRINT FRE(-1) ' Systemspeicher
```

7.4.1.2 Arrays dimensionieren

Wenn Sie in einem Programm zum Beispiel »a(1)« schreiben, legt Basic automatisch einen Array »a(10)« an. Da ab null gezählt wird, sind das 11 Elemente und aufgrund des Typs »single« schon 44 Byte. Schreiben Sie »a(1,1)« wird daraus »a(10,10)«, und das sind dann schon 484 Byte.

Also schreiben Sie besser vorher »DIM a(1)« oder DIM »a(1,1)« oder allgemein die höchste Dimension kleiner als 10, die Sie tatsächlich brauchen.

7.4.1.3 OPTION BASE nutzen

Ein Array dimensioniert mit »DIM a(1000,3)« hat nicht 3000 sondern 4000 Elemente, weil immer ab null gezählt wird. Brauchen Sie das nullte Element nicht, schreiben Sie vorher »OPTION BASE 1«. Das spart in diesem Beispiel schon 4 Kbyte.

7.4.1.4 Richtigen Daten-Typ nehmen

Wenn Sie in der DIM-Anweisung keinen Typ angeben, setzt Basic dafür den Typ Single ein. Wenn Ihre Werte ganze Zahlen sind und im Bereich von –32768 bis +32767 liegen, können Sie den Typ Integer einsetzen. Schreiben Sie dafür das Prozentzeichen hinter den Namen, zum Beispiel »dim a%(100)«. Da Integer-Zahlen nur zwei Byte belegen, sparen Sie damit genau die Hälfte.

7.4.1.5 Bis 65535 als »Unsigned Integer«

Das Limit von +32767 ist manchmal ungünstig. Wenn Sie mit Werten von 0 bis 65535 auskommen und keine negativen Zahlen benötigen, ja dann hätten Sie es in C ganz einfach, da gibt es nämlich den Typ »Unsigned Integer« (vorzeichenlose Ganzzahl). Das läßt sich aber auch in Basic simulieren, wie das Listing von Abb. 7.3 zeigt:

```
DIM  a%(100)

nochmal:
INPUT "Zahl  im  Bereich  0-65535:  ", x
IF x < 0 OR x > 65535 THEN GOTO nochmal
```

```
IF x > 32767 THEN x = 32767! - x
a%(1) = x
'Ausgabe der Zahl
x = a%(1)
IF x < 0 THEN x = 32767! - x
PRINT x
END
```

Abb. 7.3: »Unsigned Integers« in Basic sparen RAM

Bei einem Array von 100x100 sparen Sie damit rund 20 Kbyte. Was passiert?

Die Zahlen 0–32767 werden ganz normal abgelegt. Hingegen werden Zahlen im Bereich von 32768–65535 in negative Zahlen gewandelt, die zwischen –1 und –32768 liegen. Vor der Ausgabe muß das natürlich wieder umgekehrt werden. Nehmen wir an, die Zahl sei 33000. Dann wird sie mit

```
x = 32767! - x
```

als 32767 – 33000 = –233 gespeichert. Auch die Rückwandlung erfolgt als

```
x = 32767! - x
```

doch jetzt ist das 32767 – –233 und »minus minus« ergibt bekanntlich »plus«, ergo gilt 32767 + 233 = 33000.

7.4.1.6 Zahlen bis 255 in einem Byte

Sind die Zahlen kleiner oder gleich 255, kann man sie auch in einem Byte unterbringen. In C gibt es dafür den Typ Char (Zeichen), in Basic machen wir so etwas Ähnliches und nehmen eine Kette von Chars, auch String genannt. So ein String darf in Basic bis zu 32767 Zeichen lang sein. Wir können also im Extremfall mit diesem Trick 32 Kbyte einsparen. Hierzu ein Beispiel bringt Abb.7.4.

```
a$= SPACE$(300)
s=SADD(a$)

FOR i=1 TO 255
   POKE s+i, i
NEXT

WIDTH 60
FOR i=1 TO 255
   PRINT PEEK(s+i);
NEXT
```

Abb. 7.4: So speichert man Zahlen in einem Byte

Mit »a$= SPACE$(300)« wird ein String von 300 Leerstellen angelegt. Das ist wichtig, weil damit auch seine Adresse fixiert wird, jedenfalls fast. Wenn Sie einen neuen String anlegen oder mit »fre(„“)« Basic zwingen, seinen Stringbereich aufzuräumen, kann sich die Adresse verschieben. Dann müssen Sie nochmals »s=SADD(a$)« geben.

Nun werden die Bytes einfach in den String »gepokt« und dann mit PEEK wieder ausgelesen.

7.4.1.7 ' ist länger als REM

Sehr aufwendige Kommentare kosten natürlich auch Platz, jedes Zeichen belegt ein Byte, doch so schön es aussieht, der Hochstrich als REM (übrigens Alt-ä) ist nicht kürzer, im Gegenteil! Das liegt daran, daß AmigaBasic 2–4 Byte (je nach Zeile) braucht, um sich zu merken, daß der Hochstrich für REM steht. Aber Sie können ja immer eine RUN-Version ohne alle Kommentare und eine Variante mit vielen REMs »für Ihre Akten« speichern.

7.4.1.8 Nichts kostet auch Speicher

Leerzeilen im Listing lockern zwar das Bild auf, doch jede Leerzeile kostet 4 Byte. Das liegt daran, daß AmigaBasic ein Programm zeilenweise verwaltet. Die Zeilen liegen aber im RAM hintereinander. Vor jeder Zeile steht ein Zeiger auf den Beginn der nächsten Zeile. Damit kann AmigaBasic beim Suchen nach einer Sprungmarke schnell von Zeile zu Zeile springen, egal wie lang die Zeilen jeweils sind.

Übrigens: Zeilennummern kosten zwei Byte extra, sie sind aber immer kürzer als Labels (Namen für Sprungziele).

7.4.1.9 Brauchen Sie eigene Screens und Windows?

Eigene Screens und Windows kosten Speicher. Prüfen Sie also zuerst, ob Sie nicht mit dem Workbench-Screen und dessen 4 Farben auskommen können, ferner, ob nicht das Standard-Fenster von AmigaBasic reicht. Wenn nicht, so können Sie trotzdem sparen.

Der Speicherbedarf eines Screens steigt mit der Tiefe, und die bestimmt die Anzahl der Farben. Die Anzahl der Farben ist die Tiefe hoch 2. Tiefe 2 ergibt 4 Farben, Tief 3 schon 8 und Tiefe 4 dann 16 usw. Der Wert für die Tiefe ist praktisch die Anzahl der Bitplanes. Jede Bitplane belegt 640x256 Bit, also rund 20 Kbyte.

Bei Windows kommt es auf den Refresh-Typ an. Der speicherteuerste Typ heißt SuperBitMap, in Basic ist das der Typ 17, den Sie immer vermeiden sollten.

7.4.2 Entfernen allgemeiner Speicher-Schlucker

Wenn sich beim Basic-Programm nichts mehr holen läßt, müssen Sie Basic selbst mehr Speicher anbieten, der natürlich woanders eingespart werden muß. Hierfür gibt es einige Möglichkeiten in der »Startup-Sequence«.

Sichern Sie zuerst die vorhandene Startup-Sequence indem Sie sie auf einen Namen wie »ss.org« kopieren. Später können Sie dann mit »copy ss.org to startup-sequence« den alten Zustand wieder herstellen. Vergessen Sie auch nicht, nach jeder Änderung der Startup-Sequence den Amiga neu zu starten. Nun aber:

7.4.2.1 Ohne RAM-Disk

Eine RAM-Disk ist zwar schnell, sie kostet aber auch RAM. Wenn also Dateien nach RAM: oder RAD: kopiert werden, streichen Sie diese Zeilen, erst recht das »mount rad:«. Sie können auch noch im Betrieb alle Dateien in der RAM-Disk löschen. Der Speicher wird dadurch frei. Bei der resetfesten RAM-Disk (RAD:) hilft das Löschen der Dateien nichts, weil die Größe dieser RAM-Disk konstant ist, sie ist dann halt nur leer. Mit »remrad« (remove RAM-Disk) können Sie die »RAD:« allerdings auf ein Minimum schrumpfen lassen. Das »mount pipe:« können Sie auch sparen, wenn Sie nicht Daten von einem Task zum anderen schicken wollen.

7.4.2.2 Ohne RESIDENT

Residente Programme, erkennbar an einem Namen nach dem Schlüsselwort RESIDENT, bleiben zwecks schnellem Zugriff immer im Speicher. Wenn Sie diese Aufrufe löschen, werden die Programme eben von der Diskette geladen, sonst ändert sich nichts. Übrigens, wenn Sie eine Festplatte haben, lohnt sich das RESIDENT kaum, da sollten Sie immer den freien RAM bevorzugen.

7.4.2.3 Ohne extra Puffer

Entfernen Sie alle Zeilen mit »ADD BUFFERS«. Damit werden nur zusätzliche Puffer für die Disketten bzw. Festplatten angelegt. Die extra Puffer können zwar bei diskintensiven Programmen das Tempo steigern, sind aber keinesfalls lebensnotwendig.

7.4.2.4 Ohne Workbench

Sozusagen als letzte und radikalste Maßnahme können Sie auch noch die Workbench einsparen und Basic direkt aus dem CLI heraus starten. Dazu entfernen Sie aus der Startup-Sequence die am Ende stehenden Befehle

```
end cli >nil:
laodwb
```

Nun können Sie mit

```
run amigabasic
```

Basic aus dem CLI heraus starten. Natürlich können Sie auch diese Zeile in die Startup-Sequence eintragen, womit Sie dann sozusagen vollautomatisch ins Basic gelangen.

7.4.2.5 Start im Direktmodus

Da AmigaBasic als Voreinstellung nur 25 000 Byte für den Programmspeicher reserviert, kann es vorkommen, daß längere Programme gar nicht erst laden wollen bzw. schon beim Laden mit »out of memory« aussteigen. In diesem Fall tippen Sie direkt in das Ausgabefenster:

```
clear ,50000: run "NameDesProgramms"
```

Beachten Sie das Komma vor der Zahl! Das Beispiel setzt voraus, daß 50 Kbyte reichen, natürlich können Sie auch eine höhere Zahl eingeben, allerdings nicht viel mehr als 170 000. Probieren Sie größere Zahlen aus, Sie werden einen bildschönen Absturz des Amiga erleben. Noch ein Tip: Nur »Clear« setzt den Wert nicht zurück, Sie müssen dann einen neuen Wert eingeben.

7.5 Basic-Menü ist weg

Fast jedes größere Basic-Programm arbeitet mit eigenen Pull-down-Menüs, doch hier kann es Probleme geben. Das eigene Menü überschreibt das AmigaBasic-Menü. Wenn das Programm nun aus irgendeinem Grunde abbricht, bleibt das eigene Menü erhalten und AmigaBasic läßt sich nicht mehr bedienen. Zwei Lösungen bieten sich an.

Zuerst sollten Sie im Programm immer einen Error-Handler vorsehen, der im Minimum so aussieht:

```
ON ERROR GOTO Fehler
'Ihr Programm hier
Fehler:
MENU RESET
ON ERROR GOTO 0
```

Wichtig ist also das »MENU RESET«, womit das AmigaBasic-Menü wieder eingeschaltet wird. Das können Sie aber auch direkt in das Ausgabefenster eintippen (vorher anklicken), die Wirkung ist die gleiche.

7.6 Basic-Menü soll weg sein

Was im vorigen Tip ein Fehler war, kann auch Absicht sein. Wenn Sie verhindern wollen, daß ein Anwender in Ihrem Programm »herumfuhrwerkt«, lassen Sie die Menüs (oder einige davon) einfach verschwinden. In der Zeile

```
MENU x,0,0,""
```

gilt für x

1 = Projekt-Menü
2 = Edit-Menü
3 = Run-Menü
4 = Windows-Menü

Sogar die Tastenkürzel sind dann unwirksam, so daß zum Beispiel niemand bei ausgeschaltetem Menü 3 Ihr Programm mit <A>+<.> anhalten kann.

7.7 Notfalls mit <Ctrl>+<C>

Der vorherige Tip wirkt leider nur bei Leuten, die dieses Buch nicht gelesen haben. Denn auch wenn das Menü 3 ausgeschaltet wurde, und damit <A>+<.> nicht mehr wirkt, können Sie dennoch ein laufendes Programm stoppen und zwar mit der Tastenkombination <Ctrl>+<C>.

7.8 Wo sind die Locate-Koordinaten?

Der Locate-Befehl in Basic ist recht nützlich, wenn es darum geht, den Cursor auf eine bestimmte Bildschirm-Koordinate zu stellen. Diesen Punkt über den Daumen zu peilen, ist mühsam; das folgende Programm nimmt Ihnen die Arbeit ab oder?

```
WHILE NOT MOUSE(0)
  LOCATE 1,1
  PRINT USING "###";MOUSE(1);
  PRINT USING "#####";MOUSE(2)
WEND
```

Das Programm läuft so lange, bis die linke Maustaste gedrückt wird. In der Schleife werden laufend die Mauskoordinaten angezeigt. Doch Sie sehen sofort, ganz so schön ist das noch nicht, weil die Maus-Koordinaten in Bildpunkten ausgegeben werden, LOCATE aber Text-Koordinaten braucht. Machen wir es also besser:

```
DEFINT x,y
WINDOW 2

WHILE NOT MOUSE(0)
  x = INT(MOUSE(1) / 8 + 1)
  y = INT(MOUSE(2) / 8 + 1)
  LOCATE 1,1
  PRINT USING "x =###";x;
  PRINT USING "   y =###";y
  a$=INKEY$
  IF a$ <> "" THEN
```

```
    LOCATE y,x
    PRINT a$
  END IF
WEND

WINDOW CLOSE 2
```

Durch die Division durch 8 sind wir im Textraster. Als Zugabe ist es noch möglich, eine Taste zu drücken, die an der Stelle des Mauszeigers angezeigt wird.

7.9 Ausgabe-Window gleich aktiv

Sie wissen, daß man normalerweise das Basic-Ausgabe-Window erst anklicken muß, bevor man da etwas eingeben kann. Das ist per Prinzip unpraktisch und geht im vorigen Beispiel überhaupt nicht, weil dort der Mausklick das Programm beendet. Die einfachste Lösung ist die Zeile

```
WINDOW 2
```

Damit wird das Window geöffnet, gelangt nach vorne und wird das aktive Window. Nun könnte es aber sein, daß Sie sich vorher die Fenster für die Direkteingabe und das Listing gerade so schön angeordnet hatten, und jetzt werden beide durch das große Output-Window abgedeckt. Daher wird das Ausgabefenster zum Schluß mit »WINDOW CLOSE 2« geschlossen.

Wenn Sie ein selbstangelegtes Window auch gleich aktivieren wollen, müssen Sie Intuition einsetzen (siehe auch Kapitel 7.31), und das geht so:

```
LIBRARY "intuition.library"
WINDOW 2, "Titel",(20,20)-(200,200),7
ActivateWindow(WINDOW(7))
INPUT "Gib was ein ",a$
```

»ActivateWindow« ist die Intuition-Funktion, die als Parameter die Adresse der Window-Struktur braucht. Genau diese liefert die Basic-Funktion WINDOW(7).

7.10 Dateien sparsamer speichern

Daß AmigaBasic-Programme immer ein Icon haben, ist sicherlich sehr nützlich, kann man doch damit die Programme so schön von der Workbench aus starten (siehe Abschnitt 7.1). Doch leider erzeugt AmigaBasic diese Piktogramme auch für alle Dateien, die Basic anlegt, und so ein Icon gibt es nicht umsonst. Tatsächlich entsteht eine zweite Datei mit der Erweiterung ».info«, und die kostet Platz auf der Disk.

Verhindern können Sie das nicht, doch Sie dürfen diese Info-Dateien ohne weiteres löschen. Probieren Sie es aus.

```
OPEN "test" FOR OUTPUT AS 1
KILL "test.info"
PRINT #1, "Hallo"
CLOSE
```

Klicken Sie nun das Direkt-Fenster an und geben Sie »files« ein. Sie sehen, daß die Datei »test« vorhanden ist, »test.info« aber fehlt.

7.11 Maus-Tips

Die MOUSE-Funktion ist so vielseitig, daß wir sie uns einmal im Detail ansehen sollten. Die Funktion liefert Informationen über die Position des Mauszeigers und über den Status der linken Maustaste. Die rechte Taste kann nur mittelbar über die Menu-Funktion abgefragt werden. Welches Ergebnis die Funktion »x = MOUSE(n)«zurückgibt, hängt von **n** ab. **n** muß ein ganzzahliger Ausdruck zwischen 0 und 6 sein.

MOUSE(0) (Tastenstatus)

meldet den Tastenstatus und speichert gleichzeitig Werte zur Mausposition. Das Ergebnis kann eine Zahl von –3 bis +3 sein, wobei gilt:

```
MOUSE(0) = 0
```

Die Taste ist zur Zeit nicht gedrückt und wurde auch seit dem letzten MOUSE(0)-Aufruf nicht gedrückt.

```
MOUSE(0) = 1, 2 oder 3
```

Die Maustaste ist zur Zeit nicht gedrückt und wurde aber seit dem letzten MOUSE(0)-Aufruf einmal, zweimal oder dreimal gedrückt.

```
MOUSE(0) = -1, -2 oder -3
```

Die Maustaste wird niedergehalten, nachdem sie einmal, zweimal oder dreimal gedrückt wurde. Der Wert bedeutet meistens, daß die Maus zur Zeit bewegt wird.

MOUSE(1), MOUSE(2) (laufende X-Y-Position)

liefern die horizontale X-Koordinate (1) und die vertikale Y-Koordinate (2) zum Zeitpunkt des letzten MOUSE(0)-Aufrufs und zwar unabhängig vom Status der Maustaste.

MOUSE(3), MOUSE(4) (Start-X-Y-Position)

liefern die horizontale X-Koordinate (3) und die vertikale Y-Koordinate (4) zum Zeitpunkt der letzten Tastenbetätigung vor einem MOUSE(0)-Aufruf. Praktisch wird das Programm ständig MOUSE(0) abfragen, und man hat somit die Position des Beginns eines Ziehvorganges.

MOUSE(5), MOUSE(6) (Ende-X-Y-Position)

liefern die horizontale X-Koordinate (5) und die vertikale Y-Koordinate (6) nach folgender Regel:

Wenn die Maustaste nach dem letzten MOUSE(0)-Aufruf gedrückt wurde, wird die Position zum Zeitpunkt des MOUSE(0)-Aufrufs geliefert.

Wurde die Taste hingegen nach dem letzten MOUSE(0)-Aufruf nicht gedrückt, dann wird die Position zum Zeitpunkt des Loslassens der Taste geliefert. Praktisch hat man damit das Ende eines Ziehvorgangs.

Das Programm in Abb. 7.5 bemüht sich, möglichst viele MOUSE-Funktionen einzusetzen. Die drei Textzeilen »halihalo« sollen nur etwas auf den Schirm bringen. Wenn Sie links oben vom Text die linke Maustaste drücken, sie festhalten und nach rechts unten ziehen (also den Text auswählen), wird ein Rahmen um den Text gemalt. Wenn Sie dann an einer anderen Stelle klicken, wird der Rahmen (samt Inhalt) dahin bewegt.

```
CLS: PRINT: PRINT

FOR i=1 TO 3
   PRINT "   halihalo"
NEXT

WHILE MOUSE(0) = 0: WEND 'bis Taste
x1 = MOUSE(3) : y1= MOUSE(4) 'Startpunkt
WHILE MOUSE(0) =-1 : WEND 'solange Taste
x2 = MOUSE(5): y2 = MOUSE(6) 'Endpunkt
LINE (x1,y1)-(x2,y2),,b 'einrahmen

'Box speichern
s=2+(y2-y1+1)*INT((x2-x1)/16+1)
DIM box(s)
GET   (x1,y1)-(x2,y2),box

WHILE 1
WHILE MOUSE(0) = 0: WEND
  PUT (x1,y1), box 'lösche alte Box
   x1 = MOUSE(1) : y1= MOUSE(2)
  PUT (x1,y1), box 'zeichne neue
WEND
```

Abb. 7.5: Die Maus-Funktionen in der Praxis

7.12 Mini-Paint

Das vorige Programm konnte Rechtecke zeichnen, das von Abb. 7.6 erlaubt das Freihand-Malen. Solange Sie die Maustaste drücken, wird analog zur Mausbewegung auf dem Schirm gezeichnet.

```
WINDOW 2
Farbe=1
WHILE 1
  a$=INKEY$
  IF a$="x" THEN WINDOW CLOSE 2: END
  IF a$>="0" AND a$<="3" THEN Farbe=VAL(a$)
  WHILE MOUSE(0) < 0
    PSET (MOUSE(1),MOUSE(2)),Farbe
  WEND
WEND
```

Abb. 7.6: Noch ein Tip für ein Malprogramm

Der Kern des Programm ist diese Zeile:

```
PSET (MOUSE(1),MOUSE(2)),Farbe
```

»PSET« setzt einen Punkt bei (x,y) mit der Farbe »Farbe«. Der ganze Trick ist, daß anstatt von (x,y) die aktuellen Maus-Koordinaten mittels der Funktionen MOUSE(1) und MOUSE(2) eingesetzt werden.

Zusätzlich wird ständig noch die Tastatur abgefragt. Die Zeichen »0« bis »3« werden als Werte für »Farbe« übernommen (0 = Blau, 1 = Weiß, 2 = Schwarz, 3 = Rot). Blau wirkt hier als Radiergummi. Sie können natürlich mit einem eigenen Screen mehr Farben zulassen.

Noch ein Hinweis: Da Basic nicht so schnell ist wie Sie, passiert folgendes: Wenn Sie die Maus schnell bewegen, entstehen gepunktete Linien; fahren Sie langsamer, werden die Linien wieder massiv.

7.13 Schneller scrollen

Der Scroll-Befehl zum Verschieben von Bildschirmbereichen ist sehr leistungsfähig, sein Tempo kann aber noch gesteigert werden. Hier zuerst ein Beispiel:

```
CIRCLE (50,25),40
FOR i=1 TO 3000: NEXT
SCROLL (0,0)-(100,150),0,100
```

Das Programm malt einen Kreis, wartet etwas und schiebt dann den Kreis nach unten.

Und was ist der Trick dabei? Nun, man sollte meinen, daß ein kleiner Bereich schneller gescrollt wird als ein großer, weshalb sich viele Anwender bemühen, den Scroll-Bereich pixelgenau auf das Bild zu legen. Doch das ist falsch. Das Scrollen wird schneller, wenn man den Ausschnitt etwas größer wählt als das zu verschiebende Bild.

7.14 CLI-Befehle in Basic nutzen

Wie man Library-Funktionen aufruft, hatten wir schon im Abschnitt 7.3 kennengelernt. Wenn es um CLI-Befehle geht, ist die DOS-Library zuständig.

7.14.1 Das Prinzip

Zuerst das Wichtigste: CLI-Befehle werden in Basic mittels der DOS-Funktion »Execute« ausgeführt. Diese Funktion läuft aber nur, wenn sich im c-Directory der aktuellen Workbench-Diskette das Programm RUN befindet.

Sollten Sie also mit einer Sparversion von Workbench-Disk arbeiten, der RUN fehlt, kopieren Sie dieses Programm wieder hinzu. Die 2568 Byte machen den Kohl auch nicht fett.

Prinzipiell müssen wir genau das tun, was wir auch »zu Fuß« machen, nämlich ein CLI-Window öffnen und darin CLI-Befehle eingeben. Um ein CLI-Window öffnen zu können, brauchen wir die DOS-Funktion »dOPEN«, und um die zu erreichen, müssen wir die DOS-Library öffnen. Folglich beginnt unser Programm mit

```
DECLARE FUNCTION dOpen& LIBRARY
LIBRARY "dos.library"
```

Um ein CLI-Window zu öffnen, muß man einen String übergeben, der so aussieht:

```
CON:010/020/600/200/CLI-Window
```

Das kennen Sie schon aus dem Kapitel 2, daher ganz kurz: »CON:« heißt, daß wir das Device »Console« nutzen wollen. Die folgenden vier Zahlen sind – je als (x,y) die linke obere und die rechte untere Ecke des Windows. »CLI-Window« ist der Fenstertitel. Wir weisen das Ganze dem String »w$« zu, was dann ergibt:

```
n$ = CHR$(0)
w$ = "CON:010/020/600/200/CLI-Window"+n$
```

Wichtig ist das »n$« am Ende, also das »CHR$(0)«. Übergeben werden nämlich immer C-Strings, und die sind mit einem Null-Byte terminiert. Ferner wird – auch wegen der Sprache C – ein String nie direkt übergeben, sondern immer nur seine Startadresse. Diese erhalten wir in Basic mittels der SADD-Funktion, woraus folgt:

```
h&=dOpen&(SADD(w$),1005&)
```

»1005« ist der File-Modus, hier »vorhandene Datei«, Sie können aber auch 1006 (neu) nehmen. Auf jeden Fall erhalten wir in der Variablen »h&« ein sogenanntes Handle (eine Adresse) auf das Window zurück. Nun erst kann es losgehen mit

```
Execute& SADD("dir"+n$),h&,0
```

Die Funktion Execute erwartet als Parameter

- die Adresse eines C-Strings mit dem Kommando
- das Input-Handle
- das Output-Handle

Unser Befehl heißt »dir«. Er wird durch das angehängte »n$« zum C-String, »SADD« übergibt seine Adresse. Den Sinn der beiden Handles schildere ich gleich, sehen wir uns zuerst einmal das komplette Programm von Abb. 7.7 an.

```
DECLARE FUNCTION dOpen& LIBRARY
LIBRARY "dos.library"

n$=CHR$(0)
w$ = "CON:010/020/600/200/CLI-Window"+n$
h&=dOpen&(SADD(w$),1005&)

Execute& SADD("dir"+n$),h&,0
IF h& THEN dClose h&
LIBRARY CLOSE
LIST
END
```

Abb. 7.7: CLI in Basic: das Prinzip

Das Programm führt den Dir-Befehl aus und bleibt dann so lange im CLI, bis Sie »endcli« eingeben. Der List-Befehl am Ende ist auch so ein Trick. Ich setze den während der Programmerstellung immer ein, um gleich nach der Ausführung eines Programms wieder im List-Fenster zu landen und da ändern zu können.

Nun zu den Handles: Wenn Sie im CLI sind bzw. Basic vom CLI aus gestartet haben (siehe Abschnitt 7.1.3), brauchen Sie gar keine Handles. Sie können dann das ganze Programm kürzen auf

```
LIBRARY "dos.library"
n$=CHR$(0)
Execute& SADD("dir"+n$),0,0
LIBRARY CLOSE
```

(Verkleinern Sie das Ausgabe-Window von Basic, damit es Ihnen nicht das CLI-Fenster ganz abdeckt.) Wenn Sie

```
Execute& SADD("dir"+n$),h&,0
```

schreiben, geben Sie ein Input-Handle an (hier h&) und kein Output-Handle (nur eine Null). Die Folge davon ist, daß Sie im Input bleiben und das CLI mit »endcli« verlassen müssen. Drehen Sie das um, also zu

```
Execute& SADD("dir"+n$),0,h&
```

gelangen Sie automatisch ins Basic zurück, wenn der Befehl ausgeführt ist. Beide Effekte nutzt das Programm von Abb. 7.8 aus.

```
DECLARE FUNCTION dOpen& LIBRARY
LIBRARY "dos.library"

n$=CHR$(0)
CLS
INPUT "Geben Sie CLI-Befehl ein: ",c$

w$ = "NEWCON:010/020/600/200/CLI-Window"+n$
h&=dOpen&(SADD(w$),1005&)

IF c$ =  "" THEN Execute& SADD(c$+n$),h&,0
IF c$ <> "" THEN Execute& SADD(c$+n$),0,h&


PRINT "Wir sind wieder im Basic"
IF h& THEN dClose h&
LIBRARY CLOSE
END
```

Abb. 7.8: Automatisch vom CLI zurück oder nicht

Das Programm fragt nach einem CLI-Befehl. Geben Sie einen solchen ein, wird er ausgeführt, danach sind Sie wieder im Basic. Geben Sie nur <Return>, landen Sie im CLI, können darin arbeiten und kommen mit »endcli« wieder ins Basic zurück.

7.14.2 Gibt es DF1: oder andere Devices?

Das folgende Beispiel löst gleich zwei Probleme. Zum ersten muß ein Basic-Programm feststellen können, ob ein bestimmtes Device – zum Beispiel ein zweites Laufwerk – existiert, zum zweiten sollte ein CLI-Befehl auch sein Ergebnis dem Basic-Programm zur Verfügung stellen können.

Unser zweites Problem ist folgendes: Wir müssen CLI-Befehle über die Funktion »Execute« aufrufen. Diese Funktion gibt einen Wert zurück, nämlich –1, wenn sie ausgeführt wurde, und 0, wenn etwas schiefging. Das von »Execute« ausgeführte Programm hingegegen kann melden, was es will, im Funktionswert schlägt sich das nicht nieder.

Nun haben aber viele CLI-Befehle die schöne Eigenschaft, daß man ihre Ausgabe umleiten kann. Tippen Sie beispielsweise ein:

```
assign df0: exists
```

so erscheint die Meldung »DF0« auf dem Schirm. Ändern Sie das in

```
assign df2: exists
```

und Sie haben kein drittes Laufwerk, so lautet die Meldung:

```
df2: not assigned
```

Ändern Sie den Befehl hingegen in

```
assign >NIL: df2: exists
```

so wird überhaupt keine Meldung ausgegeben, denn jetzt wird sie auf das Blindgerät »NIL:« umgeleitet. Diese Form findet man in den Stapel-Dateien, wo man über »warn« das Ergebnis auswertet. Wir wollen die Meldung aber nicht im Nichts verschwinden lassen, sondern ganz im Gegenteil in einer Datei. Wir schreiben

```
assign >RAM:xxx df2: exists
```

womit das Ergebnis sehr schnell in die Datei xxx auf der RAM-Disk umgeleitet wird. Nun dürfte auch das Listing von Abb. 7.9 schon klarer sein.

```
DECLARE FUNCTION DOpen&() LIBRARY
DECLARE FUNCTION Execute&() LIBRARY
LIBRARY "dos.library"
n$=CHR$(0)
CLS
lw$ = "DF1:"
c$ = "assign >ram:xxx "+lw$+" exists"
w$="CON:10/20/600/200/CLI-Window"+n$
v&=DOpen&(SADD(w$),1006)
e=Execute&(SADD(c$+n$),0,v&)
IF v& THEN CALL DClose(v&)
LIBRARY CLOSE
OPEN "ram:xxx" FOR INPUT AS 1
LINE INPUT #1, a$
CLOSE
IF INSTR(1,a$, "not") THEN
  PRINT "Device "lw$" unbekannt"
  ELSE
    PRINT lw$" existiert"
END IF
END
```

Abb. 7.9: Prüfen, ob ein Gerät vorhanden ist

In »c$« wird der String für Execute zusammengesetzt, das dann wie gehabt ausgeführt wird. Anschließend wird die Datei »RAM:xxx« geöffnet und die darin befindliche Meldung in »a$« eingelesen. Kommt in diesem String das Wort »not« vor, gibt es das Device nicht.

7.15 Schleifen anderer Basic-Dialekte

In Kapitel 3 hatten wir ja schon die DO-Loop-Schleife von Quick-Basic (auf dem PC) vorgestellt. Wenn Ihnen weitere solche Gebilde begegnen, können Sie die auch in AmigaBasic simulieren. Prinzipiell gibt es nur folgende Arten von Schleifen:

- Endlos (DO...LOOP)
- Zählend (FOR...NEXT)
- Abweisend (WHILE...WEND)
- Nicht abweisend (DO...UNTIL)

Sie sehen, daß AmigaBasic die endlose und die nicht abweisende Schleife nicht direkt unterstützt. Die endlose Schleife ist auf zwei Arten nachzuvollziehen, nämlich als

```
Marke:
  REM Schleifenkörper
GOTO Marke
```

Eleganter sieht vielleicht das aus:

```
WHILE 1
   REM Schleifenkörper
WEND
```

Hier ist die Bedingung eine Konstante, daher läuft die Schleife endlos. Die normale WHILE-Schleife ist abweisend, weil sie nicht ausgeführt wird, wenn die Bedingung schon im Eingang nicht erfüllt ist. Eine nicht abweisende Schleife wird mindestens einmal ausgeführt, weil die Bedingung erst am Ende geprüft wird. Typisch ist dafür in anderen Basics die Form

```
DO
   a$=INKEY$
   PRINT "Bin in Schleife"
UNTIL  A$=CHR$(27)
```

Die Schleife läuft so lange, bis die Escape-Taste gedrückt wird. In AmigaBasic schreiben Sie das als

```
a$=""
WHILE  a$<>CHR$(27)
   a$=INKEY$
   PRINT "Bin in Schleife"
WEND
```

Sie sehen, daß man die Bedingung negieren muß (aus »=« wird »<>«). Außerdem muß sichergestellt werden, daß die WHILE-Schleife jetzt mindestens einmal ausgeführt wird. Das wäre nicht der Fall, wenn a$ noch zufällig den Wert »CHR$(27)« hätte. Deshalb wird a$ vorab mittels »a$=““« initialisiert.

Zur FOR-Schleife wäre anzumerken, daß sie in AmigaBasic korrekt arbeitet. »FOR i=1 TO 1« wird einmal ausgeführt, »FOR i=1 TO 0« hingegen nicht. In anderen Basic-Dialekten wird die Schleife dann einmal durchlaufen, sie ist also nicht abweisend, die Bedingung wird erst bei »NEXT« geprüft. Da es Programme gibt, die sich darauf verlassen, sollten Sie bei der Übernahme solcher Listings aufpassen.

7.16 Call ohne CALL

Der Aufruf einer SUB-Routine kann meistens ohne CALL erfolgen. So können Sie anstatt

```
CALL   NameDerRoutine(a,b,c)
```

auch schreiben:

```
NameDerRoutine   a,b,c
```

Das geht allerdings nicht, wenn die Routine keine Parameter hat. In diesem Fall hält Basic den Namen der Routine für eine Sprungmarke. Zwei Lösungen bieten sich an, nämlich

```
CALL NameDerRoutine
```

oder

```
:NameDerRoutine
```

Die Lösung mit dem Doppelpunkt beruht darauf, daß man mehrere Basic-Befehle durch einen Doppelpunkt getrennt in eine Zeile schreiben kann. Empfehlen kann ich diesen »üblen Trick« allerdings nicht, denn das Programm wird damit sehr unübersichtlich. Leute mit PC-Karte dürften sogar total verwirrt sein, denn in MS-DOS-Batchfiles bezeichnet diese Schreibweise eine Marke, also genau das, was wir hier vermeiden wollen.

Ansonsten gilt: Nach THEN und ELSE ist CALL obligatorisch.

7.17 Umlaute und Blanks vermeiden

Zwei Kleinigkeiten erzeugen in Basic Syntax-Fehler. Zuerst mag Basic auch auf dem deutschesten Amiga keine deutschen Umlaute, dito auch nicht einige andere Sonderzeichen in Variablen- und Marken-Namen. Was überhaupt erlaubt ist, steht im Basic-Handbuch. In Strings dürfen Sie aber deutsch schreiben. Zum zweiten liebt Basic keine Leerstellen oder Tabs am Ende einiger Zeilen. So führt ein Blank nach »END IF« mit Sicherheit zum Syntax-Fehler.

7.18 Listing mit Zeilennummern

Zeilennummern braucht man zwar in Basic nur selten, doch oft ist es nützlich, für Dokumentationszwecke, zum Diskutieren mit Kollegen (per Modem oder telefonmündlich) oder auch für Veröffentlichungen so ein Programm mit Zeilennummern zu versehen. Außerdem zeigt so ein Listing sehr klar die Leerzeilen auf, besonders die am Schluß.

```
DECLARE FUNCTION dOpen& LIBRARY
LIBRARY "dos.library"

INPUT "Welches Programm numerieren";p$
p$=":"+p$
SAVE p$,a
```

```
c$="type "+p$+" to "+p$+".n"+" opt n"
n$=CHR$(0)
w$ = "NEWCON:10/20/200/25/Bin im CLI"+n$
h&=dOpen&(SADD(w$),1005&)
Execute& SADD(c$+n$),0,h&
IF h& THEN dClose h&
LIBRARY CLOSE
PRINT "Listing als "p$+".n gespeichert"
END
```

Abb. 7.10: Numeriertes Listing via CLI

Die erste Lösung sieht so richtig raffiniert aus, nutzt sie doch das CLI (siehe Kapitel 7.14) und gleich zwei Optionen des CLI-Type-Befehls. Betrachten Sie diese Zeile aus Abb. 7.10 (und bitte die Blanks nicht beim Abtippen vergessen):

```
c$="type "+p$+" to "+p$+".n"+" opt n"
```

Im CLI würden Sie tippen

```
type xxx to xxx.n opt n
```

Damit wird die Datei »xxx« in die Datei »xxx.n« kopiert und dabei mit Zeilennummern versehen. In Basic wird der Datei-Name in die Variable p$ eingelesen und damit die Kommandozeile für die Execute-Funktion gebildet. Die neue Datei erhält die Erweiterung ».n«. Den Rest kennen Sie schon, doch ein Trick steckt im Listing noch. Normalerweise steht Basic im Verzeichnis Basic, das CLI ist aber nach dem Start immer auf der obersten Disketten-Ebene, es würde also die Datei nicht finden. Daher bekommt »p$« den Pfad »:p$«. Die Folge davon ist, daß sowohl die ASCII-Datei (save p$,a) als auch die mit den Zeilennummern (p$.n) im Root-Directory gespeichert werden. Leider hat der Befehl »save p$,a« auch zur Folge, daß Basic diesen Namen (p$) auch als den aktuellen Programm-Namen in sein Fenster einträgt. Wenn Sie das stört, müßten Sie zum Schluß zum Beispiel

```
SAVE ":basic/list.bas"
```

schreiben, vorausgesetzt, Sie nennen das Programm »list.bas« und speichern es im Verzeichnis »:basic«.

7.18.1 Auch ohne CLI

Wie schon angedeutet, die CLI-Lösung sieht zwar »mächtig gewaltig« aus, aber sie ist gar nicht nötig, wie Abb. 7.11 zeigt.

```
INPUT "Welches Programm numerieren";p$
p$=":"+p$
SAVE p$,a

OPEN p$ FOR INPUT AS 1
OPEN p$+".n" FOR OUTPUT AS 2

i=1
WHILE NOT EOF(1)
  LINE INPUT #1,a$
  PRINT #2,USING "#### ";i;
  PRINT #2,a$
  i = i+1
WEND
CLOSE

PRINT "Listing als "p$+".n gespeichert"
END
```

Abb. 7.11: Die Aufgabe von Abb. 7.10 ohne CLI

Sie sehen, nicht immer ist das CLI die optimale Lösung, Basic ist auch ganz schön leistungsfähig.

7.19 Guru-Meldung in Basic

Eines hat Basis nicht, nämlich einen Befehl zur Erzeugung von Guru-Meldungen. Es sei denn, Sie rechnen POKE dazu, das – richtig falsch angewandt – auch eine Reise nach Indien vermitteln kann. Nun, diesen Typ wollen wir nicht, sondern die Guru-Meldung mit Ausweg, den sogenannten Recovery-Alert. Abb. 7.12 zeigt die Lösung.

```
DECLARE FUNCTION DisplayAlert&() LIBRARY
LIBRARY "intuition.library"

Alert "Der Amiga brennt"
IF Taste& THEN
   PRINT "Linke Maustaste gedrückt"
   ELSE
   PRINT "Rechte Maustaste gedrückt"
END IF

END

SUB Alert(al$) STATIC
SHARED Taste&
```

```
t$ = MKI$(250)+CHR$(20) 'x, y von Text 1
t$ = t$ + al$ + MKI$(1)

t$ = t$ + MKI$(50)+CHR$(50) 'x, y von Text 2
t$ = t$ + "Linke Maustaste" + MKI$(1)


t$ = t$ + MKI$(450)+CHR$(50) 'x, y von Text 3
t$ = t$ + "Rechte Maustaste" + MKI$(0)

Taste&=DisplayAlert&(0,SADD(t$),70)

END SUB
```

Abb. 7.12: Eine Guru-Meldung in Basic

Das Problem an der Geschichte ist der Textstring, hier tt$. Er besteht aus drei Texten, nämlich dem der Subroutine übergebenen al$ sowie aus »Linke Maustaste« und »Rechte Maustaste«. Es können noch mehr Strings sein.

Zu jedem dieser Sub-Strings gehören die Koordinaten für x und y in Bildpunkten, wobei der x-Wert als Integer (mit MKI$) und der y-Wert als Byte (mit CHR$) angegeben werden müssen.

Dann muß jeder Sub-String mit MKI$(1) enden, mit Ausnahme des letzten, der braucht MKI$(0). Im Aufruf der Funktion mit

```
Taste&=DisplayAlert&(0,SADD(t$),70)
```

bedeutet »0«, daß es sich um einen »recoverable alert« handelt, soll heißen, der Amiga bootet nicht neu. Dann folgt die Adresse des Strings und schließlich eine Zahl für die Höhe der Box.

7.20 BasicClip nutzen

Beim Macintosh kann man mit dem Clipboard (der Zwischenablage) alles Mögliche anfangen, beim Amiga auch, und das sogar in Basic. Wenn Sie nämlich in AmigaBasic einen Bereich auswählen und dann Copy aus dem Menü ziehen bzw. <A>+<C> tippen, wird die Auswahl in eine Datei namens BasicClip in die RAM-Disk kopiert. Damit kann man zum Beispiel diese Auswahl drucken, indem man in das CLI wechselt – so ein CLI-Fenster kann ja auch nebenbei schon offen sein – und dann tippt:

```
type ram:BasicClip to prt:
```

Voraussetzung ist allerdings, daß Sie Ihren Drucker vorher per Schalter oder Steuersequenz in den Modus »Auto-LF« bringen. Basic speichert nämlich nach jeder Zeile nur ein CR (Wagenrücklauf), womit alle Zeilen übereinandergedruckt werden.

Das können Sie allerdings auch einfacher haben. Setzen Sie an den Anfang und an das Ende des Bereichs Zeilennummern oder Marken und befehlen dann

```
LLIST 1-2
```

oder

```
LLIST a-b
```

Logisch, daß BasicClip, schließlich eine ganz normale Datei, in der RAM-Disk nicht gelöscht wird, wenn Sie ein anderes Basic-Programm laden. Folglich kann man in einem Programm eine Routine auswählen und kopieren, dann ein anderes Programm laden, und sie dort einsetzen. Gerade beim Amiga ist das sehr nützlich, denn viele Programmteile, wie den Aufbau von Windows und Menüs wird man immer wieder verwenden und dabei nur leicht ändern.

7.21 MAKEDIR in Basic

Beim Umgang mit der RAM-Disk fiel mir auf, daß Basic leider einen wichtigen Befehl vermissen läßt, nämlich MAKEDIR. Auch sonst wäre es ganz praktisch, wenn man irgendwo ein neues Directory anlegen könnte.

Nun, man kann, wie Abb. 7.13 zeigt.

```
DECLARE FUNCTION CreateDir& LIBRARY

LIBRARY "dos.library"

d$=":basic/testdir"
d&=CreateDir&(SADD(d$+CHR$(0)))

IF d& THEN
  Unlock(d&)
  PRINT d$" angelegt"
ELSE
  PRINT "Fehler"
END IF

LIBRARY CLOSE
```

Abb. 7.13: MAKEDIR in Basic

Das Programm nutzt schlicht die DOS-Funktion »CreateDir«. Beachten Sie, daß Sie mit

```
d$=":basic/testdir"
```

einen kompletten Pfad angeben können, das aber nicht müssen.

Zu »CreateDir« gehört unbedingt noch »Unlock«, denn ein neues Directory ist erst einmal »locked« (verschlossen). Würden Sie das »Unlock« weglassen, wird zwar auch das Directory angelegt, doch dann können Sie nicht darauf zugreifen, es sei denn über weitere DOS-Funktionen und mit dem Handle »d&«.

Auch im CLI wird der Versuch, es nur zu listen – zum Beispiel mit »dir all dirs« – nur mit der Meldung »object in use« (in Gebrauch) quittiert.

Ist Ihnen diese Panne aus Versehen unterlaufen, vielleicht durch einen Tippfehler, hilft nur noch eines: den Amiga neu booten, in das CLI gehen und das Directory mit »delete seinName all« löschen.

7.22 Daten blitzschnell kopieren

Wer öfter mit Arrays arbeitet, kennt das Problem. Einen Array in den anderen zu kopieren, dauert deshalb so lange, weil man jedes Element anfassen muß, bei zweidimensionalen Arrays sogar in zwei verschachtelten Schleifen. Das muß nicht sein, weil a) Arrays immer dicht (im Stück) gespeichert sind und b) es eine ExecFunktion gibt, die sehr schnell einen Speicherblock auf eine andere Adresse verschiebt. Die Funktion heißt

```
CopyMem (Quelle, Ziel, Größe)
```

Quelle und Ziel sind Adressen, die Größe wird in Bytes angegeben. Wollen wir einen Array in einen anderen kopieren, dann gilt:

- Beide Arrays müssen mit DIM Speicher reserviert haben.
- Beide Arrays müssen vom selben Typ sein.
- Der Ziel-Array muß gleich oder größer als der Quell-Array sein.
- Die Adresse wird mit VARPTR(ar(0)) eingesetzt.
- Ist OPTION BASE auf 1, gilt VARPTR(ar(1))

Ferner gilt:

- Es können auch Teil-Arrays kopiert werden.
- Es darf innerhalb desselben Arrays kopiert werden.
- Die Länge muß dann nur gleich oder kleiner sein als der Abstand vom Ziel-Index bis zum Array-Ende.

Wie einfach das in der Praxis aussieht, zeigt Abb. 7.14.

```
LIBRARY   "exec.library"
DECLARE  FUNCTION  CopyMem&  LIBRARY

DIM  a%(99),  b%(99)

FOR  i=0  TO  99:  a%(i)=i:  NEXT

e&=CopyMem&(VARPTR(a%(0)),   VARPTR(b%(0)),   200&)

WIDTH  60
FOR  i=0  TO  99:  PRINT  b%(i);  :  NEXT
```

Abb. 7.14: Rasend schnell: Datenkopie mit CopyMem

Die im ROM dahinterstehende Assembler-Routine führt in einer Schleife (mit D0 als Zähler) nur ein

```
move.b    (a0)+,(a1)+
```

aus. Wüßte man hingegen, daß Quelle und Ziel auf Langwort-Grenzen liegen, könnte man mit

```
move.l    (a0)+,(a1)+
```

immer 4 Byte kopieren. Die Schleife läuft dann nur noch über ein Viertel der Länge. Folglich muß auch die Länge ein Vielfaches von vier sein. Genau diesen Fall haben die Entwickler von Exec schon bedacht. Wenn die drei Bedingungen zutreffen, können Sie anstatt »CopyMem« »CopyMemQuick« einsetzen. Versprechen Sie sich allerdings nicht allzuviel davon. Schon »CopyMem« ist so schnell, daß es – gemessen am Tempo von Basic – praktisch 0 Sekunden benötigt.

7.23 Consol-Fenster in reinem Basic

Schaut man sich einmal an, wie in Assembler ein Consol-Window (CON:) geöffnet wird, fällt etwas auf, sieht das doch im Prinzip so aus:

```
move.l   #window,d1
move.l  #1005,  d2
jsr      _LVOOPEN(a6)
...
...
window dc.b 'CON:20/20/200/200/Titel',0
```

»CON:« ist ein Device, und solche Devices kann man auch in Basic öffnen. »OPEN „PAR:"« ist wohl das beliebteste Beispiel dafür. Nun denn, probieren wir es einmal lt. Abb. 7.15 mit »CON:«

```
OPEN "CON:20/20/300/100/Titel"  AS 1 LEN=20

FIELD #1, 20 AS io$
LSET io$="Nun gib mal was ein"
PUT #1

LSET io$=""
GET #1

p  = INSTR(1,io$,CHR$(10))-1
a$ = LEFT$(io$,p)
PRINT a$, LEN(a$)


CLOSE 1
```

Abb. 7.15: So einfach sind Consol-Windows in Basic

Am einfachsten wäre das mit nur-INPUT- oder nur-OUTPUT-Windows, doch für den so häufigen Fall, daß man eine Frage stellt und auf Antwort wartet, gleich zwei Windows zu opfern, ist ja wohl nichts. Ergo nehmen wir einen Random-File. Hier besteht leider das Problem, daß der Puffer (io$) nach »PUT« nicht gelöscht wird. Also machen wir das mit »LSET io$=""«.

Nach dem Einlesen von »io$« hat der auf jeden Fall wegen der FIELD-Anweisung die Länge 20. Die Eingabe wurde aber mit einem CHR$(10) abgeschlossen. Deshalb suchen wir dessen Position im String und können dann mit »a$ = LEFT$(io$,p)« den eingegebenen String ausschneiden.

7.24 Laufschrift ruckfrei

Es gibt ja viele Beispiele für Laufschriften, doch die stammen alle noch aus der »Vor-Amiga-Zeit«. Sie basieren alle darauf, das Basic nur ab einer definierten Cursor-Position Text drucken kann. Das Ergebnis ist dann immer eine etwas ruckelnde Laufschrift. Diesen Effekt kann man in AmigaBasic leicht beseitigen, wenn man die SCROLL-Anweisung einsetzt, wie es Abb. 7.16 zeigt:

```
DEFINT a-z
a$="Das ist Laufschrift"
FOR i=LEN(a$) TO 1 STEP -1
b$=b$+MID$(a$,i,1)
NEXT
```

```
a$=b$+SPACE$(80-LEN(a$))
w=WINDOW(2)/2 - 1

WHILE 1
  FOR i=1 TO w
     IF i MOD 4 = 0 THEN
       LOCATE 10,1
       PRINT MID$(a$,i/4,1)
      END IF
      SCROLL (0,70)-(640,80),2,0
      IF MOUSE(0) THEN END
  NEXT
WEND
```

Abb. 7.16: So wird Laufschrift ruckfrei

Die erste FOR-Schleife hat die Aufgabe, den Text rückwärts zu schreiben, weil ich unbedingt von links nach rechts rollen will. Wenn Sie die andere Richtung bevorzugen, lassen Sie diese Schleife weg, ersetzen in der Zeile danach »b$« durch »a$« und ändern die Scroll-Richtung von 2 in –2.

7.25 Gadget-Abfrage mit einer Zeile

Ein Gadget – hier ein Text, umrahmt von einem Rechteck – kann auch sehr leicht in Basic erzeugt werden. Die Locate-, Print- und Line-Anweisungen machen es möglich. Die Frage ist nur, wie man so ein Gadget – es können ja viele sein, elegant abfragt. Die Lösung zeigt Abb. 7.17.

```
DEF FNinRect(x,y,x1,y1,x2,y2)=x>x1 AND x<x2 AND y>y1 AND y<y2

LOCATE 5,10: PRINT "Klick"
LOCATE 6,10: PRINT "hier"

x1=50: y1=25
x2=130: y2=55
LINE (x1,y1)-(x2,y2),1,b

WHILE NOT MOUSE(0) OR NOT FNinRect(MOUSE(1),
     MOUSE(2),x1,y1,x2,y2) 'gehört noch zur
                           'vorherigen Zeile!


WEND
```

Abb. 7.17: So einfach kann die Gadget-Abfrage sein

Der Funktion »inRecht« werden in x und y die Maus-Koordinaten übergeben, dann folgen die Maße des Rechtecks als links, oben, rechts und unten. Die Funktion ergibt nur dann wahr, wenn sich die Maus innerhalb des Rechtecks befindet.

Die zweite Bedingung ist, daß die Maustaste gedrückt sein muß. Folglich läuft die WHILE-Schleife so lange wie sie nicht gedrückt ist oder die Maus nicht im Rechteck ist.

Die Zeile wurde nur der Klarheit wegen so geschrieben. Nach den Regeln der booleschen Algebra kann man das auch sparsamer schreiben. Es gilt nämlich

```
NOT a OR NOT b = NOT (a AND b)
```

Also können Sie die Zeile so ändern:

```
WHILE NOT (MOUSE(0) AND FNinRect(MOUSE(1),
                    MOUSE(2),x1,y1,x2,y2))
```

Sie sollten diese Lösung einsetzen, weil sie auch schneller ist.

7.26 PEEK und POKE langer Adressen

In C- oder Assemblerlistings findet man ab und zu Adressen, die man mit PEEK und POKE auch in Basic nutzen könnte. Doch tippt man die Adresse mit »&H« ein, meldet Basic nur einen »Overflow«, zu deutsch, die Zahl ist zu groß. Sie ist es wirklich, aber nur für die &H-Funktion. Basic selbst kann mit seinem Typ »Long Integer« (&) durchaus Adressen halten, nur der »&H«-Operator kommt da nicht mit. Abhilfe schafft folgender Trick:

Für eine Zahl wie $DFF096 schreiben Sie:

```
&HDFF0*256+&H96
```

Kleiner Gag am Rande: Kennen Sie schon diese Möglichkeit, in einer Endlosschleife das Wort »Overflow« zu drucken?

```
a$="DFF096"
a&=VAL("&h"+a$)
```

Tatsächlich rast jetzt so lange das Wort »Overflow« über den Schirm, bis Sie das Programm stoppen. Wie das funktioniert? Eigentlich sollte es das gar nicht. AmigaBasic hat einen Bug.

7.27 Basic schneller machen

Es wird oft über das »Tempo« von Basic geklagt, doch oft genug reichen ein paar Änderungen, um ein lahmes Programm wieder flottzumachen.

7.27.1 In der Kürze liegt die Würze

In Kapitel 7.4.1 wird ausführlich geschildert, wie man Programme kürzen kann. Der Anlaß war zwar Speichermangel, doch auf das Tempo wirken sich diese Maßnahmen positiv aus.

Bedenken Sie immer, daß AmigaBasic ein Interpreter ist. Das Programm als solches ist an sich nur Text, auch wenn es im sogenannten Token-Format abgespeichert ist. Was heißt das nun wieder? Wenn Sie ein Programm mit

```
save "name",a
```

sichern, dann, und nur dann, wird es als der Text gespeichert, den Sie auch im List-Fenster sehen. Geht das Programm in den Hauptspeicher, werden sofort nach der Eingabe einer Zeile oder beim Laden von der Disk, alle Schlüsselwörter wie IF, THEN oder GOTO in Zahlen-Codes übersetzt. Diese Codes nennt man Tokens. Dank dieser Tokens wird Basic schneller. Es braucht zum Beispiel nicht mehr die 5 Buchstaben G, O, S, U und B zu GOSUB zusammenzusetzen, sondern liest dafür nur eine Zahl ein. Nichtsdestotrotz ist noch genügend Text in jeder Zeile, und die Tokens sind aus Interpretersicht auch nur Text. Diesen Text liest der Interpreter, versucht daraus einen Befehl mit seinen Parametern zu erkennen und ruft dann die passende Routine in Maschinensprache auf.

Dieser Vorgang wiederholt sich Zeile für Zeile. Manchmal hat der Interpreter es dabei recht einfach. Beginnt nämlich die Zeile mit REM, ignoriert er einfach den Rest und geht zur nächsten Zeile. Zwei Aufgaben aber bleiben, nämlich a) das REM zu erkennen und b) zur nächsten Zeile zu gehen. Auch diese Vorgänge kosten Zeit. Wie Sie leicht erkennen, kostet sogar eine Leerzeile Zeit.

7.27.2 Schleifen freihalten

Aus dem oben geschilderten Prinzip eines Interpreters ergibt sich leider auch, daß alle Zeilen einer Schleife jedesmal neu übersetzt werden. Läuft die Schleife von 1 bis 1000 und hat sie 5 Zeilen, dann werden 5000 Zeilen übersetzt.

Daraus folgt ganz klar, daß nichts in eine Schleife gepackt werden sollte, was da nicht unbedingt hineingehört. Das sind alle Berechnungen, die zum selben Ergebnis führen, dito alle Funktionsaufrufe. In einem solchen Fall rechnen Sie vor der Schleife und speichern das Ergebnis in einer

Variablen. Dito rufen Sie eine Funktion vor der Schleife auf und weisen deren Ergebnis einer Variablen zu.

Bei Formeln müssen Sie besonders aufpassen. Nehmen wir an, die Zahlen im Array a sollen in Prozent zur Basis 4711 umgerechnet werden. Dann wäre es schlecht, zu schreiben:

```
FOR i=1 TO 1000
  a(i) = a(i)/4711*100
NEXT
```

Das schreiben Sie besser als

```
f = 1/4711*100
FOR i=1 TO 1000
  a(i)=a(i)*f
NEXT
```

Stellen Sie also immer bei längeren Ausdrücken fest, was davon eine Konstante ergibt und rechnen Sie die außerhalb der Schleife aus. Bei der Gelegenheit: Wenn Sie in einer Schleife auch nur durch eine einzige Konstante dividieren müssen, dann rechnen Sie außerhalb der Schleife den Reziprokwert aus und multiplizieren mit diesem innerhalb der Schleife. Eine Multiplikation ist nämlich immer schneller als eine Division.

7.27.3 Keine Schleife mit GOTO

Man sieht häufig diese Form der Schleife

```
Anfang:
  Tue dies
  Tue das
GOTO Anfang
```

Wenn Sie so eine Endlosschleife benötigen, nehmen Sie die Form

```
WHILE 1
  Tue dies
  Tue das
WEND
```

Der Grund ist folgender: Eine Marke oder Zeilennummer muß Basic immer erst suchen, und das kostet Zeit. Bei den »echten« Schleifen hingegen (WHILE...WEND und FOR...NEXT) speichert Basic (auch) die Adressen auf dem Stack, so daß ein einziger Maschinenbefehl reicht, um das Sprungziel zu holen.

7.27.4 GOSUB ist schneller als GOTO

Eine Variante des Beispiels im Abschnitt 7.27.3 ist diese Form:

```
Anfang:
ON x GOTO a,b,c
...
END
a:
...
GOTO Anfang
b:
...
GOTO Anfang
c:
...
GOTO Anfang
```

Vergessen Sie diese Lösung sofort und setzen statt dessen »ON X GOSUB ...«, und lassen Sie die Routinen mit RETURN enden. Das GOSUB packt nämlich die Rücksprung-Adresse auf den Stack, so daß sich »RETURN« nur diese Adresse vom Stack holen muß. Auch das geht natürlich bedeutend schneller als das Suchen nach einer Marke.

7.27.5 Benutzen Sie Ganzzahlen

Der Rechner in unserem Rechner, sprich der 68000, beherrscht die vier Grundrechenarten; allerdings kann er nur mit ganzen Zahlen rechnen. Wenn Sie ihm diese anbieten bzw. Basic die entsprechenden Maschinenbefehle ausführt, geht das sehr schnell. Kommazahlen hingegen werden in aufwendigen Unterprogrammen bearbeitet, und die brauchen ihre Zeit.

Also setzen Sie Integerzahlen (%) und lange Integers (&) ein. Das geschieht entweder, indem Sie dem Variablen-Namen die Zeichen »%« bzw. »&« anhängen oder global. Mit

```
DEFINT i-n
```

deklarieren Sie alle Variablen, die mit den Buchstaben i-n und I-N beginnen, als Integers. Sie können natürlich auch »DEFINT a–z« oder »DEFINT x–z« schreiben, doch wenn Sie schon einen Bereich auswählen, sollten Sie »i–n« vorziehen, weil das traditionsgemäß (aus alten FORTRAN-Zeiten) die Buchstaben sind, die Programmierer für Ganzzahlen einsetzen.

Integers werden in einem Wort (16 Bit) gespeichert und können Werte zwischen –32768 und +32767 annehmen.

Beim PC ist »Integer« das Ende der Fahnenstange, doch der 68000 ist ein 32-Bit-Rechner. Deshalb ist hier auch der Typ »Long Integer« (&) von Vorteil. Solche Zahlen werden in einem Langwort

(32 Bit) gespeichert und können Werte zwischen –2,147,483,648 und +2,147,483,647 (2 Milliarden) annehmen. Da können Sie sogar noch zwei Stellen für die Pfennige abstreichen, und es müßte immer noch für Ihr Gehalt reichen. Auch die Langzahlen können Sie mit zum Beispiel

```
DEFLNG i-n
```

gloabl deklarieren.

7.27.6 Die Ganzzahl-Division ist sehr viel schneller

Wenn bei einer Division beide Operatoren Integers im Bereich von –32768...+32767 sind, ist die Ganzzahl-Division deutlich schneller. Sie müssen nur den Backslash (\) anstatt des Schrägstrichs (/) einsetzen.

7.27.7 Bestellen Sie die Müllabfuhr rechtzeitig

In Basic findet ein Vorgang statt, der offiziell »garbage collection« heißt, worunter ein nicht programmierender Amerikaner schlicht und ergreifend »Müllsammlung« oder »Müllabfuhr« versteht. Dahinter steckt folgendes:

Basic verwaltet Strings in einem speziellen Speicherbereich und das auf eine besonders sinnige Weise. Wenn Sie nämlich schreiben:

```
a$="abc"
b$="def"
c$="ghi"
a$="a$ ist nun länger"
```

dann hat Basic ein Problem. Die Strings werden hintereinander abgelegt. Der zweite Wert von »a$« paßt aber nicht mehr in die drei Zeichen von »abc«. Theoretisch müßten jetzt die anderen Strings so weit verschoben werden, bis genug Platz für den neuen »a$« da ist, doch das kostet Zeit. Deshalb legt Basic »a$« einfach am Ende der Liste noch einmal an und merkt sich das. Und das tut es grundsätzlich, also auch dann, wenn der neue String noch in den alten passen würde.

Wenn Sie nun oft genug denselben Strings immer wieder neue Werte zuweisen, entsteht ein immer größerer Stringbereich, in dem lauter ungültige Strings stehen. Das kann natürlich nicht ewig so weitergehen, weil irgendwann der Speicher nicht mehr reicht. Das merkt Basic und fängt an, den Müll wegzuräumen, die berühmte »garbage collection« legt los. Leider sind Basics Müllwerker nicht die schnellsten, besonders dann nicht, wenn Sie viel Müll zu sammeln haben. Das kann nun dazu führen, daß Basic plötzlich eine längere Pause einlegt und das noch an einer Stelle, wo das richtig auffällt. Was kann man tun?

Nun, man muß rechtzeitig die Müllabfuhr bestellen, sprich, die Garbage-Collection erzwingen. Dazu muß man Basic nur auffordern, den freien String-Speicher zu nennen. Dann wird es nämlich

nicht einfach den Rest nennen, sondern erst einmal aufräumen und dann rechnen, was noch frei ist. Diese Maßnahme erreichen Sie ganz einfach durch die Zeile

```
x=FRE("")
```

7.27.8 Räumen Sie auf

AmigaBasic hat die etwas ungewöhnliche Eigenschaft, mit »Save« nicht nur das Programm sondern auch den ganzen Status zu sichern. Wenn Sie nun längere Zeit ein Programm bearbeitet und dabei Variablen angelegt und wieder geändert haben, dann geht auch alles, was Sie schon längst abgehakt und vergessen hatten, mit auf die Disk.

Die Abhilfe: Speichern Sie das Programm als ASCII-Text, wozu Sie im Direktmodus eingeben müssen:

```
save    "NameDesProgramms",a
```

Nun verlassen Sie Basic und rufen es wieder auf. Wenn Sie jetzt das Programm laden, kann es geringfügig länger dauern, denn jetzt muß zusätzlich das »Tokenizing«, die Umwandlung in das Kompaktformat, laufen. Nun sichern Sie das Programm ohne die A-Option.

Kleiner Tip am Rande: Mit »a« gesicherte Dateien bekommen kein Icon.

7.27.9 Arbeiten Sie mit Tabellen

Spätestens, wenn Sie in Basic eine Analog-Uhr programmieren, werden Sie häufig die Sinus- und Cosinus-Funktion benutzen. Wenn Sie dann auch noch mit den gewohnten Grad anstatt mit Radianten arbeiten wollen, wird das eine ganz schöne Rechnerei über viele, viele Punkte.

In diesem Fall ist es sehr empfehlenswert, eine Tabelle einzusetzen.

```
DIM  sinus(90)

'So geht es zwar:

FOR i=0 TO 90
     sinus(i)=SIN(i*3.1415993#/180)
NEXT

'Doch das ist besser:

f=  3.1415993#/180
FOR i=0 TO 90
    sinus(i)=SIN(i*f)
NEXT
```

Gleich zwei Lösungen für dieselbe Aufgabe? Nein, die zweite ist schon die bessere und gleichzeitig die Anwendung der Regel aus Kapitel 7.27.2.

7.27.10 Nehmen Sie die »graphics.library«

Nehmen Sie Routinen aus der »graphics.library« anstatt der Basic-Originale. Einige Routinen wie »DrawEllipse« sind deshalb schneller als ihr Basic-Pendant (hier CIRCLE), weil AmigaBasic sie emuliert, anstatt auf die »graphics.library« zuzugreifen. Andere haben einen etwas umständlicheren Zugriff, der mehr Zeit kostet. Ein Beispiel für den Circle-Ersatz sieht so aus:

```
LIBRARY "graphics.library"
rp& = WINDOW(8)
FOR i%=20 TO 170 STEP 10
  DrawEllipse rp&,i%,i%,20,10
NEXT
```

Echt Tempo macht auch der PRINT-Ersatz, was schlicht daran liegt, daß beim Amiga alle Zeichen gemalt werden, und darauf ist die »graphics.library« spezialisiert. So könnte Ihr Quick-Print aussehen:

```
LIBRARY "graphics.library"
qprint "Das ist ein Text"

SUB qprint(a$) static
   Text WINDOW(8), SADD(a$), LEN(a$)
   IF RIGHT$(a$,1)<>";" THEN PRINT
END SUB
```

Die zweite Zeile im Unterprogramm sorgt für den Zeilenvorschub. Wollen Sie diesen unterdrükken, müssen Sie beim Aufruf ein Semikolon mitgeben, das aber Teil des Strings sein muß. Dummerweise wird das Semikolon hier mitgedruckt, aber das stellen Sie sicherlich leicht ab.

7.28 Tips zu einigen Basic-Befehlen und -Funktionen

Einige Basic-Befehle und -Funktionen haben so ihre kleinen Macken, die Sie kennen sollten. Andere bieten einige wenig bekannte Möglichkeiten, die zu nutzen sich durchaus lohnt.

.1 BEEP

Die Folge

```
PRINT "Achtung, Achtung"
BEEP
```

können Sie kürzer schreiben als

```
PRINT "Achtung, Achtung"CHR$(7)
```

.2 CLOSE

Basic schreibt Daten nicht sofort auf die Disk, sondern erst in einen Puffer. CLOSE schreibt den Puffer auf die Disk und schließt dann die Datei. Erfolgt das bei einer zum Schreiben geöffneten Datei nicht (kleiner Absturz), ist die Datei mit hoher Wahrscheinlichkeit nicht mehr zu gebrauchen, zum Beispiel weil das EOF-Zeichen fehlt. Sie sollten daher eine Datei nie länger offen lassen als unbedingt nötig.

.3 COLOR

Ohne diese Anweisung setzt Basic für die Vorder- und Hintergrundfarbe immer die Werte aus Preferences ein.

.4 CONT

CONT setzt ein gestopptes Programm immer in der nächsten Zeile fort, allerdings mit einer Ausnahme. Eine unterbrochene INPUT-Anweisung wird wiederholt.

.5 DEF FN

Der Versuch, eine rekursive Funktion zu schreiben, wird in AmigaBasic mit einem »out of memory«-Error bestraft.

.6 DELETE

Der Befehl wird auch im Programm akzeptiert, er wird sogar ausgeführt, doch dann wird das Programm abgebrochen.

.7 END

END schließt alle offenen Dateien. Läuft das Programm auf die letzte Zeile, wirkt das auch wie END.

.8 ERL

Die Funktion ERL ergibt die »Error Line«, also die Zeilennummer, in der ein Error aufgetreten ist. Die Zeile ist aber nur ansprechbar, wenn sie auch eine Zeilennummer hat. Hat sie keine, sucht Basic rückwärts, bis es eine Zeilennummer findet und nimmt diese.

.9 EXP(x)

»x« muß eine Zahl einfacher Genauigkeit <=88 und in doppelter Genauigkeit <=708 sein. Bei einem Überlauf meldet AmigaBasic zwar »Overflow«), doch dann setzt es einfach den größtmöglichen Wert (je nach Typ) ein und fährt im Programm fort.

.10 FRE und CLEAR

```
CLEAR ,33000,5000
```

reserviert 33 000 Byte für den Programmspeicher und 5000 Byte für den Stack. Der Rest ist der noch freie Systemspeicher, auf einer 1-Mbyte-Maschine ohne sonstige Tasks typisch 600 Kbyte. Doch leider können Sie die nicht dem Basic-Programmspeicher zuweisen. Bei ca. 200 Kbyte liegt die Grenze.

.11 FOR-NEXT-STEP

Wird der Wert für STEP auf null gesetzt, läuft die Schleife endlos.

.12 GET/PUT (Grafik)

Wenn Sie als Array a% einsetzen, können Sie nach dem GET diese Daten auswerten:

```
a%(0) == Breite des Ausschnitts
a%(1) == Höhe des Ausschnitts
a%(2) == Anzahl der Bit-Planes
```

Das sollten Sie abfragen, weil PUT die Meldung »illegal function call« bringt, wenn das Bild nicht in die aktuelle Größe des Windows paßt. Diese wiederum können Sie mit WINDOW(2) und WINDOW(3) feststellen.

.13 HEX$(x)

Der zulässige Bereich für »x« ist unsymmetrisch. Er reicht von –32768 bis +214783647. Zahlen im Bereich von –65536 bis –32768 führen nicht zu einer Fehlermeldung, aber zu falschen Ergebnissen. Zum Beispiel ergibt HEX$(-65535) dann 1.

.14 INKEY$

INKEY$ liest direkt aus dem Tastaturpuffer. Da dort noch Zeichen stehen könnten, sollten Sie diesen Puffer zuerst leeren. Das geschieht mit

```
WHILE INKEY$ <> "" : WEND
```

.15 INPUT

Das Fragezeichen ist sehr einfach zu unterdrücken.

```
INPUT "Geben Sie was ein: ",a$
```

Das Komma nach dem String macht's. Setzen Sie hier ein Semikolon, erscheint das Fragezeichen.

.16 INPUT$

Um auf einen Tastendruck zu warten, schreiben viele Leute so etwas wie

```
10 a$=INKEY$: IF a$="" THEN 10
```

Das kann man einfacher haben. Den gleichen Zweck erfüllt

```
a$=INPUT$(1)
```

»a$=INPUT$(n,k)« liest n Zeichen von der unter der Nummer k geöffneten Datei. Da alle Zeichen gelesen werden, auch Trennzeichen und Schlußzeichen, kann man hiermit alles lesen und damit auch beschädigte Dateien noch retten. Die Länge der Datei kann mit »L=LOF(k)« festgestellt werden. Da INPUT$ aber nicht den Filepointer aktualisiert (LOC), kann man sich so wie in Abb. 7.18 behelfen:

```
WIDTH 60
OPEN "test" FOR INPUT AS 1
L = LOF(1)
c = 0
WHILE 1
  IF c = L THEN END
  c = c + 1
  a$ = INPUT$(1, 1)
  PRINT a$;
WEND
```

Abb. 7.18: Wenn LOC nicht geht...

.17 LBOUND, UBOUND

Im Gegensatz zu den Arrays, die per Voreinstellung ab Index 0 zählen, zählen LBOUND und UBOUND ab 1. Setzen Sie »OPTION BASE 1« zum Synchronisieren.

.18 LEFT$-, MID$-, RIGHT$-Funktionen

Wird als Länge für den Sub-String null eingesetzt, ist das Ergebnis ein Leerstring.

.19 LIST, LLIST

Es ist manchmal ganz nützlich, unter Umgehung von »PRT:« und des damit verbundenen Filters ein Listing direkt zum Drucker zu senden. Das geht mit

```
LIST, "PAR:"
```

Sie können einen Drucker an der seriellen Schnittstelle aber noch direkter ansprechen bzw. darüber und via Modem sogar Listings verschicken. Das geht so:

```
LIST, "COM1:2400,N,8"
```

.20 LOAD

Wenn Basic eine Datei über »Open« im Projekt-Menü nicht laden will, probieren Sie im Direktmodus

```
load "NameDerDatei"
```

.21 LOCATE

Die möglichen Werte hängen von der aktuellen Größe des Fensters ab. Deshalb sollten Sie vorab prüfen, ob Ihre LOCATE-Parameter noch Sinn machen. WINDOW(2) ergibt die Breite und WINDOW(3) die Höhe des Fensters in Bildpunkten. Diese Werte geteilt durch 8 sind für den Standard-Font die Limits für LOCATE.

.22 LPOS(0)

LPOS hat nichts mit der Position des Druckkopfes zu tun, sondern ergibt die aktuelle Position im Druckerpuffer, und der kann viele Kbyte groß sein.

.23 MENU

Sind die Menü-Titel so lang oder so zahlreich, daß der rechte Titel über den Schirmrand hinaus schreibt, kann es eine Guru-Meldung geben.

.24 MENU(0)

MENU(0) wird nach jedem Aufruf zurückgesetzt, so daß es immer in einer Schleife abgefragt werden muß.

.25 MERGE

MERGE akzeptiert nur Text, also Basic-Programme, die vorher mit »save „Name",a« gesichert wurden.

.26 MID$-Anweisung

»MID$=« kann in einem String einen Teilstring durch einen anderen ersetzen. Letzterer darf aber nie über die Länge des großen Strings »hinausragen«. Dessen Länge wird nicht geändert, ein Einfügen ist also nicht möglich.

.27 OBJECT-Anweisungen und Bobs

Bobs werden in der Reihenfolge ihrer Priorität gezeichnet. Legen Sie diese mit »OBJECT.PRIORITY« fest, sonst ist die Reihenfolge des Auftretens zufällig.

.28 OBJECT.SHAPE

Sie haben die Wahl zwischen »OBJECT.SHAPE Nummer, Definition« und »OBJECT.SHAPE Nummer1, Nummer2«. In der zweiten Syntax wird das Objekt »Nummer1« dupliziert und erhält den Namen »Nummer2«. Diese Methode ist sehr viel schneller und sie spart Speicher.

.29 OBJECT.STOP

Kollidieren zwei Objekte miteinander, wird für beide automatisch die Stop-Anweisung ausgeführt, dito für das Objekt, das gegen eine Fensterkante läuft.

.30 OCT$(x)

Der zulässige Bereich für »x« ist unsymmetrisch. Er reicht von –32768 bis +214783647. Zahlen im Bereich von –65536 bis –32768 führen nicht zu einer Fehlermeldung, aber zu falschen Ergebnissen.

.31 ON ERROR

Tritt im Error-Handler noch ein Error auf, bricht Basic das Programm ab. Das in anderen Basic-Dialekten mögliche »ON ERROR GOTO 0« innerhalb des Error-Handlers ist in AmigaBasic nicht erlaubt.

.32 ON x GOTO/GOSUB a,b,c

Führt der Wert von x nicht zum Sprungziel, wird bei der nächsten Anweisung nach »ON x« weitergemacht. Im Falle von »ON x GOSUB« führt das oft zum Fehler »GOSUB without RETURN«.

.33 PAINT

Die PAINT-Anweisung arbeitet nur mit einem Window vom Typ 16.

.34 PATTERN

Obwohl der Array mit dem Muster nur 4 Elemente hat (0...3), muß er dimensioniert werden. Andernfalls meldet Basic »illegal function call«.

.35 PEEK und POKE

In den Formen mit »W« und »L« muß die Adresse eine gerade Zahl sein. Andernfalls meldet Basic einen »illegal function call«.

.36 POINT

POINT stellt die Farbe eines Punktes fest. Der Wert –1 bedeutet allerdings nur, daß der angegebene Punkt außerhalb des Fensters liegt.

.37 REM

REM oder »'« ist auch am Ende von DATA-Zeilen nicht zulässig.

. 38 RESUME

RESUME ohne Angabe einer Marke oder »RESUME 0« sollten Sie vermeiden. Dies führt nämlich zur fehlererzeugenden Zeile zurück. Kann dort der Schaden nicht behoben werden, hängt sich das Programm auf.

.39 RND(n)

In anderen Basic-Dialekten erzeugt RND(n) eine Zufallszahl zwischen 1 und n. In AmigaBasic können Sie das nachbilden mit

```
INT(RND*(n+1))
```

.40 RUN

```
RUN "ProgName"
```

ist CHAIN immer dann vorzuziehen, wenn keine Variablen an das aufgerufene Programm zu übergeben sind.

.41 SHARED

Wenn Variablen erst im Sub-Programm mit SHARED als global deklariert werden, trägt das zur Unübersichtlichkeit bei. Sie können aber mit »DIM SHARED« auch einfache Variablen im Hauptprogramm global deklarieren.

.42 SPC(n)

SPC(n) in einer PRINT-Anweisung gibt n Leerstellen aus. Allerdings: SPC(n) am Ende der Print-Liste unterdrückt den Zeilenvorschub. Das erreichen Sie einfacher mit einem Semikolon.

.43 STR$(x)

Unabhängig vom Typ von x stellt der erzeugte String immer eine Zahl einfacher Genauigkeit dar.

.44 SUB xxx STATIC

Weil das STATIC noch Pflicht ist, sind keine rekursiven »Subs« möglich. Deshalb können sich auch nicht zwei »Subs« abwechselnd gegenseitig aufrufen.

.45 TAB(n)

Ist n kleiner als die aktuelle Schreibposition, wird der Cursor um einen Tabulator zurückgestellt. Allerdings geht er dabei gleichzeitig auf die nächste Zeile.

.46 TAN

Bei einem Überlauf meldet AmigaBasic zwar »Overflow«, doch dann setzt es einfach den größtmöglichen Wert (je nach Typ) ein und fährt im Programm fort.

```
IF " "+x$ <> STR$(VAL(x$)) THEN Fehler
```

.47 WRITE

Wenn es Sie stört, daß »PRINT a« immer eine Leerstelle vor die Zahl setzt, schreiben Sie doch einfach »WRITE a«.

.48 RESTORE

Wird beim RESTORE-Befehl eine Zeile/Marke angegeben, in der keine DATA-Anweisung steht, wird die nächstfolgende DATA-Zeile eingesetzt.

.49 SADD

Wegen der dynamischen Speicherverwaltung ist die Adresse nur so lange gültig, wie keine neuen String-Zuweisungen erfolgen. Auch wenn »a$« schon existiert, kann eine neue Zuweisung an »a$« alle Strings verschieben.

.50 SCREEN

Wenn Sie einen eigenen Screen definieren, müssen Sie auch (mindestens) ein eigenes Window anlegen. Der von ihm belegte Speicher wird erst mit «SCREEN CLOSE n« freigegeben.

.51 SCROLL

SCROLL wird schneller, wenn der Ausschnitt etwas größer ist als das zu verschiebende Bild.

.52 SOUND WAIT

»SOUND WAIT« wartet nicht einfach, sondern »spielt« die Töne in eine Warteschlange, also einen Puffer, dessen Größe begrenzt ist. Erfolgt nicht rechtzeitig ein »SOUND RESUME«, bringt Basic eine Alarm-Meldung, in der es über Speichermangel klagt.

.53 VAL(x$)

Ist das Ergebnis der VAL-Funktion Null, muß das nicht stimmen. Testen Sie das besser mit

.54 VARPTR

Wegen der dynamischen Speicherverwaltung ist die Adresse nur so lange gültig, wie keine neuen Variablen angelegt werden.

.55 WIDTH

Die Weite muß immer um 2 Zeichen kleiner sein als die Fensterbreite (WINDOW(2)\8). Sonst ist der letzte Buchstabe einer Zeile mit der vollen Länge nicht lesbar.

.56 WIDTH „COM1:", n

Dieser Befehl ändert die Größe des COM1-Puffers (üblicherweise 255) überhaupt nicht. Statt dessen wird immer nach n Bytes ein Return eingefügt.

.57 WINDOW-Anweisung

Der Typ 1 kostet immer soviel Speicher, als ob das Window die volle Schirmgröße hätte. Der Typ 16 – das Super-Bitmap-Window – kostet viel extra Speicher und sollte nur verwendet werden, wenn das Window immer verkleinert und vergößert wird, und dennoch sein Inhalt erhalten werden soll. Der zweite Grund: Die PAINT-Anweisung arbeitet nur mit einem Window vom Typ 16.

7.29 ACBM- und IFF-Bilder in Basic

Der Umgang mit IFF-Bildern – also Grafiken im Amiga-Standard-Format – ist auch in Basic kein Problem. Alle dafür erforderlichen Programme gibt es nämlich schon im Verzeichnis »BasicDemos« auf der Extras-Diskette. Interessant davon sind:

```
LoadACBM            : Lädt eine Grafik im ACBM-Format
LoadILBM-SaveACBM   : Konvertiert IFF- in das ACBM-Format
ScreenPrint         : Druckt eine Grafik im ACBM-Format
```

Um diese Programme einfach nutzen zu können, kopieren Sie sie in das Verzeichnis, das auch Ihre übrigen Basic-Programme, die IFF-Bilder und AmigaBasic enthält. Andernfalls müßten Sie die entsprechenden Pfadnamen angeben und ggf. Disketten wechseln. Kopieren Sie wirklich und verschieben Sie nicht nur, denn wir wollen die Programme ändern, und vielleicht brauchen Sie die Originale auch.

Ferner benötigen Sie die folgenden BMAPs im selben Verzeichnis oder in »LIBS« (siehe Kapitel 7.3.4):

```
exec.bmap
dos.bmap
graphics.bmap
```

So, jetzt kann es losgehen. Zuerst wollen wir uns ein Unterprogramm besorgen, das eine Hardcopy des Schirms drucken kann.

ScreenPrint ändern

Starten Sie also Basic und laden Sie das Programm »ScreenPrint«. Von diesem löschen Sie alle Zeilen, die nur die Demo erzeugen. Das ist alles bis zur Marke »ScreenDump:«. Kurz darauf sehen Sie die Deklaration der Library-Funktion »AllocMem«. Diese Zeile müssen Sie löschen, weil sie in unserem späteren Hauptprogramm schon vorkommen wird. Knapp eine Window-Seite weiter finden Sie die Zeile

```
IF BorderFlag% = 0 THEN 'No Borders
```

Löschen Sie diese eine Zeile und das 5 Zeilen weiter folgende »END IF«. Wir wollen nämlich immer ohne Rahmen drucken. Damit wäre das Unterprogramm fertig. Um es später »mergen« zu können, speichern Sie es als ASCII, wofür Sie im Direkt-Window eintippen:

```
save "Sdump",a
```

LoadACBM ändern

Jetzt laden Sie das Programm »LoadACBM«. Hier können Sie erst einmal die zahlreichen Kommentarzeilen am Anfang löschen, womit Sie auch Platz gewinnen. Etwa 3 Window-Seiten weiter finden Sie Marke »MCleanup:« gefolgt von einer FOR-Schleife, die nur Wartezeit erzeugt. Diese Schleife löschen Sie und setzen statt dessen ein:

```
GOSUB ScreenDump
```

Dieses Unterprogramm müssen wir noch einbauen, also gehen Sie an das Ende des Listings, setzen Sie den Cursor in die nächste Leerzeile, und tippen Sie im Direkt-Fenster ein:

```
merge "Sdump"
```

Jetzt sichern Sie das ganze Programm unter dem Namen »DumpACBM«, womit Sie ein Programm haben, das Grafiken im ACBM-Format drucken kann. Das ACBM-Format ist schneller als das IFF-Format.

IFF-Bilder drucken

Wenn Sie IFF-Bilder – zum Beispiel die von DPaint – drucken wollen, können Sie diese mit »LoadILBM-SaveACBM« in das ACBM-Format konvertieren und dann unser »DumpACBM« einsetzen. Sie können sich aber auch leicht ein Routine zum Drucken von IFF-Bildern in Ihr Programm einbauen. »LoadILBM-SaveACBM« macht nämlich folgendes: Es lädt ein IFF-Bild, zeigt es an und speichert es im ACBM-Format. Folglich können Sie das ganze Programm nehmen, das Unterprogramm »SaveACBM« weglassen, die zahllosen REMs auch und den Rest (hauptsächlich das UP »LoadILBM«) nach der obigen Methode in Ihr Programm einbauen.

7.30 Fehler richtig abfangen

Basic bietet mit seinen »ON ERROR GOTO« eine sehr elegante Methode der Fehlerbehandlung, doch einige Tips dazu seien angebracht.

Zuerst dürfen Sie nie etwas tun, was in anderen Basic-Dialekten üblich ist, nämlich im Error-Handler (der Fehlerbehandlungsroutine) ein »ON ERROR GOTO 0« einbauen. Doch schauen wir uns einmal in Abb. 7.19 an, was geht.

```
ON ERROR GOTO Fehler

Quelle:
INPUT "Name der Quelldatei: ",q$
OPEN q$ FOR INPUT AS 1

Ziel:
INPUT "Name der Zieldatei: ",z$
Ziel1:
OPEN z$ FOR OUTPUT AS 2
```

```
WHILE NOT EOF(1)
  LINE INPUT  #1,a$
  PRINT #2,a$
WEND

END

Fehler:
BEEP
IF ERR=53 THEN
  PRINT "Finde Datei "q$" nicht"
  RESUME Quelle
END IF

IF ERR=57 THEN
  PRINT"Datei ist geschützt"
  PRINT "Bitte anderer "; : REM Name...
  RESUME Ziel
END IF

IF ERR=70 THEN
  LOCATE 10,1
  PRINT "Diskette schreibgeschützt"
  PRINT "Schutz entfernen und Disk wieder einlegen"
  RESUME Ziel1
END IF

PRINT "Fehler ";ERR
RESUME 0
```

Abb. 7.19: So schreibt man einen Error-Handler

Das Programm soll schlicht eine Textdatei kopieren. Bei der Frage nach dem Quell-File kann ein falscher Name oder Pfad eingegeben werden, die Zieldatei oder die ganze Diskette kann geschützt sein. Diese Fehler – es gibt noch mehr – wollen wir abfangen, die anderen sollen nur gemeldet werden.

Dabei fällt Ihnen sicherlich auf, daß beim Fehler 70 (Diskette schreibgeschützt) nicht bei der Eingabe-Aufforderung weitergemacht wird. Der Grund ist, das DOS vorher einen Requester mit der Meldung »Volume <Name> is write protected« erscheinen läßt. Dann können Sie den Schreibschutz entfernen und müssen nur die Diskette wieder einlegen (der Klick auf »Retry« ist unnötig). Damit auch der nicht englischsprachige Anwender weiß, was gemeint ist, geben wir dazu noch die Übersetzung aus. Die muß aber mit »LOCATE« auf die Zeile 10 gesetzt werden, sonst würde der Requester den Text abdecken.

7.31 Die Window-Struktur

Lassen Sie uns dieses Kapitel beenden, indem wir Basic verlassen, uns Intuition zuwenden und dann doch wieder in Basic landen. Wie das geht? Nun – lesen Sie weiter.

Intuition unterscheidet u.a. zwei Strukturen (Sammlung von Daten in einem Bereich), nämlich »NewWindow« und »Window«. Wenn Sie den umständlichen Weg über die Intuition-Library nehmen, müssen Sie eine NewWindow-Struktur definieren (Daten in den Speicher poken) und mit deren Adresse die Funktion »OpenWindow« aufrufen. Daraufhin legt Intuition die wesentlich größere Window-Struktur an, und nur damit arbeitet es weiter. Einfacher können Sie in Basic ein Window mit der WINDOW-Anweisung öffnen, auch damit entsteht die Window-Struktur. Die Adresse der Window-Struktur dieses Fensters (immer des aktuellen Fensters) können Sie mit der WINDOW(7)-Funktion ermitteln. Die ergibt die Anfangsadresse mit dem Offset 0.

In Abb. 7.20 ist die ganze Window-Struktur aufgeführt, die Erklärungen zu den einzelnen Einträgen kommt gleich. Zuerst sollten Sie folgendes wissen:

In der Spalte »C-Schreibweise« stehen die Original-Namen von Commodore. Die zu kennen, kann nicht schaden. Innerhalb dieser Spalte gilt immer die Folge »Typ Name_der_Variablen;«. Typen sind SHORT, BYTE, ULONG usw. Schon in der ersten Zeile sehen Sie aber auch das Wort »struct«. Das heißt, daß der folgende Typ selbst wiederum eine Struktur ist. In C kann man neue Typen definieren. Folglich heißt

```
struct Window *NextWindow;
```

Es gibt eine Variable vom Typ Window-Struktur mit dem Namen »*NextWindow«. Der Stern vor einem Namen hat wiederum eine besondere Bedeutung, nämlich daß es sich hierbei um einen Zeiger handelt. Ein Zeiger ist nichts weiter als eine Adresse. Konkret heißt das hier, daß in diesem Eintrag notiert ist, bei welcher Adresse im RAM die nächste Window-Struktur beginnt. Nun müssen Sie nur noch wissen, daß es vereinbarungsgemäß eine Adresse 0 nicht gibt, sondern die Null immer bedeutet, daß der Zeiger nirgendwohin oder in Nichts (NIL) zeigt. In diesem Fall heißt das, daß es kein weiteres Window mehr gibt.

7.31.1 PEEK und POKE und Typen

Da WINDOW(7) die Anfangsadresse der Window-Struktur liefert, würde ein

```
PRINT PEEKL( WINDOW(7) )
```

nur die Adresse des nächsten Windows ergeben. Auf alle anderen Werte müssen Sie unter Angabe des Offsets (siehe diese Spalte) zugreifen. Beispielsweise erfahren Sie die linke obere Ecke des Windows mit

```
PRINT PEEKW(WINDOW(7)+4)
```

Wichtig ist immer, daß Sie dabei den Typ berücksichtigen. Dabei gilt in dieser Spalte

Typ	Basic-Typ	Peek mit	Poke mit
Byte	*)	PEEK	POKE
INT	%	PEEKW	POKEW
LNG	&	PEEKL	POKEL
Zeiger	&	PEEKL	POKEL

*) Den Typ Byte gibt es in Basic nicht, Sie können aber mit POKE, das byte-weise arbeitet, direkt Zahlen im Bereich von 0-255 schreiben bzw. sie mit PEEK lesen.

C-Schreibweise	Offset	Typ
struct Window		
{		
struct Window *NextWindow;	0	Zeiger
SHORT LeftEdge;	4	INT
SHORT TopEdge;	6	INT
SHORT Width;	8	INT
SHORT Height;	10	INT
SHORT MouseY;	12	INT
SHORT MouseX;	14	INT
SHORT MinWidth;	16	INT
SHORT MinHeight;	18	INT
SHORT MaxWidth;	20	INT
SHORT MaxHeight;	22	INT
ULONG Flags;	24	LNG
struct Menu *MenuStrip;	28	Zeiger
UBYTE *Title;	32	Zeiger
struct Requester *FirstRequest;	36	Zeiger
struct Requester *DMRequest;	40	Zeiger
SHORT ReqCount;	44	INT
struct Screen *WScreen;	46	Zeiger
struct RastPort *RPort;	50	Zeiger
BYTE BorderLeft;	54	Byte
BYTE BorderTop;	55	Byte
BYTE BorderRight;	56	Byte
BYTE BorderBottom;	57	Byte
struct RastPort *BorderRPort;	58	Zeiger
struct Gadget *FirstGadget;	62	Zeiger
struct Window *Parent;	66	Zeiger
struct Window *Descendant;	70	Zeiger
USHORT *Pointer;	74	Zeiger
BYTE PtrHeight;	78	Byte
BYTE PtrWidth;	79	Byte
BYTE XOffset;	80	Byte
BYTE YOffset;	81	Byte

```
    ULONG IDCMPFlags;                82     LNG
    struct MsgPort *UserPort;        86     Zeiger
    struct MsgPort *WindowPort;      90     Zeiger
    struct IntuiMessage *MessageKey;94     Zeiger
    UBYTE DetailPen;                 98     Byte
    UBYTE BlockPen;                  99     Byte
    struct Image *CheckMark;        100     Zeiger
    UBYTE *ScreenTitle;             104     Zeiger
    SHORT GZZMouseX;                108     INT
    SHORT GZZMouseY;                110     INT
    SHORT GZZWidth;                 112     INT
    SHORT GZZHeight;                114     INT
    UBYTE *ExtData;                 116     Zeiger
    BYTE *UserData;                 120     Zeiger
  };
```

Abb. 7.20: Die Window-Struktur

In Abb. 7.20 haben die einzelnen Einträge folgende Bedeutung:

*NextWindow

Existieren mehrere Windows, ist das der Zeiger auf das nächste. Steht hier null, gibt es keines mehr.

LeftEdge, TopEdge, Width und Height

Damit wird die Lage des Windows und seine Größe beschrieben. In Abb. 7.21 entspricht

```
LeftEdge = X
TopEdge  = Y
Width    = Breite
Height   = Höhe
```

Beachten Sie bitte, daß bei Intuition alle Positionen relativ zum übergeordneten Container sind. Ein Window befindet sich im Container Screen, ein User-Gadget würde sich im Container Window befinden. Beginnt beispielweise ein

```
          Screen bei 0,20   (x,y)
und ein   Window bei 30,40  (x,y)
```

so liegt das Window relativ zum Screen-Nullpunkt bei 30,60.

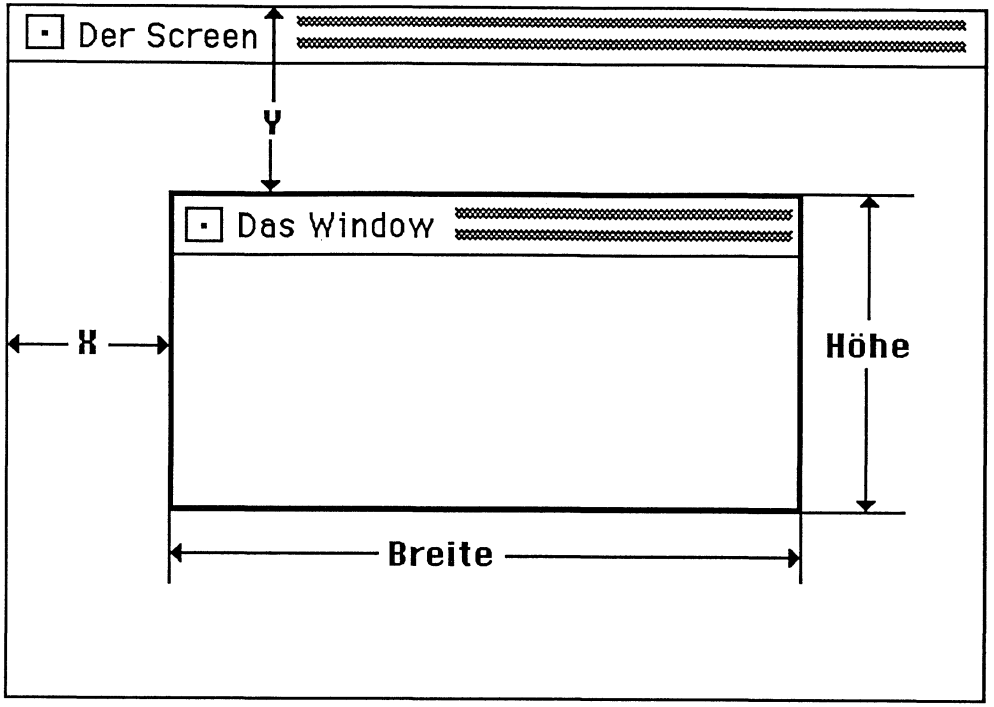

Abb. 7.21: Lage und Maße eines Windows

MouseY, MouseX

Hier können Sie die aktuelle Position des Mauszeigers erfragen, vorausgesetzt, es wurde beim Aufruf von »NewWindow« das Flag REPORTMOUSE gesetzt.

MinWidth, MinHeight

Das sind die Breite und die Höhe des Windows, die beim Verkleinern des Fensters nicht unterschritten werden können.

MaxWidth, MaxHeight

Das sind die Breite und die Höhe des Windows, die beim Vergrößern des Fensters nicht überschritten werden können.

Flags

Hier wären zuerst die Flags zum Setzen der System-Gadgets zu nennen, also

```
WINDOWSIZING : Größe des Windows ändern = 1
WINDOWDRAG   : Window verschieben       = 2
WINDOWDEPTH  : Window -> hinten/vorne   = 4
WINDOWCLOSE  : Window schließen         = 8
```

Die Zahlen am Ende jeder Zeile sind Bit-Werte der Flags. Mehrere Flags können gleichzeitig gesetzt werden, indem man diese Zahlen addiert. Alle diese Flags zusammen ergeben also einen Wert von 15. Sie können das Flagwort mit PEEKL auslesen und werden wahrscheinlich eine größere Zahl sehen, weil noch allerhand Bits dazukommen können, nämlich eine der folgenden Refresh-Arten:

```
SIMPLE_REFRESH =  64
SMART_REFRESH  =   0 (Standard)
SUPER_BITMAP   = 128
```

Ferner wird auch noch der Typ da notiert, wobei gilt:

```
BORDERLESS     = 2048
BACKDROP       =  256
GIMMEZEROZERO = 1024
SUPER_BITMAP   =  128
```

Keine Angaben zum Typ heißt »Standard-Window«. Beachten Sie, daß SUPER_BITMAP sowohl ein Typ als auch eine Refresh-Art ist.

Nun hätten wir aber noch einige Flags:

```
REPORTMOUSE    (512) erlaubt die Mausposition abzufragen.
ACTIVATE       (4096) sorgt dafür, daß das Window nach
               dem Öffnen aktiv ist.
NOCAREREFRESH (131072) heißt keine Meldung, wenn das Window
                       Refresh benötigt.
```

*MenuStrip

Hier notiert sich Intuition die Adressen der Menü-Struktur. Im Normalfall greifen Sie nicht auf diese Felder zu.

*Title

Zeiger auf einen Text, der der Window-Titel sein soll. Es muß ein C-String sein (mit 0-Byte terminiert). Ist der Text länger als das Window breit ist, wird der Rest abgeschnitten.

***FirstRequest, *DMRequest, ReqCount**

Hier notiert sich Intuition die Adressen der Requester-Listen. Im Normalfall greifen Sie nicht auf diese Felder zu.

***WScreen**

Hier steht die Adresse des Screens, 0 bedeutet, daß das Window den Workbench-Screen nutzt.

***RPort**

Diesen Zeiger auf den Rastport erhalten Sie einfacher mit der Basic-Funktion WINDOW(8). Siehe auch Kapitel 7.32.

BorderLeft, BorderTop, BorderRight, BorderBottom

Die Window-Grenzen. Wenn Sie mit einem GZZ-Window arbeiten, finden Sie hier die Lage der vier Kanten des »inner Window«.

***BorderRPort**

Ein Zeiger auf den RastPort der Fenstergrenzen.

***FirstGadget**

»*Gadget« ist ein Zeiger auf das erste User-Gadget in einer Liste von Gadget-Strukturen.

***Parent, *Descendant**

Hier notiert sich Intuition, welche Windows Vorgänger und Nachfolger sind.

***Pointer, PtrHeight, PtrWidth, XOffset, YOffset**

Der Mauszeiger ist ein Sprite. *Pointer zeigt auf die Sprite-Daten.

IDCMPFlags

Diese Flags sagen Intuition, welche Ereignisse es den Task melden soll. Am wichtigsten ist zu wissen, wann der User das Fenster schließt. Um das abfragen zu können, muß man das Flag CLOSEWINDOW setzen.

***UserPort, *WindowPort**

Intuition legt beim Öffnen diese beiden Ports an. Über den UserPort werden Nachrichten empfangen (zum Beispiel Gadget betätigt), über den WindowPort kann man Nachrichten senden.

DetailPen, BlockPen

Mit dem DetailPen werden im Window Feinheiten wie die Gadgets oder Text gezeichnet. Mit dem BlockPen werden Flächen gefüllt, zum Beispiel der Hintergrund in der Titelleiste. Die Zahlen sind die Nummern der Farbregister. Eine Standard-Kombination wäre 0 und 1, mit –1 wird der Wert der Screen-Pens übernommen.

***CheckMark**

Ein Menü-Punkt kann mit einem Häkchen markiert werden. Gefällt Ihnen dieses Häkchen nicht, müssen Sie hier einen Zeiger auf ein Image mit Ihrer eigenen Grafik einbauen.

***ScreenTitle**

Zeiger auf den Text des Screen-Titels.

GZZMouseX, GZZMouseY

Position des Mauszeigers im Falle eines Gimmezerozero-Windows.

GZZWidth, GZZHeight

Breite und Höhe des inneren GZZ-Windows

***ExtData, *UserData**

Zeiger auf Daten, die der Anwender an Window binden will.

7.32 Die RastPort-Struktur

Der »RastPort« ist eine Datenstruktur, in der die aktuelle Einstellung der Grafikumgebung notiert ist. Ob die Zeichenfarbe, die Stilart des Textes oder der Zeichenabstand, alles steht im RastPort. Ein Window braucht einen RastPort, schon damit es selbst gemalt werden kann, weshalb es die Adresse seines RastPorts kennen muß. Diese Adresse können Sie sich mit der Basic-Funktion »WINDOW(8)« beschaffen, und zwar immer für das gerade aktive Window. Der Einfachheit

halber und weil viele der Einträge in Basic und auch überhaupt nicht direkt nutzbar sind, bringt Abb. 7.22 nur eine kleine Auswahl.

Offset	Typ	Bedeutung
25	Byte	Vordergrund-Farbe
26	Byte	Hintergrund-Farbe
27	Byte	Randfarbe
28	Byte	Zeichenmodus
36	INT	Grafik-Cursor X-Position
38	INT	Grafik-Cursor Y-Position
52	Zeiger	Zeiger auf Font
58	INT	Texthöhe
60	INT	Textbreite
62	INT	Text-Basislinie

Abb. 7.22: Eine Auswahl aus der RastPort-Struktur

7.33 Anwendung der Window- und RastPort-Strukturen

Sie können alle Daten in der Window- und der Rastport-Struktur abfragen, und viele davon sind auch recht nützlich. Beachten Sie, daß einige der Daten auch mit der WINDOW-Funktion abgefragt werden können, ein direktes PEEK ist aber schneller. Besonders wenn die Abfrage in einer Schleife erfolgt, sollten Sie das direkte »peeken« vorziehen.

Sie können in alle Felder neue Werte »poken«, werden aber in einigen Fällen keine Reaktion feststellen, in anderen ganz üble, nämlich wenn die Werte oder der Typ falsch sind.

7.33.1 Stilarten 1

Die Stilart der Schrift läßt sich mit einem einfachen POKE ändern und zwar mit

```
POKE WINDOW(8)+56,s%
```

Für »s%« können Sie folgende Werte einsetzen

```
0 = normal
1 = unterstrichen
2 = fett
4 = kursiv
```

Tatsächlich handelt es sich hierbei um Bits, von denen auch mehrere gleichzeitig gesetzt sein dürfen. Praktisch müssen Sie dazu die Zahlen nur addieren. So ergibt 6 (2+4) fette Kursivschrift.

7.33.2 Stilarten 2

Die eleganteste Möglichkeit, in Basic verschiedene Stilarten einzusetzen, bietet die Grafik-Library-Funktion »SetSoftStyle«. Abb. 7.23 zeigt ein Beispiel.

```
LIBRARY "graphics.library"
DECLARE FUNCTION AskSoftStyle% LIBRARY
m%=AskSoftStyle%(WINDOW(8))
IF m%=0 THEN END
FOR i=1 TO 9
  READ s%, a$
  SetSoftStyle& WINDOW(8),s%,m%
  PRINT a$
NEXT
DATA 0,normal
DATA 1,unterstrichen
DATA 2,fett
DATA 4,kursiv
DATA 3,unterstrichen und fett
DATA 6,"fett und kursiv"
DATA 5,unterstrichen und kursiv
DATA 7,alles zusammen
DATA 0,wieder Normalschrift
```

Abb. 7.23: Stilarten ohne POKE

Sicherheitshalber sollte man vorher mit »AskSoftStyle« fragen, ob die Stilart eines Fonts auch geändert werden darf. Für diesen Fall bricht das Programm einfach mit END ab. In diesem Beispiel (Standard-Basic-Font) können Sie aber die Funktion »AskSoftStyle« auch weglassen und dann in »SetSoftStyle« für das »m%« –1 einsetzen.

In den ersten vier Data-Zeilen stehen die möglichen Stilarten. Man kann diese aber auch kombinieren. Da es sich um einzelne Bits handelt, muß man die Zahlen nur addieren. Beachten Sie auch die DATA-Zeilen als solche. Man kann durchaus verschiedene Typen (hier Zahlen und Strings) mischen. Wichtig ist, daß sie dann in der richtigen Reihenfolge gelesen werden.

7.33.3 Zeilenabstand ändern

Der vertikale Abstand zwischen den Zeichen, praktisch der Zeilenabstand ist normalerweise 8, was der Höhe des Fonts »Topaz 8« entspricht. Der Abstand läßt sich verkleinern, was nicht zu empfehlen ist, und gefahrlos vergrößern. Die Änderung erfolgt direkt mit

```
POKEW WINDOW(8)+58, a%
```

Das setzt allerdings voraus, daß Sie wissen, welcher Font mit welcher Höhe gerade aktiv ist. Eleganter schreibt man deshalb:

```
a%=PEEK(WINDOW(8)+58)
POKEW WINDOW(8)+58,a% + a% DIV 2
```

In diesem Fall wird die tatsächliche Höhe abgefragt und dann um 50% erhöht. Aus einem Achter-Abstand wird also 12.

7.33.4 Zeichenabstand ändern

Der horizontale Abstand zwischen den Zeichen ist normalerweise null, und trotzdem ist ein Abstand vorhanden. Wieso denn solches? Nun, wenn Sie sich einmal mit FED (siehe Kapitel 10.2) einen Zeichensatz ansehen, merken Sie, daß jedes Zeichen seine mögliche Breite nicht ausnutzt, sondern rechts und links noch einen Rand läßt. Diese Ränder werden aber mitgeschrieben, folglich kann man Zeichen auch näher zusammenrücken lassen und auseinander sowieso. Das geschieht mit

```
POKEW WINDOW(8)+64, a%
```

Bei einem negativen Wert von a% rücken die Zeichen zusammen, –1 klappt noch ganz gut, bei positiven Werten wird der Abstand größer, was eine elegante Methode zur Erzeugung von Sperrschrift ist.

7.33.5 Größenänderung limitieren

Oft will man verhindern, daß der Anwender mittels der Größen-Gatgets ein Fenster beliebig vergrößern oder verkleinern kann. Das können Sie leicht vorgeben mit

```
POKE WINDOW(7)+16, Xmin%
POKE WINDOW(7)+18, Ymin%
POKE WINDOW(7)+20, Xmax%
POKE WINDOW(7)+22, Ymax%
```

Über die maximal erlaubten Werte der jeweiligen Auflösung dürfen Sie allerdings nicht gehen, sonst »gurut« es. Wenn alle vier Werte null sind, ist keine Größenänderung des Windows möglich.

7.33.6 Window-Titel ändern

Sie können dem Ausgabe-Fenster von Basic einen neuen Titel geben. Dafür reicht

```
t$ = "Hallo Welt"+CHR$(0)
POKEL WINDOW(7)+32, SADD(t$)
a$=INPUT$(1)
```

Sobald Sie das Window anklicken, ändert sich der Window-Titel in »Hallo Welt«. Das Programm muß dazu im Input stehen, hier wartet es nur auf einen Tastendruck.

7.33.7 Screen-Titel ändern

Wie eben (7.33.6) können Sie auch den Screen-Titel ändern. Es ändert sich nur das Offset, also

```
t$ = "Hallo Freunde"+CHR$(0)
POKEL WINDOW(7)+104, SADD(t$)
a$=INPUT$(1)
```

Interessant daran ist, daß sich der Titel wieder zu »Workbench Screen« zurückändert, sobald man das List-Window von Basic öffnet.

8 Einbindung von Assembler-Routinen in Basic

8.1 Anforderungen an die Routinen

Ein Assemblerprogramm, das von Basic aufgerufen werden soll, muß einige Bedingungen erfüllen, die wir uns zuerst vor Augen führen sollten.

8.1.1 Lageunabhängig

Im Amiga gibt es kein sicheres Plätzchen, wie zum Beispiel das Stück nicht genutzten Video-RAM des Atari ST. Ansonsten ist diese Technik auch nicht zu empfehlen, denn sollte wirklich jemand einen unbelegten Speicherbereich beim Amiga entdecken, so werden alle da ihre Assembler-Routinen speichern wollen, was dann zu Konflikten führt.

Daraus folgt die erste Forderung an die Routine, sie muß lageunabhängig sein, neudeutsch: »position independend«. Damit sind leider Adressierungen wie

```
move.l #buffer,d2
```

nicht erlaubt. Abhilfe bringt die PC-relative Adressierung. Nun wäre eine Schreibweise wie »move.l buffer(pc),d2« natürlich Unsinn, denn »buffer« ist eine Adresse und keine Adreßdistanz. Die könnte man errechnen, einfacher ist jedoch diese Form:

```
lea buffer(pc),a0
move.l a0,d2
```

Bliebe noch ein Problem: Die PC-relative Adressierung ist beim Zieloperanden nicht möglich. Verboten ist also zum Beispiel

```
move #1,dahin(pc) ;falsch!
```

Auch hier bringen zwei Befehle Abhilfe, nämlich

```
lea dahin(pc),a0
move #1,(a0)
```

Um zum Beispiel in eine Window-Struktur zu schreiben, kann man aber auch so vorgehen:

```
lea windowdef(pc),a0
move #1,4(a0)
```

Das wäre der richtige Ersatz für

```
move #1,windowdef+4
```

8.1.2 Nur ein Segment

Sie dürfen für Programm-, Daten- und Variablenspeicher nur ein Segment bilden, doch das ist kein Problem. Dazu müssen Sie nur Anweisungen wie SECTION, DATA und BSS einfach weglassen. Der Grund ist, daß die PC-relative Adressierung nur eine vorzeichenbehaftete 16-Bit-Distanz erlaubt (+/– 32 Kbyte), der Lader aber die Segmente auf größere Distanz legen kann. Logisch, daß auch Ihr Programm nicht größer als 32 Kbyte werden darf, zumindest dürfen Befehle und zugehörige Adressen nicht um mehr als 32 Kbyte auseinanderliegen.

8.1.3 Unterprogramm muß alle Register retten

Daß die Routine mit RTS enden muß, ist klar. Schließlich soll sie auch zu Basic zurückkehren. Wichtig ist, daß Sie zu Beginn der Routine alle Register retten (movem) und vor dem RTS wieder zurückspeichern.

8.1.4 Nur Code und Daten speichern

Ihre Assemblerroutine sollten Sie nur assemblieren, nicht jedoch linken. Letzteres fügt noch diverse Infos für den Lader (des DOS) hinzu, die Sie in Basic nicht brauchen. Kann Ihr Assembler keine reinen Objekt-Module abspeichern, müssen Sie den Overhead selbst entfernen. Ein Beispiel dafür kommt noch.

8.2 Raum für die Routinen

Den Raum für die Routine muß das Basic-Programm bereitstellen. Dazu kann man in Basic einen entsprechend großen Array definieren oder auch einfach einen String verwenden, der von selbst groß genug wird, wenn man den Code dahinein lädt. Alternativ kann man über die Exec-Funktion »AllocMem« den Speicher beschaffen. Wie das alles geht, wird noch gezeigt.

8.3 Laden und Aufrufen von Assembler-Routinen

Mit Abb. 8.1 soll die Assemblerroutine vorgestellt werden, mit der wir uns zuerst befassen wollen. Das Programm öffnet ein Fenster, wartet auf eine Taste und ist dann auch schon fertig. Hier geht es mir in erster Linie um die Darstellung, wie man auf Datenbereiche lageunabhängig zugreift.

```
opt l+,p+ ;linkbar, pos. indep. wenn DevPac

_SysBase          equ 4    ;Basis von Exec
_LVOOpenLibrary   equ -552 ;Library öffnen
_LVOCloseLibrary  equ -414 ;Library schliessen
_LVOOpen          equ -30  ;File öffnen
_LVORead          equ -42  ;Lesen
_LVOClose         equ -36  ; schließen

RELO  macro
      lea \1(pc),a0
      move.l a0,\2
      endm

_main
      movem.l  d0-d6/a0-a6,-(sp)    ;fuer Basic retten
      lea      dosname(pc),a1       ;Name der DOS-Lib
      moveq    #0,d0                ;Version egal
      move.l   _SysBase,a6          ;Basis Exec
      jsr      _LVOOpenLibrary(a6)  ;DOS-Lib oeffnen
      tst.l    d0                   ;Fehler?
      beq      fini                 ;wenn Fehler, Ende
      move.l   d0,a6                ;Zeiger merken

      RELO     name,d1
      move.l   #1005,d2             ;Status = gibt es
      jsr      _LVOOpen(a6)         ;nun öffnen

      move.l   d0,d5                ;Handle merken
      tst.l    d0                   ;Fehler?
      beq      fini                 ;wenn ja, abbrechen

      move.l   d5,d1                ;von CON lesen
      RELO     buffer,d2            ;in diesen Puffer
      move.l   #1,d3                ;1 Zeichen
      jsr      _LVORead(a6)         ;Lesen aufrufen

      move.l   d5,d1                ;CON wieder schließen
      jsr      _LVOClose(a6)
```

```
        move.l   a6,a1                ;DOS-Lib-Basis
        move.l   _SysBase,a6          ;Basis Exec
        jsr      _LVOCloseLibrary(a6) ;Funktion „Schließen"
        movem.l  (sp)+,d0-d6/a0-a6
fini
        rts                           ;Return zum CLI

dosname dc.b 'dos.library',0
        cnop 0,2
name    dc.b 'CON:40/100/580/80/hit any key',0
        cnop 0,2
buffer  ds.b 2
```

Abb. 8.1: Programm 1, das von Basic her aufgerufen werden soll

Das »position independend« habe ich mit dem Makro RELO etwas rationalisiert. Der Hinweis kommt zwar spät, aber immerhin noch: Auch in Programmen, die nicht von Basic her aufgerufen werden sollen, bringt die Methode Vorteile. Dadurch entfallen nämlich im Code die Relokatier-Offsets, womit der Code kürzer und damit schneller geladen wird. Auch wenn Sie auf RELO verzichten und nur da, wo es möglich ist, ein »(pc)« einsetzen, bringt das schon etwas. Der Compiler-Switch »p+« (DevPac) ist übrigens ganz nützlich. Er erzeugt zwar keinen lageunabhängigen Code, meldet aber die lageabhängigen Zeilen als Fehler.

Wenn ein Basic-Programm eine Assembler-Routine aufrufen soll, muß diese natürlich im RAM stehen. Die einfachste Methode ist, den Binär-File mit dem Code dorthin zu laden. Doch leider ist das für den Anwender ziemlich lästig, da er außer dem Basic-Programm auch immer den zugehörigen Code-File parat haben muß. Trotzdem soll das Verfahren vorgestellt werden, denn es ist (zum Testen) so schön einfach und schnell. Abb. 8.2 zeigt die Lösung.

```
OPEN "test.obj" AS 1
l=LOF(1)
CLOSE 1

OPEN ":buch/a" AS 1 LEN=l
FIELD #1, l AS a$
GET 1,1
ass$=a$
CLOSE 1

ass&=SADD(ass$)
CALL ass&

END
```

Abb. 8.2: Methode 1: Basic und Assembler getrennt

Mittels der LOF-Funktion wird geprüft, wie lang das File ist, das Ergebnis wird in der Variablen l notiert.

Nun wird das File nochmals geöffnet, jetzt aber als Random-File. Die Recordlänge wird gleich der Filegröße gesetzt, womit nur ein »get« ausreicht, das File im Stück einzulesen (das ist wichtig). Damit steht der Code im String ass$. Mit SADD kann man nun die Adresse des Strings feststellen und der Variablen ass& zuweisen. Beachten Sie den &-Operator! Damit wird der Typ Long-Integer erzwungen, was für eine Adresse Grundvoraussetzung ist. Nun kann man schlicht mit

```
CALL Adreßvariable
```

das Maschinenprogramm aufrufen.

Wie gesagt, die Methode hat den Nachteil, daß man immer beide Module, den Code und das Basic-Programm auf der Disk haben muß, eine Tatsache, die ein reiner Anwender (beim Kopieren) leider nur zu oft vergißt. Abhilfe bringt die Lösung nach Abb. 8.3.

```
DIM a%(72)

FOR i=1 TO 72
 READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=VARPTR(a%(1))
CALL ass&
END

DATA 48E7,FEFE,43FA,0058,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0044,2C40,41FA,004C
DATA 2208,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,002A,2205,41FA,0050,2408,263C,0000
DATA 0001,4EAE,FFD6,2205,4EAE,FFDC,224E,2C79
DATA 0000,0004,4EAE,FE62,4CDF,7F7F,4E75,646F
DATA 732E,6C69,6272,6172,7900,434F,4E3A,3430
DATA 2F31,3030,2F35,3830,2F38,302F,6869,7420
DATA 616E,7920,6B65,7900,0000,0000,0000,0000
```

Abb. 8.3: Methode 2: Basic und Assembler in einem File

Das Maschinenprogramm steht als Hex-Strings in DATA-Zeilen. Mit der einfachen Anweisungsfolge in der FOR-Schleife wird es in einen Array geschrieben. Dieser Array muß (!!) vom Typ Integer sein, ansonsten würde nämlich Basic die Zahlen in das Fließkomma-Format bringen, womit von unserem Code nichts übrigbliebe.

Das erste Wort des Codes steht nun in a%(1). Mit VARPTR kann man dessen Adresse ermitteln. Den Rest kennen Sie schon.

8.4 Maschinenprogramm im Systemspeicher

Die Methode des vorherigen Kapitels hat zwei – wenn auch kleine – Nachteile. Die Adresse des Arrays ist nicht konstant und muß deshalb unmittelbar vor dem Aufruf, ggf. immer neu, ermittelt werden. Außerdem belegt der Array einen Teil des knappen(?) Basic-Speichers. Dem kann man abhelfen, indem man mit der Exec-Funktion »AllocMem« sich Systemspeicher beschafft. Dessen Adresse ändert sich nie. Abb. 8.5 zeigt das Prinzip.

```
LIBRARY "exec.library"
DECLARE FUNCTION AllocMem& LIBRARY

DEFLNG M
MEMF.PUBLIC = 1
MEMF.CHIP = 2
MEMF.FAST = 4
MEMF.CLEAR = 65536&

ass& = AllocMem&(144,MF.PUBLIC+MEMF.CHIP+MF.CLEAR)

FOR i%= 0 TO 143 STEP 2
 READ a$: adr& = ass&+i%
 POKEW adr&, VAL("&h"+a$)
NEXT

CALL ass&

FreeMem& ass&,144

END

DATA 48E7,FEFE,43FA,0058,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0044,2C40,41FA,004C
DATA 2208,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,002A,2205,41FA,0050,2408,263C,0000
DATA 0001,4EAE,FFD6,2205,4EAE,FFDC,224E,2C79
DATA 0000,0004,4EAE,FE62,4CDF,7F7F,4E75,646F
DATA 732E,6C69,6272,6172,7900,434F,4E3A,3430
DATA 2F31,3030,2F35,3830,2F38,302F,6869,7420
DATA 616E,7920,6B65,7900,0000,0000,0000,0000
```

Abb. 8.4: Methode 2a: Maschinenprogramm im Systemspeicher

Die Konstanten (genau: es sind Variablen)

```
MEMF.PUBLIC = 1
MEMF.CHIP = 2
MEMF.FAST = 4
MEMF.CLEAR = 65536&
```

bestimmen in den ersten drei Zeilen die Art des Speichers. »MEMF_CLEAR« kann eingestzt werden, wenn der Speicher nach der Reservierung auch noch gelöscht, also mit Nullen beschrieben werden soll. »MEMF.PUBLIC ist ein allgemeiner Datenbereich, der typisch für Interruptvektoren, Prozeßleitblöcke, Message-Ports u.ä. benutzt wird.

Chip-Memory und Fast-Memory kennen Sie. Wichtig in der Zeile

```
ass& = AllocMem&(144, MF.PUBLIC+MEMF.CHIP+MF.CLEAR)
```

ist das »+«, was einem logischen ODER entspricht, und das hat hier die Bedeutung »nehme Public- oder Chip-Memory, was gerade frei ist«. Natürlich hätte ich anstatt des tollen Ausdrucks

```
MF.PUBLIC+MEMF.CHIP+MF.CLEAR
```

auch schlicht 65539& schreiben können, und eine »3« hätte es auch getan, denn warum soll ich den Bereich erst löschen, wenn ich ihn sowieso gleich wieder beschreibe.

8.5 Ein Assembler-Basic-Konverter

Die bisherigen Listings ließen eine Frage offen. Wie kommt der Code in die Data-Zeilen? Die kleine Utility von Abb. 8.5 löst genau dieses Problem, natürlich in Assembler, um den geht es ja hier (Basic wäre einfacher).

```
 opt l- ;nur wenn DevPac-Assembler

* ABC = Assembler Basic Converter
* Object-File -> Basic-DATA-Zeilen
* (c) 1990 Peter Wollschlaeger
* ----------------------

_SysBase          equ 4    ;Basis von Exec
_LVOOpenLibrary   equ -552 ;Library oeffnen
_LVOCloseLibrary  equ -414 ;Library schliessen
_LVOOutput        equ -60  ;StdOut-Handle holen
_LVOWrite         equ -48  ;Schreiben (auf File)
_LVORead          equ -42  ;Lesen
_LVOOpen          equ -30  ;File oeffnen
_LVOClose         equ -36  ; schliessen
MODE_OLDFILE      equ 1005 ;File existiert
MODE_NEWFILE      equ 1006 ;File neu/ueberschreiben

* Einige simple Makros zum File-Handling
* ----------------------
```

```
move.l  \1,d1          ;Name
     move.l  \2,d2          ;Mode
     jsr     _LVOOpen(a6)
     endm

READ macro
     move.l  \1,d1          ;Handle
     move.l  \2,d2          ;Puffer-Adr
     move.l  \3,d3          ;Max.
     jsr     _LVORead(a6)
     endm

WRITE macro
     move.l  \1,d1          ;Handle
     move.l  \2,d2          ;Puffer-Adr
     move.l  \3,d3          ;Anzahl
     jsr     _LVOWrite(a6)
     endm

CLOSE macro
     move.l  \1,d1          ;Handle
     jsr     _LVOClose(a6)
     endm
* ---------------------


     movem.l  a0/d0,-(sp)          ;Parms. Kommandozeile

* DOS-Lib oeffnen:
_main
     move.l   #dosname,a1          ;Name der DOS-Lib
     moveq    #0,d0                ;Version egal
     move.l   _SysBase,a6          ;Basis Exec
     jsr      _LVOOpenLibrary(a6)  ;DOS-Lib oeffnen
     tst.l    d0                   ;Fehler?
     beq      fini                 ;wenn Fehler, Ende
     move.l   d0,a6                ;Zeiger merken


* Kommandozeile parsen
* ----------
     movem.l  (sp)+,a0/d0          ;hole Parms
     move.b   #' ',-1(a0,d0.l)     ;terminiere mit Blank weil
                                   ; mein Parser das will
     lea      quelle,a1            ;Puffer Name Quellfile
     bsr      parse                ;hole ihn
     lea      ziel,a1              ;Name Zielfile
     bsr      parse                ;dito

* Beide Files oeffnen
* ---------
     OPEN     #quelle,#MODE_OLDFILE
     move.l   d0,d4                ;Handle
     tst.l    d0                   ;ging was schief?
```

```
      bne       weiter                ;wenn nicht
      jsr       _LVOOutput(a6)        ;Hole Output-Handle
      WRITE     d0,#msg1,#len1        ;und melde Fehler
      bra       cl_lib                ;und Ende
 weiter
      OPEN      #ziel,#MODE_NEWFILE ;wie vor nun mit Zielfile
      move.l    d0,d5
      tst.l     d0
      bne       start
      jsr       _LVOOutput(a6)        ;Hole Output-Handle
      WRITE     d0,#msg2,#len2
      bra       cl_q

* Hier geht's nun los
* -------------
start
      READ d4,  #buffer,#16           ;Header skippen
loop
      READ d4,  #buffer,#16           ;Lese <=16 Bytes
      tst.l     d0                    ;End of File?
      beq       cl_z                  ;wenn ja
      asr       #1,d0                 ;gelesene Bytes durch 2
      subq      #1,d0                 ;- 1 wegen dbra-loop
      move      d0,d6                 ;Anzahl Worte merken
      WRITE     d5,#data,#6           ;write LF + 'DATA '
      lea       buffer,a4             ;Zeiger auf Quelle
conv
      move      (a4)+,d2              ;ein Wort
      lea       hbuf,a0               ; soll dahin
      bsr       hex                   ;als Hex-String
      WRITE     d5,#hbuf,#4           ;auf Zielfile
      cmp       #0,d6                 ;letztes Wort in Zeile?
      beq       no                    ;wenn nicht
      WRITE     d5,#komma,#1          ;sonst ein Komma dahinter
no
      dbra      d6,conv               ;bis Zeile fertig
      bra       loop                  ;bis EOF
cl_z
      CLOSE     d5                    ;Close Zielfile
cl_q
      CLOSE     d4                    ;und Quellfile

cl_lib
      move.l    a6,a1                 ;DOS-Lib-Basis
      move.l    _SysBase,a6           ;Basis Exec
      jsr       _LVOCloseLibrary(a6) ;Funktion „Schliessen"
fini
      rts                             ;Return zum CLI

* Einfacher Parser; kennt nur Blanks als Delimeter
* ------------------------
parse
      cmp.b     #' ',(a0)+            ;führende Blanks
      beq       parse                 ;skippen
```

```
        subq.l   #1,a0              ;wir waren 1 zu weit
p1
        move.b   (a0)+,(a1)+        ;nun kopiere
        cmp.b    #' ',(a0)          ;bis wieder ein Blank
        bne      p1
        clr.b    (a1)               ;Terminiere mit 0-Byte
        rts

* Konvertiere d2.w in ASCII-String ab (a0)
* ------------------------
hex
        moveq    #3,d1              ;für 4 Nibble
next
        rol      #4,d2              ;hole 1 Nibble
        move     d2,d3              ;nach d3 retten
        and.b    #$0f,d3            ;maskiere es
        add.b    #48,d3             ;in ASCII wandeln
        cmp.b    #58,d3             ;ist es >9 ?
        bcs      out                ;wenn nicht
        addq.b   #7,d3              ;sonst muß es A-F sein
out
        move.b   d3,(a0)+           ;1 Zeichen abspeichern
        dbra     d1,next            ;next nibble
        rts

* ---------------------------
* Datenbereich:

dosname  dc.b      'dos.library',0
         cnop 0,2

data     dc.b      10,'DATA '
         cnop 0,2
komma    dc.b      ','
         cnop 0,2
msg1     dc.b      'Quellfile nicht gefunden',10
len1     equ       *-msg1
         cnop 0,2
msg2     dc.b      'Konnte Zielfile nicht öffnen',10
len2     equ       *-msg2
         cnop 0,2

quelle   ds.b 40
ziel     ds.b 40
buffer   ds.b 16
hbuf     ds.b 4
```

Abb. 8.5: Assembler-Programm generiert Basic-Zeilen

Das Programm namens ABC wird mit der Syntax

```
ABC  Object_File  Basic_File
```

im CLI aufgerufen. Object_File ist das Maschinenprogramm, so wie es der Assembler erzeugt. Basic_File ist danach reiner ASCII-Text, in dem jede Zeile mit dem Wort DATA beginnt. In den DATA-Zeilen steht das Maschinenprogramm in Form von Strings. Jeder String ist ein Wort in hexadezimaler Notation. Die hexadezimale Schreibweise hat den Vorteil, daß man ein Programm besser lesen kann.

Zwischen mir und dem Amiga wurde vereinbart, daß in jeder Zeile 8 Worte stehen sollen. Daraus resultiert dann folgendes Schema für das Programm:

1. Zerlege Kommandozeile in die Filenamen Quelle und Ziel
2. Öffne die Files
3. Schreibe ein Linefeed plus dem Wort DATA plus 1 Blank
4. Lese 16 Byte
3. Wandle 16 Byte in 8 Hexstrings
6. Schreibe 8 Strings durch Kommas getrennt
7. Wiederhole ab 3. bis EOF(Quelle)
8. Schließe die Files

Kleine Schwierigkeit: Es bleiben zum Schluß nicht 16 Byte übrig, folglich gilt: Lese 16 Byte oder was noch da ist. Wandle davon die Hälfte in Hex-Worte.

Natürlich muß man auch prüfen, ob beim Öffnen der Files etwas schiefging und ggf. eine Fehlermeldung ausgeben.

Damit hätten wir die Aufgabe beschrieben, nun zur Lösung. Nach dem Programmstart steht die Adresse der Kommandozeile (aller Text, der dem Programmnamen folgt) im Register A0, ihre Länge in D0. Es ist üblich, zuerst diese Parameter zu retten, dann zuzusehen, ob man seine Libs öffnen kann, und danach die Kommandozeile zu interpretieren. Sie können natürlich die beiden »movem« sparen, indem Sie das Öffnen der DOS-Lib nach dem »parsen« schreiben. Wie auch immer, der Parser kopiert die beiden File-Namen von der Kommandozeile in zwei Puffer und schließt sie (DOS will es so) mit einem Null-Byte ab. Nun versuche ich, die beiden Files zu öffnen. Im Fehlerfall hole ich mittels »_LVOOutput« die Handle der Standard-Ausgabe (hier das CLI-Fenster) und gebe mit demselben WRITE-Makro die Fehlermeldung aus.

Hier liegt übrigens der Grund dafür, daß Sie das Programm nur vom CLI aus aufrufen dürfen. Auch »RUN ABC...« ist nicht erlaubt. Wenn Sie das wollen, müssen sie ein eigenes Fenster öffnen und das geht so:

```
move.l   #window,d1        ;Adresse
move.l   #MODE_OLDFILE,d2  ;Status = gibt es
jsr      _LVOOpen(a6)      ;nun öffnen
```

Mit unserem Makro wird die Sache noch einfacher, also

```
OPEN    #window,#MODE_OLDFILE
```

Danach steht die Handle in D0 und kann sowohl für die Eingabe (Read) als auch die Ausgabe (Write) eingesetzt werden.

Im Datenbereich fehlt dann noch:

```
window dc.b 'CON:40/100/580/80/Fenstertitel',0
```

Die vier Zahlen beschreiben die linke obere Ecke des Fensters, seine Breite und Höhe. Danach folgt der Titel. Nach dem Aufruf sollten Sie die Handle (D0) sichern. Vergessen Sie nicht, das Fenster wieder zu schließen.

Nun zeige mir jemand einen 68000er, bei dem sich Windows so einfach öffnen lassen (mittels System-Software)! Dieses Fenster hat zwar nicht die Mächtigkeit eines Intuition-Windows, ist dafür aber sehr bequem zu handhaben.

Ab »Hier geht's nun los« werden zuerst 16 Byte blind gelesen (kommt gleich), danach läuft die schon geschilderte Read/Write-Schleife. Zum Schluß werden die Files geschlossen und schließlich noch die DOS-Library.

Die 16 Byte, die ich anfangs überlese, sind der sogenannte File-Header.

Sie finden da zuerst ein Langwort des Inhalts $000003E7. Das heißt, hier beginnt eine Programm-Unit. Danach folgt ein Langwort des Inhalts 0, sofern Sie dem Modul keinen Namen gegeben haben (sollten Sie auch nicht tun). Das letzte Langwort könnte der Grund sein, warum Sie vielleicht die Zeile ändern und nur 12 Byte überlesen wollen. Hier steht nämlich die Länge des Moduls in Langworten minus eins.

Zu den Unterprogrammen:

Der Parser sucht einen Text ab Adresse in A0, der mit je einem Blank beginnen und enden muß. Das letzte Blank wurde vorab mit

```
move.b  #' ',-1(a0,d0.l)
```

an das Ende der Kommandozeile gezwungen, übrigens ein schönes Beispiel für die Mächtigkeit der 68000-Adressierungsarten.

Der vom Parser isolierte Text wird in den Puffer ab A1 kopiert. Da es sich um Filenamen handelt, werden die Texte mit einem Null-Byte terminiert.

Die Routine »hex« konvertiert eine Binärzahl in einen String, der den Wert in hex darstellt. Wenn Sie den Schleifenzähler wieder von 3 in 7 ändern und das »rol« in in »rol.l«, können Sie damit auch Langworte wandeln.

8.6 Die Parameterübergabe

Bis hierher war die Sache zwar einfach, aber kaum realistisch. In der Regel wird man nämlich Daten an die Assembler-Routine übergeben müssen und sich das Ergebnis (die Ergebnisse) auch von dort holen wollen. Frage also: Wie übergibt man Parameter in beiden Richtungen?

Antwort 1: Man muß auch die Ergebnis-Variablen übergeben. Das Assembler-Programm muß diese dann ändern. Genau: Man übergibt die Adressen der Basic-Variablen, die Maschinenroutine schreibt ab dort das Ergebnis hinein.

Antwort 2: Die Parameterübergabe läuft über den Stack.

Am besten erklärt man das an einem Beispiel. Ich möchte ganz simpel, daß die Assembler-Routine in einem String das letzte Zeichen duch ein x ersetzt. Dazu muß ich zwei Parameter übergeben, nämlich die Adresse des Strings und seine Länge. In Basic sähe das so aus, wie in Abb. 8.6.

```
DIM a%(13),r%(3)
FOR i=1 TO 13
 READ a$:a%(i)=VAL("&h"+a$)
NEXT

a$="hallo"
Adresse&=SADD(a$) Laenge&=LEN(a$)

ass&=VARPTR(a%(1))
CALL ass&(Adresse&,Laenge&)
PRINT a$

END

DATA 48E7,8080,206F,000C,202F,0010,5380,11BC
DATA 0078,0800,4CDF,0101,4E75
```

Abb. 8.6: Basic-String soll modifiziert werden

Wie üblich, wird wieder die Routine aus Data-Zeilen geladen, ihre Adresse steht dann in »ass&«. Adresse und Länge des Strings werden bestimmt, und dann erfolgt der Aufruf mit

```
CALL    ass&(Adresse&,Laenge&)
```

Schauen wir uns nun an, was in den Data-Zeilen steckt, Abb. 8.7 verrät das Geheimnis.

```
  opt l+,p+
  movem.l a0/d0,-(sp)  ;8 mehr auf dem Stack
;  +Returnadresse macht 12
```

```
move.l 12(sp),a0 ;Adresse
move.l 16(sp),d0 ;Laenge
subq.l #1,d0
move.b #'x',0(a0,d0.l)

movem.l (sp)+,a0/d0
rts
```

Abb. 8.7: Bedeutung der Data-Zeilen von Abb. 8.6

Ich muß mir eigentlich nur merken, daß durch das »movem« zusätzlich noch 8 Byte auf den Stack gekommen sind (2 Register) und auch noch die Return-Adresse mit 4 Byte existiert; macht in der Summe 12.

Nun kann ich einfach im 4er-Abstand, beginnend mit 12, auf die Parameter der Reihe nach zugreifen. Doch Vorsicht, so einfach ist das nur, wenn alle Parameter vom Typ Long sind. Da der Stack von den hohen zu den tiefen Adressem wächst, ist 12(sp) tiefer als 16(sp). Das heißt, der Parameter bei 12(sp) ist der letzte auf dem Stack, dieser (Adresse&) war aber der erste im Aufruf. Anders ausgedrückt: Die Parameter werden in umgekehrter Reihenfolge abgelegt.

Schauen wir uns deshalb noch ein Beispiel an. Laut Abb. 8.8 soll ein Array oder ein Teil davon ganz schnell mit dem gleichen Wert in allen (oder den ausgewählten) Elementen geladen werden.

```
DIM a%(15),r%(100)
FOR i=1 TO 15
 READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=0

Laenge&=5
Wert%=123
Adresse&=VARPTR(r%(3))

ass&=VARPTR(a%(1))
CALL ass&(Adresse&,Laenge&,Wert%)
FOR i=1 TO 10
 PRINT r%(i)
NEXT

END

DATA 48E7,C080,206F,0010,202F,0014,322F,001A
DATA 5380,30C1,51C8,FFFC,4CDF,0103,4E75
```

Abb. 8.8: Ein Array wird initialisiert

Übergeben muß ich die Adresse des Elements, ab dem der Array geladen werden soll (hier r%(3)), die Länge (Anzahl der Elemente) und natürlich den Wert. Der Wert ist aber als einziger vom Typ Integer, sprich, belegt nur 2 Byte. Doch schauen wir uns an, was Basic dafür auf den Stack packt.

Der Aufruf lautet wieder:

```
CALL ass&(Adresse&,Laenge&,Wert%)
```

Schauen wir uns die Assemblerroutine dazu an (Abb. 8.9).

```
 opt l+,p+
 movem.l a0/d0-d1,-(sp) ;12 mehr auf dem Stack
; +Returnadresse macht 16

 move.l 16(sp),a0 ;Adresse
 move.l 20(sp),d0 ;Anzahl
 move.w 26(sp),d1 ;Wert
 subq.l #1,d0 ;- 1 wegen dbra
loop move.w d1,(a0)+
 dbra d0,loop

 movem.l (sp)+,a0/d0-d1
 rts
```

Abb. 8.9: Die Assembler-Routine zu Abb. 8.8

Daß ich nun bei 16 anfangen muß, ist klar. Schließlich sind 3 Register und die Return-Adresse mit je 4 Byte auf dem Stack. Doch warum finde ich den Wert bei 26(sp) und nicht bei 24(sp)? Offensichtlich läßt sich Basic durch das %-Zeichen (was short Integer heißen soll) nicht beeindrucken, und packt auch dann 4 Byte auf den Stack. Tatsächlich können Sie in Basic auch »Wert&« schreiben und dann den Wert bei 24(sp) als Langwort abholen. Das ist an sich klarer, macht aber Probleme bei negativen Zahlen, da dann ja das Vorzeichen sozusagen in Bit 31 steht, was natürlich fehlt, wenn man nur die Bits 0 bis 15 in ein Wort übernimmt.

Wenn bei Ihren Experimenten etwas schiefgeht, schauen Sie sich auch Ihr Basic-Programm an. Ich habe beispielsweise beim Testen der Routine einen bildschönen Absturz erlebt, weil ich anstatt r% a% geschrieben habe, was dann leider zur Folge hatte, daß ich mit »Wert« die Assemblerroutine überschrieben habe, als sie mitten in der Ausführung war.

Auch sollten Sie darauf achten, keine neuen Variablen in Basic anzuziehen, wenn Sie vorher mit VARPTR oder SADD eine Adresse bestimmt haben. Die neue Variable kann die älteren verschieben, so daß dann deren Adressen nicht mehr stimmen. Am einfachsten weist man vorab allen Variablen einen Wert zu. So sinnlos ist also »ass&=0« in Abb. 8.8 gar nicht.

9 Tips für C-Programmierer

9.1 Der Aztec-C-Compiler

Aztec-C ist ein 2-Pass-Compiler. Im Pass 1 wird der Quelltext (C) in Assembler-Quelltext übersetzt. Im Pass 2 wird diese Assembler- Source dann assembliert. Pass 1 ruft automatisch Pass 2 auf, sofern dies nicht extra ausgeschaltet wird. Der Quelltext muß immer mit ».c« enden. Der Aufruf des Compilers erfolgt im einfachsten Fall mit

```
cc filename.c
```

Dem Filenamen können noch diverse, jeweils durch eine Leerstelle getrennte Optionen folgen. Gehört zu einer Option ein Filename, muß dieser direkt hinter dem Optionskenner stehen.

Folgende Optionen bietet Aztec-C:

–a Kein Assemblerlauf (Pass 2 wird nicht aufgerufen). Kann dann nachträglich mit »as name.asm« erfolgen.

–b Schaltet die nach jeweils 5 Fehlern erscheinende Frage »Weitermachen?« aus.

–d Define. Entspricht dem #define in der Form von z.B. als -dTRUE=1

+h Gefolgt von Filename: File für Symboltabelle

–i Gefolgt von Text: Directory-Pfad für Include-Files

+i Gefolgt von Filename: Filename einer existierenden Symboltabelle. Erspart das langwierige Laden der Include- Files.

+l Muß immer sein!!! Damit werden Integers 32 Bit breit, wie es der Amiga verlangt.

–o Gefolgt von Filename. Böse Falle: ergibt nicht den Objekt-File, sondern den Namen des Files, unter dem der Assembler-Quelltext gespeichert werden soll. Kann man auch weglassen, dann heißt der File wie der C-File mit der Extension ».asm«.

–s Es werden keine Warnungen ausgegeben.

–t Im Assembler-Source wird der C-Befehl als Kommentar hinzugefügt.

Wenn Sie den Assembler getrennt aufrufen, können Sie dabei die folgenden Optionen verwenden:

–i Include-File-Directory wie in Pass 1

–o Gefolgt von Filename: Name des Objekt-Files

–v Erzeugt Angaben über Speicherverbrauch

Am einfachsten ist es, mit der mitgelieferten Diskette zu booten, die auch gleich die Pfade setzt (siehe Startup-Sequence). Dann läßt sich ein File mit zum Beispiel dem Namen »hello.c« so compilieren und linken:

```
cc hello -a
as hello
ln hello.o -lm -lc
```

Natürlich können Sie das mit einem Kommando-File automatisieren, was dann so aussähe:

```
.key n
cc <n> -a
as <n>
ln <n>.o -lm -lc
```

Wird dieser Text unter dem Namen cl (compile & link) im s-Directory gespeichert, so kann der Aufruf dann erfolgen mit

```
execute cl hello
```

Auch der hier schon so nebenbei erwähnte Linker (ln) hat noch Optionen, die Sie ihm mitgeben können, nämlich:

–o Gefolgt von Filename: Name des fertigen Programms, wenn es nicht Name des C-Files ohne ».c« sein soll.

–l Gefolgt von Filename: Name einer eigenen Library

–t Gefolgt von Filename: In diesem File wird die Symboltabelle abgelegt.

9.2 Der Lattice-C-Compiler

Auch beim Lattice-C muß der Quelltext-File mit der Extension ».c« enden und auch Lattice-C ist ein 2-Pass-Compiler mit den Pass-Namen LC1 und LC2. Allerdings ist der von Pass 1 erzeugte File für uns von keinerlei Nutzen, nur LC2 kann damit etwas anfangen. Statt dessen bietet Lattice die Möglichkeit, den Assembler-Quelltext des Aztec sozusagen nachträglich zu erzeugen. Dazu wird der Object-Code, der ja schon 68000-Maschinensprache ist, mit einem Disassembler wieder in die Assembler-Mnemoniks übersetzt.

Die Optionen sowohl für den Pass 1 als auch für den Pass 2 sind so zahlreich und auch noch versionsabhängig, daß ich hierzu auf das Handbuch verweisen muß, zumal man praktisch nur eine Option (–i) wirklich braucht.

9.2.1 Lattice Version 3

Ein Kommando-File für Lattice, Version 3 sieht so aus:

```
.key prg
stack 20000
lc1 -i:include/ -i:include/lattice/ <prg>
lc2 <prg>
blink :lat/lib/c.o,<prg>.o to <prg> lib
      :lat/lib/lc.lib,:lat/lib/amiga.lib
```

Beachten Sie, daß die letzte Zeile noch zur vorletzten Zeile gehört, ich kann nur nicht über den Buchrand hinausschreiben.

9.2.2 BLINK einsetzen

BLINK (sprich B-LINK) ist ein Public-Domain-Linker, der voll zu ALINK kompatibel ist, nur wesentlich schneller. Bei den neueren Versionen gehört BLINK schon zum Lieferumfang. Haben Sie BLINK nicht, besorgen Sie sich ihn unbedingt, Sie sparen damit sehr viel Zeit.

Lattice-C läuft bei mir auf einer Festplatte. Ich habe alle Files des Systems in ein Directory namens »lat« kopiert, was man am schnellsten so erreicht:

```
makedir lat
copy df0: to lat all  ;(Lattice-Disk in Drive 0)
```

Leider hatte der Compiler dann trotz richtiger Include-Anweisungen Probleme, die sich erst gaben, als ich die Include-Files in das Root-Directory gelegt hatte, also

```
makedir :include
copy :lat/include to :include all
delete :lat/include all
```

Das o.g. Kommando-File habe ich unter dem Namen »lc« im S-Directory abgelegt. Da ich ein fauler Mensch bin, tippe ich nur x für »execute«, wofür ich in »:s/Shell-Startup« eingetragen habe

```
alias x execute
```

Damit kann nun ein Programm mit zum Beispiel dem Namen »hello.c« so compiliert und gelinkt werden:

```
x lc hello
```

9.2.3 Lattice ab Version 4

Die Version 4/5 bietet sehr viele Vorteile, ich kann das Umsteigen darauf nur dringend empfehlen. Für die Installation benutzen Sie am besten einen der mitgelieferten Execute-Files (Harddisk oder Floppy). Eine Harddisk ist empfehlenswert, ansonsten brauchen Sie zwei Disktten-Laufwerke. Nach der Installation ergänzen Sie Ihre Startup-Squence oder besser »:s/Shell-Startup« um diese Kommandos (ändern Sie dh0: in df0:, wenn Sie keine Harddisk haben):

```
path add dh0:lc/c
assign LC: dh0:lc/c
assign LIB: dh0:lc/lib
assign INCLUDE: dh0:lc/include
assign QUAD: RAM:
```

Nun können Sie den File xxx.c compilieren und linken mit

`lc -L xxx` (–L muß groß geschrieben werden)

9.3 Wenn das Fenster nicht schließt

Es existieren Compiler-Versionen, die allergisch auf Exit() reagieren. Die Programme laufen dann zwar, aber ENDCLI schließt nicht das CLI-Fenster. Um das zu vermeiden, ersetzen Sie alle »Exit()« nach Tests (z.B. nach OpenWindow) durch »exit()« (kleines e), hilft das nicht, durch »return«. Das »Exit()« vor dem Ende von main() streichen Sie ganz.

Wenn dann noch beim Start von der Workbench zusätzlich ein Fenster mit dem Programmnamen öffnet, dann ersetzen Sie »main()« durch »_main()«.

9.4 Zuerst »amiga.lib«?

Es gibt interessante Dinge in der Dokumentation von Commodore, man muß sie nur lesen, auch wenn sie dick, englisch und teuer ist. So findet man im »Exec«-Manual ab B12 ein paar Seiten mit dem Titel »amiga.lib.doc«. Diese Seiten enthalten den Hinweis, daß unter anderem die C-Funktion »printf()« im ROM vorhanden ist (siehe Kapitel 6).

Daher empfiehlt Commodore C-Programmierern, »amiga.lib« vor die Laufzeit-Bibliothek des C-Compilers zu linken, damit der Code kürzer wird. Recht haben sie, denn der Linker nimmt immer die erste Referenz.

Doch Vorsicht ist auch geboten. Wenn Sie die speziellen »printf()«-Features Ihres C-Compilers ausnutzen wollen, müssen Sie dessen Code wohl auch einbinden. Was das ROM-»printf()« kann, lesen Sie bitte in Kapitel 6 nach.

9.5 Das richtige Start-Modul

Jedes C-Programm muß mit einem Startup-Code gelinkt werden. Wie der aussieht – auch das C-Startup ist in Assembler geschrieben – können Sie in Kapitel 6.5 nachlesen.

Je nach Compiler gibt es hierzu ein paar Varianten, die Sie im Handbuch nachlesen sollten. Was eigentlich immer funktioniert, ist das Linken von »c.o« als Startup-Modul (deshalb auch als erster).

Bei Lattice ist es am praktischsten, wenn Sie anstatt »main« »_main« schreiben. Ein so entstandenes Programm läuft unter dem CLI und auch von der Workbench aus.

Ein Programm, das nur unter dem CLI läuft, sieht typisch so aus wie das in Abb. 9.1.

```
/* start_1.c    */
/* Nur unter CLI */

void main(argc, argv)
int argc;
char *argv[];

{
 int i;

 for(i = 1; i < argc; i++)
 printf("%s\n", argv[i] );
}
```

Abb. 9.1: Dieses Programm läuft nur unter dem CLI

Das Programm druckt einfach alle Argumente der Kommandozeile. Wichtiger ist jedoch, daß so ein Programm die Deklarationen dafür (argc, argv) hat.

Wollen Sie ein Programm schreiben, das nur unter der Workbench läuft, können Sie so vorgehen, wie es Abb. 9.2 zeigt. Die Struktur, die dahinter steckt, finden Sie in »startup.h« Ihres Compilers. Tiefer möchte ich auf das Thema nicht eingehen, weil mir diese Art Programme widerstrebt. Ich meine, ein Programm, das unter der Workbench läuft, sollte auch vom CLI aus aufgerufen werden können. Der kleine Unterschied liegt bei »argv«.

```
/* start_2.c       */
/* Nur unter Workbench */

#include <workbench/startup.h>
```

```
void main(argc, argv)
int argc;
struct WBStartup *argv;
{
 int i;

 if (argc) exit(); /* wenn CLI-Aufruf */

 printf("Programm %s\n", argv->sm_ArgList->wa_Name );

 for(i=0; i<500000; i++) /* warte etwas */
 ;
}
```

Abb. 9.2: Ein Programm, das nur unter der Workbench läuft

9.6 Warnmeldungen doch beachten

Es geht hauptsächlich um die Warnmeldungen in Zeilen wie dieser:

```
IntuitionBase = (struct IntuitionBase *) OpenLibrary(......);
```

Hier hat das »(struct IntuitionBase *)« nur den Zweck, den von der Funktion zurückgegebenen Long-Wert, der eine Adresse ist, in einen Zeiger, der dieselbe Adresse ist, umzuwandeln. In den Assembler-Listings sehen Sie, daß es auch ohne dieses Type-Casting geht.

Tatsächlich können Sie diese Konstrukte auch weglassen, nur erhalten Sie dann eine Warnmeldung. Lattice ist ja in diesem Punkt noch harmlos. Bei Aztec ist das Casting von Long in einen Struct-Zeiger nicht erlaubt, nur »Zeiger in Zeiger« ist möglich. Deshalb muß man bei Aztec vorher noch schreiben

```
LONG *OpenLibrary();
```

oder besser

```
struct Library * OpenLibrary();
```

Da das für alle Funktionen dieser Art gilt, tun Sie gut daran, bei Aztec generell im Deklarationsteil Ihres Programms zu schreiben:

```
#include <functions.h>
```

Die zweite Warnmeldung, die wenigstens bei mir häufig auftritt, ist

```
function returns value mismatch
```

Das tritt immer dann auf, wenn eine Funktion keinen Wert zurückgibt (void). Jeder C-Programmierer schreibt normalerweise schlicht »main()«. Hier müssen Sie nun »void main()« schreiben. Sie können allerdings auch diese Warnmeldung gefahrlos ignorieren.

Das Problem an der Geschichte ist, daß es auch Warnmeldungen gibt, die Sie genau so ernst wie Errors nehmen müssen. Deshalb ist es bedenklich, alle Warnungen per Compiler-Switch einfach abzuschalten. Ein Beispiel wäre die Meldung, daß ein Integer in einen Zeiger gewandelt wurde. Das passiert dann, wenn man einer Funktion einen Zeiger übergibt und ihn dann innerhalb der Funktion einer Zeigervariablen zuweist. Hat man nun als Funktionsargument

```
char p   anstatt   char *p
```

deklariert, passiert der Fehler, und der ist natürlich fatal. Der vergessene Stern wird als Warnung somit völlig unterbewertet.

9.7 Gefährliche Fehler

Vom vergessenen Stern habe ich eben schon erzählt; der bringt in der Regel nur unsinnige Ergebnisse. Die folgenden Fehler führen allerdings mit Sicherheit zum Absturz.

Ich habe zum Beispiel einmal nach dem OpenLibrary() geschrieben:

```
if (DosBase=0)
```

also »=« anstatt »==«. Damit wurde a) der Ausdruck wahr und das Programm lief weiter, doch b) hatte nun DosBase die Adresse 0, und das ist natürlich tödlich.

Ein zweites Beispiel, das einige Varianten hat: Man schreibt ganz ordentlich

```
struct RastPort *rp;
```

und dann später

```
Move(rp, x,y);
```

Das gibt den schönsten Crash, weil man vergessen hatte, »rp« die Adresse eines Rastports zuzuweisen. Allgemein gilt, daß der Zugriff auf solchermaßen nicht initialisierte Systemvariablen Gurus produziert. Auch Move() und Draw() außerhalb des Windows kann zu Guru-Meditationen führen.

Auch relativ gefährlich ist die Initialisierung von Strukturen der Form

```
struct NewWindow ns = {....
```

Wenn Sie sich da einmal vertippen, einen falschen Typ einsetzen oder gar etwas auslassen, kracht es.

Ansonsten war festzustellen, daß sich das System manchmal von solchen Schocks nicht erholt hatte, auch wenn es nach einer Guru-Meldung wieder gebootet hatte. Da half nur noch der Griff zur Panik-Taste (Netzschalter).

9.8 Nehmen Sie Makros

Wenn Sie keine Lust haben, im Programm immer wieder solche Strings wie »IntuitionBase->ActiveScreen->MouseX« zu tippen, dann schreiben Sie doch einfach einmal zu Programmbeginn

```
#define MX IntuitionBase->ActiveScreen->MouseX
```

und zukünftig tippen Sie nur noch »MX«. Durchforsten Sie einmal Ihre Programme, und Sie werden feststellen, daß es viele und immer wiederkehrende Bandwürmer dieser Art gibt. Definieren Sie dafür Kürzel, die Sie sich gut merken können, und packen Sie alle in ein Include-File. Das kann man ja auch einmal ausdrucken und als Spickzettel neben der Tastatur liegen lassen.

9.9 So schreiben Sie kompakte Programme

Es gibt viele Tricks, kompakte Programme zu schreiben, womit ich allerdings die Code-Größe meine und nicht Bandwurmzeilen im Quelltext. Letztere wirken sich übrigens oft nachteilig auf die Code-Größe aus. Ich will hier auch nicht Bytes zählen, nach dem Motto, daß »i+=3« kürzeren Code ergibt als »i=i+3«, denn solche Dinge werden zunehmend auch schon von den Compilern erkannt. Binsenweisheiten wie »puts« ist kürzer als »printf« möchte ich auch nicht wiederholen, also was bleibt noch?

9.9.1 »_main()« anstatt »main()«

In reinen Amiga-Programmen (unter Intuition) sollten Sie immer »_main()« anstatt »main()« einsetzen. Der Linker wird dann einen kürzeren Startup-Code einbinden, der nicht mehr die Standard-I/O-Kanäle öffnet. Damit sind zwar Funktionen wie »printf()« nicht mehr verwendbar, aber die brauchen Sie ja nicht unter Intuition, es sei denn Sie arbeiten mit dem Trick aus Kapitel 6.6, aber da nehmen wir ja »sprintf()«.

9.9.2 Amiga.lib zuerst

Wenn Sie »printf()« und Konsorten einsetzen wollen, dann sollten Sie zuerst in Kapitel 6.6 nachlesen, ob die abgemagerte Version im ROM nicht ausreicht. Wenn das der Fall ist, linken Sie »amiga.lib« vor Ihre C-Lib und schon wird der ganze Printf-Code nicht mehr in Ihr Programm eingebunden.

9.9.3 Den ROM und die »Protos« nutzen

Wieso soll man eigentlich die aufwendigen Libraries der Unix-Welt einbinden, wo doch der Amiga entsprechende Funktionen schon im ROM hat? Ein »printf(„Hello World")« in ein File namens »hallo.c« einfach so mit »lc-L hallo« kompiliert und gelinkt, erzeugt einen Programm-Code von 5320 Byte (Lattice-C, V. 4.0). Das gleiche Ergebnis erreiche ich jedoch auch mit 196 Byte, wohl gesagt, immer noch in C. Nur in Assembler geht das noch kürzer (144 Byte). Abb. 9.3 zeigt die Lösung.

```
#include "libraries/dos.h"
#include "proto/exec.h"
#include "proto/dos.h"

#define ABSEXECBASE ((struct ExecBase **)4L)
struct DosLibrary  *DOSBase;

void hallo()
{
   if (!(DOSBase = (struct DosLibrary *)
            OpenLibrary("dos.library",0)))
      return;

   Write(Output(),"Hello World\n",12);

   CloseLibrary((struct Library *)DOSBase);
}
```

Abb. 9.3: Der Kürze-Rekord: Nur 196 Byte Code.

Typisch für die Lattice-Version 4 sind die Includes aus dem Proto-Directory. Damit wird ein Dilemma von C ausgeschaltet, nämlich: C bringt alle Parameter einer Funktion auf den Stack, die Systemroutinen erwarten Sie jedoch in Registern. Folglich muß der Compiler (Linker) Code einbauen, der die Parameter vom Stack holt und in Register umlädt. Genau das wird vermieden, wenn man die Proto-Include-Files anzieht. Der Aufruf der Systemroutinen sieht dann allerdings auch nicht mehr sehr nach C aus, sondern eher wie Assembler-Makros.

Wenn Sie den Text als »hallo.c« speichern, können Sie das Programm compilieren und linken mit

```
lc -v -y hallo.c
blink hallo.o
```

Maßgebend ist dabei, daß die Ausgabe über die Write()-Funktion des AmigaDOS läuft. Weshalb soll ich da also mit »Amiga.Lib« linken? Nach dem Kompilieren mit der v- Option des Lattice-C (kein Code zum Stack-Checking), die hier obligatorisch ist, reicht deshalb auch ein schlichtes »blink hallo.o« Der Linker fügt jetzt nur den Header (16 Byte) hinzu, damit der Amiga »hallo« als ausführbares Programm anerkennt.

9.10 Compiler/Linker-Aufruf im Quelltext

Den folgenden Trick habe ich zuerst auf einer Public-Domain-Disk gesehen und bestaunt. Nach dem dritten Hingucken hatte ich es dann kapiert. Der Trick stammt von John Meissen. Setzen Sie doch einmal an den Anfang des Listings von Abb. 9.3 diese Zeilen:

```
echo; /*
 lc -v -y hallo.c
 blink hallo.o
 quit
*/
```

Das Programm wird jetzt kompiliert und gelinkt (Name hallo.c) indem man aufruft:

```
execute hallo.c
```

Wie funktioniert das?

Das Execute des CLI gibt nach »echo« eine Leerzeile aus, den Rest der Zeile ignoriert es wegen des Semikolons, weil das im CLI einen Kommentar einleitet. Danach führt Execute die Compile- und Link-Anweisungen aus, mehr nicht, denn jetzt folgt »quit«.

Der C-Compiler sieht »echo;« und legt deshalb eine Integer-Variable »echo« an. Für ihn sind die folgenden Zeilen wegen des »/*...*/« aber Kommentare. Sie können »echo« durchaus auch als C-Variable benutzen. Nur um zu zeigen, daß es geht, schreiben Sie doch einmal für die Ausgabe anstatt

```
Write(Output(),"Hello World\n",12);
```

jetzt einmal die Zeilen

```
echo = (int) Output();
Write(echo,"Hello World\n",12);
```

9.11 Grafiken speichern – ganz einfach

Es gibt zwar einige Standard-Formate wie das »IFF« zum Speichern und Laden von Grafiken, doch a) können Sie darüber in allen möglichen Büchern etwas lesen und b) ist die Sache recht kompliziert und mit viel Overhead verbunden. Häufig genug will man aber nur seine Bilder auf der Disk ablegen und wieder von da holen, also sozusagen das Format mit einem Programm koppeln. Doch das kann man auch einfacher haben.

Zur Demonstration des Verfahrens soll das Kunstwerk von Abb. 9.4 erzeugt, auf eine Disk geschrieben und dann wieder eingelesen werden.

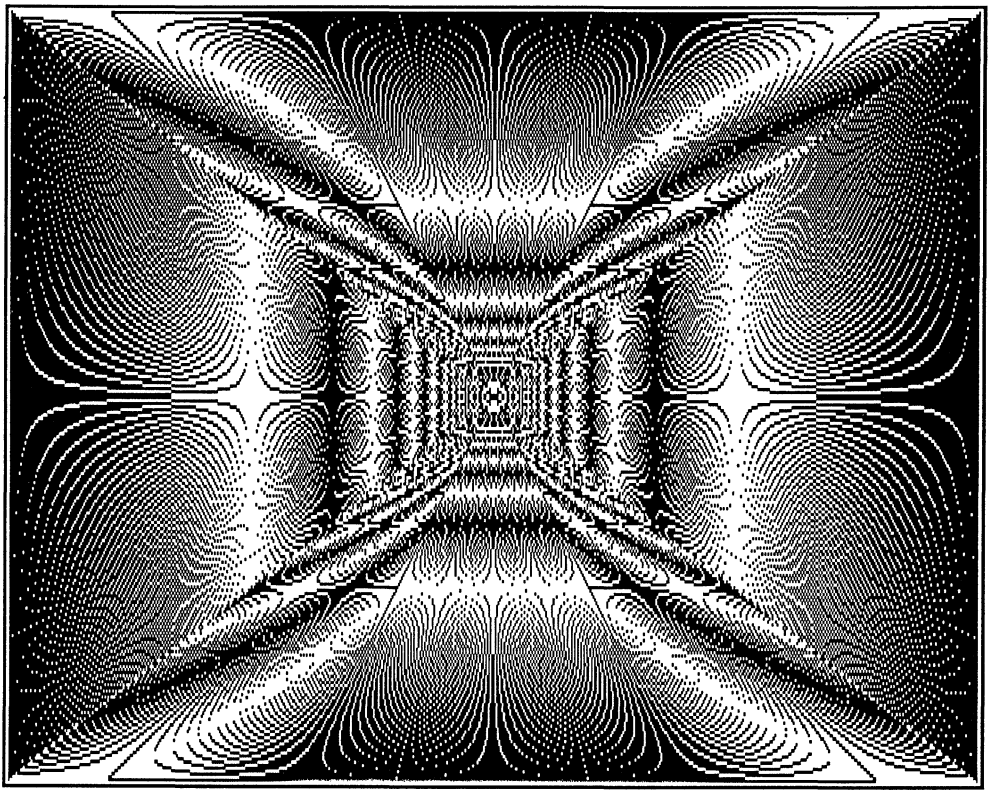

Abb. 9.4: Diese Grafik soll auf einer Disk gespeichert werden

Das Listing von Abb. 9.5 hat die Aufgabe, die Grafik zu erzeugen und auf die Disk zu schreiben.

9.11.1 Grafik-Tricks mit dem Complement-Modus

Die Grafik selbst generiert die Funktion make_bild(), und die ist schon ein Trick für sich. Jede Schleife zieht diagonale Linien über den Schirm, die Sie sich wie X-se vorstellen können, die

immer durch den Bildmittelpunkt gehen und sich mit jedem Durchgang weiter öffnen. Die erste Schleife ist für die nach links, die zweite für die nach rechts gekippten X-se zuständig. Wäre nun nicht der spezielle Draw-Modus eingeschaltet, wäre das Bild nach dem Vorgang voll mit der Zeichenfarbe gefüllt. Der Trick ist nun der COMPLEMENT-Modus. Damit werden die eben gezeichneten Linien von den neuen komplementiert, quasi stellenweise gelöscht. Das Muster entsteht dadurch, daß die schreibenden und löschenden Linien verschieden lang sind.

```
/* put_bild.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>
#include <libraries/dos.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

struct Window *Window;
struct RastPort *rp;
struct Screen *Screen;

#include "stdwd.h" /* Siehe Kapitel 5.7,
              Abb. 5.2          */
void make_bild()
{
  int i, xmax, ymax;

  xmax = Window->GZZWidth-1;
  ymax = Window->GZZHeight-1;

  SetRast(rp, 2);
  SetDrMd(rp, COMPLEMENT);

  for(i=0; i<xmax; i++) {
    Move(rp, i,0); Draw(rp,xmax-i,ymax);
  }
  for(i=0; i<ymax; i++) {
    Move(rp, 0, ymax-i); Draw(rp, xmax, i);
  }
}


void _main()
{
  int  size;
  ULONG handle;

  open_libs();
```

```
Window = (struct Window *) open_window(0,0,640,256,
    NULL,
    GIMMEZEROZERO | ACTIVATE,
    VANILLAKEY,
    NULL
   );
  if (Window == NULL) exit();

  rp = Window->RPort;

  make_bild();

  if ((handle = Open("Abb.", MODE_NEWFILE)) == 0)
    { close_all();
      exit();
    }
  Screen = Window->WScreen;
  rp = &Screen->RastPort;
  size = RASSIZE(640,256);

  Write(handle, (UBYTE *) rp->BitMap->Planes[0], size);
  Write(handle, (UBYTE *) rp->BitMap->Planes[1], size);
  Close(handle);

  Wait(1L << Window->UserPort->mp_SigBit);
  close_all();
}
```

Abb. 9.5: So kann man ein Bild aufzeichnen

9.11.2 Zugriff auf den Workbench-Screen

Aufgezeichnet werden soll der ganze Screen. Den Zeiger darauf kennen wir aber nicht, weil wir keinen eigenen Screen geöffnet haben. Das macht nichts, denn Intuition notiert netterweise in der Window-Struktur einen Zeiger auf den Screen (hier Workbench- Screen). Nachdem wir Screen als einen Zeiger auf eine Screen-Struktur deklariert hatten, können wir uns die Adresse beschaffen mit

```
Screen = Window->WScreen;
```

9.11.3 Zugriff auf die Bit-Planes

Nun brauchen wir den Rastport des Screens. Der eigentlich für das Window deklarierte Rastport-Zeiger »rp« kann dafür verwendet werden (Rastport ist Rastport). Daraus folgt:

```
rp = &Screen->RastPort;
```

Im Rastport gibt es einen Zeiger auf die BitMap-Struktur. In dieser wiederum befindet sich ein Array von Zeigern auf die Bit-Planes. Diese Planes schließlich sind nun (endlich) die RAM-Bereiche, in denen das Bild gehalten wird. Der Workbench-Screen hat 2 Planes (0 und 1). Jedes Bit in jeder Plane entspricht genau einem Bildpunkt. Die aus den 2 Bit möglichen 4 Kombinationen entsprechen 4 Farbregister-Nummern. An eine Plane kommt man nun mit (Beispiel Plane 0)

```
rp->BitMap->Planes[0]
```

heran. Um diesen RAM-Bereich auf ein File schreiben zu können, müssen wir es nur der Write-Funktion von AmigaDOS als Parameter mitgeben. Diese Funktion will natürlich auch noch die Größe des Bereichs wissen, die mit

```
size = RASSIZE(640,256);
```

ermittelt wird. Für »size« hätte ich natürlich auch 640*256/8=20480 schreiben können, aber das eingebaute Makro »RASSIZE(640,256)« sieht natürlich viel eleganter aus und erspart das Kopfrechnen.

Da die Write-Funktion einen Zeiger auf einen Puffer vom Typ UBYTE (oder char) erwartet, machen wir noch etwas »type casting to keep the compiler happy«. Das ergibt dann den schönen Ausdruck

```
Write(handle, (UBYTE *) rp->BitMap->Planes[0], size);
```

für die erste Plane (zählt ab 0). Mit zwei oder mehr dieser Anweisungen geht auch das ganze Bild auf die Disk. Ab drei Planes lohnt es sich, eine solche Zeile in eine Schleife mit »Planes[i]« zu setzen. Wenn Sie das Ganze perfekt erledigen wollen, können Sie natürlich die aktuelle Auflösung erfragen (Screen-Typ) und die Anzahl der Planes (in BitMap).

9.11.4 Zugriff auf die Farb-Register

Auffallender ist jedoch, daß ich in diesem Beispiel gar nichts zu den Farben sage. Wenn Sie das wollen, müssen Sie die Ist-Farben mit GetRGB4() erfragen, diese mitspeichern und dann mit SetRGB4 setzen.

Per Prinzip geht das so

```
for(i=0; i<=31; i++) /*alle 32 Register */
  farbe[i] = GetRGB4(&Screen->ViewPort,i);
```

Die Farben sind als Funktionsergebnis codiert in der Form »0x0RGB«. Zum Setzen mit

```
SetRGB4(&Screen->ViewPort,i,R,G,B)
```

müssen Sie dann die Nibbles aus den Worten holen. Da ich dazu schon wahrhaft abenteuerliche arithmetische Übungen gesehen habe, hier eine C-gemäße Lösung (Farbwerte in f):

```
R = f >> 8 & 0xF;
B = f >> 4 & 0xF;
G = f & 0xF;
```

9.11.5 Das Bild wieder laden

Nun, wir speichern das Bild ohne Farben und hoffen, daß zwischenzeitlich niemand andere Workbench-Farben setzt. Dann können wir nämlich ganz einfach das Bild wieder laden, wobei das dann so wie in Abb. 9.6 aussieht.

```
/* get_bild.c */
  .
  .
  .
  if ((handle = Open("Abb.", MODE_OLDFILE)) == 0)
    { close_all();
      exit();
    }

  Read(handle, (UBYTE *) rp->BitMap->Planes[0], size);

  for(i=0; i<500000; i++) /* Warten dient nur zur */
   ;                      /* Demonstration        */
   Read(handle, (UBYTE *) rp->BitMap->Planes[1], size);
  .
  .
  .
```

Abb. 9.6: Laden des Bildes von der Disk (Auszug)

Im Prinzip bleibt alles wie bisher, nur daß anstatt Write() jetzt Read() eingesetzt wird. Zwischen den beiden Read()-Aufrufen habe ich eine Pause eingebaut, aber nur, damit Sie sehen, daß auch schon nach dem Laden von Planes[0] ein Bild entsteht. Allgemein gilt: Für monochrome Darstellungen (nur 1 Farbe) reicht eine Plane.

9.12 Programm muß Preferences kontrollieren

Als Ergänzung zum vorigen Tip (9.11) kann es durchaus sinnvoll sein, auch die Bild-Einstellung in Preferences zu notieren und ggf. vor dem Laden wiederherzustellen. Sie können nämlich

durchaus die Daten in Preferences von Ihrem Programm aus ändern, und Intuition reagiert sofort darauf, indem es die neuen Werte berücksichtigt. Wenn ein Programm richtig geschrieben ist, verhält es sich übrigens immer so.

9.12.1 Die Preference-Struktur

Preferences ist (wie üblich) eine Daten-Struktur, schauen wir uns diese zunächst in Abb. 9.7 an.

```
struct Preferences
{
 BYTE   FontHeight; /* Höhe des System-Fonts (8 oder 9) */
 UBYTE  PrinterPort; /* Drucker an paral. oder ser. Port */
 USHORT BaudRate; /* der RS-232 */
 struct timeval KeyRptSpeed; /* Tasten-Wiederholrate */
 struct timeval KeyRptDelay; /* Ansprech-Verzögerung davon */
 struct timeval DoubleClick; /* Tempo Doppelklick */
 USHORT PointerMatrix[POINTERSIZE]; /*Mauszeiger-Matrix */
 BYTE   XOffset; /* X-Offset Hot Spot */
 BYTE   YOffset; /* Y-Offset  */
 USHORT color17; /*Farben Mauszeiger */
 USHORT color18;
 USHORT color19
 USHORT PointerTicks; /* Mausuntersetzung */
 USHORT color0; /Farben der Workbench */
 USHORT color1;
 USHORT color2;
 USHORT color3;
 BYTE   ViewXOffset; /* Bildlage x */
 BYTE   ViewYOffset; /*  y */
 WORD   ViewInitX, ViewInitY; /* Ursprungswerte */
 BOOL   EnableCLI; /* CLI-Schalter */
 USHORT PrinterType; /* Code */
 UBYTE  PrinterFilename[FILENAME_SIZE]; /*File-Name*/
 /* Nun folgen die diversen Drucker-Parameter, */
 /* siehe hierzu Konstanten im Anhang */
 USHORT PrintPitch;
 USHORT PrintQuality;
 USHORT PrintSpacing;
 UWORD  PrintLeftMargin;
 UWORD  PrintRightMargin;
 USHORT PrintImage;
 USHORT PrintAspect;
 USHORT PrintShade;
 WORD   PrintThreshold;
 USHORT PaperSize;
 UWORD  PaperLength;
 USHORT PaperType;
```

```
/* Nun wird's geizig: jedes Datum steckt in einem Nibble */
/* o == oberes, l == unteres Halb-Byte    */

UBYTE  SerRWBits;  /* o: Read Bits */
          /* u: Write Bits */
UBYTE  SerStopBuf; /* o: Stop Bits */
          /* u: Tabellenplatz Puffergröße */
UBYTE  SerParShk;  /* o: Parity */
          /* u:Handshake Modus */
UBYTE  LaceWB;     /* 0x01 wenn INTERLACE */
UBYTE  WorkName[FILENAME_SIZE]; /* Temp. File f. Drucker */
BYTE   sys_reserved1;
BYTE   sys_reserved2;

/* Druckerdaten für Hardcopies: */
UWORD  PrintFlags;
UWORD  PrintMaxWidth;
UWORD  PrintMaxHeight;
UBYTE  PrintDensity;
UBYTE  PrintXOffset;
UWORD  wb_Width;
UWORD  wb_Height;
UBYTE  wb_Depth;

UBYTE  ext_size; /* Nicht ändern! */
BYTE   ext_bytes[PREF_EXTBYTES];
};
```

Abb. 9.7: Die Preference-Struktur

Für die einzelnen Einträge sollten Sie möglichst die in den Include-Files vordefinierten Konstanten verwenden.

9.12.2 Der Zugriff auf Preferences

So, nun wissen wir, was da alles steht, bliebe die Frage des Zugriffs. Intuition bietet dafür folgende Funktionen:

```
GetPrefs(Puffer, Anzahl)
```

und

```
GetDefPrefs(Puffer, Anzahl);
```

GetPrefs() liest die aktuellen Preference-Daten, GetDefPrefs() hingegen die auf der Disk abgespeicherten. Puffer ist ein Speicherbereich, den Sie bereitstellen müssen, Anzahl sagt, wie viele

Bytes eingelesen werden sollen. Die Idee dahinter ist, daß sicherlich nicht immer alle Daten benötigt werden und die wichtigsten ziemlich weit vorne stehen.

Das Gegenstück ist

```
SetPrefs(Puffer,  Anzahl,  Disk-Flag);
```

Damit werden »Anzahl« Bytes aus dem Puffer in die Preference-Struktur geschrieben. Ist »Disk-Flag« TRUE, gehen die Daten in den RAM und auf die Disk, im Falle von FALSE nur in den RAM. Dies entspricht der Auswahl zwischen Save und Use im Preference-Requester. Abb. 9.8 zeigt eine praktische Anwendung.

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

struct Window *Window;
struct RastPort *rp;

#include "stdwd.h" /* Siehe Kapitel 5.7,
              Abb. 5.2         */
void _main()
{
  struct Preferences pref;

  int s = sizeof(pref);
  int i;
  BYTE x;

  open_libs();

  Window = (struct Window *) open_window(0,0,640,256,
    " bitte warten...",
    WINDOWCLOSE | WINDOWSIZING | ACTIVATE,
    CLOSEWINDOW | VANILLAKEY,
    NULL
   );
  if (Window == NULL) exit();

  GetPrefs(&pref,s);
  x = pref.ViewXOffset; /* Wert retten */

  for(i=1; i<512; i++) {
     pref.ViewXOffset = (BYTE) i+x;
```

```
      SetPrefs(&pref,s,FALSE);
   };

   pref.ViewXOffset = x; /* alter Wert zurück */
   SetPrefs(&pref,s,FALSE);

   SetWindowTitles(Window," Fertig", -1);

   Wait(1L << Window->UserPort->mp_SigBit);
   close_all();
}
```

Abb. 9.8: Zugriff auf Preferences total

Das Programm soll einfach das Bild in X-Richtung verschieben, sprich, andere Werte in »ViewXOffset« eintragen. Da es sich um einen Eintrag vom Typ »signed Byte« handelt, dürfen nur Werte von –128 bis +127 eingetragen werden. Da ich zu faul bin, dafür For-Schleifen für hin und zurück zu schreiben, ein kleiner Trick. Ich lasse die Schleife mittels »int i« bis 512 laufen, »caste« dann aber in BYTE. Im Ergebnis sehen Sie das Bild dreimal von links nach rechts über den Schirm laufen.

In diesem Listing habe ich einfach die ganze Preference-Struktur (Variable pref) genommen, was einen Puffer der Größe »sizeof(pref)« (ein paar hundert Bytes) erfordert. Will man ein paar Bytes Speicher und einige Millisekunden Ladezeit sparen, kann man wie in Abb. 9.9 gezeigt, vorgehen.

```
void _main()
{

   BYTE pref[0x77];
   int s = sizeof(pref);
   int i;
   BYTE x;
   int ViewXOffset = 0x76;
   .
   .
   .
   GetPrefs(&pref,s);
   x = pref[ViewXOffset];
```

Abb. 9.9: Zugriff auf Preferences byte-sparend (nur Auszug)

Dazu muß man halt abzählen, wie weit man in die Struktur hinein will und lädt dann nur so viele Bytes. Viel mehr als 200 Byte bringt aber dieser so beliebte Trick auch nicht.

9.13 Ausgabe von Datum und Zeit

Nun schlägt es 13, weshalb der Abschnitt auch 9.13 heißt. Doch ernsthaft: An der Ausgabe von Datum und Zeit kommt man wohl oft nicht vorbei. Das Problem läßt sich allerdings sehr elegant mit Hilfe von Intuition lösen. Es bietet dazu eine Funktion dieser Form:

```
ULONG Sekunden, Mikrosekunden;
CurrentTime(&Sekunden, &Mikrosekunden);
```

Die Funktion ergibt die Systemzeit in Sekunden und Mikrosekunden (millionstel Sekunden). Letzeres ist aber glatt geprahlt, denn die Zeit wird nicht mit dieser Genauigkeit zur Verfügung gestellt. Theoretisch kann Intuition einer Task (einem Window) die Zeit 60mal pro Sekunde melden, praktisch sind es je nach Systemauslastung deutlich größere Intervalle (ca. 10/Sek.).

Gezählt wird die Zeit ab Mitternacht vom 1. Januar 1978 in Sekunden. Unterschätzen Sie das nicht, denn wegen der Long-Variablen reicht das für über 100 Jahre. Die Mathe-Freaks mögen nun einen klugen Algorithmus erfinden, daraus Datum und Uhrzeit zu generieren, nur ich bin dazu zu faul. Statt dessen zeige ich Ihnen lieber mit Abb. 9.10 den Trick, wie man mittels der C-Funktionen time(), localtime() und asctime() viel Tipperei spart. Das sind ANSI-Erweiterungen von C, die inzwischen zu jedem anständigen Compiler gehören, Lattice ab Version 4.0 hat sie zum Beispiel.

```
/* time.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>
#include "time.h"

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

struct Window *Window;
struct RastPort *rp;

#include "stdwd.h" /* Siehe Kapitel 5.7,
              Abb. 5.2          */

void _main()
{
  struct tm *tp;
  long t;

  open_libs();
```

```
  Window = (struct Window *) open_window(0,0,640,256,
    "Die Zeit:",
    WINDOWCLOSE | WINDOWSIZING | ACTIVATE,
    CLOSEWINDOW | VANILLAKEY,
    NULL
   );   if (Window == NULL) exit();

  rp = Window->RPort;
  SetDrMd(rp, JAM1);
  SetAPen(rp, 2L);
  Move(rp,110,7);

  time(&t);
  tp = localtime(&t);
  Text(rp, asctime(tp),24);


  Wait(1L << Window->UserPort->mp_SigBit);
  close_all();
}
```

Abb. 9.10: Datum und Uhrzeit im Window-Titel

Das kleine Programm hat die Aufgabe, Datum und Zeit in die Titel-Leiste des Windows zu schreiben. Die Funktion asctime() liefert an sich einen String von 26 Zeichen. Da dieser für printf() mit »\n\0« am Ende vorbereitet ist, geben wir effektiv nur 24 Zeichen aus.

Nun kann man anstatt time() auch CurrentTime() einsetzen, nur hätten wir da zwei Probleme. Zum einen startet laut ANSI die Zeit nicht 1978 sondern 1970. Die 8 Jahre (nebst anderer Zeitzone) müssen wir addieren. Zum zweiten meldet uns Intuition die Zeit auch nur, wenn wir das Flag INTUITICKS setzen.

Letzteres können wir natürlich ausnutzen, um dieses Event in die Event-Schleife aufzunehmen, und dann laufend die Zeit anzuzeigen. Wie das alles zusammen dann aussieht, zeigt Abb. 9.11.

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>
#include "time.h"

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;
struct RastPort *rp;
 struct IntuiMessage *msg;
```

```
ULONG class;
USHORT code;

#include "stdwd.h" /* Siehe Kapitel 5.7,
              Abb. 5.2          */

void _main()
{
  struct tm *tp;
  long t,m;

  open_libs();

  Window = (struct Window *) open_window(0,0,640,256,
    "Die Zeit:",
    WINDOWCLOSE | WINDOWSIZING | ACTIVATE,
    CLOSEWINDOW | VANILLAKEY | INTUITICKS,
    NULL
   );
  if (Window == NULL) exit();

  rp = Window->RPort;

   for(;;)  /* bis CLOSEWINDOW */
   {
     Wait(1L << Window->UserPort->mp_SigBit);
     while( msg = GetMsg(Window->UserPort) )
     {
       class = msg->Class;
       code  = msg->Code;
       ReplyMsg( msg );

       switch(class)
       {
  case CLOSEWINDOW: ClearMenuStrip(Window);
                    close_all();
                    break;

  case INTUITICKS:  CurrentTime(&t,&m);
                    t = t + 252482380L + m/1000000L;
                    tp = localtime(&t);
                    Move(rp,110,7);
                    Text(rp, asctime(tp),24);
                    break;
}
     }
   }

 close_all();
}
```

Abb. 9.11: Laufende Zeitanzeige aus »CurrentTime()«.

Beachten Sie, daß ich hier gar nichts hinsichtlich Zeichen-Farbe und Modus getan habe. Per Voreinstellung ergibt das rote Schrift auf blauem Grund, was in dem weißen Titel-Streifen des Windows recht gut aussieht.

9.14 Richtiges Refresh für Windows

Wenn Sie selbst Ihr Window auffrischen, also neu zeichnen müssen, sollten Sie unbedingt die beiden Funktionen

```
BeginRefresh(Window)
```

und

```
EndRefresh(Window, fertig)
```

einsetzen. Das dürfte immer beim SIMPLE_REFRESH- und manchmal beim SMART_REFRESH-Window der Fall sein. Dahinter steckt folgendes:

Wenn Teile des Windows neu gezeichnet werden müssen, erhalten Sie eine IDCMP-Nachricht der Klasse REFRESHWINDOW bzw. IECLASS_REFRESHWINDOW, wenn Sie das Console-Device nutzen. In beiden Fällen gehen Sie in Ihre Zeichen-Routine, befehlen da zuerst BeginRefresh() und starten dann mit dem Neuzeichnen. Danach geben Sie EndRefresh(). Dadurch wird das Window für die Dauer zwischen den beiden Anweisungen in einen speziellen Zustand gebracht, der das Neuzeichnen sehr stark beschleunigt, weil effektiv nur in die Bereiche gezeichnet wird, die wirklich Refresh benötigen.

Auch wenn Sie nicht neu zeichnen wollen, sollten Sie auf eine REFRESHWINDOW-Message mindestens mit BeginRefresh() und EndRefresh() reagieren, weil das die Ihrem Window unterliegenden Strukturen (Layer) wieder optimal organisiert.

Das Argument »fertig« in EndRefresh ist vom Typ Boolean und dann TRUE, wenn das Refresh fertig ist. Der andere Fall ist theoretisch denkbar, wenn mehrere Tasks an Ihrem Window arbeiten. Da erscheint dann in den einzelnen Tasks so lange FALSE, bis der letzte auch sein Teil zum Refresh beigetragen hat.

9.15 I/O mit AmigaDOS in C

Sicherlich die einfachste Methode für die Ein- und Ausgabe von Texten ist die Anwendung von C-Funktionen, wie printf() oder scanf(). Leider sind wir damit auf das aktuelle CLI-Fenster angewiesen, eine Methode, die wir Window-Freaks inzwischen gar nicht mehr so mögen.

Eine recht bequeme Lösung mit eigenem Window bietet jedoch AmigaDOS. Das kleine Programm von Abb. 9.12 fragt Sie nach Ihrem Namen und gibt dann »Guten Tag, lieber Name« aus. Dazu gehört ein Window mit eigenem Titel, Drag-Gadget und Size-Gadget, und das alles mit diesen wenigen Zeilen:

```
#include <libraries/dos.h>

void _main()
{
 LONG handle, count, len;
 UBYTE buffer[81];

 handle = Open("con:0/20/500/100/ Ein Con-Window ",
        MODE_OLDFILE);
 count = Write (handle, "Wie heißen Sie? ", 17);
 len = Read (handle, buffer, 80);
 count = Write (handle, "\nGuten Tag, lieber ", 19);
 count = Write (handle, buffer,len);
 count = Write (handle, "\n\nTippen Sie Return", 19);
 count = Read (handle, buffer, 1);
 Close(handle);
}
```

Abb. 9.12: I/O mit eigenem Fenster in AmigaDOS

Im Listing von Abb. 9.12 habe ich bewußt auf einige Tests verzichtet, um das Prinzip besser zeigen zu können. Wenn Sie aber nicht Ihren Amiga schon bis zum Rand gefüllt haben, wird er auch so sicherlich das Con-Window noch öffnen. Das I/O läuft im DOS über die Funktionen Read() und Write(). Beide verlangen als Parameter

ein Handle,

die Adresse des Puffer mit den Daten bzw. für die Daten

und die Anzahl der zu übertragenden Bytes.

Beide Funktionen geben die tatsächlich geschriebene bzw. gelesene Anzahl Bytes zurück, eine negative Zahl bedeutet Fehler.

So weit, so gut, nur was ist ein Handle (zu deutsch Griff)? Nun, das ist vereinfacht ausgedrückt die File-Nummer, die man von Open() erhält. Sie können mit Open() ein Disketten-File öffnen. Die Open-Funktion erwartet den File-Namen und die Zugriffsart. MODE_OLDFILE öffnet ein existierendes File, MODE_NEWFILE würde ein neues anlegen. Das Besondere daran ist, daß auch die Console (Tastatur und Bildschirm) in diesem Sinne ein File ist. Da Sie (wie ich hoffe) eine Console haben, ist der Modus MODE_OLDFILE. Nur hier kann man aber auch noch einige

Parameter mehr übergeben, und das sind (durch Schrägstriche getrennt) die Parameter eines Windows. Mit der Anweisung

```
handle = Open("con:0/20/500/100/ Ein Con-Window ",
       MODE_OLDFILE);
```

wird ein Window mit der linken oberen Ecke 0/20, der Breite 500 und der Höhe 100 geöffnet. Das Window erhält den Titel »Ein Con-Window«. Als Ergebnis erhalten wir ein Handle, und damit können wir auf das Window mit Write() und Read() zugreifen.

Das Window hat zwar eine nicht zu übersehende Ähnlichkeit mit einem CLI-Window, das braucht Sie aber nicht daran zu hindern, das Programm von der Workbench aus zu starten (siehe Kapitel 1.15).

Schön einfach, und da plagen wir uns noch mit den aufwendigen Intuition-Windows? Nun, die Sache hat einen Haken, das I/O ist auf Texte (genauer: jede Art von Bytes) beschränkt.

10 Nützliche Programme auf der Extras-Diskette

10.1 MEMACS – Ein Tip für Programmierer

MEMACS ist der ideale Editor für Programmierer, da er – unter vielen anderen Programmierer-Features – erlaubt, mehrere Textdateien gleichzeitig zu bearbeiten. Gerade beim Amiga ist das von besonderer Bedeutung, weil sich viele Dinge, wie der Aufbau von Windows oder Menüs in allen Programmen wiederholen. Da ist es denn sehr praktisch, wenn man die jeweils am besten passende Routine aus verschiedenen anderen Programmen leicht ausschneiden und einsetzen kann.

Typisch für den Einsatz als Programmierer-Tool sind auch die Einschränkungen von EMACS. Die Textgröße ist auf die Größe des Hauptspeichers beschränkt, doch davon haben Programmierer immer genug. Im Zweifelsfall haben Sie viele Mbyte RAM und keine Festplatte, das ganz einfach deshalb, weil Compiler in einer RAM-Disk viel schneller laufen.

Die zweite Einschränkung ist die Zeilenbreite auf 80 Zeichen. Das ist zwar nur visuell so – der Rest ist nur nicht sichtbar bis man die Zeile teilt – doch auch Absicht. Was man auf dem Schirm nicht sofort sieht, wird oft in der ersten Debug-Phase – dem intensiven Blick auf den Quelltext – übersehen. Sie können aber auch davon ausgehen, daß lange Zeilen zwar trickreich aussehen, aber nicht unbedingt den besten Code ergeben.

Die Bedienung von MEMACS ist dank der Pull-down-Menüs recht einfach, Sie sollten jedoch zuerst die Bedeutung einiger Begriffe und das grundsätzliche Konzept dieses Editors verstehen.

10.1.1 Puffer

In MEMACS können Sie verschiedene Texte gleichzeitig bearbeiten. Jeder Text wird in einem Puffer (buffer) gehalten. Mehrere dieser Puffer können auf dem Schirm abgebildet werden. Dazu wird der Schirm in Ausschnitte geteilt. Diese MEMACS-Fenster sind keine Intuition-Fenster, sondern halt nur Rahmen. Immerhin können Sie aber mit der Maus zwischen diesen Fenstern umschalten und innerhalb eines Fensters per Mausklick den Cursor positionieren.

Ansonsten können Sie über den Befehl »Select-buffer« in einen anderen Text gehen. Dazu müssen Sie den Namen des Puffers eintippen, der nicht unbedingt der Dateiname ist. Wenn MEMACS

ohne Angabe eines Dateinamens gestartet wird, legt er einen Puffer namens »main« an. Der Dateiname wird erst beim Abspeichern vergeben, das würde dann auch »main« durch den echten Namen ersetzen. Der aktuelle Dateiname wird immer in der letzten Zeile angezeigt.

10.1.2 Zwei Betriebsarten

Wie jeder Editor kennt auch MEMAC zwei Betriebsarten, nämlich »Normal« und »Befehl«. Im Normal-Modus können Sie Text eingeben und editieren, im Befehls-Modus, der durch die Menüanwahl oder Tastenkürzel aktiviert wird, wartet MEMACS ggf. noch auf Eingaben, zum Beispiel auf den File-Namen nach dem Befehl »Read-File«. Die Befehls- bzw. Eingabezeile ist die letzte Schirmzeile. Typisch wird darin das Kommando wiederholt, zum Beispiel erscheint dann

```
read file:
```

woraufhin Sie den File-Namen – ggf. mit komplettem Pfad – eingeben müssen.

10.1.3 »Read-file« und »Visit-File«

Zwei Befehle müssen Sie strikt unterscheiden:

»Read-File« liest eine Datei in den aktuellen (gerade aktiven) Puffer und ersetzt damit dessen vorherigen Inhalt.

»Visit-File« legt einen neuen Puffer an und liest dann die Datei in diesen Puffer ein.

10.1.4 »Dot« und »Mark«

Der Punkt (Dot) ist die aktuelle Cursor-Position im Textpuffer. Jeder Puffer hat seinen eigenen Punkt. Der Menü-Befehl »Set mark« setzt eine Marke auf den aktuellen Punkt. Wenn Sie jetzt den Cursor (den Punkt) weiterbewegen, bleibt die Marke stehen, auch dann, wenn Sie Text löschen und einfügen.

Der Text zwischen der Marke und dem Punkt ist ein Block, der kopiert, bewegt oder gelöscht werden kann.

10.1.5 »Yank«

MEMACS verfolgt wie viele andere Editoren – zum Beispiel auch Microsofts Word – folgende Methode zum Kopieren und Bewegen von Blöcken:

Der Block wird zuerst gelöscht und gleichzeitig in eine Zwischenablage kopiert. Der Befehl »Einfügen« kopiert die Zwischenablage wieder in den Text und zwar unmittelbar vor den Cursor. Daraus resultieren diese Techniken (in Klammern stehen die MEMACS-Befehle):

Block markieren:
- Cursor auf Blockbeginn
- Marke setzen (Set-mark)
- Cursor auf Blockende

Block löschen:
- Block markieren
- Löschen (Kill-region)

Block verschieben:
- Block markieren
- Löschen (Kill-region)
- Cursor bewegen
- Einfügen (Yank)

Block kopieren:
- Block markieren
- Löschen (Kill-region)
- Einfügen (Yank)
- Cursor bewegen
- Einfügen (Yank)

Wird zwischen dem Einfügen (Yank) der Puffer gewechselt, wird damit der Block von einem Text in den anderen kopiert bzw. verschoben.

10.1.6 Anmerkungen zu den Menüs

Die meisten Menüpunkte sind selbsterklärend, einiges ist jedoch anzumerken.

.1 Das Project-Menü

Hier sind alle Befehle versammelt, die den aktuellen Puffer komplett beeinflussen, Ausnahme: »Visit-file«.

Rename

Die Datei des aktuellen Puffers wird umbenannt. Geben Sie allerdings auf den Prompt »new file name« nur <Return> ein, wird die Verbindung vom Puffer zur Datei aufgehoben. War das nicht Ihre Absicht, müssen Sie mit »Save-file-as« die Verbindung wiederherstellen.

Read-File

Die eingelesene Datei überschreibt den aktuellen Pufferinhalt.

Visit-File

Die eingelesene Datei öffnet einen neuen Puffer. Das geht aber nur, wenn schon wenigstens eine Datei einem Puffer zugeordnet ist.

Insert-File

Fügt die Datei in den aktiven Puffer ein, und zwar ab der Zeile oberhalb des Cursors.

Insert-buffer

Fügt den Puffer (dessen Namen anzugeben ist) in den aktiven Puffer ein, und zwar ab der Zeile oberhalb des Cursors.

Save-File

Sichert den aktuellen Puffer. Vorher mußte aber mit »Save-as-File« dem Puffer eine Datei zugeordnet werden. Ist das nicht der Fall, wird nur der Fehler »No File Name« gemeldet.

Save-mod

Schreibt den Inhalt aller Puffer, die seit dem letzten »Save« modifiziert wurden, auf die Disk.

New-CLI

Macht ein CLI-Window auf, in dem Sie wie üblich arbeiten können. Nach »ENDCLI« landen Sie wieder in MEMACS.

CLI-Command

Fordert zur Eingabe eines einzigen CLI-Befehls auf. Nach dessen Ausführung landen Sie automatisch wieder in MEMACS.

.2 Das Edit-Menü

Mit dem Edit-Menü werden die Puffer und die ihnen zugeordneten Dateien bearbeitet.

Kill-region

Löscht den markierten Block und speichert ihn in der Zwischenablage (kill buffer). Der Befehl kann für verschiedene Blöcke wiederholt werden, die dann nacheinander in der Zwischenablage gespeichert werden. Siehe auch Abschnitt 10.1.5.

Yank

Kopiert den Inhalt der Zwischenablage in den Puffer, und zwar ab der Zeile oberhalb des Cursors. Siehe auch Abschnitt 10.1.5.

Set-mark

Markiert die Cursor-Position eines Puffers. Ab diesem Zeitpunkt wird jede andere Cursor-Position als »Dot« (Punkt) bezeichnet. Der Text zwischen »mark« und »dot« ist ein Block. Siehe auch Abschnitt 10.1.5.

Die Marke kann auch mit einem Doppelklick der Maus gesetzt werden. In beiden Fällen bestätigt MEMACS das mit einem »mark set«.

Copy-region

Kopiert einen Block in die Zwischenablage. Im Gegensatz zu »Kill-region« wird damit die Zwischenablage überschrieben.

Upper-region

Wandelt alle Buchstaben des aktuellen Blocks in Großbuchstaben um.

Lower-region

Wandelt alle Buchstaben des aktuellen Blocks in Kleinbuchstaben um.

Kill-buffer

Löscht einen Puffer, der nicht der aktuelle sein darf. Es ist manchmal nötig, einen oder mehrere Puffer zu löschen, weil alle im RAM gehalten werden, und der Speicher dafür ggf. nicht mehr ausreicht.

Justify-buffer

Der Text wird linksbündig justiert, doch Achtung! MEMACS entfernt dann alle Blanks und Tabs an den Zeilenanfängen.

Quote-char

Sollen Control-Codes in den Text eingefügt werden, zum Beispiel zur Druckersteuerung aber auch für Makros, braucht man als Einleitungszeichen, damit MEMACS es nicht für das Tastenkürzel eines Menübefehls hält.

Indent

Bewegt den Cursor auf die nächste Zeile und fügt da so lange Blanks ein, bis der Cursor unter dem ersten Zeichen der vorherigen Zeile steht.

Indent

Tauscht das Zeichen unter dem Cursor mit dem davorliegenden.

Cancel

Bricht einen Befehl ab.

.3 Das Window-Menü

Die Punkte sind recht klar, weshalb hier noch die Übersetzung folgt:

One-window:	Aktuelles Window auf volle Größe
Split-window:	Aktuelles Window teilen
Next-window:	Cursor in nächstes Window
Prev-window:	Cursor in vorheriges Window
Expand-window:	Window um 1 Zeile länger
Shrink-window:	Window um 1 Zeile kürzer
Next-w-page:	Im nächsten Window (dem nicht aktiven) eine Seite weiter.
Prev-w-page:	Im vorigen Window (dem nicht aktiven) eine Seite zurück.

.4 Das Move-Menü

Diese Menü-Punkte dienen der schnellen bzw. weiten Bewegung des Cursors. Es bedeuten:

Top-of-buffer:	Auf die erste Zeile
End-of-buffer:	Auf die letzte Zeile
Top-of-Window:	Auf Window-Beginn
End-of-window:	Auf Window-Ende
Goto-Line:	Auf Zeilennummer ...
Swap-dot&mark:	Cursor-Position wird Marke, Cursor geht auf vorherige Marke.
Next-page:	Eine Window-Seite weiter
Prev-page:	Eine Window-Seite zurück
Next-word:	Ein Wort weiter
Prev-word:	Ein Wort zurück
Scroll-up:	Window um 1 Zeile nach oben
Scroll-down:	Window um 1 Zeile nach unten

.5 Die Menüs »Line« und »Word«

Diese Menüs enthalten Kürzel, um eine Zeile oder ein Wort schneller editieren zu können. Da die Punkte nicht oft benötigt werden und sie selbsterklärend sind, verzichte ich auf die Übersetzung.

.6 Das Search-Menü

Hier geht es um das Suchen und Ersetzen von Strings im aktuellen Puffer. Beim Suchen wird standardmäßig nicht zwischen Groß- und Kleinschreibung unterschieden (läßt sich mit »set« ändern), ersetzt wird jedoch immer so, wie Sie den Text eingetippt haben.

Search-forward

Fragt mit »search:« nach dem Suchstring und sucht ihn ab Cursor vorwärts.

Search-backward

Fragt mit »search:« nach dem Suchstring und sucht ihn ab Cursor rückwärts.

Search-replace

Fragt mit »search:« nach dem Suchstring und sucht ihn ab Cursor vorwärts. Beim ersten Auftreten des Suchbegriffs kommt die Frage »replace:«, die mit dem Ersatzstring zu beantworten ist. Danach ersetzt MEMACS alle weiteren Suchbegriffe automatisch und meldet zum Schluß die Anzahl mit »Replaced nn occurences«.

Query-s-r

Arbeitet wie »Search-replace«, fragt aber bei jedem Auftreten des Suchbegriffs, ob der ersetzt werden soll. Die Frage

```
Change string (y/n/c/^G)?
```

können Sie beantworten mit

y = Yes (Ja)
n = Nein
c = alle weiteren ohne Frage ersetzen
^G = abbrechen

Fence-match

Sucht das nächste Auftreten des Zeichens auf dem der Cursor gerade steht. Besonders praktisch ist das im C-Modus (siehe set-mode), weil da von der {-Klammer ausgehend die »}« gefunden wird.

.7 Das Extras-Menü

Mit dem Extras-Menü wird die Arbeitsweise von MEMACS gesteuert. Hier können auch Makros definiert werden.

Set-arg

Nach diesen Befehl erscheint das Prompt

```
arg: 4
```

was Sie amerikanisch lesen müssen als »argument four« und das dann falsch als »argument for« also »Argument für«. Tippen Sie dann beispielsweise

```
60-
```

ein, werden 60 Minuszeichen ab der Cursor-Position geschrieben. »60*« macht sich auch gut. Ich nehme das immer für die Trennlinien zwischen Programmabschnitten. Beachten Sie, daß die Zahlen nicht mit dem Ziffernblock eingegeben werden dürfen.

Set

Damit lassen sich einige Standardwerte ändern. Nach »set what:« können Sie eingeben:

Screen: Schaltet den MEMACS-Schirm zwischen einem Intuition-Window und seinem eigenen Screen um. Letzterer bietet mehr Platz.

Interlace: Schaltet diesen Modus ein oder aus. Interlace-Betrieb ist nur mit dem MEMACS-Screen möglich (siehe set screen).

Mode: Mögliche Antworten sind »cmode« für C-Programme und »wrap« für automatischen Wortumbruch.

Left: Fragt nach Spalte für den linken Rand.

Right: Fragt nach Spalte für den rechten Rand.

Tab: Fragt nach Tab-Abstand.

Indent: Fragt nach Anzahl Blanks für Einrückungen je Stufe im C-Modus.

Case: Schaltet Groß-/Kleinbuchstaben-Berücksichtigung beim Suchen an (ON) oder aus (Default ist aus).

Backup: Die möglichen Optionen sind:
OFF = Keine Sicherung durch MEMACS (Standard).
ON = Benennt die aktuelle Datei beim Speichern in »name.bak« um und speichert sie in »T:« (Vorsicht, meistens RAM).

SAVE: Eine vorhandene Datei wird nicht überschrieben.

Start-macro / Stop-macro

Alle Tastenbetätigungen und Menü-Befehle zwischen diesen beiden Kommandos werden als Makro aufgezeichnet.

Execute-macro

Führt das Makro aus, d.h. alle Tastenbetätigungen und Menü-Befehle der Aufzeichnung werden wiederholt.

Set-key

Belegt eine Taste mit einem Makro. Belegbar sind die Funktionstasten (nicht zu empfehlen), die Funktionstasten mit Shift, die Help-Taste und der ganze Ziffernblock (rechts). Sie können bis 80 Zeichen auf eine Taste legen, Menü-Befehle inklusive. Sie müssen zuerst die zu belegende Taste drücken und dann den Text eingeben. Control-Codes müssen jeweils mit <Ctrl>+<u> eingeleitet werden (siehe Quote-char).

10.1.7 Die Tastenkürzel

MEMACS kann über Menüs bedient werden, doch wenn Sie die Tastenkürzel erst kennen (und in den Fingern haben), können Sie sehr viel schneller werden. Die Tastenkürzel stehen neben den Menüpunkten und zwar in einer Kurzschreibweise, die Sie wie folgt interpretieren müssen.

Ein Kürzel wie

```
^X^S
```

heißt

```
<Ctrl>+<X> <Ctrl>+<S>
```

Also: <Ctrl> und <X> gleichzeitig drücken, dann <Ctrl> loslassen und jetzt <Ctrl> und <S> gleichzeitig drücken.

Wird eine Steuerfolge mit Escape eingeleitet, zum Beispiel als

```
ESC^V
```

heißt das

```
<Esc> <Ctrl>+V
```

Also: <Esc> drücken und loslassen, dann <Ctrl> und <V> gleichzeitig drücken.

10.1.8 Die Start-Datei

MEMACS führt bei jedem Start die Datei »emacs_pro« (ohne m) aus, wenn sie im selben Verzeichnis oder in »s:emacs« vorhanden ist. Hierfür müssen die Texte der Menü-Befehle eingesetzt werden. Zum Beispiel können Sie diese Datei dafür erstellen (mitgeliefert wird keine):

```
Set mode cmode
set case on
```

Sie können eine solche Datei auch von MEMACS aus mit »Execute-file« ausführen lassen. Zum Ausprobieren einzelner Zeilen einer solchen Datei können Sie »Execute-line« verwenden.

10.2 FED, der Font-Editor

Fehlen Ihnen bestimmte Sonderzeichen, welche die ANSI-und Amiga-Norm – aus natürlich völlig unerfindlichen Gründen – nicht berücksichtigt haben?

Kein Problem, denn Sie haben a) einen Amiga und b) den Font-(Zeichensatz-)Editor FED. Im Gegensatz zum PC werden beim Amiga die Zeichen nicht aus einem ROM gelesen, sondern aus Dateien auf der Workbench-Diskette. Solche »weiche Ware« kann man natürlich ändern, und der mitgelieferte FED ist extra dafür da.

Eine Regel sollten Sie allerdings nie vergessen. Ist nämlich ein Font erst einmal »verhunzt«, ist es sehr mühsam, den Originalzustand wiederherzustellen. Arbeiten Sie also immer nur mit einer Kopie der Workbench-Diskette.

10.2.1 Was FED nicht kann

FED kann keine Farb-Zeichensätze bearbeiten und auch keine Fonts, die größer als 32 Bildpunkte sind. Ferner sollten Sie schon jetzt notieren, daß alle Menüpunkte zu den Stilarten, Beispiel »Make italic«, und zur Lage, Beispiel »All Right«, immer auf den ganzen Font wirken. Natürlich können Sie ein einzelnes Zeichen völlig neu und dann in jeder Stilart malen.

10.2.2 Der Start

FED finden Sie in der Schublade »Tools«. Das Programm wird durch einen Doppelklick auf sein Icon gestartet und macht dann mit Abb. 10.1 auf.

Um einen vorhandenen Zeichensatz bearbeiten zu können, müssen Sie ihn zuerst von der Diskette laden. Wählen Sie dazu »Open« aus dem Projekt-Menü. Klicken Sie so lange auf die Auf- oder Ab-Pfeile, bis der gewünschte Font auf dem roten Streifen steht. Dann klicken Sie auf »LOAD IT«

(lade ihn) oder auf »WHOOPS!«, wenn Sie doch keinen Font laden wollen. Haben Sie einen Font geladen, sind Sie in etwa auf dem Stand von Abb. 10.1.

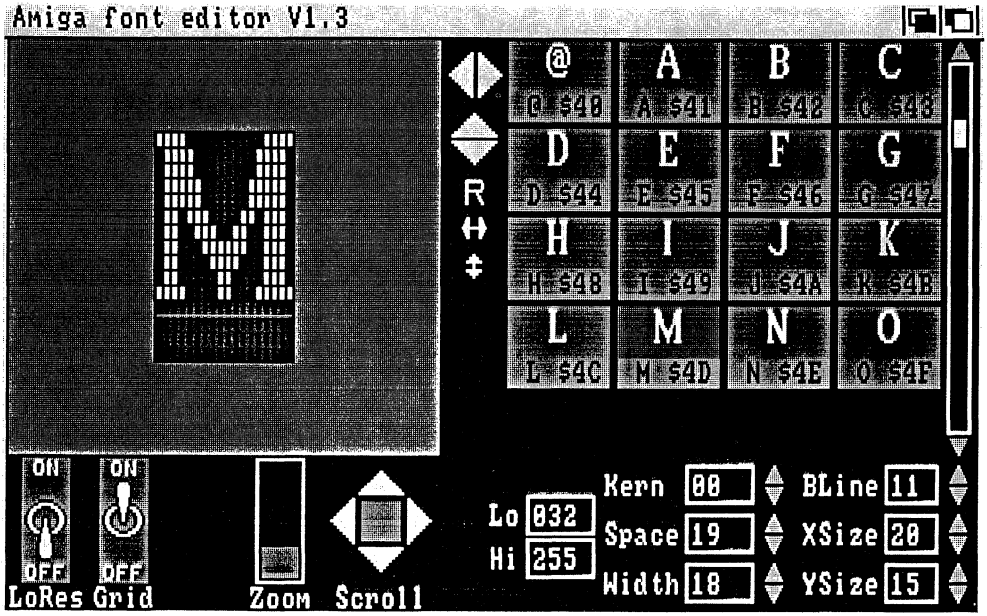

Abb. 10.1: Die Bedienoberfläche von FED

Das Fenster rechts im Bild stellt eine Auswahl der möglichen Zeichen dar. Diese tragen die Codes (Nummern) von $20 bis $FF, in dezimaler Schreibweise wären das 32 bis 255. Die ASCII-Codes 0 bis 31 sind also nicht erreichbar. Dieser Bereich ist aber auch den Control-Codes (Steuerzeichen) vorbehalten, die nicht editiert werden sollten. Wenn Sie das unbedingt wollen, können Sie im Feld »Lo« die »032« durch »000« ersetzen.

Sie müssen jetzt so lange durch das Fenster mit dem Zeichensatz rollen, bis das Zeichen, das Sie bearbeiten wollen, in Ihr Blickfeld kommt. Dann klicken Sie das Zeichen an, und schon wird es vergrößert in die Arbeitsfläche mit dem Gitterraster kopiert. Wie hoch das Gitter ist, hängt von der Zeichenhöhe ab. Die Breite des Gitters ändert sich bei proportionalen Fonts mit fast jedem Zeichen.

Ein Zeichen neu zu malen, ist ganz einfach. Ein Klick auf einen weißen Bildpunkt löscht ihn, ein Klick auf die dunkle Fläche erzeugt einen weißen Punkt. Sie können auch bei gedrückter Maustaste freihändig malen. Die Änderungen sehen Sie (fast) sofort im rechten Fenster, so daß Sie direkt die Wirkung in der Originalgröße beurteilen können. Folgt die Änderung nicht im Auswahlfenster, müssen Sie das Zeichen nur dort anklicken, und der aktuelle Stand erscheint.

10.2.3 Die Bedienelemente

In Abb. 10.1 sehen eine ganze Menge Gadgets (Bedienelemente), für die gilt:

Schalter LoRes

In der Stellung »ON« wird die niedrige Aufösung von 320x200 eingeschaltet, bei »OFF« ist die Auflösung 640x200. Wenn Sie einen Font in der niedrigen Auflösung einsetzen wollen, sollten Sie ihn auch in diesem Modus bearbeiten.

Zoom

Mit diesem Schieberegler können Sie die Arbeitsfläche vergrößern und wieder verkleinern. Das ändert aber nichts an der Auflösung, sondern ist wohl eher für Leute gedacht, die ihre Brille verlegt haben.

Scroll

Durch Klicken auf eines der vier weißen Dreiecke können Sie den Ausschnitt der Arbeitsfläche verschieben. Praktisch braucht man das nur, wenn man vorher »gezoomt« hatte.

Dreiecke links/rechts/auf/ab

Mit jedem Klick auf eines dieser Dreiecke verschieben Sie das aktuelle Zeichen um je einen Bildpunkt in die entsprechende Richtung. Doch seien Sie vorsichtig! Wenn ein Zeichen zu weit über den Rand hinausgeschoben wird, können Sie es nicht mehr zurückholen.

R für Reverse

Ein Klick auf den Buchstaben unter den Verschiebe-Dreiecken invertiert das aktuelle Zeichen. Es wird dann zu dunkler Schrift auf hellem Hintergrund.

Spiegelung

Mit einem Klick auf den Rechts-/Links-Pfeil (<->) können Sie ein Zeichen horizontal spiegeln. Das macht sich zum Beispiel gut, wenn Sie Klammern konstruieren. Sie malen »{«, kopieren es zum Code für »}« und spiegeln es da. Mit dem Auf-/Ab-Pfeil () können Sie vertikal spiegeln.

Felder Lo und Hi

Hiermit bestimmen Sie den Bereich eines Fonts und damit auch seinen Speicherplatz auf der Diskette. Geben Sie zum Beispiel für Lo 65 und für Hi 90 an, hat der Font nur Zeichen für die Tasten <Shift>+A bis <Shift>+Z (die Großbuchstaben A–Z).

Feld Kern

Mit dem Kerning ist das Unterschneiden gemeint. Sie können damit zum Beispiel erreichen, daß im Wort »Tee« das e unter das Dach des T rückt. »Gekernte« Fonts arbeiten bei den meisten Zeichen bei 01, bei einigen mit 0 und einigen wenigen mit –01. Die Wirkung ist allerdings mit FED nicht zu beurteilen, und eine Kunst ist das richtige Kerning schon, weil es dann für den ganzen Font richtig durchgezogen werden muß. Lassen Sie das Kerning besser bei 0. Kerning ist nur bei proportionalen Zeichensätzen möglich.

Felder Space und Width

Damit wird der Abstand und die Breite der Zeichen festgelegt. Beide Werte gehören eng zusammen und sind nur bei proportionalen Zeichensätzen änderbar.

Feld BLine

Die Basislinie wird als grüne Linie im Editierbereich dargestellt. Sie ist die Grundlinie über der Buchstaben wie a, b oder c beginnen, und unter der die Unterlängen von Buchtaben wie g, j, oder p liegen. Wenn Sie sich an Ihre ersten Schuljahre erinnern, wo Sie auf liniertem Papier geschrieben haben, dann ist das in etwa die Basislinie. Normalerweise sollten aber Buchstaben ohne Unterlängen einen Bildpunkt über der Basislinie beginnen. Beachten Sie auch, daß einige Zeichen wie zum Beispiel die Klammern eine leichte Unterlänge von 2 Punkten haben sollten.

Felder XSize und YSize

Diese Felder geben die Nennbreite und die Nennhöhe des Fonts an. Sie wirken immer auf den ganzen Font. »YSize« entspricht dem Punktwert, ist also zum Beispiel 15 bei Courier-15, während sich »XSize« typisch an der Breite des Zeichens »0« orientiert.

10.2.4 Die Menüs

Das Projekt-Menü

ist wohl selbsterklärend, anzumerken wäre nur: »New« heißt, daß Sie einen Font ganz bei Null beginnen wollen, was eine nicht zu empfehlende Übung ist. Sie sollten einen Font zuerst unter einem anderen Namen mit »Save as« abspeichern und nach jedem gelungenen Zeichen ihn wieder mit »Save« sichern.

Das Edit-Menü

ist recht gefährlich, weil ausgenommen »Erase« alle Punkte global auf den ganzen Font wirken und nicht bzw. schwer rückgängig zu machen sind. Es bedeuten:

```
Make italic      : Alle Zeichen kursiv
Make bold        : Alle Zeichen fett
Make underlined  : Alle Zeichen unterstrichen

Copy to          : Kopiert das aktuelle Zeichen in andere
                   Felder und zwar in jedes, das beim aktiven
                   Copy-Befehl angeklickt wird. Durch noch
                   einen Aufruf des Menü-Punktes muß der
                   Copy-Modus wieder ausgeschaltet werden.
Erase            : Löscht das aktuelle Zeichen

                   Alle Zeichen um einen Punkt schieben nach
All Right        : rechts
All Left         : links
All Up           : oben
All Down         : unten
```

Das Attribut-Menü

Sie können einem Font mehrere Attribute zuordnen, da jedes nur ein Bit setzt. Betroffen ist immer der ganze Font. Es bedeuten:

```
Font type -> FIXEDWIDTH    : Feste Breite
          -> PROPORTIONAL  : Proportionalschrift

Font style -> Normal      : Normal
           -> Italic      : Kursiv
           -> Bold        : Fett
           -> Underlined  : Unterstrichen
           -> Extended    : Erweitert (gedehnt)
```

10.2.5 Starten Sie FixFonts

Im Verzeichnis »FONTS:« gibt es für jeden Font eine Datei und ein Directory. In letzterem befinden sich für jede Größe eines Fonts die Daten, wohingegen die gleichnamige Datei (mit dem Extender ».font«) auch Angaben über den Directory-Inhalt enthält. Wenn Sie nun einen neuen Font einem Font-Verzeichnis hinzufügen (oder daraus löschen), muß das die »Oberdatei« natürlich auch wissen. Genau diesen Job erledigt das Programm »FixFonts« in der System-Schublade.

10.3 Der Icon-Editor

Mit IconED können Sie Icons – offiziell Piktogramme geheißen – erstellen und vorhandene Icons ändern. Ich bleibe übrigens ziemlich konsequent beim Begriff »Icon«, weil dann jeder weiß, was gemeint ist, und nicht etwa Commodores Disketten nach dem »PiktoED« oder der Picto.Library absucht.

10.3.1 Der Start

IconED im Verzeichnis »Tools« wird durch einen Doppelklick auf sein Icon gestartet. Nachdem die Belehrung über Typ-Konflikte – die Lösung dazu finden Sie im Kapitel 1.7 – weggeklickt ist, erscheint Abb. 10.2, allerdings mit immer demselben Icon in allen Rahmen.

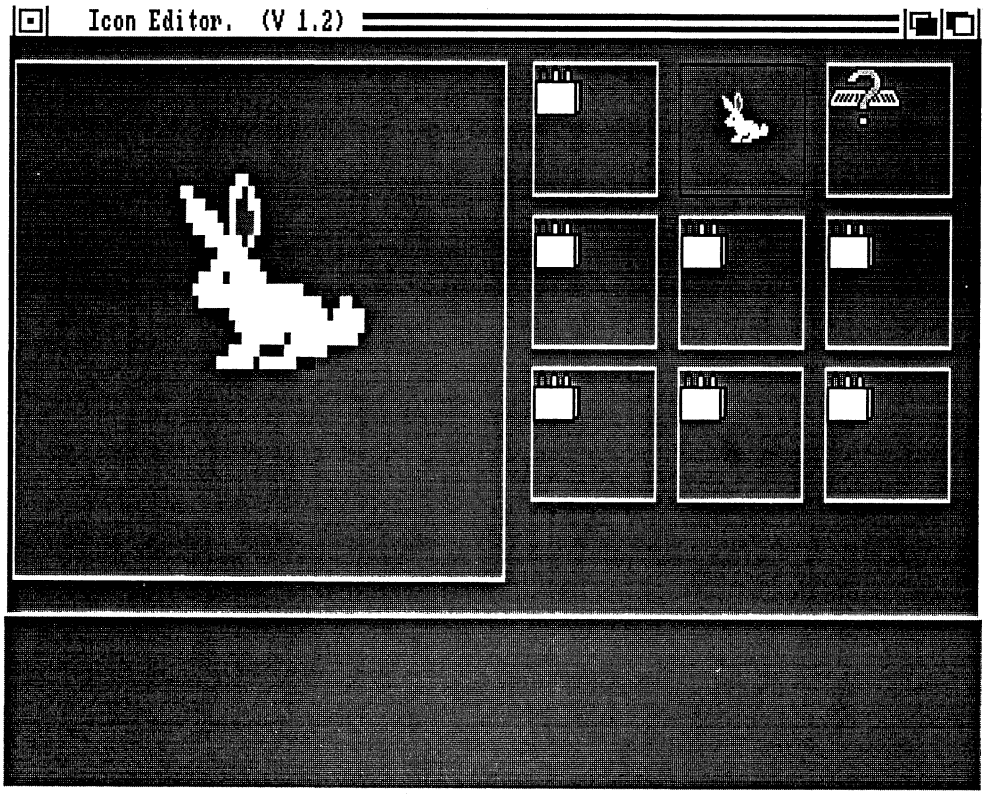

Abb. 10.2: Die Bedienoberfläche von IconED

Das ist – wie unschwer zu erkennen, das Icon von IconED selbst. Der große Rahmen ist die Arbeitsfläche, die neun kleinen haben den Zweck, weitere Icons für die Bearbeitung in Reserve zu halten. Ein paar mehr als man braucht, sind das schon.

Um ein neues Icon zu laden, klicken Sie einen der kleinen Rahmen an und wählen dann »Load Data« aus dem Disk-Menü. Im folgenden Requester tippen Sie in das Textfeld den Namen einer Info-Datei ein, wobei Sie den Extender ».info« weglassen. Den kompletten Pfad sollten Sie allerdings angeben, »:trashcan« wäre ein Beispiel. Danach müssen Sie noch auf »Load Icon Image« klicken.

10.3.2 Das Editieren

Das zuletzt geladene Icon wird automatisch auch im Editor-Rahmen dargestellt. Wollen Sie ein anderes bearbeiten, müssen Sie nur sein Abbild in einem der kleinen Rahmen anklicken. Nun können Sie loslegen. Da, wo Sie mit der Maus hinklicken, wird ein Punkt gesetzt, halten Sie die linke Maustaste gedrückt, können Sie auch Linien und Kurven ziehen.

Die Zeichenfarbe – per Voreinstellung weiß – können Sie mittels des Menüs »Color« ändern. Um etwas wegzuradieren, müssen Sie es mit der Hintergrundfarbe (per Voreinstellung blau) übermalen.

10.3.3 Die Extras

Nach einer Änderung könnnen Sie das Icon mit »Save Data« im Disk-Menü speichern. Dabei heißt »Save Full Image«, daß alles gespeichert wird, während bei »Frame And Save« ein Ausschnitt gesichert wird. Wenn Sie diesen Punkt angewählt haben, müssen Sie danach die linke obere Ecke Ihres gedachten Auswahlrechtecks im Editorfeld einmal anklicken, dann das Rechteck ziehen, es ist jetzt sichtbar, und wieder klicken, wenn das Rechteck die richtige Größe hat. Vor dem zweiten Klick können Sie die Aktion noch abbrechen, indem Sie auf »Cancel The Save« gehen.

Das Copy-Menü

hat einen interessanten Punkt, und der heißt »Merge With Frame«. Der Rahmen laut Sub-Menü-Nummer wird dem Bild hinzugemischt. Vorher sollten Sie allerdings mittels des Move-Menüs dafür sorgen, daß beide Bilder unterschiedliche Positionen haben. »From Frame« hingegen ersetzt nur das Bild durch das des angewählten Rahmens.

Das Move-Menü

erlaubt unter »In Frame« Ihr Bild im Rahmen zu verschieben (wenn es klein genug ist). Ist »Single« eingestellt, bewegt jeder Klick auf einen der vier Pfeile das Bild um einen Punkt, bei »Repeat« läuft das Bild so lange, wie Sie auf einen Pfeil klicken. »Exchange With Frame« tauscht das aktuelle Icon mit einem anderen, nämlich dem, dessen Rahmen-Nummer Sie im Sub-Menü anwählen. Damit kann man ganz gut einen Zwischenstand ablegen.

»Undo Frame« wählen Sie besser nicht an, das löscht nur den Rahmen.

Das Text-Menü

hat nur den Punkt »Write Into Frame«, das heißt, damit läßt sich in einen Rahmen Text einschreiben. Beachten Sie, daß ein Icon immer auch automatisch einen sichtbaren Namen hat. Im

Text-Requester tippen Sie anstatt »text« Ihren Text ein. Durch mehrmaliges Klicken auf die Felder »Foreground« (Vordergrundfarbe), »Background« (Hintergrundfarbe), Font und Mode wählen Sie dann die Attribute und klicken schließlich auf »Position«. Es erscheinen wie bei »Move« vier Pfeile, mit denen Sie den Text positionieren können.

Das Misc-Menü

hat auch nur einen interessanten Punkt, und der heißt »Flood Fill«. Wählen Sie zuerst aus dem Color-Menü die Füll-Farbe und dann aus dem Misc-Menü »Flood Fill«. Der Text »Flood Fill Activated« im Fenster-Titel zeigt an, daß IconEd jetzt »scharf« ist. Wenn Sie in ein Rechteck (oder sonstige Figur) Ihres Bildes klicken, wird es mit der Füllfarbe eingefärbt.

10.4 IconMerge

Haben Sie sich schon einmal gewundert, wieso beim Anklicken des Mülleimers – vornehmer »trash can« oder »garbage« – dessen Deckel aufgeht? Dahinter steckt ein simpler Trick. Es gibt nicht ein Icon, sondern derer zwei. Normalerweise wird das Icon »Deckel zu« dargestellt. Klicken Sie es an, wird das Icon »Deckel auf« angezeigt.

Diesen Effekt können Sie für Ihre Icons auch sehr einfach erreichen. Sie erstellen mit IconEd zwei Icons, eines für den Ruhe- und eines für den Auswahl-Zustand. Wichtig ist, daß beide Icons dieselbe Größe haben. Dann starten Sie das Programm IconMerge, zu deutsch, den Icon-Mischer, per Doppelklick auf sein Icon von der Workbench aus.

Nach dem Start müssen Sie einige Fragen beantworten. Zuerst werden Sie nach dem Modus als »m« oder »s« gefragt. »m« steht für »merge« oder »mischen«, das »s« für »split«. Sie können also mit diesem Programm auch ein Doppel-Icon in seine zwei Teile zerlegen (splitten).

Nun fragt Sie IconMerge nach den Namen der drei Info-Dateien. Sie geben nur die Namen ohne den Extender ».info« ein und zwar für:

Icon1: das Normal-Icon
Icon2: das Auswahl-Icon
Icon3: das gemischte Icon

Beim Splitting ist es genau umgekehrt. Icon1 ist das Gesamt-Icon, Icon2/Icon3 sind die Namen für Einzel-Icons.

Alternativ können Sie IconMerge auch vom CLI aus aufrufen, wobei für das Mischen diese Syntax gilt:

```
iconmerge <icon1> <icon2> <icon2> opt m
```

Für das Splitten müßten Sie »opt s« schreiben.

S

Stichwortverzeichnis

Symbole

&-Operator 209
.i. 52
.info 21
_LVOClose 207
_LVOCloseLibrary 207
_LVOExecute 123
_LVOOpen 123, 207
_LVOOpenLibrary 207
_LVORead 207
_main 225
_main() 224
_main()« anstatt »main() 228
_stdin 115
_stdout 115
2-Pass-Compiler 221, 222
32-Bit-clean 92
32-Bit-Konstanten 104
32-Bit-Operanden 104

A

absolute Adressen 84
abweisende Schleife 165
ACBM 190
ActivateWindow 157
Adreß-Error 97
ALIAS 46
alias 46, 76, 223
AllocMem 206, 210, 211
AllocRemember 85, 87, 89
allokierten Speicher 85
Amiga 1000 18
Amiga-Assembler 115
Amiga-Standard-Format 190
Amiga.lib 115, 229
amiga.lib 224
ANSI-Erweiterungen von C 240
Apple ImageWriter 62
Archive 78
Archive-Bit 78
argv 225
Arrays 151, 152
asctime 240
Assembler-Basic-Konverter 211
Assembler-Routinen 207
ASSIGN 45
assign 164
Astartup.obj 115
Ausgabe-Umleitung 115
AvailMem() 117
awrite 60
Aztec-C 221

B

Basic schneller 177
Basic-Editor 140
Basic-Menü 155
BasicClip 170
BEEP 182
BeginRefresh 243
Beenden von Amiga-Basic 139
Bildröhre 17
Binärzahl in einen String 216
Black and White 64
BLINK 223
Blitter 64, 81
blitzschnell kopieren 172
Blitzstart 24
BlockPen 200
BMAP 143, 145
BMAP-Datei 145
Bobs 187
Boot-Diskette 75
BorderBottom 199
BorderLeft 199
BorderRight 199
BorderTop 199
Bottom Up 135
Breakpoint 100
BSS 206
Bus-Error 97

C

C-Funktionen im ROM 112
C-String 162
Calcomp ColorMaster 62
Call 166
CALL obligatorisch 167
CALLEXEC 124
CALLLIB 125
Canon PJ 1080A 62
CBM MPS 1000 62
CBM MPS 1250 62
CheckMark 200
Chip-Daten 86
Chip-Memory 86, 87, 89, 211
Chip-RAM 86
CHK-Befehl 97
CLEAR 150, 184
CLEAR-Befehl 150
CLI 34, 35, 48
CLI-Befehle 77, 164
– in Assembler 122
– in Basic 161
CLI-Programme 109
– von der Workbench 33
CLI-Startup 38
CLI-Window 161
Clipboard 170
CLOSE 183
CLR vermeiden 104
CLR-Befehl 104
CMD 58
Code in die Data-Zeilen 211
COLOR 183
Color Register 81
COMMSEQ-Flag 129
Complement-Modus 231
CON 161
Condition Code 92
Con-Handler 39
Console-Device 93
CONT 183

Control-Codes 47
ConvertFD 143, 144
Copper 81
copy 58, 73
CopyMem 172
CopyMemQuick 173
CopyPrefs 31
CreateDir 172
CurrentTime 240, 241
Cursor-Steuertasten 141
Custom-Chips 86

D

DATA 206
DATA-Zeilen 209
Datei beschädigt 75
Dateien finden 76
– sparsamer 157
Daten-Typ 151
Datum 240
Dead-End-Alert 85
Debugger 95
DECLARE FUNKTION 146
DEF FN 183
DEFINT 179
DEFLNG 180
Deklarieren 148
DELETE 183
Density 62, 64
DetailPen 200
Device 164
DevPac-Assembler 115
DF1: 71
Diabolo C150 62
Dichte 62, 64
DIM-Anweisung 151
DIP-Schalter 68
dir 74, 76
Direct Memory Access 82
Directory 73
Direktadressierung 104
Direktmodus 155
DISK 21
Disk aufräumen 75
DiskDoctor 74
Diskette 18
Disketten-Wechsel 77
Disk-Icons 19
Disk-Libraries 149
DisplayAlert 170
Dither 63
Dithering 63
Division durch 0 97
DMA 82
DO-Loop-Schleife 165
DOS-Library öffnen 134
Dot 248
DRAW 21
Drucker installieren 67
Drucker-Software 55
Druckertreiber 55, 61
Drucker-Update 55

E

Echo-Befehle 18
eigenes Prompt 38
Einsatz von Libraries 145
Einschalten 17
END 183
Endlosschleife 178
EndRefresh 243
Entwicklungsbaum 136
ENV 50, 51
EPROM 55
Epson 62
ERL 183
Error validating disk 74
Error-Handler 155
Escape-Sequenzen 47, 65, 68
Escape-Taste 57
Event 93
Exec 82
Execute 51, 163
EXECUTE-Befehl 124
Execute-Funktion 122
Exit 224
EXP(x) 183

F

F-S 63, 64
Farbregister 81, 234
Fast Fonts 54, 142
Fast Memory 87
Fast-Memory 211
FD 143
FD-File 143
FED 256
Fenster 224
– zu 23
FF 54, 142
FGO-Regel 100
File already open 56
File-Transfer zum PC 58
FindTask 111
FixFonts 260
Flags 91, 198
Floyd-Steinberg 63
Font-Editor 256, 257, 259
FOR-NEXT-STEP 184
FOR-Schleife 166
forbid(); 86
Form Feed 61
FORMAT 79
format 72
Formular-Vorschub 61
Fraction 63
FRE 150, 184
FreeRemember 85, 89
Funktionsergebnisse 84, 91

G

Gadget 175
Gadget-Abfrage 175
Ganzzahl-Division 180
Ganzzahlen 179
GARBAGE 21
garbage collection 180
gefährliche Fehler 227
GET/PUT (Grafik) 184
GETENV 51
GOSUB 179
GOTO 178
Grafiken speichern 231
GraphicDump 63
graphics.library 182
Graustufe 63
Größenänderung 203
Guru-Meldung in Basic 169

H

Haltepunkt 100
Handle 131, 244
Handles 163
Hardcopies 61
Hardware-Unabhägigkeit 82
Height 196
HEX$(x) 184
HighCyl 72
History-Buffer 37
HP DeskJet 62
HP LaserJet 62
HP PaintJet 62
HP ThinkJet 62

I

Icon-Editor 260
IconED 260
IconMerge 263
Icons 20
IDCMP 93
IDCMP-Flags 93
IDCMPFlags 199
IFF-Bilder 190
– drucken 192

illegaler Befehl 97
ImageWriter 67
Include-Files 89
INCONX 34
Indirektion 84
Info-Datei 20
Inhaltsverzeichnis ansehen 141
InitPrinter 64
INKEY$ 184
INPUT 184
INPUT$ 185
Input-Device 93
Input-Stream 93
install 75
InstallPrinter 67
Integer 63
interaktiver Modus 79
INTUITICKS 241
I/O mit AmigaDOS 130, 243

K

Kickstart 18
kleine RAD 28
Kommando-Datei 33
Kommando-Interpreter 35
Kommandozeile 122, 134, 215, 216
– auswerten 122
kompakte Programme 228, 229
Konsol-Fenster 173
kopieren 73

L

lageunabhängig 205, 207
Lattice ab Version 4 224
– Version 3 223
Lattice-C 222, 223
Laufschriften 174
LBOUND 185
lea 105
Leerzeichen 48
LEFT$ 185
LeftEdge 196
Libraries 143
– gleichzeitig öffnen 147
– in Basic 143
Library 83
Library-Unterprogramme 143
Line-A-Emulator 97
Line-F-Emulator 97
LIST 185
List-Fenster 140, 141, 143
Listing 142
LLIST 185
LOAD 186
LoadACBM 191
LoadACBM ändern 191
LoadILBM-SaveACBM 191
LoadWB 31
localtime 240
LOCATE 186
Locate-Befehl 156
Locate-Koordinaten 156
LOF-Funktion 209
Long Integer 176
löschen 78, 79
LowCyl 72
LPOS(0) 186

M

Macintosh 67
Magnetfelder 18
main 225
main() 224
MAKEDIR 171
Makro 46
– RELO 208
Makro-Befehl 124
Makros 124, 228
Makrosprache 46
Mark 248
Maschinenprogramm 209
– im Systemspeicher 210
MaxHeight 197
MaxWidth 197
MEMAC, zwei Betriebsarten 248
MEMACS 247
MEMF.CHIP 211
MEMF_CLEAR 211
MEMF.PUBLIC 211
MENU 186
MENU(0) 186
Menüs mit Makros 126
MenuStrip 198
MERGE 186
MF.CLEAR 211
MF.PUBLIC 211
MID$ 185
MID$-Anweisung 186
MinHeight 197
Mini-Paint 160
MinWidth 197
MKI$ 170
MODE_OLDFILE 216
Monitor 17
MORE 49
mount 72
MountList 72
mountlist 36
MOUSE(0) 158
MOUSE(1) 158
MOUSE(2) 158
MOUSE(3) 159
MOUSE(4) 159
MOUSE(5) 159
MOUSE(6) 159
MOUSE-Funktionen 159
MouseX 197
MouseY 197
movem 206
MOVEM-Befehl 108
moveq 103
MP_SIGBIT 84
MUL 105
Multitasking 82, 84
Multitasking-System 83
Music Construction Set 61

N

NEC P6/6/7/9/2000 62
neue Zeichen 23
NewWindow-Struktur 194
Nullmodem 59, 95
Nur im CLI 20

O

OBJECT-Anweisungen 187
OBJECT.SHAPE 187
OBJECT.STOP 187
Objekt-Module 206
OCT$(x) 187
ohne CLI 168
ohne Workbench 154
Okidata 62
ON ERROR 187
ON ERROR GOTO 192
ON x GOTO/GOSUB 187
OpenLibrary 83
Optimierung des Basic-Programms 149
OPTION BASE 151
Optionen 221
Ordered 63
Out of Memory 149
Overflow 176

P

p-Bit 78
PAINT 187
Palette 31, 32
PAR 56, 65
Parameterübergabe 217
Parser 216
Partitionen 72

Pascal-Strings 134
patchen 39
path 44
PATTERN 187
PC-DOS-Befehle 46
PC-Parallelschnittstelle 60
PC-relative Adressierung 108, 205, 206
pc-relatives Ziel 108
PEEK 176, 188, 194
permit();; 86
Pfadfinder 43
POINT 188
Pointer 199
POKE 176, 188, 194, 195
Polling 94
Polling-Methode 83
position independend 205, 208
positionsunabhängig 107
Postvorschriften 93
Preference-Farben 31
Preferences 31, 33, 62
– kontrollieren 235
Preference-Struktur 236
print 121
Printf 118
printf 114, 224
printfiles 58
Privilegverletzung 97
Programmspeicher 150
PROJEKT 21
Prompt 38
protect 77, 78
Proto-Directory 229
Protos 229
PRT 56
PSET 160
PtrHeight 199
PtrWidth 199
Pure 78

Q

Q-Befehle 103
QUICK 79
Quick-Basic 59, 165
Quick-Befehle 103

R

RAD 24
RAM 24
RAM-Disk 24, 40, 73, 154
RangeRand 116
RasInfo-Struktur 148
RASSIZE 234
RastPort 200, 201
RastPort-Struktur 200
Read-file 248
Recovery-Alert 169
Refresh 243
REFRESHWINDOW-Message 243
Register retten 206
relabel 71
REM 153, 188
RememberKey 89
Remember-Struktur 89
ReplyMessage 86
resetfeste RAM-Disk 24
RESIDENT 40, 154
residentfähig 41
Resource-Manager 86
RESTORE 189
RESUME 188
Return-Adresse 219
Return-Code 92
Return-Wert 93
reverse 47
richtig springen 109
RIGHT$ 185
RND(n) 188
ROM 18
ROM Kernel Reference Manuals 113
ROM-Kernel-Manual 110
ROM-Libraries 148
ROM-Wack 95, 96
ROM-Wack-Display 97
RPort 199
RUN 41, 188

S

s-Bit 78
SADD 189
Sammeldruck 58
Schleife endlos 166
Schleifen 177
– anderer 165
schneller PC-Druck 60
schneller Schieben 106
Schnellkopie 73
Schnellstart 26
Schreibmaschine 56
Scratch 91
SCREEN 189
Screen-Titel ändern 204
ScreenPrint 191
– ändern 191
ScreenTitle 200
Script 78
SCROLL 190
SCROLL-Anweisung 174
Scroll-Befehl 160
search 76
SECTION 206
Segment 206
Seiko 5300 62
SER 56
SetPrefs 238
SetRGB4 234
SetSoftStyle 148
SHARED 188
Shell 35
– läuft 36
Shell-Editor 36
Shell-Startup 38
Shift-Count 106
SIMPLE_REFRESH 243
Skip-Befehl 53
SMART_REFRESH 243
Smoothing 64
Snapshot 22
Software-Orientierung 82
SOUND WAIT 190
Spannungsspitzen 17
SPC(n) 188
SPEAK 43
Speicher messen 149
Speicher-Schlucker 153
Sprache 43
sprintf 121
Sprite-Prozessor 81
Stack 105
– aufräumen 113
Stackpointer 113
Standard-Escape-Sequenzen 65
Standard-I/O-Kanäle 228
Stapelspeicher 150
Start-Datei 256
Starterlaubnis 86
Start-Modul 225
Startup-Code 109, 110, 225
Startup-Menü 52
Startup-Modul 115
Startup-Sequence 25, 27, 30
STATIC 189
Status-Register 92
Stilarten 57, 201
STR$(x) 189
SUB 189
Supervisor-Modus 97
Systemspeicher 150

T

TAB 189
Tabellen 181

TAN 189
Task 83
Tastatur 22
Tastaturbelegungen 23
Tastenkürzel 255
Threshold 63
Title 198
TOOL 21
Top Down 135
TopEdge 196
Trace-Bit gesetzt 97
TRAP 97
Trap-Befehl 97
TRAPV-Befehl 97
Treiber 55
TYPE 75
Type 42
– in hex 42
Type-Casting 226
Typen 194

U

UBOUND 185
Umgebungsvariablen 50
Umlaute 167
undelete 74
UnDo 142
universelle Escape-Sequenzen 66
UNIX 35
Unlock 172
Unsigned Integer 151
Update auf Shell 39

V

VAL 190
value mismatch 227
VARPTR 190, 209
Verschieben von Bildschirm-
bereichen 160
verschiedene Betriebssysteme 18
Visit-File 248

W

Wait 94
Warnmeldungen 226, 227
wd_UserPort 84
WIDTH 190
Width 196
Window 201
WINDOW-Anweisung 190, 194
Window-Struktur 194
Workbench-Programme 86, 109
Workbench-Screen 233
WRITE 189
WScreen 199

X

XCPT 97
XOffset 199

Y

Yank 248
YOffset 199

Z

Zeichenabstand ändern 203
Zeilen löschen 141
Zeilenabstand ändern 202
Zeilennummern 167, 178
Zeit 240
Zeitzone 241
Zerlege Kommandozeile 215
Zugriff auf Preferences 237
Zweierpotenz 105

Profi-Tips und Power-Tricks für den Amiga

Der Autor kennt den Amiga – drei Bücher und zahlreiche Zeitschriften-Artikel beweisen es – doch inzwischen kennt er noch mehr, nämlich die vielen großen und kleinen Probleme der Leser.

Darüber hätte er natürlich noch ein Buch, sozusagen das ganz große Amiga-Buch, schreiben können, doch das ist es nicht. Die große Linie kennen die Anwender, was Ihnen oft nur fehlt, sind die kleinen Kniffe, die Tips und Tricks, die Lösungen zu den alltäglichen Problemen. Genau das ist das Hauptthema dieses Buches.

Wie kann ich ein CLI-Programm von der Workbench aus starten oder was tu ich, wenn ein Programm ein Laufwerk verlangt, das ich gar nicht habe? Muß eine Hardcopy so lange dauern und häßlich aussehen, oder wie werden meine Basic-Programme schneller? Das sind nur einige der vielen Fragen, auf die das Buch die Antworten gibt.

Darüber hinaus wird aber noch eine andere Linie verfolgt. Der Amiga bietet so zahlreiche Möglichkeiten – zwei Betriebssysteme und Multitasking, wer hat das schon? – daß man sie gar nicht alle erlernen kann oder Jahre dafür braucht. Da helfen die Tips und Tricks, die gezielt so ausgewählt wurden, daß sie die tägliche Arbeit erleichtern. Wie man mit ALIAS viel Tipparbeit spart, warum man EXECUTE auch weglassen kann, oder wie man den Amiga dazu bringt, gleich mit Basic zu starten, wären nur drei Beispiele.

Schließlich gibt es noch viele Tips und Lösungen für Programmierer. Bevorzugt wird hier Basic, doch auch Einsteiger in Assembler und C wurden nicht vergessen. Der eingebaute Debugger wird entzaubert, ein ganzes Kapitel »Überlebensregeln« hilft Fehler zu vermeiden, und viele heiße Profi-Tricks sind auch dabei. Wie man extrem kompakte C-Programme schreibt (von 5000 auf 200 Byte kürzt) oder wie Assembler-Programme C-Funktionen im Amiga-ROM nutzen können, gehören in diese Kapitel.

Aus dem Inhalt:

- Workbench-Tips
- CLI-Tips
- Drucker-Tips
- Tips für Programmierer
- Tips zum Umgang mit Diskette und Festplatte

Hard- und Software-Anforderungen:

Amiga mit Kickstart/Workbench 1.2 und 1.3

ISBN 3-89090-960-4

4 001057 909601 03900

Markt&Technik

DM 39,– sFr 37,– öS 304,–