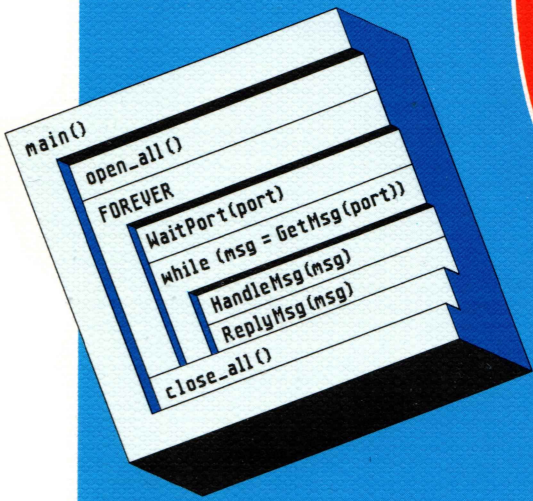
BLEEK
JENNRICH
SCHULZ

Das große C Buch



DATA BECKER

DISKETTE IM BUCH

AMIGA

Bleek
Jennrich
Schulz

Das große C-Buch zum Amiga

DATA BECKER

2. überarbeitete Auflage 1989

ISBN 3-89011-191-2

Copyright © 1988

DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf

Text verarbeitet mit Word 4.0, Microsoft
Ausgedruckt mit Hewlett Packard LaserJet II
Druck und Verarbeitung Mohndruck, Gütersloh

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle technischen Angaben und Programme in diesem Buch wurden von den Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler sind die Autoren jederzeit dankbar.

Vorwort

Nachdem die Amiga-Euphorie allmählich abgeklungen war und man festgestellt hatte, daß der Amiga nicht nur eine tolle Grafik- und Sound-Maschine, sondern auch für die sogenannten "ernsthaften" Anwendungen zu gebrauchen ist, wurden die Rufe nach Literatur zum Programmieren dieser ernsthaften Anwendungen immer lauter. In diesem Buch wollen wir Ihnen daher näherbringen, wie Sie viele der Betriebssystemfunktionen des Amiga sinnvoll für Ihre eigenen Programme anwenden können.

Da dies alles auf der Ebene der Programmiersprache C abläuft, wird vor dem eigentlichen "Programmierlehrteil" die Funktionsweise eines C-Compilers erläutert. Diesen Part übernimmt Bruno Jennrich, der seine langjährige Programmier-Erfahrung hier einbringt. In diesem Teil werden Sie mit den Ursachen bekannt gemacht, die z.B. die "Casts" notwendig machen usw. So können Sie vor allen Dingen bei großen Projekten Zeit und Nerven sparen. Wenn Ihnen bisher einige Besonderheiten der Sprache C Schwierigkeiten gemacht haben und dies zu Fehlern und Programmabstürzen führte - nach diesem Buch machen Sie höchstens noch Flüchtigkeitsfehler.

Danach erhalten Sie eine Einführung in das bisher recht stiefmütterlich behandelte Thema "Intuition". Wolf-Gideon Bleek, ein Spezialist auf diesem Gebiet, wird Sie in die Tiefen der Benutzeroberfläche Intuition hinabführen. Danach werden Sie keine Probleme mehr haben, Ihre eigenen Programme mit Windows, Screens, Menüs, Gadgets und anderen Bestandteilen von Intuition übersichtlich und komfortabel zu gestalten.

Im dritten Teil, für den Peter Schulz, bekannt durch seinen "Profimat" und das Buch "3-D-Grafikprogrammierung", verantwortlich zeichnet, werden schließlich die Intuition-Kenntnisse eingebettet in ein Projekt - Stück für Stück entwickeln Sie mit Peter Schulz einen Editor. Dabei bleiben heikle Themen wie z.B. die Tastaturabfrage, der Diskettenzugriff usw. keineswegs unan-

getastet. Sie lernen also den Umgang mit den Devices und anderen, auch stiefmütterlich behandelten Librarys kennen.

Insgesamt ist dieses Buch also ein Werk, das der ernsthafte Programmierer immer griffbereit neben seinem Amiga liegen haben sollte.

Die Autoren *Februar 1988*

Inhaltsverzeichnis

1.	**Die Sprache C**	**15**
1.1	Überbleibsel der Geschichte	16
1.2	Der Aztec-Compiler	17
1.2.1	Die Installation	18
1.2.2	Das Compilieren	22
1.2.3	Die Verwendung von Make	22
1.2.4	Die Commercial-Version	27
1.4	Der Compiler in Aktion	28
1.4.1	Files vergleichen	28
1.4.2	Hexdump eines Files	37
1.4.3	Ein Monitor	42
1.5	Der Lattice-C-Compiler 4.0	51
2.	**Der Compiler unter der Lupe**	**55**
2.1	Dem Compiler auf die Finger geschaut	61
2.1.1	Interne Organisation der Variablen & Strukturen	62
2.1.2	Software-Organisation der Variablen & Strukturen	75
2.1.3	Die Übersetzung der Steueranweisungen	81
2.1.4	Der Aufruf von Prozeduren	90
2.2	Der Assembler	95
2.3	Der Linker	101
2.3.1	Das "large data"- und "small data"-Modell	105
2.3.2	"large code" und "small code"	106
2.4	Das Debuggen von Programmen	110
2.4.1	Der Aztec-DB	112
2.4.2	Der Aztec-Source-Level-Debugger	129
2.5	Programmiertechniken	145
2.5.1	Der Zugriff auf absolute Speicherstellen	146
2.5.2	Schalter in der Kommandozeile	147
2.5.3	Funktionstabellen	149
2.5.4	File-Handling	152
2.5.4.1	Das Öffnen und Schließen von Dateien	152
2.5.4.2	Daten lesen und schreiben	156

2.5.4.3 File-Informationen erfahren 162
2.5.5 Directorys anzeigen lassen 174
2.5.5.1 Directory anzeigen - inklusive aller Unterverzeichnisse, Wildcards und Unterbrechungen 177
2.5.6 Spielereien mit der Systemzeit 196

3. Intuition und C für Fortgeschrittene 203

3.1 Windows, die Grundlage jeder Ein- und Ausgabe 205
3.1.1 Die Window-Parameter und wie man sie auswählt ... 205
3.1.2 Wie verwaltet Intuition unser Window? 214
3.1.2.1 Der Zugang zur Intuition-Library 214
3.1.2.2 Die NewWindow-Struktur 216
3.1.2.3 Die Window-Struktur 233
3.1.2.4 Eine Übersicht aller Window-Funktionen 240
3.1.3 Programmsammlung "Anwendungen mit Windows" ... 247
3.1.3.1 Windows für jeden Zweck 248
3.1.3.2 Programmteile für Textverarbeitung 255
3.1.3.3 Window für ein neues CLI 257

3.2 Screens, eine elementare Darstellungsgrundlage 260
3.2.1 Die Möglichkeiten beim Öffnen eigener Screens 261
3.2.1.1 Die NewScreen-Struktur 262
3.2.1.2 Vorab das erste Screen-Listing 267
3.2.1.3 Die Screen-Struktur 271
3.2.1.4 Die Screen-Funktionen 277
3.2.2 Anwendungsbeispiele für Screens 282
3.2.2.1 Allgemeine Beispiele 282
3.2.2.2 Programmteil "Screen" für Textverarbeitung 284

3.3 Ausgabe - denn ohne geht es nicht 284
3.3.1 Textausgabe, unsere wichtigste Kommunikation 285
3.3.1.1 Position, Farbe, Zeichenmodus 286
3.3.1.2 Die IntuiText-Struktur 287
3.3.1.3 Der Zeichensatz macht alles interessanter 288
3.3.1.4 PrintIText, die Ausgabe kann beginnen 290
3.3.2 Linienausgabe und was man damit anfängt 294
3.3.2.1 Farbe, Position und und und 295
3.3.2.2 Die Border-Struktur 295

3.3.2.3 Koordinatentabelle zum Linienzeichnen 296
3.3.2.4 DrawBorder-Funktion mit Anwendungsbeispielen 297
3.3.3 Grafikausgabe, jetzt wird's interessant 299
3.3.3.1 Größe, Position, Farben und vieles mehr 300
3.3.3.2 Die Image-Struktur 300
3.3.3.3 Die Grafikdaten in der Image-Struktur 302
3.3.3.4 Die DrawImage-Funktion und ihre Anwendung 303
3.3.4 Beispiele für jede Grafikanwendung 305
3.3.4.1 Auf einfache Weise viele Texte definieren 306
3.3.4.2 Border-Hilfen, die man immer braucht! 308
3.3.4.3 Symbole sagen mehr als Worte 311

3.4 Gadgets, einfacher Informationsaustausch 312
3.4.1 Verschiedene Gadgets, verschiedene Anwendungen .. 312
3.4.1.1 Das Boolean-Gadget 315
3.4.1.2 Das Proportional-Gadget 326
3.4.1.3 Das String-Gadget 331
3.4.2 Gadgets in Aktion 338
3.4.2.1 Die Gadget-Funktionen 338
3.4.2.2 Das neue Sizing-Gadget 357

3.5 Requester: Viele Informationen auf einmal 358
3.5.1 Der Automatik-Requester 359
3.5.2 Der System-Requester 364
3.5.3 Wir bauen uns den Requester selbst 366
3.5.3.1 Die Requester-Struktur 367
3.5.3.2 Eine Requester-Struktur einrichten 369
3.5.3.3 Die Requester-Funktionen 373
3.5.4 Requester: wann es der Benutzer will 375

3.6 Alerts, wenn es ernst wird 377
3.6.1 Verwendungszwecke 378
3.6.2 Aufbau und Einstellungen für einen Alert 378
3.6.3 Alerts für Programmprojekte 383
3.6.4 Auswertung der Guru-Meditation 386

3.7 Abfrage des IDCMP 391
3.7.1 Vorbereitung und Nachrichtenempfang 392
3.7.1.1 Die IntuiMessage-Struktur 398
3.7.1.2 Die MsgPort-Strukturen 400

3.7.1.3 Das Warten auf eine Nachricht 401
3.7.2 Wir werten die Nachrichten aus 403
3.7.2.1 Die altbekannten Gadgets 404
3.7.2.2 Das Window macht Meldung! 405
3.7.2.3 Requester werfen den Ablaufplan über den Haufen . 407
3.7.2.4 Menü-Flags mit Auswertung 407
3.7.2.5 Mäuse laufen gerne. Aber wohin? 410
3.7.2.6 Erkennunng der Tastatureingaben 413
3.7.2.7 Die Diskettenstation 417

3.8 Menüs, die Auswahl der Konsumgesellschaft 417
3.8.1 Der allgemeine Aufbau eines Menüs 418
3.8.2 Wir basteln ein Menü 419
3.8.2.1 Die Menüleiste als Grundstock 423
3.8.2.2 Die ersten Menüpunkte & die MenuItem-Struktur ... 427
3.8.2.3 Kleine Extras in der Menügestaltung 431
3.8.2.4 Komplexere Verschachtelungen: Untermenüs 436
3.8.2.5 Ein Anleitung zum Design-Menü 439
3.8.2.6 Grafik-Menü mit Grafiken 441
3.8.3 Die Abfrage der gesamten Menüleiste 445
3.8.4 Die Arbeit mit Source-Code-Utilities 449
3.8.4.1 Die Bedienung von PowerWindows 450
3.8.4.2 Eine Menüleiste erstellen "lassen" 451
3.8.4.3 Der entstandene Quelltext 453
3.8.4.4 So verarbeitet man den Quelltext weiter 466

3.9 Kontakt über das Console.Device 467
3.9.1 Aufbau der Kommunikationsleitungen 468
3.9.2 Wir empfangen die ersten Daten 470
3.9.3 Wie läuft nun die Ausgabe? 472
3.9.4 Die Steuersequenzen des Console.Device 473
3.9.5 Auswertung aller Nachrichten vom Console.Device .. 476
3.9.6 Zum Schluß der CLI-Editor 484

3.10 Einrichten selbstdefinierter Tastaturtabellen 485
3.10.1 Aufbau der Tastaturtabellen 485
3.10.2 Die Arbeit mit den Tabellen 488
3.10.3 Eine Tastaturtabelle selber zusammensetzen 492

3.11 Speicherverwaltung 495
3.11.1 Die Speicherorganisation des Amiga 496
3.11.2 Erste Gehversuche mit AllocMem() 497
3.11.3 Verbesserung mit AllocEntry() 500
3.11.4 Die beste Lösung 503

4. Betriebssystemprogrammierung 507

4.1 Planung des Editors 507
4.2 Ein Programmgerüst 518
4.3 Schritt für Schritt 529
4.3.1 Öffnen eines Fensters 529
4.3.2 Von RAWKEY nach ASCII 539
4.3.3 Die Speicherverwaltung 543
4.3.4 Testumgebung 557
4.3.5 Zeilen löschen 561
4.3.6 Ausgeben von Text im Fenster 568
4.3.7 Der Cursor 593
4.3.8 Texteingabe 630
4.3.9 Kommandozeile 680
4.4 Befehlsübersicht des Editors 737

Anhang 739
Anhang A: Funktionen 739
Anhang B: Lattice-Compiler & Linker-Optionen 752

Stichwortverzeichnis 761

1. Die Sprache C

Im Jahre 1972 setzte sich Dennis Ritchie daran, die aus der Sprache BCPL (Basic Combined Programming Language) entstandene Sprache B zu überarbeiten. Diese Überarbeitung wurde die Sprache C.

Sie wurde entwickelt, um das Unix-Betriebssystem leichter auf PDP-Rechnern (der Firma DEC), insbesondere auf der PDP-7, implementieren zu können. Damals hätte sich Dennis Ritchie sicherlich nicht träumen lassen, daß seine Sprache heute einen solchen Boom - vor allem auf dem Homecomputer-Sektor - erlebt.

Dies liegt wohl daran, daß C eine äußerst effiziente Hochsprache ist. Diese Effizienz kommt den Homecomputern sehr entgegen, die ja meistens an Speichermangel leiden, und unter Zeitdruck stehen, bzw. niedrige Taktfrequenzen und langsame Programmausführungszeiten haben.

Der Code, den der C-Compiler erzeugt, ist sehr kompakt und schnell auszuführen, was denjenigen Programmierern, die lieber auf Maschinensprache verzichten und dennoch schnelle Programme schreiben müssen oder wollen, sehr entgegen kommt.

Leider hat sich die Idee der Super-Portabilität nicht realisieren lassen. Ursprünglich sollte jedes C-Programm auf jedem C-Compiler laufen. Aber unterschiedliche Betriebssysteme machten unterschiedliche Compiler nötig, und so wurde die Kompatibilität nahezu unmöglich.

Ein kleines Beispiel zeigt die Schwierigkeiten auf, die sich für die "Super-Portabilität" ergeben:

Schon zwischen Amiga und Atari ST treten so starke Differenzen auf - obwohl beides 680000er Rechner sind - , daß ein ST-Programm, das auf Betriebssystemroutinen des GEM, VDI oder AES zurückgreift, nur auf dem Amiga zum Laufen gebracht

werden kann, wenn man die Betriebssysteme beider Rechner genau kennt und weiß, welcher Betriebssystemaufruf des ST durch welchen Betriebssystemaufruf des Amiga ersetzt werden muß. Das ganze verkompliziert sich natürlich dann, wenn ein Programm vom IBM PC auf den Amiga übertragen werden soll. Die Tatsache nämlich, daß beim IBM PC die sogenannte Segment-Adressierung benutzt wird, macht dort umständliche FAR-Deklarationen (engl.: far = deutsch: weit) von Variablen und Funktionen notwendig. Und auch die Tatsache, daß MS-DOS-Funktionen fast ausschließlich über Interrupts aufgerufen werden, macht bei der Übertragung von IBM-Programmen die genaue Kenntnis der Betriebssysteme zur Voraussetzung. Man muß wissen, welche Funktion des MS-DOS welche Funktion des Amiga-DOS übernehmen könnte.

1.1 Überbleibsel der Geschichte

Sicher wissen Sie, daß verschiedene Teile des Amiga-Betriebssystems in C entwickelt wurden. Sie werden sich sicher fragen, wie das möglich war - schließlich ist ein Betriebssystem doch das, was den Computer mit Leben erfüllt. Da aber auch ein C-Compiler ohne Betriebssystem nicht funktioniert, kann das doch eigentlich nicht ganz richtig sein!

Es stimmt aber doch! Und zwar wurde das Amiga-Betriebssystem auf SUN-Workstations entwickelt. Über die serielle Schnittstelle hatte man von der SUN aus Kontrolle über den Amiga. Dies ist übrigens der Grund, warum beim Aufruf des ROMWACK (des Amiga-Debuggers) nach der seriellen Schnittstelle gefragt wird. Beim Auftreten eines Fehlers hatte man dann nämlich die Möglichkeit, diesen von der SUN aus zu finden und zu beheben.

Die Tatsache, daß das Amiga-Betriebssystem auf einer SUN entwickelt wurde, hat aber einen Nachteil: Da nämlich der Amiga zunächst von der Firma Amiga entwickelt, und die Entwicklung dieses Rechners von Commodore erst begonnen wurde, als Amigas finanzielle Grundlage zu schwinden begann, ist hierin wahrscheinlich der Grund zu sehen, daß auch die Sprache

BCPL bei der Betriebssystementwicklung Verwendung fand. Diese Tatsache macht sich manchmal unangenehm bemerkbar, da in der Sprache BCPL Zeiger etwas umständlich verwaltet werden.

BCPL-Zeiger dürfen nämlich nur auf durch 4 teilbaren Adressen zeigen. Bei der Verwendung eines BCPL-Pointers muß dessen Wert also zunächst mit vier multipliziert werden, um die tatsächliche Adresse zu erhalten. Wenn Sie eine Adresse einem solchen BCPL-Zeiger zuweisen wollen, müssen Sie diese zunächst durch 4 teilen.

Auch Strings haben ein etwas abweichendes Format. Erstens werden nur die oben beschriebenen BCPL-Zeiger verwendet. Das erste Byte, auf das dieser Zeiger zeigt, enthält dabei allerdings nicht wie zu erwarten das erste Zeichen des Strings, sondern die Anzahl der Zeichen des Strings. Erst dann folgen die einzelnen Zeichen.

Jedoch sollen uns die Probleme im Zusammenhang mit den BCPL-Zeigern und -Strings nicht weiter interessieren, da diese Zeiger recht selten auftreten - eben nur bei der Programmierung des AmigaDOS (dos.library). Wir wollten nur demonstrieren, daß die Entwicklung des Amiga scheinbar recht umständlich vonstatten ging.

1.2 Der Aztec-Compiler

Doch wenden wir uns von den Kompatibilitätsproblemen ab und ausschließlich dem Amiga zu. Hier stehen wir nämlich vor der Glaubensfrage, welchen Compiler wir verwenden wollen. Die eingefleischten Lattice-Programmierer schwören auf die Zuverlässigkeit des Lattice-Compilers - es wurden ja auch einige Teile des Amiga-Betriebssystems auf dem Lattice-Compiler verfaßt -, während Aztec-Programmierer auf die Geschwindigkeit ihres Compilers setzen.

Wir werden Ihnen in diesem Buch beide Compiler vorstellen und erklären, wie Sie diese zu benutzen haben, um optimale C-Programme zu erstellen.

Wir verwenden in diesem Buch jedoch den Aztec-Compiler (V3.6). Um den Aztec-Compiler einsetzen zu können, muß natürlich bekannt sein, wie man den Comiliervorgang beeinflussen kann. Dies geschieht mit Hilfe von Compiler-Optionen.

1.2.1 Die Installation

Kommen wir also zum Compiler-Paket der Firma Manx. Es enthält vier Disketten. Neben dem Compiler, Assembler und Linker befinden sich auf diesen Disketten auch sämtliche Include-Files und die verschiedenen Bibliotheken, die zum compilierten und assemblierten Programm hinzugelinkt werden können (siehe auch Kapitel 2.3). Zusätzlich befinden sich einige nützliche Programme auf den Disketten, die Ihnen z.B. erlauben, eigene Routinen in die zuzulinkenden Librarys aufzunehmen etc.

Doch beschäftigen wir uns damit, was für den Betrieb des Compilers unbedingt wichtig ist: Auf der Diskette SYS1 des Pakets finden Sie neben einer vollständigen, nutzbaren Workbench-Diskette drei weitere Unterverzeichnisse: bin/, include/ und lib/.

Im Verzeichnis SYS1:bin/ befinden sich alle nötigen Programme für den Betrieb des Compilers: also der Compiler selbst (File cc), der einen Assembler-Source erzeugt, der Assembler (as), der diesen Source assembliert, und der Linker (ln), der den Objektcode des Assembler-Source mit den Librarys, die im Unterverzeichnis SYS1:lib enthalten sind, verbindet, um ein lauffähiges Programm zu erzeugen.

Im Unterverzeichnis SYS1:include sind die einzelnen Header- oder Include-Files enthalten, die in jedes C-Programm eingebunden (included) werden müssen, das auf Betriebssystemstrukturen zugreift. (Die Include-Files werden manchmal auch Bindings genannt.) Leider kann es bei der Verwendung der V3.6-Includes zu Problemen kommen, denn um das Compilieren der

Include-Files zu beschleunigen, hat man aus allen Include-Files der Version 3.6 alle Kommentare entfernt. Bei verschiedenen Include-Files hat man auch das Präprozessor-Kommando #endif am Ende einiger Include-Files entfernt. Dies führt beim Compilieren zu Fehlern. Sollte einmal ein diesbezüglicher Fehler auftauchen, müssen Sie das Include-File suchen, in dem das #endif fehlt, und dieses Kommando am Ende des Include-Files einfügen. (Das Kommando #endif fehlt z.B. im Include-File intuition/intuition.h.)

Auf der zweiten Diskette des Paketes - SYS2: - finden Sie neben Assembler-Include-Files weitere Librarys, die Sie je nach Bedarf zu Ihren Programmen linken können (siehe auch Kapitel 2.3).

Mit diesen beiden Disketten kann das C-Programmieren auch schon losgehen.

Jedoch kann man mit den Hilfsprogrammen der dritten Diskette - SYS3: - das Compilieren wesentlich einfacher gestalten (siehe auch Make).

Diskette - SYS4: - enthält schließlich im Unterverzeichnis arc/ sämtliche Assembler- und C-Aufrufe der einzelnen Betriebssystemroutinen. Diese Assembler- und C-Files wurden assembliert und compiliert, zur c.lib, m.lib und s.lib zusammengefaßt und können in dieser Form zu Ihren Programmen gelinkt werden.

Doch kommen wir zum Compilieren von Programmen:

Schon Diskette 1 reicht aus, um kleine Programme zu compilieren. Dazu müssen Sie aber erst einmal eine Startup-Sequenz anlegen:

```
setmap d          ;Deutschen Zeichensatz laden
stack 10240       ;Stack sollte immer 10 KByte oder mehr
                  ;betragen
copy df0:lib/c.lib to ram:
                  ;Häufigst zu linkende Library in RAM-
                  ;Disk kopieren
bin/set INCLUDE=df0:include
                  ;Wo sind die Include-Files?
bin/set    CLIB=ram:!df0:lib
                  ;Wo findet man die Librarys?
```

```
bin/set  CCTEMP=ram:
                    ;Wohin mit den temporären Files?
bin/setdat          ;Datum und Uhrzeit einstellen
                    ;(Wichtig für Make!)
run bin/mclk        ;Uhr mit Speicheranzeige starten
```

Diese Startup-Sequenz sorgt zunächst einmal dafür, daß der deutsche Tastaturtreiber geladen und der Stack auf 10240 Bytes erhöht wird. Dies ist eine Präventivmaßnahme, die Sie bei der Entwicklung von Programmen vor einem Stack Overflow schützen soll. (Eine weitere Maßnahme gegen den Stack-Überlauf lernen Sie später noch kennen.) Dann wird die Linker-Library, die zu jedem C-Programm gelinkt werden muß, in die RAM-Disk kopiert, um den Zugriff auf diese Funktionsbibliothek zu beschleunigen. Diese Bibliothek ist nicht das gleiche wie z.B. die Graphics.Library, welche die Adressen der einzelnen Grafikfunktionen des Betriebssystems enthält.

Die Librarys, die vom Linker zum C-Programm gelinkt werden, bestehen nur aus kleinen Maschinensprache-Segmenten, die unter Verwendung der von Ihnen eröffneten System-Librarys die eigentlichen Betriebssystem-Routinen aufrufen.

Doch zurück zur Startup-Sequenz. Nun werden nämlich die Umgebungsvariablen des Compilers gesetzt. bin/set INCLUDE=df0: include sorgt dafür, daß bei der Angabe von z.B. #include exec/types.h, das Unterverzeichnis df0:include nach dem angegebenen Include-File durchsucht wird. Wurde das File exec/types.h im Directory df0:include gefunden, so wird dieses eingebunden (included). Ähnlich verhält es sich mit der Umgebungsvariablen CLIB. Diese Variable bestimmt, welches Unterverzeichnis nach den zuzulinkenden Bibliotheken durchsucht werden soll. Durch bin/set CLIB = ram:!df0:lib wird beim Linken von Bibliotheken zuerst die RAM-Disk nach der zu linkenden Library durchsucht. Wenn das File dort nicht gefunden werden konnte, wird das Verzeichnis df0:lib durchsucht. (Sollen mehrere Verzeichnisse durchsucht werden, so werden die einzelnen Directorys durch ! abgetrennt.)

Die Umgebungsvariable CCTEMP gibt an, wo die temporären Assembler-Files abgespeichert werden sollen. Der Compiler er-

zeugt temporäre Assembler-Source-Files im angegeben Unterverzeichnis, die dann später zu einem kompletten Source-File zusammengefaßt und vom - automatisch aufgerufenen - Assembler assembliert werden.

Als nächstes wird in der Startup-Sequenz das Datum und die aktuelle Uhrzeit erfragt. Diejenigen unter Ihnen, die eine Akkugepufferte Uhr besitzen, sollten die Uhzeit und das Datum in der Startup-Sequenz so einstellen, wie es in der Anleitung zu ihrer Uhr steht. Alle anderen sollten aber die Uhrzeit und das Datum korrekt eingeben, zumal dies eigentlich nur beim Anschalten des Rechners bzw. ersten Start des Compilers notwendig ist. Nach einem Reset brauchen Sie bei der Erfragung der Uhrzeit und des Datums jeweils nur <Return> einzugeben, da Datum und Uhrzeit Datum nach einem Reset nicht gelöscht werden. (Die Uhr läuft zwar nicht weiter, aber das macht sich meist nur in wenigen Sekunden Zeitverlust bemerkbar.)

Zum guten Schluß der Startup-Sequenz wird das Programm mclk gestartet, welches dafür sorgt, daß in einem kleinen Fenster, das in der Kopfzeile des CLI-Windows plaziert wird, die aktuelle Uhrzeit und das noch freie Chip- und Fast-Memory angezeigt werden. Dies ist allerdings nicht unbedingt notwendig, doch kann es bei der Entwicklung von Programmen interessant sein, ob das Programm den belegten Speicher auch wieder restlos freigibt. (Dieses Programm ist also recht nützlich bei der Programmentwicklung, weil Sie hier die Möglichkeit haben zu erkennen, ob Ihr selbstverfaßtes Programm den belegten Speicher auch wieder freigibt. Wenn nicht, müssen Sie wohl noch einen Fehler im Programm haben.)

Nach dieser Initialisierung ist es möglich, kleine Programme mit nur einem Diskettenlaufwerk zu compilieren. Für größere Programme wird man aber nicht umhinkommen, mit zwei Laufwerken (bzw. Disketten) zu arbeiten. Dann kann man z.B. die C-Sources auf die zweite Diskette schreiben.

1.2.2 Das Compilieren

Jetzt stellt sich allerdings die Frage, wie man ein Programm compiliert. Eine Möglichkeit besteht darin, sich ein Batch-File zu erstellen, das durch execute ausgeführt wird.

Ein Batch-File der einfachsten Bauart sieht so aus:

```
.key file
SYS1:bin/cc <file>
SYS1:bin/ln <file>.o -lRAM:c
```

Das Programm file.c wird erst einmal durch den Compiler "gejagt", der automatisch den Assembler aufruft, der den vom Compiler erzeugten Assembler-Source assembliert. Das vom Assembler erzeugte Objekt-File wird dann mit der c.lib zu einem lauffähigen Programm zusammengelinkt.

Vorausgesetzt der Name des Batch-Files ist comp, wird ein C-Programm durch execute comp c-prog compiliert. Den Compilier-Vorgang können Sie dann durch Ctrl-C beim Compilieren und Linken unterbrechen. (Rufen Sie den Assembler selber auf, können Sie auch beim Assemblieren die Abarbeitung des Batch-Files unterbrechen.) Dies liegt daran, daß der FAILAT-Level ohne eine Änderung Ihrerseits nach dem Laden der Workbench auf 10 steht. Der Compiler und Linker brechen den Compilier- bzw. Link-Vorgang nach Ctrl-C mit exit(10) ab. Dadurch wird auch die Ausführung des Batch-Files unterbrochen.

Eine weitere Möglichkeit, die Abarbeitung des Batch-Files zu unterbrechen, ist Ctrl-D. Das Batch-File wird allerdings erst dann unterbrochen, wenn das aktuelle CLI-Kommando komplett abgearbeitet wurde. Wurde der Compiler gestartet und danach Ctrl-D gedrückt, muß man so lange warten, bis der Compiler mit seiner Arbeit fertig ist.

1.2.3 Die Verwendung von Make

Batch-Files sind für größere Projekte, die meist aus mehreren Modulen bestehen, äußerst unpraktisch. Schreiben Sie sich z.B.

ein Batch-File, das alle Module compiliert und zusammenlinkt, so müssen Sie, obwohl Sie vielleicht nur eine Kleinigkeit in einem einzigen Modul geändert haben, solange warten, bis alle Module compiliert und zusammengelinkt sind.

Sie können natürlich auch das geänderte Modul von Hand, also durch einen direkten Compiler-Aufruf compilieren und ebenso "von Hand" mit den anderen Modulen zusammenlinken. Dies ist aber auf die Dauer sehr mühsam. Außerdem müßten Sie den Compiler mit SYS1:bin/cc CPRG aufrufen. Dies ist auch auf die Dauer sehr umständlich. (Wenn Sie allerdings mit PATH SYS1:bin dafür sorgen, daß neben dem aktuellen und dem Verzeichnis SYS1:c auch das Verzeichnis SYS1:bin nach CLI-Kommandos durchsucht wird, reicht der einfache Aufruf cc CPRG, um den Compiler zu starten. Wie Sie vielleicht bemerkt haben, muß das Anhängsel .c bei der Übergabe des zu compilierenden Programms an den Compiler nicht unbedingt mit angegebenen werden. Stellt der Compiler fest, daß dieses Anhängsel (Tag) fehlt, hängt er es automatisch an den angegebenen File-Namen an. Auch dem Linker muß nicht unbedingt das Tag .o für die zu linkenden Objekt-Files übergeben werden.)

Um den Compilier-Vorgang zu erleichtern enthält der Aztec-Compiler neben Beispielprogrammen und weiteren Librarys unter den Hilfsprogrammen auf der Diskette SYS3: im Verzeichnis bin das Tool Make. (Hier befindet sich übrigens auch der Debugger DB.)

Mit Make ist es möglich, große Programme, die der Übersichtlichkeit halber immer aus mehreren Modulen bestehen sollten, sehr einfach zu compilieren. Man muß sich nicht selbst daransetzen, das gerade geänderte Programmmodul zu compilieren und zu den alten Modulen zuzulinken. Dies alles übernimmt Make.

Sie müssen allerdings vorher dieses Tool (zu deutsch Werkzeug und im übertragenen Sinn Hilfsprogramm) in das Unterverzeichnis c der Diskette sys1: kopieren.

Sie brauchen jetzt nur noch das Make-File zu erstellen, daß sich im aktuellen Verzeichnis befinden muß. Betrachten wir dazu einmal folgende Module:

```
mod1.c
------
func1()
{
     printf ("Funktion 1 ruft Funktion 3 auf !\n");
     func3();
}

mod2.c
------
func2()
{
    printf ("Funktion 2 wird von Funktion 3 aufgerufen !\n");}

mod3.c
------
func3()
{
     printf ("Funktion 3 ruft Funktion 2 auf !\n");
     func2();
}

main.c
------
main()
{
     printf ("MAIN ruft Funktion 1 auf !\n");
     func1();
}
```

Das Make-File beschreibt nun, wie die einzelnen Files voneinander abhängen. So hängt z.B. das Objekt-File mod1.o vom Source-File mod1.c ab. Dies äußert sich in der Zeile mod1.o: mod1.c, was nichts anderes besagt, als daß die Befehle, die in der nächsten Zeile folgen, nur ausgeführt werden sollen, wenn mod1.c älter als mod1.o ist, oder wenn mod1.o noch gar nicht existiert.

Um festzustellen, welches File älter als ein anderes ist, müssen natürlich die Uhrzeit und das Datum richtig eingestellt worden sein. Das passiert ja in der Startup-Sequenz mit dem Aufruf von setdat. Da immer das Datum der letzten Änderung eines Files mit abgespeichert wird, kann Make immer feststellen, wie alt ein

File ist. Die Uhrzeit muß immer korrekt eingestellt sein, um zu verhindern, daß bei einer Änderung ein falsches Datum mit dem Files abgespeichert wird.

Doch zurück zum Make-File. Die erste Zeile unseres Make-Files lautet also so:

```
mod1.o: mod.c
```

In der nächsten Zeile folgt nun, was geschehen soll, wenn mod1.o älter ist als mod1.c:

```
cc mod1
```

Wenn mod1.o älter ist als mod1.c - also nach dem letzten Compilier-Vorgang eine Änderung an mod1.c vorgenommen wurde, wird dafür gesorgt, daß das neue Objekt-File zu mod1.c erzeugt wird (man sagt auch, daß mod.o "upgedated" wird).

Das Kommando, das nach einer solchen File-zu-File-Beziehung (mod1.o: mod1.c) ausgeführt werden soll, muß mindestens ein Zeichen eingerückt werden. Das Make-File, daß aus den vier C-Sources (s.o.) die zugehörigen Objekt-Files erzeugt, sieht also wie nachfolgend aus:

```
mod1.o: mod1.c
 cc mod1
mod2.o: mod2.c
 cc mod2
mod3.o: mod3.c
 cc mod3
main.o: main.c
 cc main
```

Solch ein Make-File kann ganz einfach mit dem Editor ED erstellt werden. Es muß unter dem Namen "makefile" im aktuellen Verzeichnis vorliegen. Um aber z.B. dafür zu sorgen, daß ein Objekt-File "upgedated" wird, kann man Make wie folgt aufrufen:

```
make main.o.
```

Dieser Aufruf von Make sorgt dafür, daß im Bedarfsfalle das Objekt-File zu main.c erzeugt wird. Doch nützen einem die

einzelnen Objekt-Files ungelinkt gar nichts. Deshalb hängt man an das Make-File noch folgende Zeilen an:

```
func: mod1.o mod2.o mod3.o main.o
 ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c
```

Diese beiden Zeilen besagen einfach, daß das File func, welches das lauffähige Programm darstellt, von den vier Objekt-Files abhängig ist und daß, falls eines dieser Objekt-Files jüngeren Datums als func ist, das File func neu zusammengelinkt wird.

Make untersucht auch jedes der Objekt-Files, und wenn Make feststellt, daß zwischenzeitlich eine Änderung an einem Modul vorgenommen wurde, erzeugt es das neue Objekt-File zum geänderten C-Source, so daß das Programm func immer auf dem neuesten Stand ist.

Nun noch einmal das komplette Make-File, das durch den Aufruf make func das lauffähige Programm erzeugt:

```
mod1.o: mod1.c
 cc mod1
mod2.o: mod2.c
 cc mod2
mod3.o: mod3.c
 cc mod3
main.o: main.c
 cc main
func: mod1.o mod2.o mod3.o main.o
 ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c
```

Das waren die einfachen Funktionen von Make. Doch wenn man das Make-File so ändert, daß der Linker-Aufruf (die letzten beiden Zeilen) in die erste Zeile geschrieben werden, braucht man gar nicht mehr make func aufzurufen, sondern kann sich auf Make beschränken. Bei einfachem Aufruf von Make wird nämlich das erste Kommando des Make-Files ausgeführt.

Wollen Sie mehrere Kommandos nach einer File-zu-File-Beziehung ausführen, müssen Sie darauf achten, daß alle Kommandos mindestens eine Zeile eingerückt sind. Sonst brauchen Sie eigentlich nur noch auf den korrekten Aufruf der Befehle zu

achten, wobei Sie jedes beliebige CLI-Kommando (z.B. cd, delete etc.) verwenden können, z.B:

```
func: mod1.o mod2.o mod3.o main.o
 ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c
 delete mod1.o
 delete mod2.o
 ...
```

Eine weitere Möglichkeit von Make ist die Macro-Definition. In unserem Make-File kommt zweimal der String mod1.o mod2.o mod3.o main.o vor. Wenn man diesen nun zu Beginn des Make-Files als Macro definiert, braucht man den String nur einmal explizit zu schreiben. Dies kommt besonders Programmen mit sehr vielen Modulen zu gute:

```
OBJECT = mod1.o mod2.o mod3.o main.o

func: $(OBJECT)
 ln -o func $(OBJECT) -lRAM:c
 ...
...
...
```

Man muß nicht immer den gleichen String mehrmals schreiben. Paßt der Macro-String aber einmal nicht mehr in eine Zeile, so können Sie diesen genauso wie in C-Programmen mit dem Back-Slash (\) "verlängern":

```
OBJECT = mod1.o mod2.o mod3.o mod4.o mod5.o mod6.o\
  mod7.o mod8.o mod9.o mod10.o main.o
```

Sie können nun eigene C-Programme mittels Make schreiben.

1.2.4 Die Commercial-Version

Der Vollständigkeit halber wollen wir Ihnen noch kurz die Commercial-Version des Aztec-Compilers vorstellen. Diese unterscheidet sich von der Developers-Version nur darin, daß auf einer vierten Diskette die einzelnen Betriebssystemaufrufe aller Funktionen als Assembler-Source vorliegen. Einige Support-Funktionen für den Betrieb der Devices werden als C-Prozeduren angegeben.

Die meisten Maschinensprache-Aufrufe sind folgender Form:

```
movem.l 4(sp), d0              ;Parameter vom Stack holen
...                            ;Rücksprungadresse!
move.l #[BasePointer],a6       ;Library-Base in Register
jsr _LVOFunction(a6)           ;Funktion aufrufen
rts
```

Es gibt aber auch Routinen, die mit den Parametern noch komplizierte Berechnungen und Veränderungen durchführen.

Vielleicht ein kleiner Tip für alle diejenigen unter Ihnen, die jetzt erst mit dem Gedanken spielen, den Aztec-Compiler käuflich zu erwerben: Überlegen Sie genau, inwieweit Sie das Betriebssystem programmieren wollen und ob es notwendig ist, die Commercial-Version mit den meist gleichen Maschinensprache-Aufrufen zu kaufen. Normalerweise tut es die Developers-Version nämlich auch, und der sehr große Kostenunterschied (nahezu DM 500) darf nicht vergessen werden - zumal die arc-Files der Commercial-Version nicht von besnders hohem Gebrauchswert sind.

1.4 Der Compiler in Aktion

Bevor wir nun aber zur Funktionsweise des Compilers kommen, wollen wir diesen ein wenig benutzen und auf einige Eigenarten hinweisen.

1.4.1 Files vergleichen

Als Einstieg in die Aztec-Programmierung soll uns ein Programm dienen, das zwei Files miteinander vergleicht und die Unterschiede sowie die Position, an der ein Unterschied auftritt, anzeigt. Dazu muß dem Programm zunächst mitgeteilt werden, welche Files es vergleichen soll. Dies wird über die sogenannte Kommandozeile bzw. über die Kommando-Parameter getan. Die Parameter der Kommandozeile werden dem Programm über die Parameter der Funktion main() mitgeteilt:

```
main (argc, argv)
int   argc;
char      **argv;
{...}
```

argc (argument counter) enthält die Anzahl der angegebenen Parameter. Dabei muß man beachten, daß der Name des Programms auch als Parameter angesehen wird. Ist argc also gleich 2, so wurde tatsächlich nur ein Parameter angegeben.

Über den Zeiger argv kann man auf die Parameter (-Strings) zugreifen. Interessant ist es, sich diesen Zeiger einmal genau anzusehen:

```
char **argv;
```

Dieses Gebilde stellt nämlich einen Zeiger auf einen Zeiger vom Typ "char" dar. Zeiger vom Typ char sind aber nichts anderes als Zeiger auf Strings. Aufgrund der engen Verwandtschaft von Array und Zeiger kann man auf die einzelnen Parameter mittels argv[1], argv[2] etc. zugreifen. argv[1] ist die Adresse des ersten Parameterstrings. argv[0] enthält die Adresse des Programmnamenstrings. (Man kann char **argv; auch durch char *argv[]; ersetzen, was den Array-Zugriff auf die Parameterstrings verdeutlicht.)

Nun müssen wir testen, ob auch tatsächlich die korrekte Anzahl Parameter übergeben wurde. In unserem Beispiel benötigen wir zwei Parameter - nämlich die beiden File-Namen der Files, die verglichen werden sollen. argc muß also den Wert 3 haben (Programmnamen beachten!). Stellen wir eine Abweichung fest, wird eine Nachricht an den Benutzer ausgegeben, die ihm kurz erklärt, welche Parameter der Aufruf enthalten muß:

```
if (argc != 3)
{
   printf ("USAGE: compare FILE1 FILE2 \n");
   exit (10);
}
```

Guter Stil wäre es, auch auf ein ? als Parameter zu reagieren, einen Benutzungshinweis zu geben und dann die geforderten Parameter direkt von der Tastatur einzulesen. Diese Option wird

von vielen CLI-Befehlen unterstützt, doch ist für unser kleines Programmbeispiel dieser Aufwand ungerechtfertigt.

Die Parameter-Strings stehen uns also in argv[1] und argv[2] zur Verfügung (argv = argument values). Diese Strings können wir als Parameter für den C-Lib-Befehl fopen() verwenden.

fopen() benötigt neben dem File-Namen des zu öffnenden Files noch einen zweiten Parameter, der angibt, wie auf das File zugegriffen werden soll. fopen (Name, "r") öffnet ein bestehendes File, um dieses zu lesen, während fopen (Name, "w") ein neues File erzeugt, das nur beschrieben werden kann.

In unserem Beipiel kommt nur der Zugriff auf bestehende Files in Frage:

```
File1 = fopen (argv[1],"r");
File2 = fopen (argv[2],"r");
```

Nun wurde also versucht, die beiden zu vergleichenden Files zu öffnen. Da es aber nicht ausgeschlossen ist, daß sich der Benutzer bei der Angabe der zu vergleichenden Files irrt und eines dieser Files vielleicht gar nicht existiert, sollte man testen, ob fopen() die Files auch tatsächlich öffnen konnte:

```
if (File1 == 0L)
{    printf  ("%s kann nicht eröffnet werden !!!\n",argv[1]);
     CloseIt (10);
}

if (File2 == 0L)
{    printf ("%s kann nicht eröffnet werden !!!\n",argv[2]);
   CloseIt (10);
}
```

Sie wundern sich vielleicht, warum in der Bedingung File1 == 0L der Buchstabe L hinter der 0 steht. Dies teilt dem Compiler mit, daß er den Zeiger File1 mit einem Long-Wert vergleichen soll. Man könnte dieses 0L auch durch "(long) 0" ersetzen. Doch ist dies auf die Dauer ein wenig lästig.

Es existiert allerdings eine weitere Möglichkeit, dafür zu sorgen, daß Konstanten als "Long-Variablen" aufgefaßt werden: die

Compiler-Option +L. Hierbei ist allerdings darauf zu achten, daß auch die Int-Variablen, also nicht nur die Konstanten, zu Long-Variablen konvertiert werden. Sie sollten in Verbindung mit der Option +L allerdings die c32.lib zum Programm linken, da nur hier dafür gesorgt werden kann, daß der Argument-Counter (argc) von main(argc,argv) tatsächlich eine Long-Variable ist.

Greifen Sie bei Verwendung der c.lib und der Option +L auf diesen Zähler zu (z.B. if (argc == 3)), so werden ja 4 anstatt nur 2 Bytes für die Angabe des Wertes dieser Variablen benutzt. Dies verfälscht das Ergebnis aber in fatalem Maße, da die beiden Bytes der vorigen Int-Variable nun die beiden höherwertigen Bytes der jetzigen Long-Variablen (+L-Option) sind und die Zeiger auf die Strings um jeweils zwei Bytes verschoben werden.

Doch zurück zum eigentlichen Problem: Ist der von fopen() zurückgegebene Filedeskriptor gleich 0, liegt ein Fehler bei der Öffnung des Files vor. Dann wird eine Routine angesprungen (CloseIt()), die dafür sorgt, daß vorher geöffnete Files geschlossen werden, eine Nachricht an den Benutzer ausgegeben und das Programm verlassen wird. (Ein kleiner Tip: Programme, die z.B. Speicherbereiche belegen oder andere Operationen vornehmen, die wieder rückgängig gemacht werden müssen, sollten eine Routine zur Verfügung stellen, die automatisch alles das, was erfolgreich belegt werden konnte, wieder freigibt. Dies spart Programmierarbeit.)

Wenn die Files aber ordnungsgemäß geöffnet werden konnten, können wir uns mit dem Vergleich der beiden Files beschäftigen. Jetzt folgt die Hauptschleife unseres Programms, die die beiden Files Byte für Byte nach Gemeinsamkeiten bzw. Unterschieden durchsucht.

Dazu wird aus beiden Files das jeweils nächste Byte gelesen, und diese beiden Bytes werden dann verglichen. Sind die Bytes identisch, werden die nächsten beiden Bytes gelesen. Bei unterschiedlichen Bytes werden die beiden gelesenen Bytes ausgegeben, und die Position innerhalb der Files, an der der Unterschied aufgetreten ist, erscheint auch.

Am besten ist es, wenn man die beiden Files innerhalb einer While-Schleife untersucht, die dann abgebrochen wird, wenn eines der beiden Files vollständig gelesen wurde. Das Ende eines Files wird mit Hilfe der Funktion feof() festgestellt, und mit fgetc kann man das jeweils nächste Byte des Files lesen.

```
filepos = 0;
while (!feof(File1) && !feof(File2))
{
   Byte1 = fgetc(File1);    Byte2 = fgetc(File2);
   if (Byte1 != Byte2) printf ("$%08x  %02x <> %02x\n",
      filepos,Byte1,Byte2);
   filepos++;
}
```

In dieser Schleife werden immer die beiden gelesenen Bytes (Byte1 und Byte2) miteinander verglichen, und bei einem Unterschied wird z.B. folgender Text ausgegeben:

```
$00000154  $12 <> $3a
```

Neben den sich unterscheidenden Bytes wird auch ausgegeben, an welcher Stelle der Unterschied aufgetreten ist. Dies macht es allerdings nötig, eine Zählvariable (filepos) "mitlaufen" zu lassen, die die Anzahl der eingelesen Bytes bzw. die Leseposition innerhalb des Files enthält.

Nach dem Vergleich wird diese Zählvariable um eins inkrementiert, und wenn in beiden Files noch Daten enthalten sind, wird mit dem Vergleich fortgefahren. Enthält aber eines der beiden Files keine Daten mehr, wird die Schleife verlassen.

Die File-Statistik, die angibt, wie viele Bytes die einzelnen Files enthalten, wird durch folgende Programmsegmente erzeugt:

```
if (feof(File1) && !feof(File2))   /* File1 enthält keine Daten */
                                   /* mehr, aber File2.         */
{
   printf ("Comparison terminated !!!\n");
   printf ("%s out of data at %08lx\n",argv[1],filepos-1);
   while (!feof(File2))
   {
     fgetc (File2);
     filepos++;
   }
```

```
   printf ("%s out of data at %08lx\n",argv[2],filepos-1);
}     else
if (!feof(File1) && feof(File2))   /* File2 enthält keine Daten */
                                   /* mehr, aber File1.         */
{
   printf ("Comparison terminated !!!\n");
   printf ("%s out of data at %08lx\n",argv[2],filepos-1);
   while (!feof(File1)
   {       fgetc(File1);
      filepos++;
   }
   printf ("%s out of data at %08lx\n",argv[1],filepos-1);
}
```

Die beiden If-Anweisungen testen, welches File vollständig gelesen wurde. Im jeweiligen Rumpf wird dann der Name des vollständig gelesenen Files und dessen Länge ausgegeben. Das jeweils andere File wird bis zum Ende gelesen und für jedes gelesene Byte wird der filepos-Zähler erhöht. Nachdem das Ende des Files erreicht wurde, wird auch dieser File-Name sowie die Länge des Files ausgegeben:

```
Comparison terminated !!!
File1 out of data at $00000170
File2 out of data at $00000200
```

Noch ein paar Worte zum filepos-Zähler. In der ersten While-Schleife, in der der Vergleich der beiden gelesenen Bytes ausgeführt wird, wird filepos immer erhöht, wenn eines der beiden Files noch Bytes zur Verfügung stellt. Testet man aber mit feof() das Ende eines Files, so wird von fgetc() innerhalb der Schleifen immer ein Byte mehr als tatsächlich vorhanden gelesen. (Dies ist übrigens auch die Ursache dafür, daß beim Vergleich verschieden langer Files ein Byte mehr "verglichen" und evtl. ausgegeben wird.) Wir müssen also dafür sorgen, daß dieses "Byte zuviel" bei der Ausgabe der Größe von filepos abgezogen wird.

Haben beide Files die gleiche Länge, wird dies durch folgende If-Anweisung festgestellt.

```
(else)
if (feof(File1) && feof(File2))
   printf ("\"%s\" and \"%s\" have got the size: %08lx",
           argv[1],argv[2],filepos-1);
```

Es erfolgt dann z.B. folgende Ausgabe:

```
FILE1 and FILE2 have got the size: $00000123
```

Nun muß nur noch dafür gesorgt werden, daß die geöffneten Files geschlossen werden und das Programm verlassen wird. Dies wird durch einen Aufruf der schon oben erwähnten Routine CloseIt() erreicht:

```
CloseIt (Error_Code)
int      Error_Code;
{
   if (File1 != 0) fclose (File1);
   if (File2 != 0) fclose (File2);
   exit (Error_Code);
}
```

Diese Routine sorgt daür, daß die geöffneten Files geschlossen werden (File-Deskriptor != 0) und das Programm über exit() abgebrochen wird. Der Error_Code, der exit() übergeben wird, wird an das CLI weitergeleitet und kann so z.B. in Batch-Files dafür sorgen, daß das Batch-File abgebrochen wird (FAILAT).

Hier aber das komplette Programm, in dem auch alle verwendeten Variablen definiert werden:

```
/*******************************************************/
/*                      COMPARE                        */
/*                (c) Bruno Jennrich                   */
/*                                                     */
/*                                                     */
/* Dieses Programm vergleicht zwei Files miteinander.  */
/*******************************************************/
#include "stdio.h"

FILE *File1,                          /* File-Deskriptoren */
     *File2;

long filepos;                         /* Aktuelle File-Position */

unsigned char Byte1,                  /* gelesenes Byte */
              Byte2;

main (argc, argv)
int   argc;
char      **argv;
{
   if (argc != 3)
```

```
{
   printf ("USAGE: compare FILE1 FILE2 \n");
   exit (10);
}

File1 = fopen (argv[1],"r");        /* Files öffnen */
File2 = fopen (argv[2],"r");

if (File1 == 0L)                /* File-Open() Error */
{
   printf ("%s kann nicht eröffnet werden !!!\n",argv[1]);
   CloseIt (10);
}

if (File2 == 0L)                /* File-Open() Error */
{
   printf ("%s kann nicht eröffnet werden !!!\n",argv[2]);
   CloseIt (10);
}

filepos = 0;                  /* Aktuelle File-Position = 0 */
while (!feof(File1) && !feof(File2))
{
   Byte1 = fgetc (File1);          /* Bytes lesen */
   Byte2 = fgetc (File2);
   if (Byte1 != Byte2)             /* Unterschied ausgeben */
      printf ("$%08lx  $%02x <> $%02x\n", filepos,Byte1,Byte2);
   filepos++;                      /* File-Position erhöhen */
}

if (feof(File1) && !feof(File2))  /* File1 enthält keine     */
                                  /* Daten mehr, aber File2. */
{
   printf ("Comparison terminated !!!\n");
   printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-1);
   while (!feof(File2))
   {
      fgetc (File2);
      filepos++;
   }
                              /* File zuende lesen */
   printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-1);
}
else
if (!feof(File1) && feof(File2))  /* File2 enthält keine     */
                                  /* Daten mehr, aber File1. */
{
   printf ("Comparison terminated !!!\n");
   printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-1);
   while (!feof(File1))
   {
      fgetc (File1);
      filepos++;
   }
```

```
                                        /* File zuende lesen */
      printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-1);
   }
   else
   if (feof(File1) && feof(File2))
      printf ("\"%s\" and \"%s\" have got the size: %08lx\n",
              argv[1],argv[2],filepos-1);

   CloseIt (0);
}

CloseIt (Error_Code)
int      Error_Code;
{
   if (File1 != 0) fclose (File1);
   if (File2 != 0) fclose (File2);
   exit (Error_Code);
}
```

Da dieses Programm aus einem einzigen Modul besteht, lohnt es sich nicht, dafür extra ein Make-File anzulegen. Ein Batch-File tut es hier auch:

```
.key file
cc <file>
ln +Cb -o<file> -Lram:c
```

Dieses Batch-File kann (und sollte) übrigens für alle in den Kapiteln 1 und 2 enthaltenen Programme verwendet werden.

Die von diesem Batch-File erzeugten Programme werden immer in den Chip-Memory des Rechners geladen. Auch wenn Sie mehr als 512 KByte Speicher haben sollten, wird das Programm in diesen unteren 512 KByte angesiedelt (wenn dort noch Speicher frei ist).

Sollten Sie einmal ein Programm verfassen, das nicht mit dem noch freien Chip-Speicher auskommt (beim 1000er ca. 320 KByte), so können Sie das Programm ohne die +Cb-Option linken, müssen dann aber dafür sorgen, daß die Strukturen, die unbedingt unterhalb der 512-KByte-Grenze liegen müssen, auch dort liegen (z.B. Speicherzuweisung für Strukturen: *Zeiger_auf_Struct = AllocMem (sizeof (struct ...), MEMF_CHIP)).

1.4.2 Hexdump eines Files

Das obige Programm läßt sich leicht so umschreiben, daß es einen Hexdump eines Files ausgibt. Ein "Hexdump" ist eine Ausgabe folgender Form:

```
$00000000    $61 $62 $63 $64 $65 $65 $66 $67 $68    abcdefghi
$00000009    $69                                    j
```

Es werden also die augenblickliche Leseposition (erste Spalte), die hexadezimale Repräsentation der gelesenen Zeichen (zweite Spalte) und die einzelnen gelesenen Zeichen als ASCII-Zeichen ausgegeben.

Das Programm muß zunächst so modifiziert werden, daß nur ein einziges File geöffnet wird. Vorher muß allerdings auch hier die korrekte Anzahl der übergebenen Parameter getestet werden:

```
main (argc, argv)
int   argv;
char       *argv[];
{
   if (argc != 2)
   {
     printf ("USAGE: dump FILE \n");
     exit (10);
   }

   _File = fopen (argv[1],"r");

   if (_File == 0)
   {
     printf ("\"%s\" kann nicht eröffnet werden !!!\n",
              argv[1]);
     CloseIt (10);
   }
   ...
}
```

Dies ist Ihnen ja schon aus dem obigen Programm bekannt. Was nun aber neu ist, ist der Aufbau einer "Conversion_Table":

```
for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
for (   ;i<128;i++) Conversion_Table[i] = (char)i;
for (   ;i<160;i++) Conversion_Table[i] = '.';
for (   ;i<256;i++) Conversion_Table[i] = (char)i;
```

Diese "Conversion_Table" wird für die Ausgabe der gelesenen Bytes als Zeichen benötigt. Man kann nämlich nicht alle ASCII-Codes ausgeben. Der ASCII-Code 12 sorgt zum Beispiel dafür, daß der Bildschirm gelöscht wird. Damit dies oder ähnliches nicht passiert, werden in einer Tabelle alle druckbaren Zeichen abgespeichert. Die Zeichen, die nicht ausgegeben werden können - das sind die ASCII-Codes von 0-31 und von 128-159 - werden durch einen Punkt repräsentiert (siehe AmigaBASIC-Handbuch, dort Anhang A).

Um diese Tabelle aufzubauen, verwenden wir vier For-Schleifen. Eine Besonderheit dabei ist, daß wir bei den drei letzten For-Schleifen auf die Initialisierung der Laufvariablen i verzichten. Dies ist auch gar nicht nötig. Nach jeder FOR-Schleife hat die Laufvariable i immer den Wert, den die Variable für die Abarbeitung der nächsten For-Schleife enthalten muß.

Nun können wird die einzelnen Bytes lesen, nicht ohne vorher die File-Position auf 0 zu setzen:

```
filepos = 0;
while ((ok = Lies()) != 0)
{
   printf ("$%08lx   ",filepos);
   filepos += ok;

   for (i=0;i<ok;i++) printf ("$%02x ",Byte[i]);

   printf ("  ");

   if (ok < WIDTH) for (   ;i<WIDTH;i++) printf ("     ");

   for (i=0;i<ok;i++)
     printf ("%c",Conversion_Table[Byte[i]]);

   printf ("\n");
}
```

In der Bedingung der While-Schleife wird nun dafür gesorgt, daß Bytes aus dem File gelesen werden. Dazu wird die Routine Lies aufgerufen, die dafür sorgt, daß entweder WIDTH = 12-Bytes, oder die noch verbleibenden Bytes eines Files gelesen werden (plus eines, aufgrund der Tatsache, daß feof() erst dann

wahr (TRUE) wird, wenn ein Zeichen mehr als das letzte gelesen wurde).

Über die Variable z wird die Anzahl der gelesenen Bytes zurückgegeben:

```
int Lies()
{
   int i;
   int z;
      for (i=0;i<WIDTH;i++)
     if (!feof(_File))
     {
        Byte[i] = fgetc(_File);
        z++;
     }

   return(z);
}
```

width ist eine Konstante, die zu Beginn des Programms definiert wird. Sie gibt an, wie viele Bytes in einer Zeile dargestellt werden sollen. Im Probeausdruck zu Beginn dieses Kapitels ist width gleich 9. Bei der Ausgabe des Hexdumps auf ein 80-Zeichen breites CLI-Window kann width den Wert 12 haben. So wird die ganze Breite des Windows für die Ausgabe genutzt.

Bei der Ausgabe des Hexdumps wird zunächst die File-Position ausgegeben (printf ("$%08lx ",filepos);). Danach wird die File-Position um den Betrag der gelesenen Bytes erhöht (filepos += ok;). Das Ergebnis von Lies() ist nämlich die Anzahl der tatsächlich gelesenen Bytes (while ((ok = Lies()) != 0).

Nach der Ausgabe der File-Position (und einiger Leerzeichen zur Spaltentrennung) muß nun jedes gelesene Byte in hexadezimalem Format ausgegeben werden. Dies geschieht in einer FOR-Schleife (for (i=0;i<ok;i++) printf ("$%02lx ", Byte[i]);). Hierbei belegen die hexadezimalen Zahlen zwei Zeichen. Den Zahlen 0 bis f wird eine 0 als Füllzeichen vorangestellt.

Ist nun die tatsächliche Anzahl gelesener Bytes kleiner als width, werden für die fehlenden Bytes Leerzeichen ausgegeben (if (ok < WIDTH) for (i=ok;i<WIDTH;i++) printf (" ");). Es kann

zwar nur am Ende des Files geschehen, daß nicht mehr width-Bytes gelesen werden können, aber dennoch sieht es nicht besonders professionell aus, wenn die Ausgabe der hexadezimalen Zahlen und der ASCII-Zeichen durcheinandergebracht wird, wie z.B. so:

```
$00000000 $12 $23 $10 $07 $06 $07 $c2 $82 $85   .........
$00000009 $12 $23 $10 $07 $06    .........
```

Deshalb werden in der For-Schleife immer die gleiche Anzahl Leerzeichen ausgegeben, die von einer hexadezimalen Zahl "verbraucht" werden.

Danach brauchen nur noch die gelesenen Bytes als ASCII-Zeichen ausgegeben zu werden (for i=0;i<ok;i++) printf ("%c", Conversion_Table[Byte[i]]);). Dazu wird die vorher aufgebaute "Conversion_Table" mit dem Wert der vorher gelesenen Bytes indiziert. Nach der Ausgabe der ASCII-Zeichen wird ein Carriage Return ausgegeben (printf ("\n");), damit die weitere Ausgabe in der nächsten Zeile erfolgen kann.

Wird die While-Schleife verlassen (alle Bytes des Files wurden gelesen), müssen Sie nur noch dafür sorgen, daß das File wieder geschlossen wird. Dies wird durch CloseIt(0) erreicht, wobei CloseIt() diesmal so aussieht:

```
CloseIt (Error_Code)
int      Error_Code;
{
   if (_File == 0) fclose (_File);
   exit (Error_Code);
}
```

Hier noch einmal das ganze Programm, auch wieder mit der Definition aller Variablen und Arrays:

```
/*********************************************************/
/*                        DUMP                           */
/*                 (c) Bruno Jennrich                    */
/*                                                       */
/*                                                       */
/* Dieses Programm erzeugt ein Hexdump eines Files.      */
/*********************************************************/
#include "stdio.h"
```

```
#define WIDTH 12L                          /* 12 Zahlen pro Zeile */
                                           /* 80-Zeichen Bildschirm */

FILE *_File;                               /* Filedeskriptor */
long filepos;                              /* Aktuelle File-Position */
unsigned char Byte[WIDTH];      /* Buffer für einzulesende Bytes */
unsigned char Conversion_Table[256];
                                /* ASCII - Printable conversion */
int ok;                         /* Anzahl der gelesenen Bytes */

int Lies()
{
   int i;
   int z = 0;

   for (i=0;i<WIDTH;i++)
      if (!feof(_File))
      {
         Byte[i] = fgetc(_File);
         z++;
      }

   return (z);
}

main (argc, argv)
int   argc;
char       *argv[];
{
   int i;                                  /* Laufvariable */

   if (argc != 2)                          /* zu wenig Parameter */
   {
      printf ("USAGE: dump FILE \n");
      exit (10);
   }

   _File = fopen (argv[1],"r");            /* File öffnen */

   if (_File == 0)                         /* File-Open Error */
   {
      printf ("\"%s\" kann nicht eröffnet werden !!!\n",argv[1]);
      CloseIt (10);
   }

   for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
   for (   ;i<128;i++) Conversion_Table[i] = (char)i;
   for (   ;i<160;i++) Conversion_Table[i] = '.';
   for (   ;i<256;i++) Conversion_Table[i] = (char)i;
                               /* Conversion_Table aufbauen */

   filepos=0;                  /* Aktuelle Fileposition gleich 0 */
   while ((ok = Lies()) != 0)
   {
```

```
      printf ("$%08lx   ",filepos);
                              /* Aktuelle File-Position ausgeben */
      filepos+=ok;            /* Aktuelle File-Position erhöhen */

      for (i=0;i<ok;i++) printf ("$%02x ",Byte[i]);
                                        /* Hexadezimale Zahlen */

      if (ok < WIDTH) for (i=ok;i<WIDTH;i++) printf ("    ");
                                        /* Evt. "Leer-Zahlen" */
      printf ("  ");                    /* Spaltenabtrennung */

      for (i=0;i<ok;i++) printf
         ("%c",Conversion_Table[Byte[i]]);
                                        /* "ASCII"-Darstellung */
      printf ("\n");                    /* Carriage-Return */
   }
   CloseIt (0);                         /* Tschüs */
}

CloseIt (Error_Code)
int      Error_Code;
{
   if (_File != 0) fclose (_File);      /* Wenn File offen, */
                                        /* File schliessen */
   exit (Error_Code);
}
```

1.4.3 Ein Monitor

Die hexadezimale Ausgabe läßt sich durch kleinere Änderungen auch für einen Monitor benutzen. Mit einem Monitor ist es möglich, sich Speicherbereiche des Amiga anzusehen. Der Monitor zeigt also die verschiedenen Speicherinhalte an.

Dazu müssen Sie dem Programm die Anfangs- und Endadresse des Speicherbereichs mitteilen, der angezeigt werden soll. Auch dies geschieht wieder über die Kommando-Parameter. Diesmal werden wieder zwei Parameter übergeben:

```
main (argc, argv)
int   argc;
char      **argv;
{
   if (argc != 3)
   {
      printf ("USAGE : mon ANFANG ENDE \n");
      exit (10);
```

```
   }
...
}
```

Nachdem abgeklärt wurde, daß die richtige Anzahl Parameter übergeben wurde, kann man nun beginnen, diese zu konvertieren. Die angegebenen Anfangs- und Endadressen liegen ja noch als ASCII-Strings vor und müssen deshalb in eine Zahl umgewandelt werden. Wir haben uns entschieden, den String in eine hexadezimale Zahl umzuwandeln. Fast jeder Monitor arbeitet mit hexadezimalen Zahlen - also auch unser Monitor.

Leider stellt uns der Compiler keine Routine zur Verfügung, die einen ASCII-String in eine Hex-Zahl wandelt. Diese müssen wir selber schreiben:

```
unsigned char *atoh (String)
char               *String;
{
   int i;
   unsigned char *hex;
   unsigned long faktor;

   hex = (unsigned char *)0L;
   faktor = 1L;

   if (strlen (String) <= 8)
     for (i=(strlen(String)-1);i>=0;i--)
     {
          if ((String[i] >= '0') && (String[i] <= '9'))
          {
               hex += (String[i] -'0')*faktor;
               faktor *= 0x10;
          }
               if (((String[i] & MASK) >= 'A') &&
              ((String[i] & MASK) <= 'F`))
          {
               hex += ((String[i] & MASK) - 'A'+0xa)*faktor;
               faktor *= 0x10;
          }
     }
   return (hex);
}
```

Diese Routine setzt zunächst den Rückgabewert hex auf 0 und einen "faktor" auf 1. Die folgenden Schritte werden nur dann ausgeführt, wenn die Länge des eingegebenen Zahlen-Strings 8

Zeichen nicht überschreitet. Dies deshalb, weil der Adreßbereich von 4 Gigabytes, den der Amiga theoretisch adressieren kann, durch die Adressen 0 - ffffffff abgedeckt wird. Die höchste Adresse ist also ffffffff. Diese Zahl enthält als String aber nur 8 Zeichen.

Hat der eingegebene Zahlen-String weniger oder genau 8 Zeichen, so wird die Hauptschleife dieser Routine ausgeführt. In dieser Schleife wird der String Zeichen für Zeichen betrachtet - allerdings wird mit dem letzten Zeichen des Strings begonnen und mit dem ersten aufgehört.

In dieser Schleife wird bei der Initialisierung der Laufvariablen allerdings der Wert 1 von der Länge des Strings abgezogen (for (i=(strlen(String)-1);i>=0;i--)). Das geschieht, weil das erste Zeichen des Strings String[0] und nicht String[1] ist. Damit wir bei den in der Schleife folgenden Zugriffen auf Zeichen des Strings nicht immer wieder den Wert 1 abziehen müssen, wird dies einmal zu Beginn der Schleife bei der Initialisierung der Laufvariablen getan.

Innerhalb der Schleife werden nun zwei Fälle unterschieden:

1. Das Zeichen des Strings ist eine Ziffer (0-9).
2. Das Zeichen des Strings ist ein Buchstabe (A-F bzw. a-f).

Bei der Berechnung des Ergebnisses wird nun getestet, ob das gerade aktuelle Zeichen des Zahl-Strings eine Ziffer (0-9) oder ein Hex-Zeichen (a-f) ist.

Im ersten Fall wird vom ASCII-Wert der eingelesenen Ziffer der ASCII-Wert der Ziffer 0 abgezogen. Dies geschieht aus einem sehr einleuchtenden Grund: Durch diese Subtraktion erhält man nämlich den Zahlenwert, den diese Ziffer darstellt. "1 - 0" ergibt also den Zahlenwert 1. "2 - 0" ergibt den Zahlenwert 2 etc. Diese Zusammenhänge kann man sehr leicht aus einer ASCII-Tabelle (z.B. aus dem AmigaBASIC-Handbuch) ersehen. Der ASCII-Code der Ziffer 0 beträgt nämlich 48, der der Ziffer 1 beträgt 49 und so weiter, bis zur Ziffer 9, die den ASCII-Code 57 hat.

Die so erhaltene Zahl wird mit dem "faktor" multipliziert und zum bisherigen Ergebnis addiert. Darauf wird der "faktor" mit dem Wert 0x10 multipliziert. Die Muliplikation des Faktors mit 0x10 = 16 kommt einer Stellenverschiebung gleich.

Im Dezimalsystem bedeutet die Zahl 123 bekanntlich:

3 Einer + 2 Zehner + 1 Hunderter

Ähnlich sieht es bei Hexadezimal-Zahlen aus, nur daß hier die Zahl 16 und nicht die Zahl 10 als Basis gewählt wurde. Die hexadezimale Zahl 123 bedeutet also:

3 Einer + 2 Sechzehner + 1 Zweihundertsechsundfünfziger

Oder anders geschrieben:

3 * 0x1 + 2 * 0x10 + 1 * 0x20

"faktor" durchläuft nun der Reihe nach die Werte 0x1, 0x10, 0x20 etc. Nun wird vielleicht auch klar, warum der String von hinten nach vorne durchgearbeitet wird. Der Computer weiß garantiert, daß das letzte Zeichen des Strings die Einer-Stelle der Zahl repräsentiert. Das vorletzte Zeichen steht an der Sechzehner-Stelle, das dritte Zeichen an der 256er Stelle etc.

Doch kommen wir zum zweiten Fall. Hier werden die Hex-Ziffern a-f untersucht. Da wir nicht davon ausgehen, daß Sie nur Klein- oder nur Großbuchstaben verwenden, konvertieren wir die Zeichen zunächst in Großbuchstaben. Dies geschieht durch Ausblendung des Bits 4. (String[i] & MASK" = "String[i] & (255L-32L)) (siehe ASCII-Tabelle). Von der gerade zu bearbeitenden Hex-Ziffer wird der ASCII-Wert des Zeichens A abgezogen, um auch hier die Werte 0, 1, 2 etc. zu erhalten. Da die Zahl A aber den Wert 0xa sprich 10 darstellt, muß zum erhaltenen Wert noch der Wert 0xa addiert werden. Mit der Berechnung des Ergebnisses kann wie oben fortgefahren werden.

Die Anfangs- und Endadressen, die den anzuzeigenden Speicherbereich bestimmen, können nun mittels dieser Funktion errechnet werden:

```
lowadress  = atoh (argv[1]);
highadress = atoh (argv[2]);
```

Beachten sollten Sie, daß Zeichen, die weder Ziffer noch Hex-Zeichen sind, nicht berücksichtigt werden. Der Zahl-String 12$$5 liefert den gleichen Wert wie 125.

Nun wird getestet, ob die Anfangsadresse kleiner als die Endadresse des zu durchleuchtenden Speicherbereichs ist. Der Speicher wird nämlich in aufsteigender Reihenfolge bearbeitet:

```
if (lowadress > highadress)
{
   printf ("ANFANG muß kleiner als ENDE sein !\n")
   exit (10);
}
```

Wenn die Anfangsadresse größer als die Endadresse ist, wird das Programm sofort verlassen. Ist es jedoch so, daß die Anfangsadresse kleiner als die Endadresse ist, können wir die aus obigem Programmbeispiel (s. dump) bekannte Conversion_Table aufbauen:

```
for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
for (   ;i<128;i++) Conversion_Table[i] = (char)i;
for (   ;i<160;i++) Conversion_Table[i] = '.';
for (   ;i<256;i++) Conversion_Table[i] = (char)i;
```

Ähnlich wie wir mit filepos in den oberen Programmen gearbeitet haben, um die aktuelle File-Position darzustellen, arbeiten wir hier mit der Variablen pos, die die Adresse der jeweils auszugebenden Speicherstelle enthält. Zu Beginn erhält diese den Wert von lowadress:

```
pos = lowadress;
```

Nun können wir mit der Darstellung der Speicherinhalte beginnen. Doch diese sollte man innerhalb einer weiteren Routine ausgeben, da wir hier selber testen müssen, wann die Ausgabe beendet werden soll.

Beim Hexdump eines Files war dies nicht so schwer, da uns die Funktion Read() mitgeteilt hat, ob wir das File weiter auslesen

konnten oder nicht. Read() hat uns auch mitgeteilt, wie viele Bytes wir ausgeben mußten.

Die Anzahl der jeweils noch auszugebenden Zeichen müssen wir selber berechnen. Dies ist gar nicht schwer:

```
width = WIDTH;
while (pos < highadress)
   if ((highadress-pos) >= width) Show (WIDTH);
   else Show (highadress-pos);
```

Solange der Wert von pos die Adresse der höchsten auszugebenden Speicherstelle nicht erreicht hat, werden Bytes ausgegeben. Dabei wird unterschieden, ob noch so viele Bytes ausgegeben werden sollen, wie in eine Zeile passen (Show (width)), oder weniger.

Ist die Anzahl der auszugebenden Bytes kleiner als die Anzahl der Bytes, die in einer Zeile dargestellt werden können (width), so werden nur noch die tatsächlich darzustellenden Bytes dargestellt (Show (highadress-pos)) - kein Byte mehr und kein Byte weniger.

Um allerdings zu testen, ob nur noch wenige (weniger als width) Bytes auszugeben sind, muß man die Anzahl der auszugebenden Bytes (highpos-pos) mit der Anzahl der Bytes, die in einer Zeile dargestellt werden können, vergleichen. Leider versteht es der Aztec-Compiler nicht, Long-Konstanten (z.B. 12L) auch im Vergleich als Long-Konstanten anzusehen (Eine Ausnahme bildet das ==, bei dem Long-Konstanten nicht verarbeitet werden). Bei Zuweisungen und als Parameter von Prozeduren werden Long-Konstanten korrekt behandelt.

Wir haben uns deshalb einen Trick einfallen lassen: Wir weisen einfach den Wert der Konstanten einer Long-Variablen zu und verwenden diese Variable beim Vergleich. Dies versteht der Compiler. (Bei Verwendung der +L-Option und der c32.lib brauchen Sie diesen Trick nicht anzuwenden.)

Nun aber zu der Routine, die für die Ausgabe der Bytes auf dem Bildschirm verantwortlich ist. Sie ist Ihnen, wie oben schon

erwähnt, im großen und ganzen bekannt. Wir haben nur ein paar kleine Änderungen vorgenommen:

```
Show (ok)
long  ok;
{
   int i;

   printf ("$%08lx    ",pos);

   for (i=0;i<ok;i++)
      printf("$%02x ",*(pos+i));

   if (ok < WIDTH) for (    ;i<WIDTH;i++) printf ("    ");

   printf (" ");

   for (i=0;i<ok;i++)
     printf ("%c",Conversion_Table[*(pos++)];

   printf ("\n");
}
```

Diese Routine gibt zunächst die Adresse des ersten dargestellten Bytes einer Zeile aus. Dann wird jedes Byte - wie oben schon - in hexadezimaler Notation ausgegeben. Auch daß bei einer kleineren Anzahl auszugebender Bytes Leerzeichen ausgegeben werden, um die ASCII-Zeichen tatsächlich in einer Spalte ausgeben zu können, ist schon bekannt.

Neu ist allerdings der Zugriff auf pos. Auf diese Variable - oder vielmehr Zeiger - wird zunächst mit *(pos+i) zugegriffen. Dies bedeutet nichts anderes als: Addiere zur aktuellen Adresse von pos den Wert i, und gib dann den Inhalt dieser neuen Adresse zurück.

Ähnlich ist es mit *(pos++). Dies heißt allerdings: Inkrementiere die Adresse, auf die pos zeigt, und liefere den Inhalt dieser neuen Adresse zurück.

In beiden Fällen sind die Klammern nötig. * hat eine größere Priorität als ++. Würde man die Klammern weglassen, hätte *(pos++) die Bedeutung: Hole den Inhalt der Speicherstelle, auf die pos zeigt, und inkrementiere diesen Wert (*pos++).

Hier noch einmal das komplette Programm:

```
/*********************************************************/
/*                          MON                          */
/*                  (c) Bruno Jennrich                   */
/*                                                       */
/*                                                       */
/* Dieses Programm gibt einen Speicherbereich als        */
/* Hexdump aus.                                          */
/*********************************************************/

#define WIDTH 12L           /* Anzahl darzustellender Bytes */
#define MASK (255L-32L)     /* für Konvertierung in Groß- */
                            /* buchstaben.                */

unsigned char Conversion_Table[256];
                            /* Conversion Tabelle */
unsigned char *lowadress,   /* kleinste darzustellende Adresse */
              *highadress;  /* größte darzustellende Adresse */

unsigned char *pos;         /* aktuelle Adresse, bzw. Position */
                            /* im Speicher.                    */

unsigned char *atoh (String)   /* Ziffern-String nach Hex */
char                *String;
{
   int i;
   unsigned char *hex;          /* Ergebnis */
   unsigned long faktor;        /* Stellenfaktor */

   hex = (unsigned char *)0L;   /* Ergebnis = 0 */
   faktor = 1L;                 /* Faktor gleich 1 (Einer Stelle) */

   if (strlen (String) <= 8)    /* mehr als 8 Zeichen */
      for (i=(strlen(String)-1);i>=0;i--)
      {
         if ((String[i] >= '0') && (String[i] <= '9'))
         {                      /* Ziffern */
            hex += (String[i] - '0')*faktor;
            faktor *= 0x10;
         }

         if (((String[i] & MASK) >= 'A') &&
             ((String[i] & MASK) <= 'F'))
         {                      /* Hex-Ziffern */
            hex += ((String[i] & MASK)-'A'+0xa)*faktor;
            faktor *= 0x10;
         }
      }

   return (hex);
}
```

```
main (argc, argv)
int   argc;
char      **argv;
{
   int i;
   if (argc != 3)       /* Anzahl der Parameter ist falsch */
   {
      printf ("USAGE: mon ANFANG ENDE\n");
      exit (10);
   }

   lowadress  = atoh (argv[1]);   /* Niedrigste und höchste */
   highadress = atoh (argv[2]);   /* Adresse ausrechnen     */

   if (lowadress > highadress)    /* Fehler bei Eingabe */
   {
      printf ("ANFANG muss kleiner als ENDE sein !!!\n");
      exit (10);
   }

   for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
   for (   ;i<128;i++) Conversion_Table[i] = (char)i;
   for (   ;i<160;i++) Conversion_Table[i] = '.';
   for (   ;i<256;i++) Conversion_Table[i] = (char)i;
                                  /* Conversion_Table aufbauen */

   pos = lowadress;               /* aktuelle Position gleich */
                                  /* niedrigser Adresse */

   while (pos < highadress)        /* Ausgabeschleife */
      if ((long) (highadress-pos) >= WIDTH) Show (WIDTH);
      else  Show ((long) (highadress-pos));
}

Show (ok)
long ok;
{
   int i;

   printf ("$%08lx   ",pos);          /* Position ausgeben */

   for (i=0;i<ok;i++)
      printf ("$%02x ",*(pos+i));     /* Byte als Hex ausgeben */

   if (ok < WIDTH) for (   ;i<WIDTH;i++)
      printf ("    ");                /* Platz mit Leerstellen */
                                      /* füllen                */
   printf (" ");

   for (i=0;i<ok;i++)
      printf ("%c",Conversion_Table[*(pos++)]);
                                      /* Bytes als Hex */
```

```
    printf("\n");                    /* Carriage-Return */
}
```

Beachten sollten Sie, daß der Inhalt der angegebenen Endadresse nicht ausgegeben wird. mon 100 200 zeigt also die Inhalte der Speicherstellen Hex 100 bis einschließlich Hex 1ff.

1.5 Der Lattice-C-Compiler 4.0

Neben dem Aztec-Compiler existiert noch ein weiterer guter C-Compiler: der Lattice-Compiler. Dieser Compiler war der erste C-Compiler, der für den Amiga zur Verfügung stand. Auf ihm wurden auch einige Betriebssystemroutinen entwickelt.

Dieser Compiler wird auf 3 Disketten geliefert. Auf der ersten Diskette (Lattice_C_4.0.1) befinden sich der Compiler (lc) und der Linker (blink). Auf der zweiten Diskette findet man die Linker-Librarys und C-Include-Files. Mit diesen Include-Files hat es allerdings eine besondere Bewandtnis: Sie sind komprimiert. Dadurch ist es möglich, den Vorgang des Include-File-Compilierens zu beschleunigen. Diese Komprimierung bezieht sich allerdings nicht darauf, daß aus diesen Files alle Kommentare entfernt wurden wie beim Aztec-Compiler, sondern diese Files sind sozusagen vor-compiliert. Auf der zweiten Diskette findet man außerdem einige Source-Files (unter anderem den Startup-Code, die Routine, die Ctrl-C abfängt usw.). Die dritte Diskette des Lattice-Pakets schließlich enthält neben einigen Beispielprogrammen nochmals alle Include-Files, diesmal aber unkomprimiert.

Doch kommen wir dazu, wie man den Lattice-Compiler installiert bzw. mit Hilfe einer Startup-Sequenz dafür sorgt, daß man compilieren kann. Dazu müssen zunächst vier Umgebungsvariablen mit Hilfe des ASSIGN-Befehls gesetzt werden:

```
ASSIGN LC: Lattice_C_4.0.1:c              ;Wo ist der Compiler?
ASSIGN INCLUDE: Lattice_C_4.0.2:include   ;Wo sind die Includes?
ASSIGN LIB: Lattice_C_4.0.2:lib           ;Wo sind Linker-Libs?
ASSIGN QUAD: ram:                    ;Wohin mit temporären Files?
```

Wie Sie sehen, werden die ersten beiden Disketten zum Compilieren eines Files benötigt. Dies macht es fast unumgänglich, den Lattice-Comiler mit mindestens zwei Laufwerken oder einem Laufwerk und einer Harddisk zu betreiben (Lattice gibt als beste Konfiguration für den Betrieb vor: 2 Laufwerke, 1 Harddisk (mit mind. 20 MB), 2,5 MB Speicher). Natürlich läßt sich der Lattice-Compiler auch nur mit einem Laufwerk betreiben. Allerdings müssen dann sehr oft Diskettenwechsel durchgeführt werden.

Wenn Sie die obigen Befehle in Ihre Startup-Sequenz eingefügt haben (neben der Installation des deutschen Tastaturtreibers usw.), kann es eigentlich schon losgehen. Leider stellt der Lattice kein Make-ähnliches Utility zur Verfügung, so daß der Benutzer mit Hilfe von Batch-Files Programme compilieren muß.

Das Batch-File einfachster Art sieht so aus:

```
.key file
LC:lc -L <file>
```

Dieses Batch-File sorgt dafür, daß das angegebene File compiliert und direkt im Anschluß daran gelinkt wird, so daß ein lauffähiges Programm entsteht.

Betrachten wir einmal ein Batch-File, das den Linker explizit aufruft:

```
.key file
LC:lc <file>
LINK:blink FROM LIB:c.o <file>.o TO <file> LIB LIB:lc.lib
lib:amiga.lib
```

Betrachten wir einmal den Linker-Aufruf genauer:

Zunächst wird festglegt, welche Objekt-Files zusammengelinkt werden sollen (FROM). In unserem Beispiel wird zunächst der Startup-Code, der ja zu Beginn eines jeden Programms stehen muß, angegeben. Darauf folgt das eigentliche Compilat (<file>.o). Das TO-Schlüsselwort bestimmt, welchen Namen das lauffähige Programm erhalten soll (<file>). Nun folgt das Schlüsselwort LIB (nicht zu verwechseln mit der Umgebungsva-

riable LIB:!). Die nach diesem Schlüsselwort folgenden Librarys werden zum Linken verwendet. Dabei enthält die lc.lib alle Lattice-spezifischen Routinen (strlen(), fopen() usw.), während die amiga.lib alle Aufrufe der Amiga-Betriebssystemroutinen (Write(), Move(), Draw() etc.) enthält.

Mit Hilfe dieses Batch-Files ist es also möglich, ein Programm zu compilieren. (Wenn mehrere Module zusammengelinkt werden sollen, brauchen Sie die Namen der einzelnen Module nur nach dem FROM-Schlüsselwort aufzulisten, getrennt durch ein + oder Leerzeichen).

Da wir, wie oben schon gesagt, im Buch auf den Aztec-Compiler eingehen, und dessen Optionen im Verlauf des Buches erläutern, finden Sie im Anhang eine Erklärung der Lattice-Compiler- und Linker-Optionen.

2. Der Compiler unter der Lupe

Allen drei Versionen des C-Compilers gemeinsam ist der Compiler, Assembler und Linker. Wir wollen uns nun mit dem Compiler, dem Assembler und dem Linker und deren Optionen beschäftigen und dem Compiler beim Compilieren ein wenig auf die Finger schauen.

Dazu betrachten wir zunächst den Aufruf des Compilers. Er erfolgt nach dieser Form:

```
cc [>AusgabeFile] [Optionen] Prog[.c]
```

Dabei ist alles, was in eckigen Klammern steht, optional, muß also nicht unbedingt angegeben werden. Angegeben werden muß der Name des zu compilierenden Files. Dabei ist es dem Compiler egal, ob Sie statt "cc Programm" "cc Programm.c" angeben.

Das Ausgabe-File und die Compiler-Optionen können Sie optional angeben. Das Ausgabe-File ist besonders bei der Programmentwicklung von Nutzen. Hier werden alle Texte, die der Compiler ausgibt, hineingeschrieben - also auch die eventuellen Fehlermeldungen. Geben Sie z.B. cc >prt: Programm an, wird folgender Text auf dem Drucker ausgeben, wenn das Programm keine Fehler aufweist:

```
"Aztec C68K 3.4a  1-25-87  (C) 1982-1987 by Manx Software
                                              Systems, Inc.
Aztec 68000 Assembler 3.4a  1-25-87"
```

Doch können Sie die Texte auch auf ein Disketten-File lenken - z.B. auf errors.c - und dieses ASCII-File dann in Ruhe mit dem Ed anschauen (cc >errors.c Programm).

Die Möglichkeit der Umlenkung der Ausgabe der Texte auf ein anderes File nennt man "Re-Direction". Diese Möglichkeit läßt sich bei fast jedem CLI-Kommando anwenden, so auch bei Assembler und Linker. Mit dir >prt: kann man z.B. das aktuelle Directory auf den Drucker ausgeben.

Bei der Ausgabe der Compiler-Texte auf den Drucker o.ä. sollten Sie beachten, daß der Benutzer nach jeweils fünf aufgetretenen Fehlern (nicht Warnings) aufgefordert wird, den Fortgang der Compilation zu bestätigen. Dies hat den Vorteil, daß die Fehlermeldungen, die ja normalerweise auf dem Bildschirm ausgegeben werden, nicht so schnell an Ihnen vorbeihuschen. Sie haben die Möglichkeit, alle Fehler zu betrachten, obwohl es vielleicht so viele sind, daß nicht alle auf einmal auf den Bildschirm passen.

Bei der Ausgabe der Fehler auf einen Drucker oder ein File ist diese Bestätigung aber nicht notwendig, da die Texte ja "konserviert" werden, so daß Sie den Zwang zur Bestätigung durch die Compiler-Option -B (Eselsbrücke: "minus Bestätigung") aufheben sollten.

Im übrigen hat ein Fehler im C-Programm immer folgendes Format:

```
Listing_der_Fehlerhaften_Zeile
^
Programmname.c:Fehlerhafte_Zeile: ERROR Nummer:
                                        Kurzbeschreibung:

Z.B:

}
^
test.c:23: ERROR 69: missing semicolon
```

Das Potenz-Zeichen (^) gibt an, wo sich - nach Meinung des Compilers - der Fehler in der fehlerhaften Zeile befindet. Beim "missing semicolon"-Fehler, der die Nummer 69 hat, merkt der Compiler den Fehler immer erst einen Befehl später. Da es ein ungeschriebenes Gesetz ist - an das sich leider nicht alle Programmierer halten -, daß in eine Zeile nur ein Befehl geschrieben wird (abgesehen von If-Anweisungen mit nur einem Befehl im Rumpf), tritt dieser Fehler immer eine Zeile später auf.

Kommen wir aber nun zu den Compiler-Optionen, von denen es eine ganze Menge gibt:

Compiler-Optionen

-A

Der Assembler wird nicht automatisch vom Compiler gestartet (minus Assembler-Option).

-DSymbol[=Wert]

Diese Option weist dem angegebenen Symbol den optional anzugebenden Wert zu. *-DName=Wert* kommt einem #define Name Wert gleich, während *-DName* dafür sorgt, daß das Symbol den Wert 1 erhält (#define Name 1) (Define-Option).

-IDirectory

Die Include-Files werden aus dem hier angegebenen Directory gelesen. Wollen Sie, daß mehrere Unterverzeichnisse durchsucht werden sollen, können Sie die zu durchsuchenden Directorys durch ein ! trennen (*-IDir1!Dir2!Dir3*) - (Include-Option).

-OFilename

Das Objekt-File (oder der Assembler-Source - bei Verwendung der Option -A) wird in das angegebene File geschrieben (Output-Option).

-S

Die Ausgabe von Warnings wird unterdrückt (Silent-Option).

-T

Im vom Compiler erzeugten Assembler-Source werden die einzelnen C-Zeilen als Kommentar angegeben. Diese Option ist allerdings nur in Verbindung mit -A sinnvoll (Text-Option).

-B

Bei der Ausgabe von Fehlermeldungen auf einen Drucker o.ä. ist es sinnlos, daß nach der fünften ausgegebenen Fehlermeldung auf eine Bestätigung Ihrerseits zum Fortsetzen der Compilation gewartet wird. Die Fehlermeldungen gehen in diesem Falle ja nicht verloren, wie es bei der Ausgabe auf den Bildschirm geschehen würde, wenn noch mehr Fehlermeldungen ausgegeben

würden. Sie können nun mit dieser Option den Zwang zur Bestätigung unterdrücken (minus Bestätigung-Option).

-ENummer

Diese Option veranlaßt den Compiler, eine Ausdruck-Tabelle mit *Nummer* Einträgen zu allokieren. Diese Ausdruck- bzw. Expression-Table wird zur Auswertung von For-Schleifen, Switch-Case-Anweisungen etc. benötigt (Rekursion).

-LNummer

Diese Option veranlaßt den Compiler, *Nummer* Einträge für die lokale Symboltabelle zu reservieren. Diese lokale Symboltabelle wird für Variablendefinitionen und -deklarationen innerhalb von Prozeduren bzw. Blöcken (von { und } eingeschlossene Programmsegmente - z.B. bei If, For, Switch etc.) verwendet.

-YNummer

Diese Option veranlaßt den Compiler, eine Case-Tabelle mit *Nummer* Einträgen anzulegen. Diese Case-Tabelle wird für die Übersetzung der Switch-Anweisungen benötigt.

-ZNummer

Diese Option veranlaßt den Compiler, eine String-Tabelle mit *Nummer* Einträgen anzulegen. Die String-Tabelle wird zur Speicherung der Strings während der Compilation benutzt.

+B

Diese Option verhindert die Erzeugung von public.begin im Assembler-File. Dies ist bei der Benutzung mehrerer Module wichtig, um zu verhindern, daß in jedem Modul der Befehl jmp.begin erzeugt wird, was eigentlich nur im ersten Modul geschehen muß.

+C

Es wird "large code" erzeugt (siehe Kapitel 2.3).

+D

Es wird "large data" erzeugt.

+HFilename

Mit dieser Option können Sie die vom Compiler für ein Programm erzeugte Symboltabelle abspeichern.

+IFilename

Mit dieser Option können Sie die vorher mit +H abgespeicherte Symboltabelle wieder laden. Sie können z.B. die von einem Programm verwendeten Include-Files compilieren, abspeichern und mittels +I einlesen. Dadurch wird ein Geschwindigkeitsvorteil erreicht, da die Include-Files nicht immer wieder neu compiliert werden müssen.

+L

Diese Option veranlaßt den Compiler, alle Int-Variablen als Long-Variablen zu interpretieren. Bei Verwendung der c32.lib ist es nun möglich, Lattice-Programme zu übernehmen.

+Q

Die in einem Programm verwendeten Strings werden ins Datensegment geschrieben.

+ff

Berechnungen werden mit einfacher Genauigkeit durchgeführt. m.lib muß zum Objekt-File des Programms gelinkt werden.

+fi

Berechnungen werden mit doppelter Genauigkeit durchgeführt. Sie müssen bei Verwendung der "double-precision floating-point routines" die ma.lib oder die mx.lib beim Linken verwenden.

+f8

Es wird der 68881-Mathe-Coprozessor für die Berechnungen verwendet. Zur Benutzung dieses Prozessors, der nachträglich eingebaut werden kann, müssen Sie zum Programm die m8.lib linken.

+m

Vor Eintritt in jede Routine wird geprüft, ob ein Stack-Overflow (Stack-Überlauf) aufgetreten ist. Wenn ja, meldet sich der Compiler mit einer dahingehenden Fehlermeldung.

-n

Es werden Debugging-Informationen in der Assembler-Source eingebettet, die vom Assembler weiterbearbeitet werden.

+P

Diese Option ist identisch mit den Optionen +C +D +L. Weiterhin werden die Register D2 und D3 vor jedem Funktionsaufruf gerettet. Diese Option verlangt, daß die cl32.lib zum Programm gelinkt wird.

Einige dieser Optionen erklären sich von selbst. So z.B. die Option -a, die man sich vielleicht durch die Eselsbrücke "minus Assembler" merken könnte. Sie veranlaßt den Compiler nämlich, den Assembler nicht zu starten, wie er es sonst tut.

Damit der Compiler aber nicht umsonst compiliert, müssen Sie angeben, wohin das Assembler-Source-File, das der Compiler normalerweise in der RAM-Disk erzeugt (siehe CCTEMP-Variable) und nach Gebrauch wieder löscht, geschrieben werden soll. Dazu müssen Sie den Namen des Assembler-Source-Files mittels der -O(Output)-Option angeben. Der separate Aufruf von Compiler und Assembler, den Sie ja nun selber aufrufen müssen, verlängert sich damit schon auf:

```
cc -A -O Programm.asm Programm.c
as Programm.asm
```

Nicht nur, daß mehr Parameter an cc übergeben werden müssen - nein, Sie müssen auch den Assembler "von Hand" aufrufen, der aus Programm.asm das Objekt-File Programm.o erzeugt. (Es hat sich eingebürgert, daß Assembler-Source-Files die Extension .asm erhalten, ebenso wie beim C-Source das Tag .c an den File-Namen angehängt wird. Objekt-Files hingegen erhalten das Tag .o) Geben Sie beim Compilieren nur die Option -O an (ohne

-A), so erhält das Objekt-File, das der vom Compiler aufgerufene Assembler erzeugt, den angegebenen Namen.

Normalerweise brauchen Sie den Namen für das Objekt-File aber nicht anzugeben, da automatisch ein Objekt-File mit dem Namensstamm des Programm-Files, aber auf .o geänderter Extension, erzeugt wird. Ein kleiner Tip am Rande: Sollte Ihr Programm einmal so groß sein, daß Sie um jedes Byte Speicherplatz auf der Diskette kämpfen müssen, so sollten Sie das Objekt-File nach jedem Link-Vorgang mit "delete" wieder löschen. Dies kann sowohl bei der Professional-Version im Batch-File als auch bei der Developers-Version im Make-File geschehen. Dies ist allerdings nur dann interessant, wenn Sie mehrere Programme, die jeweils nur aus einem Modul bestehen, auf einer Diskette haben.

2.1 Dem Compiler auf die Finger geschaut

Um die weiteren Compiler-Optionen zu erläutern, wollen wir uns den Compiler ein wenig genauer ansehen. Wir wollen beobachten, wie der Compiler mit einem Programm, das als ASCII-File vorliegt, umgeht.

Zunächst untersucht der Compiler das Programm nach offensichtlichen Fehlern. Der Compiler weiß z.B., daß nach jedem Befehl ein Semikolon folgen muß. Findet der Compiler das Semikolon nach einem Befehl nicht, quittiert er das mit einer Fehlermeldung. Wenn nach einer If-Anweisung die zu testende Bedingung fehlt, wird das festgestellt und dem Programmierer unmißverständlich mitgeteilt. Es gibt aber auch Fehler, die der Compiler nicht erkennen kann und die nur Sie als Programmierer feststellen können - meist dann, wenn Ihr Programm abstürzt oder nicht das tut, was Sie von ihm verlangen. Dann tritt der Debugger in Aktion (siehe Kapitel 2.5).

Ein typischer Fehler dieser Art ist z.B. die Verwechslung von == und = bzw. ein vergessenes Gleichheitszeichen (=) bei der Bedingung einer If-Anweisung (z.B. if (a=1) {..}). Der Rumpf der If-Anweisung wird nie ausgeführt, und der Programmierer wun-

dert sich, warum. Wenn sich der Benutzer allerdings nicht darüber wundert, daß der If-Rumpf nicht ausgeführt wird, weil er es gar nicht erwartet, so wundert er sich darüber, daß die Variable a auf einmal den Wert 0 hat.

Beliebt ist aber auch der Zugriff auf ein nicht existentes Array-Element:

```
int Array[4];
```

Dieses Array enthält die Elemente 0, 1, 2 und 3, also insgesamt 4 Elemente. Weisen Sie nun aber an Array[4], das "fünfte" Element, einen Wert zu, kann es passieren, daß Sie eine andere, für das Programm sehr wichtige Variable ändern und so den Programmabsturz "vorprogrammieren". Da, wie Sie gleich noch erfahren werden, Arrays auch auf dem System-Stack abgespeichert werden, könnte es auch passieren, daß Sie bei einem Array-Zugriff die Rücksprungadresse einer Subroutine zerstören, was verheerende Folgen hat.

Bei solchen Fehlern hilft Ihnen meist nur noch ein gutes Auge und ein Gespür dafür, wo solche Fehler verborgen sein könnten - oder der Debugger (dieser war es, der uns einen solchen fehlerhaften Array-Zugriff aufdeckte).

Doch nach dem Test auf Fehler folgt die Übersetzung der Befehle und Anweisungen in Maschinensprache. Diese Übersetzung kann direkt nach der Fehlerüberprüfung eines Befehls geschehen oder nachdem der Compiler das ganze Programm nach Fehlern durchsucht hat und festgestellt wurde, daß keine Fehler mehr enthalten sind.

2.1.1 Interne Organisation der Variablen und Strukturen

Wurde sichergestellt, daß in dem zu compilierenden Programm keine Syntax-Fehler usw. enthalten sind, kann mit der Übersetzung des Programms begonnen werden. Meist besteht ein C-Programm aber nicht nur aus Befehlen, sondern auch aus Variablen-Definitionen und -Deklarationen.

Bevor wir uns den eigentlichen C-Befehlen widmen, betrachten wir deshalb, wie Variablen und Strukturen "übersetzt" werden:

Erst einmal muß festgestellt werden, welchen Typs die einzelnen Variablen sind. Die Sprache C kennt folgende "skalare" Typen:

int, long char, short, unsigned char, unsigned int, unsigned long, float, double und den Zeiger.

Zusammengesetzte Typen sind das Array (oder der Vektor), die Struktur, der Aufzählungstyp, die Bitfelder und die Union.

Der Compiler codiert jeden dieser Typen und speichert den Variablennamen und dessen Typ-Code in der sogenannten (globalen) Symboltabelle ab. Wir wollen an dieser Stelle darauf hinweisen, daß die folgenden Symboltabellen die eines Modell-C-Compilers und nicht die des Aztec-Compilers sind. Dennoch arbeitet der Aztec-Compiler grundsätzlich nach dem gleichen Prinzip beim Compilieren wie unser Modell-Compiler.

```
int Wert;
char Value;

Symboltabelle:
--------------
"Wert"/Code_für_Int/End_Code/"Value"/Code_für_Char/End_Code
```

Bei Strukturen sieht dies ähnlich aus. Auch hier folgt zuerst der Name der Struktur, auf den ein Code (ein einzelnes Byte) für den Beginn der Struktur folgt. Dann werden alle in der Struktur enthaltenen Variablen aufgelistet:

```
struct Nonsense
{
   int hallo;
   char na;
   unsigned char wie_gehts;
};

Symboltabelle:
--------------
"Nonsense"/Code_für_Struktur_typ/"hallo"/Code_für_Int/
"na"/Code_für_char/"wie_gehts"/Code_für_unsigned_char/End_Code
```

Dies ist eine Struktur-Deklaration. Die einzelnen Codes haben wir hier durch Texte angedeutet (z.B. Code_für_char anstatt ei-

nes nichtssagenden Bytewertes (z.B. 0x21)). Mit ihr wird ein Struktur-Typ festgelegt. Erst nach einer Struktur-Definition (struct Nonsense Nonsense_Struktur) kann mit der Struktur gearbeitet werden. Sie haben also sozusagen einen Struktur-Typ geschaffen. Dies ist ja auch mittels typedef möglich (typedef struct {...} Struktur_Typ_Name). Mit typedef lassen sich aber auch bekannte Typen zu anderen zusammenfassen:

```
"typedef unsigned char BYTE"

Symboltabelle:
--------------
"BYTE"/Code_für_typ_beginn/
Code_für_unsigned/Code_für_char/End_Code
```

Das Include-File exec/types.h enthält eine Menge solcher neuer Typen, die von fast allen anderen Include-Files benutzt werden. Sie sind also unabdingbar für die Benutzung von Include-Files und müssen immer vor Verwendung anderer Include-Files eingebunden werden.

Doch zurück zu den Strukturen. Die Symboltabelle zu einer Struktur-Definition sieht wie folgt aus:

```
struct Nonsense Nonsense_Struktur;

Symboltabelle:
--------------
"Nonsense_Strukur"/Code_für_Strukturbeginn/
"Nonsense"/Code_für_Struktur_typ/End_Code
```

Man kann die Nonsense_Struktur direkt definieren. Die Struktur steht dann nicht mehr für weitere Definitionen zur Verfügung:

```
 struct
{
   int hallo;
   char na;
   unsigned char wie_gehts;
} Nonsense_Struktur;

Symboltabelle:
--------------
"Nonsense_Struktur"/Code_für_Strukturbeginn/"hallo"/Code_für_Int/
"na"/Code_für_char/"wie_gehts"/Code_für_unsigned_char/End_Code
```

Eine Definition der Form struct Nonsense_Struktur Name ist hier nicht mehr erlaubt. Bei der Deklaration von Bitfeld, Array, Aufzählungstyp und Union sieht es ähnlich wie bei der Struktur aus:

```
ARRAY:
------
int Augenzahl[6];

Symboltabelle:
--------------
"Augenzahl"/Code_für_Array/Code_für_Int/6/End_Code

AUFZÄHLUNGSTYP:
---------------
enum farbe = {rot, gruen, blau};

Symboltabelle:
--------------
"farbe"/Code_für_Enum/"rot"/0/"gruen"/1/"blau"/2/End_Code
```

Interessant beim Aufzählungstyp ist, daß die einzelnen Aufzählungstypen (rot, gruen, blau) im Programm durch Integer-Zahlen repräsentiert werden. farbe = 0 würde also dasselbe bedeuten wie farbe = rot.

Der Aufzählungstyp kann mit einer Flut von #define-Anweisungen verglichen werden. Statt enum farbe = {'rot', 'gruen', 'blau'}; könnte man schreiben:

```
#define rot 0
#define gruen 1
#define blau 2
...
int farbe;
```

Bei den Define-Anweisungen sollte man beachten, daß ihre Symboltabellen nur aus Strings bestehen, die an der Stelle, in denen ein mit #define definiertes Symbol auftritt, eingesetzt werden und so ohne große Probleme compiliert werden können.

```
Symboltabelle für Defines:
--------------------------
"rot"/Code_für_Define/"0"/End_Code/"gruen"/Code_für_Define/
/"1"/End_Code/"blau"/Code_für_Define/"2"/End_Code
```

Insbesondere bei mathematischen Ausdrücken, die mit Define-Anweisungen definiert werden, ist Vorsicht geboten:

```
#define Summe  a+b

...
printf ("%d  %d",Summe,Summe*10);
```

Der printf-Befehl gibt zunächst die Summe der beiden Variablen a+b aus. Sollte die zweite Ausgabe das Zehnfache der Summe darstellen, werden Sie hier wahrscheinlich enttäuscht. Der Compiler generiert aus Summe*10 nämlich a+b*10. Er addiert zum Wert von a den zehnfachen Wert von b. Um dies zu vermeiden, sollten Sie die Summe klammern:

```
#define Summe (a+b)
...
```

Im Zusammenhang mit der Define-Anweisung sind die #ifdef-, #ifndef- und #endif-Direktiven zu nennen. Diese treten sehr häufig in den Include-Files auf und dienen meist dazu, schon compilierte Include-Files nicht ein zweites oder drittes Mal zu compilieren, da dies zu Fehlern führen würde (redefinierte Symbole etc.). Da die meisten Include-Files auf anderen Include-Files aufbauen, werden innerhalb eines Include-Files meistens weitere Includes getätigt:

```
/* **** Include-File **** */

#ifndef EXEC_TYPES_H            /* Wenn EXEC_TYPES_H nicht   */
                                /* definiert,                */
#include "exec/types.h"         /* dann include exec/types.h */
#endif

#ifndef GFX_H                   /* Wenn GFX_H nicht definiert, */
#include "graphics/gfx.h"       /* dann include graphics/gfx.h */
#endif
```

Anhand der Symboltabelle kann der Compiler sehr einfach feststellen, ob ein Symbol definiert (#ifdef) oder nicht definiert (#ifndef) ist, und kann dann - je nachdem ob #ifdef oder #ifndef verlangt wurde - über das Schicksal der im #if...-endif-Block eingeschlossenen Statements entscheiden. Jedes Include-File definiert zu Beginn ein Symbol mit dem Namen Include-File_H.

Wenden wir uns nun den Bitfeldern zu. Diese sind ein neues Feature, das der Aztec erst seit der neuesten Version (3.4a) kennt. Da Bitfelder ziemlich unbekannt sind, wollen wir ihre Benutzung hier erläutern:

Wie der Name Bitfeld schon sagt, handelt es sich bei Bitfeldern um Datenorganisationen, die mit einzelnen Bits arbeiten. Deklariert werden sie ähnlich wie Strukturen:

```
struct Bitfeld
{
   unsigned 2:;              /* Bit 0 und 1 */
   unsigned 1: a;            /* Bit 2 */
   unsigned 1: b;            /* Bit 3 */
   unsigned 1: c;            /* Bit 4 */
   unsigned 3:;              /* Bit 5-7 */  };

struct Bitfeld bf;
```

Diese Deklaration erzeugt ein 8 Bit breites Bitfeld. Sie können nun auf Bit 2, 3 und 4 mittels bf.a, bf.b und bf.c zugreifen. Dabei können Sie an diese Variablen nur die Werte 0 und 1 zuweisen. Das ist ja auch nur logisch, da ein Bit nur die beiden Zustände 0 (aus/nein) und 1 (an/ja) kennt.

Auf die Bits 0, 1, 5, 6 und 7 können Sie nicht zugreifen, da diesen keine Namen gegeben wurden. Diese Bits wurden durch die Anweisungen unsigned 2:; beziehungsweise unsigned 3:; "übersprungen".

Die Symboltabelle zu obiger Bitfeld-Deklaration und -Definition sieht wie folgt aus:

```
Symboltabelle:
--------------

"Bitfeld"/Code_für_typ_Beginn/""/Code_für_Bit0/""/Code_für_Bit1/
"a"/Code_Bit2/"b"/Code_für_Bit3/"c"/Code_für_Bit4/""/
Code_Bit5/""/Code_für_Bit6/""/Code_für_Bit7/End_Code

Definition:
-----------
"bf"/Code_für_Strukturbeginn/Code_für_typ/"Bitfeld"/End_Code
```

Wenden wir uns nun aber dem letzten nicht skalaren Datentyp zu - der Union.

Sie bezeichnet eine Variable, die mehr als einem Datentyp angehören kann:

```
union zahl
{
    int i;
    float f;
};

union zahl z;

Symboltabelle:
--------------

"Zahl"/Code_für_Union_typ/"i"/Code_für_int/"f"/
Code_für_float/End_Code

Definition:
----------

"z"/Code_für_Union_typ/"zahl"/End_Code;
```

Die Variablen i und f belegen denselben Speicherplatz und können somit einmal in der einen, einmal in der anderen Art und Weise interpretiert werden.

Betrachten wir nun Zeiger auf die einzelnen Variablen, Strukturen und Arrays, so muß nur der Code für den Zeiger eingeführt werden:

```
int *Zeiger;
struct Nonsense *Sensible;

Symboltabelle:
--------------

"Zeiger"/Code_für_Zeiger/"Sensible"/Code_für_Int/EndCode/
"Sensible"/
Code_für_Zeiger/"Nonsense"/Code_für_Struktur_typ/EndCode
```

Die so aufgebaute Symboltabelle wird beim Übersetzungsvorgang immer dann benötigt, wenn mit einer Variablen gearbeitet werden soll. Will man z.B. die Variable i inkrementieren (i++), so muß der Compiler wissen, welchen Typs diese Variable ist.

Angenommen, i sei ein Int-Objekt, so weiß der Compiler, daß diese Variable 16 Bit breit ist und er nicht add.l #1,Adresse_von_i oder add.b #1,Adresse_von_i, sondern add.w #1,Adresse_von_i für den Assembler-Source generieren muß. Damit der Compiler den Symbol-Eintrag für eine Variable schnell findet, werden immer zuerst die Namen der Variablen und dann die Typen abgespeichert.

Ähnlich ist es mit der Zuweisung von Werten an Struktur-Elemente, z.B.: Sensible->hallo = 0. Zunächst muß der Compiler herausfinden, ob "Sensible" überhaupt ein Zeiger ist. Aus der Symboltabelle erfährt er: "Sensible"/Code_für_Zeiger und weiß, daß Sensible ein Zeiger ist. Nun muß er erfahren, auf was für ein Objekt Sensible zeigt. Zeigt Sensible nämlich auf eine Int-Variable, so ist die Zeile Sensible->hallo sinnlos, da Int-Variablen keine weiteren Elemente, wie die Strukturen, enthalten.

Aus der Symboltabelle erfährt der Compiler aber, daß Sensible auf eine Struktur zeigt. Der ->-Operator ist also auch erlaubt. Sensible->hallo kann übrigens auch durch (*Sensible).hallo ersetzt werden, wobei zu beachten ist, daß der Punkt (.) eine höhere Priorität als der Stern (*) vor Sensible hat, weswegen die Klammerung notwendig ist.

Würden Sie Sensible.hallo programmieren, so würde der Compiler das mit einer Fehlermeldung quittieren (falsche Pointer-Benutzung).

Nun muß der Compiler nur noch überprüfen, ob die Struktur, auf die Sensible zeigt, überhaupt ein Element namens "hallo" enthält. Auch dies erfährt er aus der Symboltabelle. Zunächst zeigt "Sensible" auf ein "Nonsense"-Objekt, und in der Symboltabelle für Nonsense läßt sich das Element hallo ausmachen. Nun braucht der Compiler nur noch zu überprüfen, ob die Zuweisung von 0 an die Variable hallo der Nonsense-Struktur erlaubt ist, und wenn ja, muß er bestimmen, wie diese Zuweisung aussehen soll. Nachdem der Compiler aus der Symboltabelle für Nonsense erfahren hat, daß hallo ein Int-Objekt ist, kann er clr.w Adresse_von_Sensible->hallo angeben.

Der Compiler überprüft während des Compilier-Vorgangs bei jedem Zugriff auf eine Variable, welchen Typs diese Variable ist. Wollen Sie z.B. eine Variable an eine andere Variable unterschiedlichen Typs zuweisen, generiert der Compiler beim Compilieren ein Warning, das Sie darauf hinweist, daß eine illegale Typzuweisung durchgeführt wurde.

Bei normalen Variablen wie int, char und long nimmt der Compiler Typzuweisungen untereinander ohne Cast nicht übel. Dies liegt daran, daß bei der Zuweisung von z.B. einem Int-Objekt an ein Char-Objekt nur die unteren acht Bits der Int-Variablen an die Char-Variablen übergeben werden bzw. die fehlenden Bits bei einer Zuweisung von char nach long mit Nullen aufgefüllt werden und eventuell eine Vorzeichenerweiterung ext.w bzw. ext.l durchgeführt wird.

Problematisch wird diese Typkonvertierung aber im Zusammenhang mit Zeigern:

```
int  *Eingabe;
char *String = 'Der Amiga ist spitze !!!';

String = Eingabe;
```

Der Compiler ist nicht sicher, ob sich der Programmierer im klaren darüber ist, daß er zwei Zeiger mit unterschiedlichen Typen einander zuweist. Deshalb gibt der Compiler bei diesem Befehl ein Warning aus, das den Benutzer auf eine evtl. "falsche" Zeigerzuweisung hinweist.

Es kann aber durchaus gewünscht sein, daß solch eine Zuweisung eines Int-Zeigers an einen Char-Zeiger geschieht, weil man vielleicht auf das höherwertige Byte der Int-Variablen, auf die der Int-Zeiger zeigt, zugreifen will. Durch einen Char-Zeiger ist das möglich, da Char-Objekte nur ein Byte breit sind und nach String++ der Zeiger String auf die nächste Byte-Adresse zeigt.

Will man als Programmierer, daß diese Zuweisung geschieht, sollte man den Compiler mit Hilfe eines Casts davon in Kenntnis setzen:

```
String = (char *)Eingabe;
```

Das obige Programmsegment weist neben dem Cast noch zwei weitere Besonderheiten auf: Erstens wird eine Variable bei der Definition initialisiert, und zweitens werden Strings verwendet.

Bei der automatischen Initialisierung kommt zum Tragen, wo die Variable definiert wurde. Wurde die Variable nämlich außerhalb einer Funktion definiert, so wird der Speicherplatz, den diese Variable belegt, mittels dc.<b/w/l> Wert festgelegt:

```
C:
--

char *String = "Der Amiga ist spitze !!!";

Maschinensprache:
-----------------

   dseg                                    ;Datensegment
   ds 0                                    ;0 Bytes reservieren
   public _String                          ;globle Variable
_String:                                   ;Speicher mit Adresse
   dc.l .1+0                               ;des Strings belegen
   cseg                                    ;Codesegment
.1    dc.b 68,101,114,32,65,77,73,71,65,32,105,115,116,32,115
   dc.b 112,105,116,122,101,32,33,33,33,0     ;ASCII-Codes
   ds 0                                    ;Null-Byte
   ...
   public _main
_main:                                     ;ab hier gehts los
```

Bei Initialisierungen innerhalb einer Funktion sieht das etwas anders aus:

```
C:
--

proc()
{
   char *String = "Der Amiga ist spitze !!!"
   ...
}


Maschinensprache:
-----------------

   public _proc                        ;Prozedur ist global
_proc:
   link a5,#.2                         ;Stack-Speicher belegen
```

```
   movem.l .8,-(sp)                    ;Register retten
   lea .1,a0                           ;Adresse des Strings -> a0
   move.l a0,-4(a5)                    ;a0 -> Variable _String
   ...
   movem.l (sp)+,.8
   unlk a5
   rts
.2 equ -4
.3 reg
.1    dc.b 68,101,114,32,65,77,73,71,65,32,105,115,116,32,115
   dc.b 112,105,116,122,101,32,33,33,33,0
   ds 0                                      ;Null-Byte
```

Wie Sie sehen, nehmen die Strings eine ungewohnte Form an. Sie können nun nicht mehr die einzelnen Buchstaben erkennen, sondern nur noch ihren ASCII-Code, in den sie vom Compiler Zeichen für Zeichen umgerechnet und "abgespeichert" werden. (Zu den häufig auftauchenden Assembler-Direktiven global, public, desg und cseg lesen Sie bitte das Kapitel 2.2.)

An dieser Stelle ist es vielleicht angebracht, Sie auf die Breite der einzelnen Variablen hinzuweisen:

Char-Objekte sind immer ein Byte breit und haben somit einen Wertebereich von -128 bis 127 oder 0-255, wenn man "unsigned char"-Variablen verwendet. Int-Variablen hingegen belegen schon zwei Bytes mit einem Wertebereich von -32768 bis 32767 bzw. 0-65535. Long-Objekte sind 4 Bytes breit mit einem Wertebereich von -268435458 bis 268435457 bzw. 0-4294967295. Float-Variablen enthalten 4 Byte und Double-Variablen 8 Byte.

Zeiger sind Variablen, die die Anfangsadresse einer Variablen oder Struktur enthalten. Aufgrund der Tatsache, daß der 68000er theoretisch 4 Gigabytes (ca. 4 Milliarden Bytes) adressieren kann, müssen Zeiger einen Wertebereich von 0 bis 4294967295 haben. Dies wird dadurch erreicht, daß Zeiger immer 4 Bytes belegen, egal welchen Typs sie sind. Wie ein Adreßregister, mit dem ja auch auf jede Speicherstelle zugegriffen werden kann, enthalten die Zeiger immer 4 Bytes.

Wenn Sie einen Zeiger auf ein Long-Objekt de- oder inkrementiereren wollen, sollten Sie beachten, daß zur alten Adresse vier Bytes hinzugezählt bzw. abgezogen werden. Der Zeiger zeigt

dann also auf das nächste Long-Objekt. Ebenso ist es mit Strukturen. Angenommen, eine Struktur enthält 95 Bytes. Inkrementieren Sie den Zeiger mittels <Zeiger>++ oder <Zeiger>=<Zeiger>+1, so wird die Adresse um 95 Bytes erhöht.

Wir sind jedoch die ganze Zeit davon ausgegangen, daß unser Programm bei der Variablendeklaration und -definition nur Variablen enthält, die außerhalb von Funktionen definiert bzw. deklariert wurden. Nun kann man aber auch Variablen innerhalb einer Prozedur definieren, die nur für diese Prozedur bekannt sind. Dazu folgendes Programm:

```
int i;
main () {...}
prroc()
{
   int i;
   ...
   i = ...;
}
```

Wie Sie sehen, taucht die Variable i zweimal auf. Wird in der Prozedur Proc in irgendeiner Programmzeile auf i zugegriffen, so weiß der Compiler, daß das eigens in dieser Routine definierte i gemeint ist. Um dies zu wissen, benötigt er aber eine zweite Symboltabelle. Würde der Compiler den Symbol-Code für die Variable i auch in die globale Symboltabelle schreiben, so hätte er keine Möglichkeit, zwischen dem globalen i, das vor main() definiert wurde, und dem i in proc() zu unterscheiden. Welches von beiden sollte er verwenden?

Die zweite - lokale - Symboltabelle wird deshalb immer dann benutzt, wenn innerhalb einer Funktion Variablen deklariert und benutzt werden. Ist die Funktion aber übersetzt, so steht der gesamte Speicherplatz der lokalen Symboltabelle der nächsten Funktion zur Verfügung. Wird mit einem Befehl der Routine auf eine Variable zugegriffen, wird zunächst die lokale Symboltabelle durchsucht. Konnte die Variable dort nicht gefunden werden, so durchsucht der Compiler zusätzlich die globale Symboltabelle. Besonders im Zusammenhang mit der Benutzung der Librarys ist die Definition der Library-Base-Pointer innerhalb oder außerhalb einer Funktion besonders wichtig. Definieren Sie

z.B. den Zeiger auf die Intuition-Library (struct IntuitionBase *IntuitionBase) innerhalb einer Funktion (z.B. in main()), so können Sie die Funktionen von Intuition nicht benutzen.

Diese benötigen die Variable _IntuitionBase, die aber nur durch die Assembler-Direktive global _IntuitionBase,4 für andere Module und Linker-Librarys zugänglich gemacht werden kann. Wird IntuitionBase innerhalb einer Prozedur definiert, so wird nur Stack-Speicher für diesen Zeiger belegt, auf den nicht mit dem Label _IntuitionBase zugegriffen werden kann.

Aber zurück zur lokalen Symboltabelle. Natürlich benötigt auch sie Speicherplätze. Diese werden ohne Angabe Ihrerseits vom Compiler auf 1040 Bytes festgelegt. Doch meldet sich der Compiler einmal mit der Fehlermeldung: "Local table full (Use -L)", so sollten Sie dem Rat des Compilers folgen und mit der Option -L den Speicher für die lokale Symboltabelle erhöhen. Dabei sollten Sie beachten, daß ein Eintrag 26 Bytes groß ist. Mit -L40 werden 1040 Bytes reserviert. 1040 Bytes werden vom Compiler auch ohne Angabe der Option -L reserviert.

Die globale Symboltabelle kann übrigens auch abgespeichert werden - die lokale nicht. Dies ist besonders dann von Nutzen, wenn Sie Programme schreiben, die sehr viele Include-Files enthalten.

Den Include-File-Block, der nur die Include-Anweisungen enthält, können Sie separat compilieren und die so unter Verwendung der Option +H erzeugte Symboltabelle abspeichern (cc +HCompiled_Includes Include_Block.h), wobei der Inlude-Block z.B. so aussieht:

```
#include "exec/types.h"
#include "graphics/gfx.h"
...
#include "graphics/gels.h"
```

Danach können Sie dann diese Symboltabelle, die auf Diskette im File "Compiled_Includes" vorhanden ist, mit der Option +I wieder einlesen (cc +ICompiled_Includes Programm.c). Sie sparen so erstens die Zeit zum Einlesen der einzelnen Include-Files

und zweitens die Zeit, die für den Aufbau Ihrer Symboltabelle in diesem Fall benötigt wird.

2.1.2 Software-Organisation der Variablen und Strukturen

Die Inhalte der Variablen und Strukturen müssen in Speicherstellen festgehalten werden. Den dazu benötigten Speicherplatz fordern die Prozeduren vom System-Stack an, wenn keine anderen Speicherklassen (extern, static, auto, register) für die Variablen definiert wurden. Dazu wird vom Compiler mit Hilfe der Symboltabelle zunächst die Anzahl der Bytes berechnet, die den einzelnen Prozeduren für Variablen zur Verfügung gestellt werden müssen.

Da der Compiler aus globaler und lokaler Symboltabelle weiß, welche Variablen verwendet werden, und er außerdem genau über die Breite der einzelnen Variablen unterrichtet ist, ist diese Berechnung für ihn ein Kinderspiel. Auch bei der Verwendung der sizeof()-Anweisung, mit der man die Anzahl der Bytes erhält, die eine Struktur oder Variable belegt, greift der Compiler sehr intensiv auf die Symboltabelle - in ähnlicher Weise wie bei der Variablenspeicher-Zuweisung - zurück.

Es kann aber zu ernsthaften Problemen kommen, wenn die Variablen mehr Speicherplatz benötigen als, der Stack zur Verfügung stellt. Doch Sie können mit der +m-Option das sogenannte Stack-Checking einschalten. Vor jedem Einsprung in eine Routine wird die Routine _stkchk#() aufgerufen, die bei einem Stack-Overflow (Stack-Überlauf) die Routine _stkover() anspringt. Bei einem Stack-Overflow verursacht _stkover() sofort einen Programmabbruch, und manchmal wird - zur Freude des Programmierers - auch ein Reset-Requester (Finish all disk activities, ...) angezeigt.

Im Handbuch zum Aztec steht geschrieben, daß _stkover() den Benutzer freundlich auf den Stack-Overflow hinweist (mit der Nachricht "Stack overflow!!"), aber leider funktioniert das nicht immer reibungslos. Manchmal tritt der Guru in Aktion. Aber Sie können eine eigene _stkover()-Routine schreiben. Allerdings

sollte diese Routine in Maschinenspache geschrieben werden, denn in dieser können Sie einen Alternativ-Stack benutzen, wobei zu beachten ist, daß das Adreßregister a7, das ja zugleich der Stackpointer ist, auf die höchste Adresse des Alternativ-Stack-Speicherbereichs zeigt, da ein Stack nach unten "wächst".

Beim Linken dieser neuen _stkover()-Routine sollten Sie weiterhin beachten, daß eine Routine gleichen Namens schon in der c.lib enthalten ist. Der Linker hat also zwei Maschinensprache-Routinen mit dem Namen ___stkover: zur Verfügung - erstens Ihre und zweitens die aus der c.lib (Dem Namen der C-Prozedur wird in Maschinensprache ein Unterstreicher vorangestellt!). Der Linker ist aber normalerweise (ohne Benutzung einer Linker-Option) so fair, Ihrer Routine den Vortritt zu lassen, da diese für den Linker eher als die der c.lib auftaucht.

Sie sollten die Stack-Überprüfung sowieso nur in der Entwicklungsphase benutzen. Die Programme werden durch den immer wiederkehrenden Aufruf von _stkchk#() beim Eintritt in jede Funktion oder Routine immer langsamer und größer.

Doch kommen wir nun dazu, wie sich die Anlage von Variablen im Assembler-Source tatsächlich niederschlägt. Dazu folgendes kleine C-Programm:

```
int c;
main()
{
   int j;
   j = 0;
   ...;
}

proc()
{
   int i;
   c = 0;
   ...;
}
```

Der Einfachheit halber verwenden wir hier Int-Variablen. Doch im Prinzip funktioniert die Allokation bzw. Reservierung des Speicherplatzes für Strukturen, Arrays etc. genauso.

Betrachten wir nun den vom Compiler erzeugten Assembler-Source:

```
;:ts = 8                    ;Sinnlos!?!
   global _c,2              ;weist Assembler an, zwei Bytes für
                            ;_c zu reservieren. Ähnlich wie
                            ;"_c: ds.b 2"
   public _main             ;_main ist für alle Module bekannt
                            ;"genauso wie _c"
_main:
   link     a5,#.2          ;a5 nach -(sp), sp nach a5, sp + #.2
   movem.l  .3,-(sp)        ;Register auf Stack (keine Register
                             ;Liste angegeben!) Variablen-Stack
   clr.w -2(a5)              ;Zugriff auf Variable j
   ...
.4
   movem.l (sp)+,.3         ;gerette Registerinhalte vom Stack
                            ;zurück an Register
   unlnk   a5               ;a5 nach sp,-(sp) nach a5
   rts

.2 equ -2                   ;2 Bytes für Int-Objekt
.3 reg                      ;zu rettende Register (Liste leer)
   public _proc             ;proc ist für alle Module bekannt.
   link a5,#.9              ;Variablen-Stack anlegen
   movem.l .10,-(sp)        ;Register retten
   clr.w _c                 ;c = 0
   ...
.11
   movem.l (sp)+,.10
   unlnk a5
   rts
.9 equ -2                   ;2 Bytes für i
.10 reg                     ;keine zu rettenden Register
   public .begin
   dseg                     ;Beginn des Datensegmentes
   end
```

Der Variablenspeicher für Variablen aus Prozeduren wird also vom Stack geholt, während der Speicher für globale Variablen, die außerhalb von Funktionen definiert wurden, "normalen" Speicherplatz belegen. Bei Strukturen und Arrays sieht diese Deklaration ähnlich aus - nur mit mehr zu reservierenden Bytes. Bei Unions hingegen wird nicht für jede in der Union enthaltene Variable Speicher reserviert, sondern nur soviel, wie die "größte" Variable der Union benötigt. Beim Zugriff auf ein Union-Element wird auf diesen Speicherbereich zugegriffen:

```
C:
--

union zahl
{
   int i;
   float f;
}

union zahl z;

main()
{
   z.f = 1.1;
   ...
   z.i = 10;
   ...
}

Maschinensprache:
-----------------

   global _z,4                  ;Bytes für Union reservieren
                                ;(global)
   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
   move.l #$8ccccd41,_z         ;z.f = 1.1 (float representation);
   ...
   move.w #10,_z                ;z.i = 0
   ...
   movem.l -(sp),.3
   unlk a5
   rts
.2 equ 0                        ;keine lokalen Variablen
.3 reg
   public .begin
   dseg
   end
```

Beim Zugriff auf ein Struktur- oder Array-Element muß neben der Anfangsadresse der Struktur bzw. des Arrays auch noch der Offset des angesprochenen Elements berechnet werden:

```
C:
--

main()
{
struct Struktur
   {
      int Element1;
      long Element2;
      float Element3;
   }

struct Struktur s;
struct Struktur t;

   s.Element1 = 0;
   ...
   s.Element2 = s.Element3;
   ...
   s=t;
}

Maschinensprache:
-----------------

   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
   clr.w  -10(a5)                    ;s.Element1 = 0
   ...
   move.l -4(a5),d0                  ;s.Element3 -> d0
   jsr Ffix#                         ;float nach long konvertieren
   move.l do,-8(a5)                  ;d0 -> s.Element2
   ...
   lea -10(a5),a0                    ;&s nach a0
   lea -20(a5),a1                    ;&t nach a1
   move.l (a1)+,(a0)+                ;Bytes umschaufeln
   move.l (a1)+,(a0)+
   move.w (a1)+,(a0)+
.4
   movem.l (sp)+,.3
   unlk a5
   rts
.2 equ -20
.3 reg
   public .begin
   dseg
   end
```

Wie Sie sehen, benutzt dieses Programm Variablen des Typs "float". Das Programm muß deshalb mit der Option +fi compi-

liert werden. Bei der Verwendung von double-Variablen muß die Option +ff benutzt werden. Bei der Benutzung des 68881-Mathe-Coprozessors müssen Sie Ihre Programme mit der +f8-Option compilieren. Da in allen drei Fällen Mathe-Routinen benötigt werden, die in der c.lib nicht enthalten sind, muß die m.lib, ma.lib bzw. mx.lib neben der c.lib zum Objekt-File gelinkt werden.

Wir wollen uns aber nun den schon oben angesprochenen Speicherklassen zuwenden: Mit der Speicherklasse "extern" sagen Sie dem Compiler, daß die Variable in einem anderen Modul zu finden ist. An dieser Stelle muß der Compiler also keinen Speicherplatz für diese Variable deklarieren, da dies in einem anderen Modul geschehen muß. Geschieht dies allerdings nicht, so meldet sich der Linker zu Wort, der eine "unresolved reference" - einen ungelösten Zugriff - erkennt.

Benutzen Sie die extern-Anweisung bei der Variablendefinition bzw. -deklaration innerhalb einer Prozedur, so wird die globale Symboltabelle nach der geforderten Variablen durchsucht. Wenn Sie solch eine globale Variable benutzen wollen, sollten Sie immer die Speicherklasse "extern" bei der Deklaration angeben. So wissen Sie immer, welche Variablen in der Prozedur benutzt werden; welche lokal und welche global sind:

```
int i;
proc()
{
  extern int i;          /* Zugriff aus globale i (Deklaration) */
         int j;          /* lokales (auto) j (Definition)       */
  ...
}
```

Die Speicherklasse "static" gibt dem Compiler Aufschluß über die Art des Speichers, die für eine Variable verwendet werden soll. Normalerweise sind alle Variablen einer Prozedur "auto". Das heißt, daß der Speicherplatz bei Eintritt in die Funktion allokiert wird (vom Stack) und beim Austritt aus der Funktion wieder freigegeben wird. Legen Sie nun fest, daß eine Variable "static" ist, so wird der Speicherplatz für die Variable vom normalen Programmspeicher genommen. Beim Austritt aus einer Prozedur bleibt der Wert der Variablen erhalten, so daß Sie bei

einem eventuellen Wiedereintritt mit dem alten Variablenwert rechnen können. Für die Zuweisung des static-Speicherplatzes wird die bss-Direktive des Assemblers verwendet. Diese funktioniert ähnlich wie die global-Anweisung, nur daß der Speicherplatz für die Routine, in der die static-Variable definiert wurde, gültig ist. Dies wird dadurch erreicht, daß auf diese Variable nicht mit ihrem Namen - wie bei global _c,2 - sondern mit einem ganz normalen Label (z.B. .4) zugegriffen wird.

Insbesondere in rekursiven Prozeduren machen sich static-Variablen nützlich. Dort können sie z.B. als Zähler dienen, oder zur schnellen Weitergabe von Parametern, ohne daß diese über den Stack ausgetauscht werden müssen (siehe 2.1.4).

Bei der Speicherklasse "register" wird ein Register zur Speicherung eines Variablenwertes herangezogen. Grundsätzlich kann man jeden skalaren Variablentyp als register-Variable definieren. Doch sollte man beachten, daß bei Double-Variablen, die die Speicherklasse "register" haben, notwendigerweise einige Bytes abgeschnitten werden müssen. Double-Variablen sind nämlich 8 Bytes breit, aber die Adreß- und Datenregister des 68000er können nur 4 Bytes aufnehmen.

Die Register D0, D1, D2, D3, A2 und A3 können vom Compiler zum Speichern von Werten benutzt werden.

2.1.3 Die Übersetzung der Steueranweisungen

Widmen wir uns nun den Steueranweisungen. Sie bestimmen, in welcher Reihenfolge verschiedene Routinen und Befehle abgearbeitet werden sollen.

Nachdem nämlich die Fehleruntersuchung keine (offensichtlichen) Fehler zutage gebracht hat und alle Variablen und Strukturen untersucht und angelegt wurden, startet der eigentliche Compilier-Vorgang, der die einzelnen C-Befehle und Prozeduraufrufe in Maschinensprache übersetzt. Zunächst wollen wir die Kontrollstrukturen (For, While, Switch, If) "übersetzen":

```
for (i=0;i<5;i++) j += 2;
```

Übersetzt sieht diese Programmsequenz so aus:

```
        clr.l -4(a5)            * i=0 (Initialisierung von
                                *      i)
.1:     add.l #2,-8(a5)         * j += 2 (Schleifenrumpf)

        add.l #1,-4(a5)         * i++    (Inkrementierung)
        cmp #5,-4(a5)           * i<5    (Abbruchbedingung)
        blt .1                  * ja, dann zurück zum Anfang
```

Zunächst stößt der Compiler auf das "Wörtchen" "for". Dies sagt ihm, daß er es mit einem Schleifenbefehl zu tun hat. Doch im Gegensatz zu While wird bei For die Abbruchbedingung erst dann überprüft, wenn der Schleifenrumpf ausgeführt wurde. Bei While wird hingegen die Abbruchbedingung vor dem Eintritt in den Schleifenrumpf überprüft.

Hat der Compiler diesen Befehl erkannt, wird die Initialisierung der Variablen i untersucht. Der Compiler stellt fest, daß die Variable i auf 0 gesetzt werden soll. Dies schlägt sich in dem Maschinensprachebefehl clr.l -4(a5) nieder. Es wird angenommen, daß der Speicherplatz der Variablen i in -4(a5) reserviert worden ist (Stack-Speicherplätze). Die Variable j befindet sich in -8(a5).

Nach der Initialisierung der Variablen i muß sich der Compiler nun um den Schleifenrumpf, also um den Befehl j += 2 kümmern. Die Abbruchbedingung und die Inkrementierung der Variablen i bleiben noch außer acht. Der Compiler geht nun zum Befehl j += 2 über. Da dies der erste (und einzige) Befehl ist, der im Schleifenrumpf enthalten ist, muß vor diesen Befehl ein Label, eine Sprungmarkierung, gesetzt werden, damit er wiederholt angesprungen werden kann. Nach dem Setzen der Sprungmarkierung wird der eigentliche Befehl übersetzt (add.l #2,-8(a5)). Nun muß der Compiler sich der Inkrementierung der Variablen i widmen (add.l #1,-4(5)) und die Abbruchbedingung untersuchen (cmp.l #5,-4(a5); blt .1).

Hier können Sie schon erkennen, daß die Linearität der C-Befehle in Maschinensprache erhalten bleibt - nur in umgekehrter Richtung.

Der Schleifenrumpf wird vor der Inkrementierung von i und diese vor der Abbruchbedingung übersetzt. Dies läßt auf einen rekursiven Algorithmus schließen. Trifft der Compiler auf die Initialisierung der Variablen i, befindet er sich auf Rekursionsstufe 1. Da die Initialisierung der Variablen i vor den Schleifendurchläufen geschehen sein muß, wird diese sofort übersetzt (clr.l -4(a5)). Dann gelangt der Compiler zur Abbruchbedingung auf Rekursionsstufe 2. Nach der Abbruchbedingung gelangt er zur Inkrementierung von i auf Stufe 3, und danach auf Stufe 4 zum Schleifenrumpf.

Bis jetzt wurde aber noch nichts übersetzt. Der Compiler ist nur bis zum Rumpf, der ja in der Schleife mehrmals ausgeführt werden soll, herabgestiegen. Da nach dem Rumpf nichts mehr folgt, wird zunächst dieser vollständig übersetzt. Danach werden die Inkrementierung (Ebene 3) und die Abbruchbedingung (Ebene 2) übersetzt.

Natürlich ist der Rumpf einer solchen Schleife nicht immer so unkompliziert wie in diesem Beispiel. Enthält der Rumpf z.B. weitere For-Schleifen, so geht das ganze Spiel der Rekursion von vorne los. Aber gerade diese Rekursivität beim Compilieren macht C zu einer der vielseitigsten, aber auch schwer zu beherrschenden Programmiersprachen. Man kann z.B. in die Abbruchbedingung Funktionsaufrufe einbetten.

Einfache Programme kann man schon nach wenigen Stunden Einarbeitungszeit schreiben. Aber Programme eines anderen Autors zu lesen, das erfordert sehr viel mehr als nur ein paar Stunden Einarbeitungszeit. Haben Sie jedoch einmal begriffen, wie ein Compiler sich durch komplizierte und verschachtelte Programme durcharbeitet, dann kann dabei eigentlich nichts mehr schiefgehen.

Zurück zu den Kontrollstrukturen. So ähnlich wie die For-Schleife wird auch die While-Schleife übersetzt. Allerdings arbeitet der Compiler hierbei nicht so "rekursivintensiv" wie bei For. Bei While wird die Bedingung, die direkt nach dem Schlüsselwort While angegeben wird, sofort übersetzt. Bevor der Schleifenrumpf ausgeführt wird, wird die Abbruchbedingung

getestet. Der Übersetzer merkt sich nur, daß er noch ein Label an das Ende des While-Rumpfes setzen muß, um bei Nichterfüllung der While-Bedingung den Schleifenrumpf überspringen zu können:

```
C:
--

  i = 0;                    /* Initialisierung muß selbst */
                            /* vorgenommen werden         */
  while (i<5)               /* While-Schleife             */
     {
       j += 2;              /* Schleifenrumpf              */
       i ++;
     }

Maschinensprache:
-----------------

        clr.l -4(a5)         * i = 0;

.1:     cmp #5,-4(a5)        * i<5
        bge .2               * nein
        add.l #2,-8(a5)      * { j += 2;
        add.l #1,-4(a5)      *   i ++; }
        bra .1:              * zurück zum Anfang der Schleife
.2:     ...
```

Der While-Befehl kann also in einem Durchgang, ohne Rekursion, übersetzt werden. Somit können Sie so viele Whiles ineinanderschachteln wie Sie wollen. For-Schleifen können Sie jedoch nicht beliebig schachteln. Der Compiler stellt Ihnen eine Verschachtelungstiefe von 20 For-Schleifen zur Verfügung. Sie können aber mittels der Option -e die "Expressiontable" - die Ausdruckstabelle - vergrößern. Ein Eintrag reserviert 14 Bytes, so daß bei "-e80" 1120 Bytes für die Expressiontable reserviert werden. Diese 1120 sind übrigens der Default-Wert, den der Compiler ohne Änderung Ihrerseits zur Verfügung stellt. Wollen Sie die Expressiontable aber vergrößern, so müssen Sie mehr als 80 Einträge belegen.

Allerdings existiert noch eine zweite Art der While-Schleife:

```
C:
--
i = 0;
do {
     j+=2;
     i++;
   } while (i<5);
```

Bei dieser While-Schleife wird der Rumpf wenigstens einmal ausgeführt, und erst nach der Ausführung wird die Schleifenbedingung überprüft. Auch hier geht die Übersetzung wieder linear vor sich. Der Compiler muß nur an die Zeile, in der das "do" steht, ein Label setzen und dieses bei zutreffender Schleifenbedingung anspringen.

Eine weitere Kontrollstruktur ist die Switch-Anweisung. Mit ihrer Hilfe können Sie ja bekanntlich eine aus mehreren Alternativen wählen. Eine typische Switch-Anweisung sieht z.B. so aus:

```
switch (i)
  {
     case 3:
        i++;
     break;

     case 4:
        i--;
     break;

     default:
       i=0;
     break;
  }
```

Der Compiler macht daraus:

```
     move.l -4(a5),d0          * Inhalt von i nach d0
     bra .4

.6:  add.l #1,-4(a5)           * case 3: i++;
     bra .5                    * break;

.7:  sub.l #1,-4(a5)           * case 4: i--;
     bra .5                    * break;

.8:  clr.l -4(a5)              * default: i=0;
     bra .5
```

```
.4:  sub.l #3,d0
     beq .6
     sub.l #1,d0
     beq .7
     bra .8
.5:  ...
```

Zunächst überträgt der Compiler den Wert der Variablen i in das Datenregister d0. Weiter kümmert er sich zunächst nicht um die Variable i. Im weiteren Verlauf wird jeder Case-Break-Rumpf übersetzt, indem jeweils vor den ersten Befehl eines solchen Rumpfes ein Label gesetzt wird und die einzelnen Anweisungen übersetzt werden. Am Schluß eines jeden Rumpfes steht eine Verzweigung an die Stelle im Programm, die nach der Switch-Anweisung ausgeführt werden soll.

Doch wie wird die Switch-Anweisung vom Compiler bearbeitet? Natürlich spielt auch hier die Rekursion eine große Rolle. Zunächst wird nämlich dafür gesorgt, daß die Variable i in das Datenregister d0 übertragen wird. Auch die Verzweigung nach .4 (bra .4) wird programmiert.

Danach wird eine Rekursionsstufe herabgeschritten, nicht ohne im Hinterkopf zu behalten, daß von allen Case-Break-Rümpfen nach .5 gesprungen werden muß. In der ersten Rekursionsstufe werden alle Rümpfe übersetzt. Dann wird diese Rekursionsstufe beendet, und der Compiler befindet sich wieder am Anfang der Switch-Anweisung. Diese wird jetzt aber nicht wie die For- und While-Anweisungen überlesen, sondern auf die einzelnen Fälle untersucht.

Liest der Compiler jetzt "case 3", so wird von d0 der Wert 3 abgezogen und auf 0 getestet, und eventuell wird verzweigt. Beim zweiten Case wird nach dem Wert 4 gefragt. Damit nicht noch einmal der Wert der Variablen i in das Datenregister geladen und dann der Wert 4 davon abgezogen werden muß, berechnet der Compiler die Differenz des ersten und zweiten Case (3-4). Ist diese Differenz negativ (wie hier), so wird der Betrag dieser Differenz von d0 abgezogen (sub.l #1,d0). Ist die Differenz positiv, wird der Betrag jedoch addiert (add.l #1,d0). Danach kann dann wieder getestet werden, ob das Datenregister den Wert 0 hat.

Fallen alle diese Case-Vergleiche negativ aus, wird automatisch der Default-Rumpf angesprungen (der nicht immer angegeben werden muß, dann wird aber einfach die Switch-Anweisung verlassen).

Aber natürlich sieht eine Switch-Anweisung nicht immer so geordnet aus wie die obige. Erstens können mehr als zwei Cases enthalten sein, und zweitens müssen diese nicht geordnet sein:

```
switch (i)
  {
     case 1:
        i++;
     break;
          case 0:
        i--;
     break;
     case 6:
        i=0;
     break;

     case 2:
        i=0;
    break;
  }
```

Abgesehen davon, daß dies nicht besonders elegant programmiert ist - der Compiler nimmt's nicht übel. Doch wie geht er hier vor? An folgendem Assembler-Code wollen wir dies verdeutlichen:

```
     move.l -4(a5),d0          * Inhalt von i nach d0
     bra .4

.6:  add.l #1,-4(a5)           * case 1: i++;
     bra .5                    * break;

.7:  sub.l #1,-4(a5)           * case 0: i--;
     bra .5                    * break;

.8:  sub.l #1,-4(a5)           * case 6: i--;
     bra .5                    * break;

.9:  sub.l #1,-4(a5)           * case 2: i--;
     bra .5                    * break;

.10: dc.w .7-.11-2             * case 0:     (0)
     dc.w .6-.11-2             * case 1:     (2)
     dc.w .9-.11-2             * case 2:     (4)
```

```
    dc.w .5-.11-2              * default:
    dc.w .5-.11-2              * default:
    dc.w .5-.11-2              * default:
    dc.w .6-.11-2              * case 6:      (12)
  .4:  cmp.l #7,d0
    bcc .5                     * default
    asl.l #1,d0                * d0 * 2
    move.w .10(pc,d0.w),d0
.11:
    jmp (pc,d0.w)
  .5:  ...
```

Wie Sie sehen, wird diese Switch-Anweisung ähnlich übersetzt wie die obige. Nur die Auswahl der einzelnen Fälle wurde hier etwas anders geregelt.

Zunächst werden die Case-Rümpfe in der Reihenfolge übersetzt, in der sie im C-Programm auftauchten. Erst in der Sprungtabelle, die die Adressen (relativ zum PC) aller Case-Rümpfe enthält, werden sie in eine Reihenfolge gebracht. Die Offsets werden dann in der Reihenfolge der Cases, auf die sie "zeigen", sortiert und abgespeichert.

Sie können unter normalen Umständen maximal 100 Case-Bedingungen abfragen. Sollten Sie einmal mehr Fälle abfragen wollen - was allerdings sehr unwahrscheinlich scheint - so können Sie mit Hilfe der Option -Y Speicherplätze für weitere Case-Bedingungen anfordern. Mit -Y100 würden 100 Einträge für die Case-Bedingungen reserviert. Ein Eintrag besteht dabei aus 4 Bytes, wobei 2 Bytes für den Offset und die anderen beiden Bytes für die Integer-Zahl, die bei Case getestet werden soll, benötigt werden.

Bei der Auswahl einer Case-Bedingung wird der Wert der Variablen i (enthalten in d0) verdoppelt (linksshift) und als Index auf diese Sprungtabelle verwendet. Da diese Tabelle nur aus Offsets besteht, die in WORDs angegeben werden, muß i nur verdoppelt und nicht vervierfacht werden, wie es z.B. nötig gewesen wäre, wenn die Tabelle aus den absoluten Adressen der Case-Rümpfe bestünde.

Da einige Fälle nicht abgefragt werden (case 3:, case 4:, case 5:), wird die Sprungtabelle an diesen Stellen mit der Adresse des

Befehls, der nach der Switch-Anweisung ausgeführt werden soll (Label .5), aufgefüllt.

Liegen die Fälle, die getestet werden sollen, allerdings sehr weit auseinander, so werden diese nicht durch eine Sprungtabelle ausgewählt. Dies wäre ziemlich Speicher-uneffizient, da unter Umständen mehrere Male die Default-Adressen (zum Label .5) abgespeichert werden müßten. Deshalb greift man hier wieder auf die erste Methode zurück, überträgt die zu testende Variable in das Datenregister d0 und subtrahiert bzw. addiert die verschiedenen CASE-Werte und testet d0 auf 0.

Doch kommen wir nun zur einfachsten Kontrollstruktur: Die If-Else-Anweisungen. Diese können so linear übersetzt werden, wie sie in C programmiert wurden:

```
if (i == 1) i = 0;
else
   i++;
```

In Maschinensprache sieht das so aus:

```
     cmp.l #1,-4(a5)       * i == 1  ?
     bne .4                * nein
     clr.l -4(a5)          * ja
     bra .5                * weiter
.4:  add.l #1,-4(a5)       * else i++;
.5:  ...
```

Das einzige Problem, das sich bei der Übersetzung der If-Anweisung stellt, ist das Label. Der Compiler merkt sich jedoch, welche Label-Nummer er z.B. einem Branch-Befehl angeben muß und welches Label vor einen Befehl geschrieben werden muß.

Ein weiteres Problem stellt sich jedoch bei der Auswertung mathematischer Terme. Wie geht der Compiler z.B. mit "i = (j*5)+20" um? Auch hier spielt die Rekursion wieder eine große Rolle. Der Compiler arbeitet sich auf die unterste Klammerebene vor, rechnet die Klammer aus und benutzt das Ergebnis für die Berechnung der nächsthöheren Klammerebene.

Dies jetzt im einzelnen zu beschreiben, würde an dieser Stelle zu weit führen. Es sei jedoch soviel gesagt: sind Sie sich nicht sicher, welche Operation Priorität vor einer anderen hat, so sollten Sie lieber ein paar Klammern mehr setzen. Dadurch stellen Sie die Portabilität Ihres Programms sicher, auch wenn ein anderer Compiler vielleicht mathematische Ausdrücke etwas anders übersetzt.

Eine weitere Steueranweisung, die allerdings nicht den Gefallen der Prozedural-Freunde findet, ist die Goto-Anweisung:

```
goto Label;
...
...
Label: ...
```

Dabei wird einfach ein bra- oder jmp-Befehl für das Assembler-Source generiert. Allerdings ist zu beachten, daß das Label, das angesprungen werden soll, in derselben Prozedur steht wie der Goto-Befehl.

2.1.4 Der Aufruf von Prozeduren

Neben den Steueranweisungen kann man natürlich auch eigene Prozeduren schreiben oder schon vorhandene benutzen. Dabei ist zu beachten, daß es zwei Arten von Prozeduren gibt:

1. Funktionen, die einen Rückgabewert liefern.
2. Routinen, die keinen Rückgabewert liefern.

Beiden gemeinsam ist, daß ihnen Parameter übergeben werden können. Dabei ist zu beachten, daß die Parameter über den Stack übergeben werden, z.B:

```
func(i,j,k)
int  i;
long   j;
struct Nonsense *k;
{
   int l;
   l=0;    func (i,k,k)
}
```

In Maschinensprache sieht diese - zugegebenermaßen etwas zwecklose - rekursive Routine so aus:

```
_func:
   link a5.#.2              ;Speicher für lokale Variablen
   movem.l .3-(sp)          ;Register auf Stack
   move.l 14(a5),-(sp)      ;Parameter k für erneuten
                            ; Aufruf auf Stack
   move.l 10(a5),-(sp)      ;j auf Stack
   move.l  8(a5),-(sp)      ;i auf Stack

   jsr _func                ;Routine aufrufen
      lea 10(sp),sp         ;Stack restaurieren
      movem.l (sp)+,.3      ;Register wieder laden
   unlnk a5                 ;Lokalspeicher freigeben
   rts

.2 equ -2                   ;zwei Bytes für Int-Objekt
.3 reg                      ;Registerliste
```

Dazu vielleicht eine kleine Abbildung, die noch einmal verdeutlicht, wie die lokalen Variablen auf dem Stack abgelegt werden:

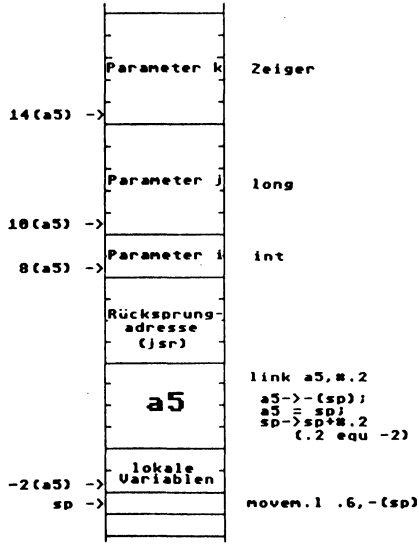


Abbildung 2.1: Stack

Sie sehen, daß mit positiver Indexierung von a5 die Eingabe-Parameter angesprochen werden (z.B. 8(a5)), während mit negativer Indexierung (-2(a5)) die lokalen Variablen, die auf dem Stack abgelegt werden, angesprochen werden.

Zu beachten ist, daß beim Zugriff auf die Eingabe-Parameter der alte Registerwert von a5, der durch den LINK-Befehl auf dem Stack abgelegt wird, und die Rücksprungadresse, die bei jedem JSR abgespeichert wird, übersprungen werden müssen. Sonst ist nur noch zu beachten, daß die Eingabe-Parameter in umgekehrter Reihenfolge auf dem Stack abgelegt werden.

Ein nicht zu unterschätzendes Problem stellt die Tatsache dar, daß der Compiler nicht überprüft, ob die richtige Anzahl von Parametern und die richtigen Typen an eine Routine übergeben wurden.

Angenommen, eine Routine erwartet 3 Long-Variablen als Eingabe-Parameter, aber Sie übergeben dieser Routine nur 3 int-Werte. Die drei int-Variablen belegen nur 6 Stack-Bytes, während in der Routine auf 12 Bytes (3*4 (long)) zugegriffen wird.

Wenn Sie nun einen dieser Parameter verändern (im oberen Beispiel z.B. k=0 <=> clr.l 14(a5)), was in C durchaus möglich (und erlaubt?!?) ist, kann es passieren, daß Sie z.B. eine Rücksprungadresse verändern, weil bei der Parameterübergabe nur 6 statt der geforderten 12 Stack-Bytes belegt wurden.

Doch wenden wir uns den Funktionen zu. Auch hier werden die Eingabe-Parameter auf dem Stack abgelegt. Dies geschieht genauso wie bei den Routinen. Der einzige Unterschied zwischen Routinen und Funktionen besteht darin, daß Funktionen einen Rückgabewert liefern. Der Typ des Rückgabewertes muß von Ihnen festgelegt worden sein, denn Sie können mit dem Rückgabewert genauso weiterrechnen wie z.B. mit Variablen und Konstanten.

Der Typ des Rückgabewertes wird dabei bekanntlich vor dem Funktionsnamen angegeben. Allerdings gibt es bei den Funktionen auch den Typ "void", der besagt, daß diese Funktion keinen

Rückgabewert besitzt, also gar keine Funktion ist. Wird der Typ einer Funktion nicht von Ihnen festgelegt, so bestimmt der Compiler, daß diese Routine einen Rückgabewert vom Typ "int" hat. Nur Prozeduren mit dem Rückgabewert "void" sind also wahre Routinen.

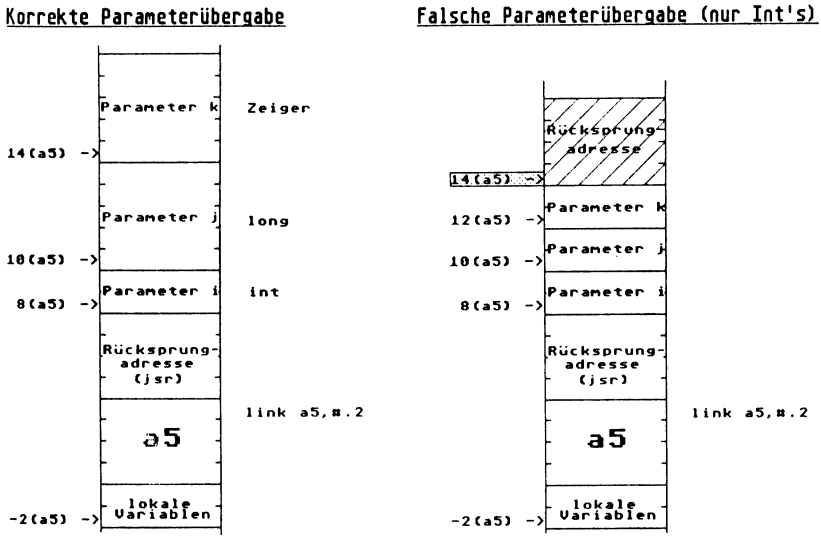


Abbildung 2.2: Stack und Parameterübergabe

Der Typ des Rückgabewertes einer Funktion wird genauso wie ein Variablentyp in der Symboltabelle abgespeichert:

```
void Funktion (...,...,...)

Symboltabelle:
--------------
"Funktion"/Code_für_Prozedur/Code_für_Void
```

Wenn sich eine Funktion in einem anderen Modul oder in einer Bibliothek befindet, kann man den Typ einer solchen "externen" Funktion ähnlich festlegen wie den Typ einer "externen" Variablen:

```
extern struct *MsgPort FindPort();

Symboltabelle:
--------------
"FindPort"/Code_für_externe_Prozedur/Code_für_Struktur_typ/
"MsgPort"/
```

Sie können davon ausgehen, daß der Compiler nicht "meckert", wenn Sie den Rückgabewert von FindPort(), der ein Zeiger auf eine MsgPort-Struktur ist, an einen Zeiger des Typs MsgPort übergeben. Grundsätzlich spielen bei Zeigerzuweisungen die Objekte der beteiligten Zeiger keine Rolle. Zeigervariablen sind immer 4 Bytes lang, so daß bei Zuweisungen keinerlei Konvertierungen vorgenommen werden müssen. Es gibt allerdings eine zweite Art von Zeigern - die BCPL- oder CPTR-Zeiger. Diese sind ein Überbleibsel der Sprache BCPL. Ihr Inhalt wird nur um 2 Bits nach rechts geshiftet, so daß diese Zeiger nur auf Langwort-Adressen zugreifen können. BCPL-Zeiger kommen allerdings nur als Eingabe-Parameter vor und das nur sehr selten, z.B. bei einigen I/O-Routinen.

Funktionen in anderen Modulen können Sie mittels "extern"-Deklarationen auch andere Rückgabewerte zuweisen. Angenommen, die Routine atol() wird in Modul2 definiert und gibt eigentlich ein Lang-Wort zurück. Definieren Sie nun in Modul1 mittels der extern-Deklaration, daß diese Routine einen Rückgabewert vom Typ "int" haben soll (extern int atol()), kann sich der Compiler nicht beschweren, da ihm die Symboltabelle für das zweite Modul nicht zur Verfügung steht. Allerdings sollten Sie beachten, daß diese Deklaration vor Benutzung der Routine vorgenommen werden muß. Auch wenn Sie Ihre eigenen Funktionen schreiben, die nicht vom Typ "void" sind, müssen Sie darauf achten, daß diese vor Benutzung definiert (oder deklariert) werden, damit sie zur weiteren Benutzung in die Symboltabelle aufgenommen werden können.

Bei Funktionen spielen die Typen der Eingabe-Parameter allerdings keine Rolle. Es gibt aber Compiler, die auch die Eingabe-Parameter auf "Typenreinheit" überprüfen, so daß Fehler wie der oben beschriebene nicht auftreten können. Der Aztec-Compiler jedoch gehört nicht zu diesen.

Vielleicht noch ein Wort dazu, wie der Rückgabewert Maschinensprache-technisch behandelt wird. Dieser wird im Datenregister d0 mittels der return()-Anweisung übergeben (auch Zeiger). Zu beachten ist hierbei, daß der Funktionstyp und der Typ der Variablen, die mit return() zurückgegeben wird, übereinstimmen.

2.2 Der Assembler

Nachdem der Compiler das Assembler-Source erzeugt hat, kann der Assembler (Aufruf: as [>Ausgabe-File] [Optionen] Programm.asm) mit seiner Arbeit beginnen. Er muß das Objekt-File aus dem Assembler-Source erzeugen.

Das Objekt-File besteht dabei soweit wie möglich aus ausführbarem Maschinen-Code. Nur die unaufgelösten externen Referenzen müssen noch mit Hilfe des Linkers gelöst werden.

Betrachten wir dazu folgendes Assembler-Source:

```
   dseg                    ;Datensegment
   ds 0                    ;0 Bytes reservieren
   public _Str             ;globle Variable
_Str:
   dc.l .1+0            ;Speicher mit Adresse des Strings belegen
   cseg                    ;Codesegment
   dseg                       ;Datensegment
.1    dc.b 72,97,108108,111,0   ;"Hallo"
   cseg                       ;Codesegment
   global _i,2
   public _main
_main:                        ;ab hier geht's los
   link     a5,#.3
   movem.l  .4,-(sp)
   add.l 1,_str    ;str++;
   jsr _exit
.5
   movem.l (sp)+,.4
   unlk   a5
   rts

.3 equ 0
.4 reg

public _exit
public .begin
```

```
dseg
end
```

Aus diesem Assembler-Source erzeugt der Assembler nun relokatiblen Code. Relokatibel bedeutet, daß das Programm an jeder beliebigen Stelle im Computer lauffähig ist. Es ist also nicht auf eine bestimmte Anfangsadresse festgelegt.

Jedoch ist der Assembler-Source gar nicht so angelegt, daß es relokatiblen Code erzeugen würde. Dieser relokatible Code benutzt nämlich die Adressierung "Programmzähler relativ mit 16-Bit-Adreßdistanzwert" (z.B. jmp $10(pc)). Das Assembler-Source enthält aber nur Befehle mit absoluter Adressierungsart (z.B. jsr _exit).

Beim Assemblieren aber werden diese absoluten Befehle in relative umgewandelt. Dabei spielen allerdings die beiden Direktiven "public" und "global" eine Rolle. Diese sorgen nämlich dafür, daß das angegegebene Label in die sogenannte Label-Tabelle aufgenommen wird, in der alle globalen Label des Moduls stehen.

In einem ersten Durchgang untersucht der Assembler das Programm auf solche Label. Gleichzeitig untersucht er, ob ein Label der Label-Tabelle eine Entsprechung im Assembler-File findet. Dies ist z.B. bei main der Fall. Zunächst wird dieses Label in die Label-Tabelle aufgenommen (public _main) und dann definiert (_main:). global _i,2 sorgt dafür, daß das Label _i in die Label-Tabelle aufgenommen wird. Der Speicherplatz für diese Variable wird aber erst vom Linker zur Verfügung gestellt. (Die Direktive bss reserviert auch Speicher für eine Variable. Nur wird das Label, das mit diesem Speicherplatz assoziiert wird, nicht in die Label-Tabelle aufgenommen. Somit eignet sich bss sehr gut für die Anlage von Static-Variablen innerhalb von Funktionen.)

Greift nun ein Befehl auf eine mit einem Label spezifizierte Speicherstelle zu, so wird anhand der Label-Tabelle untersucht, ob dieses Label im gleichen Modul definiert wurde. Ist dies der Fall, kann der Adreßdistanzwert von der augenblicklichen Position zum Label berechnet und für den Befehl eingesetzt werden.

Bei der Berechnung des Adreßdistanzwertes ist es allerdings wichtig zu wissen, ob das Label eine Speicherstelle im Code-Segment oder eine Speicherstelle im Datensegment beschreibt. Aus dem ersten Durchgang des Assemblierens weiß der Assembler, wie viele Bytes das Programm enthalten wird. Greift man mit einem Befehl dann auf ein Label im Datensegment zu, so muß beachtet werden, daß das Datensegment an das Code-Segment angehängt wird.

Code- und Datensegment sind zwei getrennte Bereiche. Steht vor einem Befehl des Assembler-Sources die Direktive dseg, so werden alle nachfolgenden Daten in das Datensegment geschrieben, bis ein cseg folgt, das den Beginn des Code-Segments beschreibt.

Im folgendem Listing können Sie genau sehen, daß Code- und Datensegment zwei getrennte Bereiche sind. Immer, wenn die Direktive dseg oder cseg auftaucht, ändert sich die Adresse in der zweiten Spalte (der Code-Segment- bzw. Daten-Segment-Programmzähler).

```
Aztec 68000 Assembler 3.4a 1-25-87
     1 0000:                        dseg
     2 0000:                        ds 0
     3 0000:                        public _str
     4 0000:                    str:
     5 0000:  xxxx xxxx             dc.l .1+0
     6 0004:                        cseg
     7 0000:                        dseg
     8 0004:                    .1
     9 0004: 4861 6c6c 6f00         dc.b  72,97,108,108,111,0
    10 000a:                        cseg
    11 0000:                        global _i,2
    12 0000:                        public _main
    13 0000:                    _main:
    14 0000: 4e55 0000              link a5,#.3
    15 0004:                        movem.l .4,-(sp)
    16 0004: 52ad xxxx              add.l #1,_str
    17 0008: 4eba xxxx              jsr _exit
    18 000c:                    .5
    19 000c:                        movem.l (sp)+,.4
    20 000c: 4e5d                   unlk a5
    21 000e: 4e75                   rts
    22 0010: 0000 0000          .3 equ 0
    23 0010: 0000               .4 reg
    24 0010:                        public _exit
    25 000c:                        public .begin
```

```
26 000c:                        dseg
27 000a:                        end
```

Dieses Listing ist das mit der Option -l erzeugte Listing des obigen Assembler-Files. Hier können Sie ganz genau erkennen, daß der Assembler die absolut adressierenden Befehle in relativ adressierende Befehle umwandelt.

Der Maschinen-Code (Hex-Code) für "jmp _exit" wäre $4eb9. Im Listing (Zeile 17) steht für "jmp _exit" aber der Hex-Code $4eba. Dieser steht für jsr XXXX(pc). Der Offset zur Routine _exit kann noch nicht bestimmt werden, da diese Routine nicht in diesem Modul definiert ist.

Deshalb setzt der Compiler für diesen Offset einen Querverweis ein, der auf das Label _exit der Label-Tabelle zeigt, die im Objekt-File mit abgespeichert wird.

Dieses Listing wird im ersten Pass des Assemblers erzeugt. Dies ist daran zu erkennen, daß in Zeile 5 lauter x stehen. Dies geschieht deshalb, weil im ersten Pass noch nicht bekannt ist, an welcher "Speicherstelle" der String im Datensegment steht.

Ähnlich ist es in Zeile 16 (add.l #1,_str). Da der Beginn des Datensegments noch nicht feststeht, kann der Offset zur Variablen _str noch nicht berechnet werden.

Das Objekt-File, das der Assembler erzeugt, besteht nun aus dem soweit wie möglich ausführbaren Maschinen-Code, einem Header, der die Größe des Programms, die Anzahl der Labels in der mit abgespeicherten Label-Tabelle etc. angibt, und aus der Label-Tabelle:

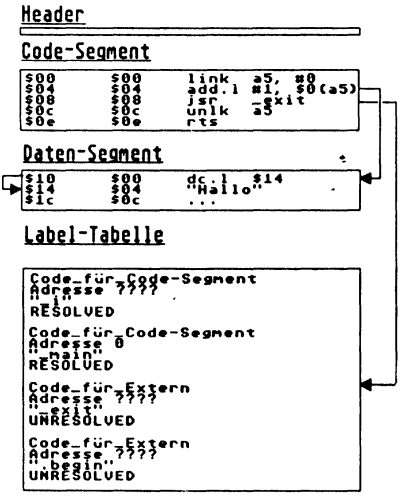


Abbildung 2.3: Label

Doch betrachten wir noch einmal das Listing und den Assembler-Source. Beim Vergleich fällt auf, daß der Assembler die Befehle movem.l .4,-(sp) (in Zeile 15) und movem.l -(sp),.4 (in Zeile 19) nicht übersetzt.

Diese Befehle hat der Assembler "wegoptimiert", da sie offensichtlich sinnlos sind. Die Befehle haben nur dann eine Aufgabe, wenn Register zu retten sind. Dies ist aber höchstens dann der Fall, wenn Register-Variablen benutzt werden.

Diese Optimierung ist allerdings nicht die einzige, die der Compiler durchführt. So werden z.B. Verzweigungen zum nächsten Befehl eliminiert:

```
    bra .7
.7 ...
```

Diese Konstruktion ist ja auch sinnlos, da der Befehl beim Label .7 sowieso, auch ohne den Branch-Befehl, ausgeführt wird. Weiterhin werden alle jsr-Befehle soweit wie möglich durch bsr-

Befehle ersetzt. Diese sind schneller in der Ausführung und sorgen somit für ein kleines Speed-Up Ihres Programms.

Jedoch ist die Option -n nicht die einzige Assembler-Option:

Assembler-Optionen

-OFilename

Der Objekt-Code wird in das hier angegebene File geschrieben.

-IDirectory

Diese Option legt wie beim Compiler fest, welches Unterverzeichnis nach Include-Files durchsucht werden soll.

-L

Diese Option veranlaßt den Assembler, ein Listing des Assembler-Sources zu erzeugen. Das Listing wird im ersten Pass des Assemblers erzeugt, in das File Filename.lst geschrieben und kann mit dem Editor ED untersucht werden.

-N

Der Objekt-Code wird nicht optimiert.

-SNummer

Es wird eine Squeeze-Tabelle mit *Nummer* Einträgen erzeugt. Ein Eintrag ist ein Byte groß. Ohne Angabe der Größe der Tabelle werden vom Assembler 1000 Bytes (Einträge) reserviert. Die Squeeze-Tabelle wird für die Optimierung des Objekt-Codes benötigt.

-V

Es wird eine Speicherverbrauchs-Statistik ausgegeben (Verbose-Option).

-ZAP

Das angegebene Assembler-Source-File wird nach dem Assemblieren gelöscht.

-C
Es wird "large code" erzeugt (siehe Kapitel 2.3).

-D
Es wird "large data" erzeugt.

-EName[=Wert]
Dem Symbol *Name* wird der angegebene Wert zugewiesen. Diese Zuweisung beim Assembler-Aufruf kommt einem "*Name* EQU *Wert*" gleich. Der Wert kann optional angegeben werden. Wird er nicht explizit angegeben, bekommt das Symbol den Wert 1.

2.3 Der Linker

Die Aufgabe (Aufruf: ln [>Ausgabe-File] Objektfiles.o [Optionen]) des Linkers besteht nun darin, die unaufgelösten Referenzen der einzelnen Module zu lösen. Dazu werden die Objekt-Files (Module) zumeist mit den Linker-Librarys gelinkt.

Betrachten wir dazu noch einmal das letzte Programmbeispiel. Dort wurde die Routine _exit aufgerufen. Diese Routine wurde aber nicht in dem Modul definiert, in dem sie aufgerufen wurde. Die Definition von _exit erfolgt in der Linker-Library c.lib. Diese muß also mit dem Objekt-File, das den _exit-Aufruf enthält, gelinkt werden.

Dabei liest der Linker zunächst das Objekt-File ein. Dann untersucht der Linker die Label-Tabelle des Objekt-Files. Aus der Label-Tabelle erfährt er, daß _exit eine unaufgelöste externe Referenz ist. Dies vermerkt der Linker in einer weiteren Tabelle, in der alle ungelösten Referenzen aufgelistet werden. Wird nun in einem Modul eine dieser unaufgelösten Referenzen gelöst, wird dies in der Tabelle der unaufgelösten Label vermerkt.

Nun untersucht der Linker die aneinandergereihten Module. Bis jetzt sind nämlich z.B. die Befehle, die eine externe Routine aufrufen, noch nicht vollständig. Zwar existiert z.B. schon der Hex-Code für den PC-relativen JSR-Befehl, aber der Offset der

Routine, die angesprungen werden soll, ist noch nicht bekannt. Diese Offsets können aber in einem zweiten Durchgang errechnet und an die jeweiligen Stellen eingetragen werden.

Betrachten wir dazu das zweite Listing in diesem Kapitel. In Zeile 17 erkennen Sie den Befehl jsr _exit. Der Assembler hat für diesen Befehl den Hex-Code $4eba xxxx generiert. jsr _exit belegt eigentlich 4 Bytes. Der Assembler benutzt aber zunächst nur 2 Bytes - nämlich die, die für die Codierung des PC-relativen JSR's verwendet werden ($4eba). Die restlichen zwei Bytes, die die Adreßdistanz für diesen Befehl angeben, bleiben zunächst uninitialisiert. Diese Initialisierung übernimmt der Linker, der im zweiten Durchgang alle benötigten Offsets errechnen und in die jeweiligen Stellen eintragen kann.

Kommen wir nun aber zu einer Frage, die Sie sich sicher auch schon gestellt haben: Werden beim Linken eines Objekt-Files mit einer Linker-Library alle Funktionen der Linker-Library oder nur die vom Objekt-File tatsächlich angesprochenen Funktionen eingebunden? Hier geht der Aztec-Linker sehr ökonomisch vor. Er hängt an Ihr Programm nur die tatsächlich verwendeten Routinen an, während er beim Linken zweier Objekt-Files alle Funktionen zusammenbindet.

Doch auch hier stellt sich eine entscheidende Frage: Wenn man nämlich dem Linker sowohl durch ln OFile.o -lram:c als auch durch ln OFile.o ram:c.lib dazu veranlassen kann, ein Modul bzw. Objekt-File mit der c.lib zu linken, und die c.lib nur ein Objekt-File ist, das sehr viele Funktionen enthält, wieso werden dann an das OFile nur die Routinen der c.lib angehängt, die tatsächlich benötigt werden, und nicht die gesamte c.lib?

Hierzu läßt sich sagen, daß die Linker-Librarys keine Objekt-Files im Sinne der Files sind, die vom Assembler erzeugt werden. Die Linker-Librarys bestehen nämlich aus einem Zusammenschluß mehrerer normaler Objekt-Files. Allerdings sind diese Librarys in einer etwas anderen Art und Weise geordnet. Während nämlich in herkömmlichen Objekt-Files die Label-Tabelle an das Ende des Files gestellt wurde, steht diese bei den

Linker-Librarys am Anfang. Dann erst folgt der ausführbare Maschinen-Code.

Soll der Linker nun eine Library zu einem Objekt-File linken, was wohl fast immer der Fall sein dürfte, so erkennt er an einem speziellen Code zu Anfang der Library, daß er eine Library und nicht ein normales Objekt-File untersucht. Der Linker untersucht nun also die Label-Tabelle der Library. Mit Hilfe dieser kann er sofort die ungelösten externen Referenzen der Objekt-Files lösen. Er merkt sich dabei auch, welche Routinen der Library benutzt wurden.

Wenn der Linker die Label-Tabelle der Library durchgelesen hat, kommt er zum ausführbaren Code der Library. Er "greift" sich diejenigen Funktionen heraus, die er sich oben gemerkt hat, und hängt sie an die normalen Objekt-Files an. Im zweiten Durchgang kann er dann alle benötigten Offsets berechnen und einsetzen.

Doch ist der Linker nicht nur für die Lösung der ungelösten externen Referenzen zuständig. Er sorgt auch dafür, daß der sogenannte Startup-Code zu den Objekt-Files gelinkt wird. Dies wird durch die Maschinensprache-Anweisung public .begin bewirkt. .begin wird im ersten Durchgang in die Liste der unaufgelösten Referenzen aufgenommen, und im Verlauf - beim Linken der c.Lib - an das Programm angehängt.

Weiterhin wird für jedes Modul bzw. Objekt-File, das die Anweisung public .begin enthält, der Maschinensprache-Befehl jmp .begin erzeugt (natürlich in PC-relativer Form). Dies ist allerdings nur für das erste der zu linkenden Objekt-Files notwendig. Dieser Befehl muß der erste des ganzen Programms sein. In den anderen Modulen nimmt dieser Befehl aber nur Speicherplätze (jeweils 4 Bytes) weg. Deshalb sollten Sie bei der Verwendung mehrerer Module dafür sorgen, daß nur beim ersten Modul das Label .begin "publiziert" wird. Alle anderen Module sollten Sie mit der +b-Option compilieren.

Doch kommen wir dazu, welche Aufgabe der Startup-Code hat. Zunächst einmal werden von diesem Startup-Code die DOS- und

die MathFFP-Library geöffnet. Dann hat der Startup-Code zu unterscheiden, ob das Programm von der Workbench aus oder vom CLI gestartet wurde.

Beim Start von Workbench muß zunächst auf die Startup-Message gewartet werden. Werden mit dieser Message Parameter übergeben, so werden diese als Lock-Struktur interpretiert und dazu benutzt, das gelockte Directory als das aktuelle Directory zu kennzeichnen. Dann wird, falls erforderlich, das Tooltype-Window geöffnet und zur Standard-Ein-/Ausgabe deklariert.

Beim Start vom CLI aus wird nur dafür gesorgt, daß der Benutzer die Anzahl der Parameter und die Adresse auf das Parameter-Array erhält. Diese werden vom DOS über den CIS (Command Input String) zur Verfügung gestellt. Der Startup-Code für das CLI muß die Parameter nur noch so aufbereiten, daß sie durch main (argc, argv) übergeben werden können.

Nach der Aufbereitung des Parameter-Strings wird die Routine main des C-Programms wie eine Prozedur aufgerufen. Irgendwo im Startup-Code, der teilweise in Maschinensprache und teilweise in C verfaßt wurde, taucht der C-Befehl main(argc, argv); auf. Direkt danach aber steht der C-Aufruf _exit(0);. Dieses _exit() sorgt dafür, daß die DOS- und MathFFP-Library geschlossen werden, der in Zusammenhang mit der Aufbereitung der Kommando-Parameter belegte Speicherplatz freigegeben, und der Befehl exit() aufgerufen wird. Dieses exit() ist genau die Routine, mit der Sie Ihr Programm zum frühzeitigen Abbruch bewegen können.

Doch befassen wir uns wieder mit den Linker-Librarys. Da der Startup-Code immer zu einem C-Programm gelinkt werden muß, muß die c.lib immer ein Bestandteil Ihres Programms sein. Es gibt aber noch andere Librarys. Zum Beispiel die m.lib. Diese ist immer dann zum Objekt-File zu linken, wenn in diesem mit der Fließkomma-Arithmetik gearbeitet wird. Allerdings reicht es nicht, bei der Verwendung der Fließkomma-Routinen einfach diese Linker-Library zum Objekt-File zu linken. Weiterhin müssen Sie das Programm mittels der Compiler-Option +ff compilieren. Um die doppelt genaue Fließkomma-Arithmetik zu be-

nutzen, müssen Sie das Programm mit der Option +fi compilieren und mit der mx.lib linken. Bei der Verwendung des 68881-Floating-Point-Prozessors müssen Sie die m8.lib benutzen und das Programm mittels +f8 compilieren.

Aber auch die m.lib und c.lib sind nicht die einzigen Librarys des Compiler-Pakets. Haben Sie sich schon einmal auf den Disketten umgesehen, so finden Sie z.B. dort auch eine c32.lib, eine cl.lib und eine cl32.lib. Auch die anderen Librarys treten in diesen Formen auf: ml.lib, m32l.lib etc.

Die Linker-Librarys, die ein l in ihrem Namen haben, müssen zu den Objekt-Files gelinkt werden, wenn Sie das "large data"- und "large code"-Speichermodell verwenden wollen:

2.3.1 Das "large data"- und "small data"-Modell

Kommen wir zunächst zu den Unterschieden zwischen "large data" und "small data". Wie der Name schon sagt, beziehen sich diese beiden Speichermodelle auf das Datensegment. Beim "small data"-Modell kann das Datensegment nur 64 KByte groß sein. Dies liegt daran, daß hier der Zugriff auf die Daten über ein Adreßregister geschieht, das auf die Mitte des Datenbereichs zeigt, und durch negative und positive Offsets zwischen 0 und 32768 nur 64 KByte "abgreifen" kann. Hier wird die "PC-relativ mit 16-Bit Adreßdistanzwert"-Adressierung benutzt.

Beim "large data"-Modell hingegen wird nicht mit Offsets gearbeitet, wenn auf eine Speicherstelle im Datensegment zugegriffen werden soll. Hier wird über die absoluten Adressen auf die Daten zugegriffen. Da aber zunächst nicht bekannt ist, an welcher Speicherstelle das Programm beginnt, müssen nach dem Laden des Programms alle Befehle, die auf absolute Speicherstellen des Datensegments zugreifen, korrigiert werden. Dies ist der Grund, warum es länger dauert, bis ein "large data"-Programm anläuft. Auch ist der Zugriff auf absolute Speicherstellen langsamer als der PC-relative Zugriff auf Speicherstellen. Der Vorteil beim "large data"-Modell hingegen ist der, daß das Daten-

segment nicht auf eine Größe von 64 KByte beschränkt ist, sondern theoretisch den gesamten freien Speicher belegen kann.

2.3.2 "large code" und "small code"

Die Modelle "large code" und "small code" befassen sich, wie der Name schon sagt, mit dem Code-Segment des Programms. Allerdings dreht es sich hier weniger um den ausführbaren Code als um die Daten, die auch im Code-Segment abgespeichert werden.

Beim "large Code"-Modell dürfen Daten im gesamten Code-Segment liegen, unter Umständen sogar über den gesamten Speicher verstreut. Das "small code"-Modell hingegen erlaubt nur, daß die Daten in einer Umgebung von 32 KByte um den Programm-Counter angelegt sind.

Ist das "small code"-Programm allerdings größer, so wird für die Speicherbereiche außerhalb dieser 32 KByte eine Adreßtabelle mit absoluten Adressen angelegt, über die indirekt auf die außerhalb liegenden Speicherbereiche zugegriffen wird.

Auch bei Verzweigungen außerhalb dieser 32 KByte wird mit einer Sprungtabelle gearbeitet. Doch nimmt die Korrektur dieser Sprung- und Adreßtabelle meist weniger Zeit in Anspruch als die Korrektur der Befehle eines "large code"-Programms, das ja ausschließlich absolute Adressierungen benutzt. Allerdings muß beim "small code"-Modell ein Adreßregister abgestellt werden, das in die Mitte des Daten-Segmentes reicht, während "large code" absolut auf das Datensegment zugreift. Grundsätzlich ist die absolute 32-Bit-Adressierung auch langsamer als die 16-Bit-Adreßdistanzwert-Adressierung.

Wollen Sie entweder "large code" oder "large data" verwenden, müssen Sie das Programm mit der Option +C bzw. +D compilieren und assemblieren. Beim Linken müssen Sie dann in jedem der beiden Fälle eine der Librarys mit einem 1 im Namen hinzu linken (z.B. cl.lib oder cl32.lib).

Jetzt fragen Sie sich sicher, was es mit der 32 im Namen der c32.lib bzw. cl32.lib auf sich hat. Diese 32 steht für 32 Bit Integer. Die in diesen Librarys verwendeten Int-Variablen sind nicht 16, sondern 32 Bit breit.

Dies ist insbesondere im Zusammenhang mit der Kompatibilität von Lattice- und Aztec-Programmen wichtig. Lattice-Programme übergeben nämlich keine 16-Bit-Int-Variablen. Variablen und Konstanten werden vom Lattice-Compiler auf 32-Bit-Größe erweitert. Diese Erweiterung wird beim Aztec-Compiler nicht mehr durchgeführt, so daß bei Verwendung einer Int-Variable als Funktionsparameter tatsächlich nur 16 anstatt 32 Bit übergeben werden. Die Library-Routine verlangt aber 32 Bit und kann so unter Umständen Parameter falsch interpretieren oder gar Symstemabstürze veranlassen.

Die ungenaue Lattice-Programmierung versucht man beim Aztec-Compiler nun dadurch zu kompensieren, daß man alle Int-Variablen und alle Konstanten auf 32 Bit erweitert. Dies wird durch die +L-Compiler-Option erreicht.

Die C-Support Funktionen der c32.lib, cl32.lib etc. wurden nun mit dieser +L-Option compiliert und können als Fehlerquelle bei Lattice-Aztec-Konvertierungen ausgeschlossen werden.

Hier nun eine Liste aller Linker-Optionen:

-OFilename

Das ausführbare Programm wird unter dem hier angegebenen Namen abgespeichert. Wird diese Option nicht verwendet, erhält das ausführbare Programm den Namen des Objekt-Files ohne die Extension .o.

-LFilename

Mit dieser Option legen Sie die Linker-Librarys fest, die zum Objekt-File des Programms gelinkt werden sollen.

-FFilename

Der Linker liest die Argumente aus dem hier angegebenen ASCII-File. Dies ist nützlich, wenn Sie dem Linker sehr viele Parameter übergeben wollen.

-T

Diese Option veranlaßt den Linker, die Linker-eigene Symboltabelle als ASCII-File abzuspeichern. Der Name dieses ASCII-Files wird durch den Namen des Objekt-Files mit auf .sym geänderter Extension gebildet. Dieses ASCII-File enthält Informationen über das File - in welchem Segment der Code steht und in welchen Hunks (=Abschnitten) ausführbare Befehle, Systemvariablen und Programmvariablen enthalten sind. Die Hunks werden durch die Label ___Hx_org (org = origin) und ___hx_end eingeschlossen, wobei für x die jeweilige Hunk-Nummer eingetragen wird. In Programmen, die nur aus einem Modul bestehen, werden die Programmvariablen zwischen ___H2_org und ___H2_end abgespeichert. Das Programm beginnt bei ___H0_org.

-W

Diese Option veranlaßt den Linker, Informationen für den Debugger db abzuspeichern. Diese Informationen sind die Namen der Label sowie die Offsets zum Programmbeginn, an der die Label stehen (siehe -T-Option).

-V

Es wird eine Hunk-Statistik ausgegeben.

+O[i]

Diese Option veranlaßt den Linker, das nachfolgende Objektmodul in das nächste bzw. in das angegebene Code-Segment zu schreiben.

+C[cdb]

Diese Option veranlaßt den Linker, dafür zu sorgen, daß das Code-Segment (+Cc), das Data-Segment (+Cd) oder beide Segmente (+Cb) des ausführbaren Programms in den Chip-Memory geladen werden.

+F[cdb]

Diese Option veranlaßt den Linker, dafür zu sorgen, daß das Code-Segment (+Fc), das Data-Segment (+Cd) oder beide Segmente (+Fb) des ausführbaren Programms in den Fast-Memory geladen werden.

+A

Diese Option veranlaßt den Linker, jedes Objektmodul an einer Langwortadresse beginnen zu lassen.

-C, -Q

Diese Optionen veranlassen den Linker, Informationen für den Source-Level-Debugger abzuspeichern (Extension: .dbg).

+Q

Diese Option veranlaßt den Linker, nicht mehr die Prozeduren der Objekt-Files, die gerade gelinkt werden, auszugeben.

-M

Der Linker beschwert sich nun, wenn Sie in einem Ihrer Module eine Funktion einer Linker.Library definieren. Ohne -M überschreibt Ihre neue Prozedur die Linker-Library-Prozedur (siehe Stack-Overflow).

+L

Diese Option veranlaßt den Linker anzunehmen, daß die folgenden Objektmodule "wahre" Amiga-Linker-Librarys sind. Auch der Versuch, die c32.lib mit Hilfe dieser Option zu linken, brachte keinen Erfolg.

+s,
+ss,
+sss

Ohne Angabe von +s, +ss und +sss wird der gesamte Code des Programms in einen Hunk geschrieben. +s veranlaßt den Compiler, den ausführbaren Code eines jeden Objektmoduls und die in den Modulen benutzten Linker-Library-Routinen in separate Hunks zu schreiben. +ss bedeutet, daß der Code in einen Hunk

geschrieben wird, solange der Code die Größe von 8 KByte nicht übersteigt. Nach 8 KByte wird einer neuer Hunk benutzt. +sss schließlich veranlaßt den Linker, für jedes Modul jeweils einen Hunk zu verwenden. Dabei sollten Sie beachten, daß Linker-Librarys Zusammenschlüsse mehrerer Module sind und so jede benutzte Routine in einen Hunk geschrieben wird.

2.4 Das Debuggen von Programmen

Was tun, wenn trotz aller Vorsichtsmaßnahmen das Programm, das man mühevoll entwickelt hat, nicht so läuft, wie man es eigentlich wünscht oder sogar abstürzt?

Hier gibt es - grobgesagt - zwei Vorgehensweisen:

1. Man läßt sich mit Hilfe des printf()-Befehls Variablen-Inhalte, Strings usw. ausgeben, und kontrolliert diese Werte.
2. Man verwendet einen Debugger, mit dem man den Ablauf des Programms bestimmen und testen kann.

Meistens ist es jedoch so, daß man eine Symbiose beider Vorgehensweisen wählt, da entweder die Variablenüberprüfung mit Hilfe des Debuggers zu mühsam ist, oder andererseits bestimmte Zugriffe (vor allen Dingen Zugriffe mit Hilfe von Zeigern) nicht durch den printf()-Befehl sichtbar gemacht werden können.

Betrachten wir einmal die Vorgehensweise mit Hilfe des printf()-Befehls.

Angenommen, Ihr Programm besteht aus mehreren Funktionen. Um nun festzustellen, in welcher Funktion der Fehler ist, der zum Programmabsturz führt, können Sie als ersten Befehl immer printf(Name der Funktion\n); aufrufen. Meistens haben Sie die Möglichkeit, festzustellen, welche Routine zuletzt angesprungen wurde, die den Fehler erzeugt hat.

Auch innerhalb von Schleifen kann man gut mit der printf()-Methode arbeiten. Wenn man sich z.B. nicht sicher ist, wieviele

Array-Elemente in einer Schleife initialisiert werden, kann man die Elementnummern bzw. den Inhalt der Schleifenvariablen ausgeben lassen. Ebenso kann man bei String-Manipulationen vorgehen, wenn man nicht sicher ist, hinter welches Zeichen z.B. die String-Endemarke (der ASCII-Code 0 - ausgedrückt durch \000) gesetzt wird.

Manchmal braucht man aber gar nicht soweit zu gehen, jede einzelne Routine in dieser Form zu kennzeichnen. Nach ein paar Monaten Programmierpraxis kann man sich auf seine Eingebungen verlassen und an der vermuteten Stelle nach dem Fehler suchen. Oder wenn man sich beim Programmieren an einer Stelle nicht ganz sicher war, kann man eben diese Stelle einer genauen Untersuchung unterziehen.

Auch bestimmte Guru-Meditations können Ihre Vorahnungen bestätigen. Besonders die Trap-Codes der Guru-Meditation (siehe Tabelle 3.16) können wertvolle Hinweise geben:

Error-Code 00000003 weist Sie auf einen Adressierungsfehler hin. Das bedeutet, daß bei einem Wort- oder Langwortzugriff auf eine ungerade Adresse zugegriffen wurde. Das kann Sie veranlassen, Ihre WORD- und LONG-Pointer noch einmal zu überprüfen.

Der Fehlercode 00000004 (Illegaler Opcode) weist Sie darauf hin, daß bei der Benutzung von Librarys oder Devices ein Fehler beim Aufruf einer Funktion stattgefunden hat. Besonders bei den Devices, denen ein sogenannter Device-Block übergeben werden muß, kann solch ein Fehler auftreten, da in dem Device-Block die aufzurufenden Routinen enthalten sind. Tritt nun ein Fehler bei der Übergabe dieses Device-Blockes an eine Funktion auf, kann es sehr leicht passieren, daß eine Speicherstelle angesprungen wird, die gar keinen ausführbaren Code enthält.

Auch der Fehler 00000008 sollte Sie stutzig machen. Dieser Fehler tritt dann auf, wenn Maschinensprachebefehle ausgeführt werden sollen, die nur im Supervisor-Modus aufgerufen werden dürfen. Dieser Fehler kann insbesondere bei der Manipulation

von Tasks auftreten. Wenn man hier nicht sehr vorsichtig ist, kann es leicht zu solchen Systemabstürzen kommen.

Neben diesen hausgemachten Debugging-Tips stehen Ihnen aber vor allem beim Aztec-Compiler zwei sehr kraftvolle Debugger (Entwanzer) zu Verfügung:

2.4.1 Der Aztec-DB

Um den Aztec-DB in Verbindung mit Ihrem Programm benutzen zu können, sollten Sie das Programm mit Hilfe der Option -w linken. Diese Option sorgt dafür, daß die einzelnen Symbole des Programms (die Namen der globalen Variablen und aller Funktionen) auch abgespeichert werden. Dies erhöht die Lesbarkeit des Programms ungemein, denn der dargestellte Maschinencode wirkt doch sehr konfus, wenn man nicht ein paar Anhaltspunkte wie Variablen und Funktionen hat. Es ist zwar möglich, ein Programm, daß nicht mit der Option -w gelinkt oder gar vollkommen in Assembler geschrieben wurde, zu debuggen. Allerdings muß man hier sehr viel Geduld haben, sich viele Notizen machen, um nicht den Überblick zu verlieren.

Doch kommen wir dazu, wie man den Debugger DB verwendet. (Auch für Lattice-Benutzer kann es evt. recht interessant sein zu wissen, wie man den DB verwendet, da damit auch ihre Programme untersucht werden können. Allerdings muß man hier auf die Symbol-Tabellen verzichten.)

Zunächst einmal muß der DB aufgerufen werden. Dies geschieht einfach durch Eingabe von "DB" <Return> auf der CLI-Ebene. Der Debugger wird geladen, gestartet und ist sofort einsatzbereit. Besonders die Version 3.6 des Debuggers ist sehr komfortabel. Man hat hier nämlich die Möglichkeit, im File SYS:s/.dbinit die Befehlssequenz anzugeben, die direkt nach dem Aufruf des Debuggers ausgeführt werden soll. (Der DB befindet sich übrigens auf der Diskette SYS2: im Verzeichnis bin).

Normalerweise steht im File .dbinit der Befehl al. Dieser Befehl bedeutet: "Load Task". Es wird also auf das nächste Programm

gewartet, das gestartet wird. Um also Ihr Programm zu debuggen, brauchen Sie dieses nur wie gewohnt zu starten. Daraus erklärt sich auch, daß der Debugger sofort nach Eingabe des al-Kommandos zur CLI-Ebene zurückkehrt, bzw. das eigene Fenster ganz in den Hintergrund setzt.

Sie können nun im CLI-Fenster Ihr zu debuggendes Programm starten. Doch Vorsicht! Achten Sie darauf, daß das CLI-Fenster auch tatsächlich aktiviert ist. Ist dies nämlich nicht der Fall, werden alle Tastatureingaben an den Debugger gesendet, der diese zunächst als Wunsch zur Unterbrechung des al-Befehls und weiterhin als neuen Befehl interpretiert!

Doch betrachten wir, was nach dem Start des zu debuggenden Programms geschieht:

Zunächst erscheint wieder das Fenster des Debuggers im Vordergrund. In das Fenster wird die Meldung "Loading Syms..." geschrieben. Dies weist Sie darauf hin, daß versucht wird, das Programm sowie die durch die Linker-Option -w erzeugte Symboltabelle zu laden. Wenn solch eine Tabelle nicht geladen werden kann, weil sie für das zu debuggende Programm nicht erzeugt wurde, erscheint eine diesbezügliche Meldung.

Doch betrachten wir, was passiert, wenn Programm samt Symboltabelle geladen wurde. Zunächst wird der erste Befehl des Programms dargestellt. Achtung! Bei C-Programmen ist mit dem ersten Befehl des Programms nicht der Befehl gemeint, der der erste der main()-Funktion ist. Der erste Befehl eines C-Programms sieht wie folgt aus:

```
__HO_org jmp .begin
```

Normalerweise wird der Befehl jmp .begin zu Beginn eines jeden Moduls erzeugt. Allerdings wird nur derjenige jmp.begin-Befehl ausgeführt, der im ersten gelinkten Modul steht. Um also Speicherplatz zu sparen, können Sie bei mehreren Programmmodulen die Erzeugung von jmp .begin bei jedem Modul unterbinden (siehe auch Compiler- und Assembler-Optionen).

Doch was bedeutet dieser Befehl? Dieser Befehl springt den Startup-Code eines jeden C-Programms an. Dieser befindet sich in der Linker-Library c.lib (Aztec) und dient dazu, die Exec- und Dos-Library für das C-Programm nutzbar zu machen und die Kommando-Parameter für die Routine main() aufzuarbeiten.

Wenn sich jemand für diesen Startup-Code interessiert, kann er ihn mit dem Kommando s <Return> single-steppen lassen. Mit s wird jeder einzelne Befehl des Programms abgearbeitet, und nach der Abarbeitung werden die Registerinhalte ausgegeben (inklusive Programm-Counter (Pc), Status-Register (Sr) und den Condition-Flags (x - Extension- (Erweiterungs)-Flag, n - Negativ-Flag, z - Zero (Null) - Flag, v - Overflow- (Überlaufs)-Flag, c - Carry- (Übertrags)-Flag). Nachdem die Registerinhalte angezeigt wurden, wird der nächste abzuarbeitende Befehl angezeigt.

Wenn beim Single-Steppen auf den Aufruf einer Routine gestoßen wird, deren Funktion man kennt, braucht man nicht jeden einzelnen Befehl dieser Routine durchzugehen. Man kann durch das t- (Trace)-Kommando dafür sorgen, daß diese Routine ausgeführt wird. Das funktioniert aber nur bei JSRs. Wenn also der nächste Befehl z.B. jsr _OpenWindow ist, können Sie durch t <Return> veranlassen, daß das Window geöffnet wird. So brauchen Sie nicht jeden einzelnen Befehl dieser Routine mit s ausführen zu lassen. Erstens werden Sie wahrscheinlich sowieso nicht so wahnsinnig viel davon verstehen, was bei der Ausführung solch eines Befehls vonstatten geht, und zweitens ist das Single-Steppen eines solchen Befehls meist sehr zeitaufwendig. Diese Zeit sollten Sie lieber darauf verwenden, Ihre eigenen Routinen zu debuggen. Außerdem kann man davon ausgehen, daß die Systembefehle fehlerfrei funktionieren. (Es sei denn, daß man bei der Übergabe von Parametern Unsinn gemacht hat. Dies muß man aber im Vorfeld des Aufrufs dieser Routine erkennen und nicht während der Abarbeitung dieses Befehls.)

Sicher werden Sie auch festgestellt haben, daß das Single-Steppen des Startup-Codes sehr zeitaufwendig ist und eigentlich nicht viel bringt. Wie gelangt man nun an die Stelle eines Programms, die wirklich von Interesse ist? Hierbei helfen uns soge-

nannte Break-Points (Unterbrechungsstellen), an denen ein Programm unterbrochen wird.

Mit Hilfe eines solchen Break-Points kann man sehr einfach zur Funktion main() gelangen, die ja der eigentliche Ausgangspunkt eines jeden C-Programms ist.

Mit bs _main setzt man einen Break-Point auf die Routine _main(). (Sicher ist Ihnen der Unterstreicher aufgefallen, der vor jede Routine gesetzt wurde. Bei der Übersetzung eines C-Programms in Maschinencode wird dafür gesorgt, daß vor jede Routine und globale Variable ein Unterstreicher gestellt wird.) Wenn man nun den Debugger mit Hilfe des g-Kommandos veranlaßt, das Programm auszuführen, wird zunächst der Startup-Code abgearbeitet. Wenn dann die Routine _main abgearbeitet werden soll, stellt der Debugger fest, daß hier ein Break-Point gesetzt wurde und unterbricht den Ablauf des Programms. Sie haben nun die Möglichkiet, mit Hilfe von s und t durch den Rest des Programms zu wandern.

Die Kommandos al (Load Task), s (Single-Step), t (Trace), g (Go = Programm unter Kontrolle des Debuggers ablaufen lassen) und bs (Break-Point-Setting) reichen also vollkommen aus, um ein Programm zu debuggen. Wenn das Programm beim Debuggen abstürzt, sehen Sie den Befehl, der das Programm zum Absturz gebracht hat. Meistens stürzt ein Programm während des Debuggings aber nicht in gewohnter Weise mit einer Guru-Meditation ab. Da der Debugger vollkommene Kontrolle über Ihr Programm und das System hat, ist er in der Lage, Sie mit einer Debugger-Meldung auf einen aufgetretenen Fehler hinzuweisen. Aber leider ist auch der Debugger nicht immer in der Lage, Systemabstürze abzufangen. Insbesondere bei Interrupt- und Task-Umschaltungsvorgängen kann es passieren, daß selbst der Debugger solche Fehler nicht mehr abarbeiten kann. Dies liegt zum großen Teil daran, daß der Debugger sehr intensiv mit Interrupts und Task-Manipulationen arbeitet.

Neben den oben aufgezählten einfachen Kommandos des Debuggers kann dieser aber wesentlich mehr. Da diese Kommandos für den Hausgebrauch kaum gebraucht werden, wollen wir diese

hier auflisten und von Fall zu Fall nur etwas näher auf diese Befehle eingehen.

Zunächst einmal ein Überblick über die Syntax der Parameter:

Wenn bei einem Befehl der Ausdruck "AUSDR" als Parameter hat, so kann "AUSDR" folgendes Aussehen haben:

```
AUSDR = TERM [binärer Operator-Term],
```

wobei die binären Operatoren folgendes Format haben:

Binäre Operatoren:

```
+  Addition
-  Subtraktion
*  Multiplikation
/  Division
%  Modulo-Devision
&  Bit-weises UND
|  Bit-weises ODER (inklusiv)
^  Bit-weises ODER (exklusiv)
```

TERM hat folgendes Format:

```
TERM:
   REGISTER,
   KONSTANTE,
  -TERM,
   ADDR, (Adresse)
  *ADDR, (Inhalt der Adresse)
  .,
   (AUSDR)
```

REGISTER:

Der Parameter REGISTER hat folgendes Format: A0, A1, D0, D1, etc. Weiterhin sind folgende Register bekannt: PC und SP (Synonym für A7 (Stackpointer)).

KONSTANTE:

Dieser Parameter ist eine hexadezimale, binäre, oktale oder dezimale Konstante:

```
0xabcd (hexadezimal)
0o1247 (oktal)
0b0110 (binär)
123.   (dezimal)
```

Falls keiner der Operatoren 0x, 0o, 0b oder . (Dezimalpunkt) verwendet wird, so wird die Konstante bezüglich der aktuellen Basis (siehe n-Kommando) interpretiert.

ADDR

Dieser Parameter legt eine Adresse fest. Dabei sind sowohl _main, 0x1264789 und _main+1e2 gültige Adressen im Sinne des DB.

**ADDR*

Mit Hilfe dieses Parameters erhalten Sie den Inhalt des Longwords, auf das ADDR zeigt. Es ist auch möglich, den Inhalt der Speicherstellen zu erfahren, auf die ein Register zeigt: *A7 (Inhalt des obersten Stack-Langwortes). Beachten Sie, daß Klammern gesetzt werden müssen bei Ausdrücken wie *SP+6, die *(SP+6) bedeuten sollen.

.

Dieser Parameter enthält die Anfangsadresse für ähnliche Kommandos. Mit db 0x200, 20. ist es z.B. möglich, die 20 Bytes, die ab Adresse 0x200 stehen, auszugeben. Wenn Sie nun den Punkt in Verbindung mit dem dw-Kommando verwenden, werden ab Adresse 0x200 die 20 folgenden WORDs ausgegeben. Beachten Sie, daß dieser Punkt nicht mit dem Dezimalpunkt zu verwechseln ist und für verschiedene Befehle verschiedene Bedeutungen haben kann (siehe db und dw).

@[Name einer Funktion]

Dieser Parameter ist insbesondere in Verbindung mit dem g-Kommando (Go) interessant. Folgender Aufruf sorgt dafür, daß ans Ende der gerade aktuellen Funktion ein Break-Point gesetzt wird und daß beim Erreichen dieses Break-Points das Programm unterbrochen wird:

g @
Wenn Sie wollen, daß das Programm am Ende einer bestimmten Funktion unterbrochen wird, müssen Sie den @-Parameter wie folgt benutzen:

g @Name_Der_Funktion
Wenn bei der Ausführung des Go-Kommandos das Ende der spezifizierten Funktion erreicht wird, wird das Programm unterbrochen.

Neben dem AUSDRUCK existieren noch BEREICHe und CMDLISTEN (Kommandolisten):

```
BEREICH:
  ADDR,ZÄHLER
   ADDR>ADDR
   ADDR
   ,ZÄHLER
```

Bei BEREICHen wird wie folgt vorgegangen: Wird der BEREICH durch ADDR,ZÄHLER definiert, so werden z.B. ab der angegebenen Adresse ZÄHLER-Bytes, Words oder Longwords ausgegeben. Bei ADDR>ADDR werden - je nach aufgerufenem Befehl - die zwischen diesen Adressen liegenden Bytes, Words oder Longwords ausgegeben.

Wird nur die ADDRESSE angegeben, so wird die vorher festgelegte Anzahl auszugebender Bytes etc. verwendet.

Bei alleiniger Angabe eines Zählers wird die zuletzt erreichte Adresse (z.B. ADDR+ZÄHLER des vorigen Befehls) verwendet.

Fehlt diese Angabe, so wird die zuletzt erreichte Adresse und der alte Zähler verwendet. Beachten Sie, daß der Zähler je nach verwendetem Befehl unterschiedlich interpretiert werden kann. Er kann z.B. die Anzahl der auszugebenden Bytes bestimmen oder aber die Anzahl der auszugebenden Linien eines Assembler-Dumps.

CMDLISTEN sind Kommandolisten und werden wie folgt definiert:

CMDLISTE: Kommando [; Kommando ...]
Die Kommandoliste besteht also aus aneinandergereihten Kommandos, die durch ein Semikolon getrennt werden. Diese sind:

add
Alle im System vorhandenen Devices anzeigen.

adi
Alle im System vorhandenen Interrupts anzeigen.

adl
Alle im System vorhandenen Librarys anzeigen.

adp
Alle im System vorhandenen und mit einem Namen versehenen Message-Ports auflisten.

adr
Alle im System enthaltenen Resources anzeigen.

ai
Informationen über einen Task anzeigen. Wenn ein Programm debugged werden soll, werden nach ai die Informationen über den Programmtask ausgegebn. Ist kein Programm geladen, können Sie durch Eingabe einer Nummer einen Task auswählen, dessen Informationen Sie betrachten wollen. Der DB informiert Sie natürlich auch darüber, welche Nummer zu welchem Task gehört.

ak
Dieser Befehl versucht, die Routine _abort oder exit() aufzurufen, um das Programm, das gerade debugged wird, aus dem Speicher zu entfernen. Wenn das Programm auf diese Art und Weise nicht entfernt werden kann, muß das System neugebootet werden. Es gibt allerdings noch die Möglichkeit, mit Hilfe des Setzens des Programm-Counters das Programm zu verlassen: rPC=_exit und g.

al, aL

Lade das Programm, das als nächstes vom CLI oder der Workbench aus gestartet wird, in den Debugger. Bei al wird dabei die Symboltabelle mit geladen, während bei aL auf die Symboltabelle verzichtet wird.

an, aN

Erzeuge ein neues Debugger-Window. Bei an wird die Symboltabelle des Hauptfensters verwendet, während aN eine eigene Symboltabelle enthält. an läßt sich also gut dazu verwenden, einen Untertask, der vom zu debuggenden Programm erzeugt wird, zu debuggen, währen aN für ein ganz neues Programm benutzt werden kann.

am

Zeige verfügbaren Speicher an.

ap, aP

Diese Funktion ist sehr interessant. Sie dient dazu, abgestürzte Programme zu debuggen. Wenn also der DB schon gestartet wurde oder aus einem anderen CLI noch aufgerufen werden kann, ist es möglich, herauszufinden, wo die Guru-Meditation erzeugt wurde.

Dazu brauchen Sie nur ap (mit Symboltabelle, die der DB nachzuladen versucht) bzw. aP (ohne Symboltabelle) aufzurufen. Danach brauchen Sie nur den "Finish all Disk-Activities..."-Requester zu "canceln" und können das Programm, das zum Systemabsturz geführt hat, debuggen. Dies funktioniert allerdings nur dann, wenn alle Systemstrukturen erhalten worden sind. Wenn also bestimmte Speicherbereiche Überschrieben wurden und so wichtige Informationen fehlen, kann auch das ap-Kommando nichts bewirken.

aq

Schließe sofort alle DB-Fenster und beende die DB-Sitzung.

ar

Wenn ein Task zum Debuggen auserwählt wurde, wird dieser solange angehalten, bis der Benutzer s, t oder g verwendet. Der Debugger behält aber immer die Kontrolle über diesen Task, solange der Task nicht beendet wurde. Mit ar ist es möglich, einen zum Debuggen ausgewählten Task wieder aus den Klauen des DB zu entfernen (Allow Task to resume).

as, aS

Mit diesem Kommando können Sie einen Systemtask zum Debuggen auswählen. Nach dem Aufruf dieses Kommandos werden Sie nach der Nummer des zu debuggenden Tasks gefragt. (as verwendet die Symboltabelle, aS nicht.)

at

Nach diesem Kommando werden alle Systemtasks, sowie deren Identifikationsnummern für as und ap, angezeigt.

bb

bw

bl

bb ADDR [== [AUSDR]]

bb ADDR [!= [AUSDR]]

bw ADDR [== [AUSDR]]

bw ADDR [!= [AUSDR]]

bl ADDR [== [AUSDR]]

bl ADDR [!= [AUSDR]]

Mit Hilfe dieser Kommandos ist es möglich, sogenannte Memory-Break-Points zu setzen. Mit Hilfe dieser Memory-Break-Points kann man eine Programmunterbrechung veranlassen, sobald der Wert der Speicherstelle, der durch ADDR bestimmt wird, vom alten Wert abweicht (bb ADDR) bzw. gleich oder ungleich dem angegebenen Wert ist (bb != 0, oder bb == 0). Je nach Befehl bestimmt ADDR die Adresse eines Bytes, Words oder Longwords.

bc ADDR

bC

Diese Kommandos erlauben es, alle Break-Points zu löschen (bC). Mit bc ADDR wird nur der angegebene Break-Point gelöscht.

bd

Dieser Befehl listet alle Break-Points auf. Dabei wird - wenn möglich - der Symbolname des Breakpoints angegeben, die Anzahl der "Break-Point-Treffer", die Anzahl der "Break-Point-Ignorier-Rate" sowie der Befehl, der bei Antreffen eines Break-Points ausgeführt werden soll:

```
adress          hits    skip   command
_main            0       0
_GetDir          1       3     db FIB+20.,30.
```

Diese Liste sagt aus, daß zwei Break-Points existieren. Und zwar am Anfang der Routine _main und _GetDir.

Die Angaben bei *hits* sagen aus, wie oft schon auf den jeweiligen Break-Point getroffen wurde. *skip* gibt an, wieviele Break-Point-Treffer lang nichts geschehen soll. _GetDir ist schon einmal angesprungen worden, aber erst nach dem 4ten Aufruf dieser Funktion wird tatsächlich eine Programmunterbrechung ausgelöst, wobei dann der unter command stehende Befehl ausgeführt wird.

bh [BEREICH]

Mit Hilfe dieses Befehls ist es möglich, eine Veränderung innerhalb des durch BEREICH festgelegten Speicherabschnittes festzulegen. Dazu wird über den BEREICH eine Checksumme gebildet, die nach jedem ausgeführten Befehl neuberechnet, und mit der Original-Checksumme verglichen wird. Beachten Sie,so daß durch Break-Point-Kommandos (insbesondere durch bh) die Abarbeitungszeit des Programms drastisch verlängert werden kann.

bq

Dieses Kommando erlaubt Ihnen, die Checksummenberechnung- und -überprüfung des Bereichs von Adresse 0 bis Adresse 255 an- bzw. auszuschalten. Nach dem Start von DB ist diese Überprüfung angeschaltet. Immer dann, wenn eine Veränderung in den untersten 256 Bytes festgestellt wird, wird die Meldung "Low memory Checksum different !!!" ausgegeben.

Wenn Sie diese Überprüfung abschalten, ist der Single-Step (Kommando: s) ein wenig schneller (Quick-Step).

br [ADDR]

Mit Hilfe dieses Kommandos wird der hits-Zähler des angegebenen (br ADDR), oder aller Break-Points (br) auf 0 gesetzt.

[#] bs ADDR [;CMDLISTE]

Bei Verwendung dieses Kommandos wird ein Break-Point festgelegt. Dabei wird die durch das Zeichen # repräsentierte Zahl der SKIP-Zähler angegeben. CMDLISTE enthält die/den Befehl(e), die nach dem Erreichen des Break-Points ausgeführt werden sollen. Beispiel:

```
3 bs _GetDir db _FIB+20.,20.
```

bt, bT

Diese beiden Kommandos versetzen Sie in die Lage, den Trace- (bt) bzw. Back-Trace-Modus (bT) an- bzw. auszuschalten.

Nach bt werden alle Routinen, die angesprungen werden, sowie die Übergabeparameter (als WORDS) angegeben.

Nach bT wird der jewelige Name der Routine, die verlassen wird, ausgegeben.

bu [ADDR]

Dieses Kommando veranlaßt den DB dazu, nach dem Auftreten eines Break-Points die hier angegebene Routine anzuspringen. Die Routine, die aufgerufen wird, muß eine C-Routine sein.

Das heißt, daß die C-spezifischen Register gerettet werden müssen.

Der Routine wird ein Langwort-Array übergeben, das folgende Werte enthält:

```
Die Werte von D0-D7.
Die Werte von A0-A7.
Den Wert des Status-Registers.
Den Wert des Programm-Counters.
```

bu 0 sorgt dafür, daß die Break-Points unbeachtet bleiben.

cs

Dieses Kommando sorgt dafür, daß die Symboltabelle gelöscht wird.

db [BEREICH]

dw [BEREICH]

dl [BEREICH]

d

Diese Kommandos geben den Bereich in Bytes (db), Words (dw) oder Longs (dl) aus. Bei der Angabe des Bereiches ist die Start-Adresse optional. Geben Sie nur die Endadresse oder einen Zählwert an, so wird der Speicher ab der letzten dargestellten Speicherstelle ausgegeben. Wenn Sie nur d angeben, wird der Bereich im Format des vorigen Befehls (Byte, Word, oder Long) ausgegeben.

dc

Alle Code-Symbole (Funktionsnamen) anzeigen lassen.

dd

Alle Daten-Symbole (Variablen) ausgegeben.

dg

Dieses Kommando gibt alle Symbole der Symboltabelle mit der jeweiligen Adresse aus.

ds

Mit Hilfe dieses Kommandos können Sie zurückverfolgen, welche Routinen mit welchen Parametern aufgerufen wurden. Ausgehend von der aktuellen Funktion wird die Routine, die diese Routine aufgerufen hatte, deren aufrufende Routine usw. bis zum Aufruf von _main() inklusive Parameter (Word-Format) ausgegeben (Backtracking).

[#] g [@ Funktionsname] [ADDR] [;CMDLISTE]
[#] G [@ Funktionsname] [ADDR] [;CMDLISTE]

Diese Kommandos führen das zu debuggende Programm aus. Bei g werden die Break-Points, die durch bs gesetzt wurden, berücksichtigt, während G nur den Break-Point berücksichtigt, der beim Aufruf dieses Kommandos spezifiziert wird.

gibt den "Skip-Count" des Break-Points an. ADDR ist die Adresse, wo der Break-Point gesetzt werden soll. Die CMDLISTE schließlich bestimmt, welche Kommandos nach dem Auftreten eines Break-Points ausgeführt werden sollen.

"@ Funktionsname" bestimmt den Namen der Funktion, bei dessen Ende ein Break ausgeführt werden soll.

ls Programmname

Mit Hilfe dieses Kommandos kann die Symboltabelle eines Programms erneut geladen werden. Dies ist dann recht nützlich, wenn die alte Symboltabelle verändert wurde. Symbole, die mit v in die Symboltabelle geschrieben wurden, bleiben erhalten.

ma AUSDR

Dieses Kommando reserviert Speicherplätze. EXPR gibt dabei die Anzahl der zu belegenden Bytes an. Ihnen wird durch Ausgabe der Anfangsadresse mitgeteilt, wo der von ihnen belegte Speicher zu finden ist. Beim Verlassen des DB wird dieser Speicher automatisch freigegeben.

mb ADDR AUSDR1 [AUSDR2 ..]

mw ADDR AUSDR1 [AUSDR2 ..]

ml ADDR AUSDR1 [AUSDR2 ..]

Mit Hilfe dieser Befehle können Sie Speicherstellen verändern. Dazu geben Sie die Anfangsadresse des zu verändernden Speicherbereichs an (ADDR) und bestimmen durch AUSDR1 (AUSDR2 ...), welche Werte in diesen Speicherbereich geschrieben werden sollen. Mit mb werden Bytes, mit mw Words und mit ml Longwords geschrieben.

mc BEREICH = ADDR

Mit Hilfe dieses Kommandos können Sie einen Bereich mit den ab ADDR stehenden Bytes vergleichen. Es werden jeweils die Unterschiede ausgegeben (ADDR1 != ADDR2).

mf BEREICH = AUSDR

Dieses Kommando füllt den angegebenen BEREICH mit dem in AUSDR spezifizierten Wert (Byte).

mm BEREICH = ADDR

Mit Hilfe dieses Kommandos können Sie den BEREICH an die angegebene Adresse kopieren

ms BEREICH = AUSDR1 [AUSDR2 ...]

Mit Hilfe dieses Kommandos kann ein Bereich nach der angegeben Byte-Folge (AUSDR1...) durchsucht werden. Immer dann, wenn die Byte-Folge gefunden wurde, wird die Anfangsadresse ausgegeben.

nX

Berechnungsbasis für alle Berechnungen und Angaben verändern:

```
nb: binär
nd: dezimal
nx: hexadezimal
no: oktal
```

q

Mit Hilfe dieses Kommandos wird der DB verlassen. Der zu debuggende Task wird freigegeben, alle DB-Fenster werden geschlossen.

r [reg=AUSDR]

Alle Registerinhalte (inklusive SR und PC) werden ausgegeben. Ist außerdem ein 68881-Mathemathik-Co-Prozessor eingebaut, werden auch dessen Register ausgegeben (Aufruf ohne Parameter). Mit Hilfe des r-Kommandos ist es auch möglich, Registerinhalte zu verändern (Beispiel: rPC = _exit).

[#] s [;CMDLISTE]
[#] S [;CMDLISTE]
[#] t [;CMDLISTE]
[#] T [;CMDLISTE]

Diese Kommandos dienen zum Single-Steppen durch ein zu debuggendes Programm. s führt den nächsten Befehl aus und gibt im Anschluß daran alle Registerinhalte aus. Bei S wird auch der nächste Befehl ausgeführt - allerdings wird hier auf die Ausgabe der Registerinhalte verzichtet.

Nach gleichem Schema arbeiten die Kommandos t- und T. Allerdings wird hier eine ganze Routine - aufgerufen durch JST - abgearbeitet. Trifft man also innerhalb eines Programms auf einen JSR-Befehl, so gelangt man nach t bzw. T an den Befehl nach der Ausführung der Unterroutine.

gibt die Anzahl der Aufrufe von s, S, t, T an, die getätigt werden sollen.

u [BEREICH]
U [BEREICH]

Mit Hilfe dieser Kommandos wird der angegebene Bereich disassembliert. Bei u werden dazu die Symbole verwendet, während U mit absoluten Adressen arbeitet.

v SYMBOL = AUSDR

V SYMBOL = AUSDR

Diese beiden Kommandos erzeugen ein neues Symbol (v) bzw. verändern ein bestehendes (V). SYMBOL ist hierbei eine Zeichenkette.

Symbole, die mit v erzeugt werden, sind Code-Symbole (siehe dc).

xmc

xm = CMDLISTE

x?

Mit Hilfe dieses Befehls können Sie Macros definieren. Dazu brauchen Sie nach x nur einen Buchstaben anzugeben und in der CMDLISTE zu bestimmen, welche Befehle ausgeführt werden sollen.

Mit x(Buchstabe) wird das definierte Macro aufgerufen. Mit x? können Sie sich alle definierten Macros ausgeben lassen. Aufgrund der Tatsache, daß Sie nur einen Buchstaben nach x angeben können, können Sie maximal 26 Macros definieren.

= AUSDR

Mit Hilfe dieses Kommandos können Sie einen Ausdruck ausgeben lassen. = 1+2 erzeugt z.B:

```
0x00000003  +3   3   03   %11   '....'   3
    ^        ^   ^    ^    ^      ^      ^
    |        |   |    |    |      |      |
   hex    signed dez okt  bin   ASCII  String
          dez
```

< File

> File

>> File

Ein- und Ausgaben umlenken:

< File sorgt dafür, daß die Eingaben nicht mehr von der Tastatur, sondern aus dem angegebenen File erfolgen. Dies geschieht solange, bis das File vollständig gelesen wurde.

> File sorgt dafür, daß alle Ausgaben (Registerinhalte, etc.) in das angegebene File geschrieben werden. Dies geschieht solange, bis entweder ein neues Ausgabe-File festgelegt oder nur > aufgerufen wird. Während der Ausgabe auf ein File wird auch das Window wie gewohnt verwendet.

>> File sorgt dafür, daß nur Kommandos in das angegebene File geschrieben werden. Dies ist z.B. dann recht nützlich, wenn der Debugging-Vorgang sehr kompliziert ist und häufig wiederholt werden muß (siehe: < File). Diese Option ist zu vergleichen mit dem Spielstand-Speichern bei einem Adventure - nicht allein deshalb, weil das Debuggen zu einem aufregenden und nervenaufreibenden Abenteuer werden kann).

?

Dieses Kommandos gibt eine Übersicht über alle DB-Kommandos aus.

2.4.2 Der Aztec-Source-Level-Debugger

Neben dem DB wird von der Firma Aztzec ein weiterer Debugger vertrieben. (Er ist nicht Bestandteil des Compilerpakets und muß gesondert erworben werden.) Dieser Debugger ist ein Meisterwerk der Programmierkunst. Mit ihm ist es nämlich möglich, C-Programme auf C-Source-Ebene (daher der Name Source-Level-Debugger) zu debuggen. Das bedeutet, daß Sie jeden einzelnen C-Befehl Ihres Programms debuggen können. Sie debuggen also nicht den Maschinencode, den der Compiler erzeugt, sondern die einzelnen C-Befehle. Man könnte den Source-Level-Debugger mit einem C-Interpreter vergleichen.

Wie benutzt man diesen Debugger nun? Zunächst muß auch hier dafür gesorgt werden, daß das C-Programm in einem bestimmten Format vorliegt. Dazu muß das Programm mit der Option -n compiliert werden. Hiermit werden dem Linker Informationen

bereitgestellt, um ein .dbg-File zu erzeugen, daß die vom Source-Level-Debugger benötigten Informationen enthält (Symbol-Tabelle).

Um den Linker zu veranlassen, daß solch ein .dbg-File erzeugt wird, muß beim Linken die Option -g verwendet werden. Zu beachten ist, daß die Option -g vor der Angabe aller zu linkenden Module angegeben wird, damit alle Symbole (Variablen und Strukturen) in diesem .dbg-File enthalten sind:

```
cc Programm -n
ln -q Programm -lc
```

Kommen wir nun zum Aufruf des Debuggers. Aufgerufen wird der SDB nach folgendem Format:

```
SDB [Optionen] [Programm [argv[1] argv[2] ...]]
```

Sie können dem SDB also direkt beim Aufruf das zu debuggende Programm (inklusive Kommando-Parameter) übergeben. Weiterhin können einige Optionen angegeben werden:

SDB-Optionen

-a

Normalerweise wird der SDB im C-Modus benutzt. Wenn Sie allerdings diese Option verwenden, arbeitet der SDB im Assembler-Modus - also so wie der DB.

-w

Normalerweise verwendet der SDB eigene Windows, um das zu debuggende Programm, Fehlermeldungen usw. darzustellen. Wenn Sie allerdings die Option -w verwenden, wird das CLI-Fenster als Ausgabefenster verwendet. (Diese Option ist recht nützlich, wenn das zu debuggende Programm sehr umfangreich ist, und mit dem Speicher gehaushaltet werden muß.)

-p[PromptString]

Wenn Sie die Option -w verwenden, haben Sie mit Hilfe dieser Option die Möglichkeit, das Prompt nach Ihren Wünschen zu

gestalten. Wenn Sie die Option -p nicht verwenden, wird das Prompt sdb? verwendet. Wenn Sie nicht die Option -w verwenden, wird als Promt CMD? ausgegeben.

-e

Der SDB nutzt 3 verschiedene Dartellungsbereiche innerhalb seines Fensters. Der oberste Bereich wird zur Darstellung des C-Programms benutzt. Der mittlere Bereich stellt die Eingabezeile dar, in der Ihre Kommandos angegeben werden. Der unterste Bereich wird zur Ausgabe von Daten verwendet (Registerinhalte, Variableninhalte, Fehlermeldungen, Assemblerlistings usw.). Beim Aufruf des Debuggers werden die Kommandos nur in der Kommandozeile dargestellt. Wenn Sie den SDB aber mit dieser Option aufrufen, werden die Kommandos außerdem noch in dem unteren Datenbereich angezeigt. Dies ist dann recht nützlich, wenn Sie alle Ausgaben in ein File umlenken, um später nicht den Überblick darüber zu verlieren, welche Ausgabe durch welchen Befehl hervorgerufen wurde.

-mFILENAME

Mit Hilfe dieser Option werden alle Eingaben aus dem angegebenen File getätigt. Dies ist z.B. dann sehr nützlich, wenn immer einige Schritte ausgeführt werden müssen, bis man an eine fehlerhafte Programmstelle gelangt.

-sPFAD

Der SDB geht davon aus, daß sich das zu debuggende Programm im aktuellen Verzeichnis befindet. Sie können allerdings mit Hilfe der Option -s dafür sorgen, daß andere Verzeichnisse oder Disketten durchsucht werden sollen (z.B. -sDisk1:;Disk2:C-Sources;... oder -sDisk1:!Disk2:C-Sources!...).

-n

Normalerweise werden alle definierten Macros abgespeichert (xxxx.mac). Wenn Sie vermeiden wollen, daß diese Macros geladen werden, können Sie dies mit Hilfe dieser Option verhindern.

Display-Optionen

Die nun folgenden Optionen beziehen sich auf die Farben in den drei Darstellungsbereichen des SDB-Fensters:

-csT,H,hT,hH

Mit Hilfe dieser Option bestimmen Sie die Farben für Text, Hintergrund, highlighted Text und highlighted Hintergrund im obersten Darstellungsbereich (C-Programm) des SDB-Fensters. Die Werte, die Sie hier angeben können, müssen zwischen 0 und 3 liegen und geben die jeweilige Workbench-farbe an, mit der Text und Hintergrund dargestellt werden sollen.

-ccU,T,H,P

Mit Hilfe dieser Option legen Sie die Farben für die Kommandozeile fest (Umrandungsfarbe des Kommandobereichs, Textfarbe, Hintergrund- und Promptfarbe).

-cdT,H

Mit Hilfe dieses Kommandos geben Sie die Farben für den Datenbereich des SDB-Fensters an (Text und Hintergrund).

Beachten Sie, daß Sie alle 4 bzw. 2 Farbangaben bei der Farbwahl festlegen müssen.

Wenn Ihnen die Default-Farben des SDB-Fensters nicht gefallen, es Ihnen aber auch zu lästig ist, diese bei jedem Aufruf des SDB neu zu definieren, können Sie mit Hilfe der Umgebungsvariable SDBOPT diese Farben permanent erhalten:

```
set "SDBOPT=-cs1,2,3,0 -cc1,2,3,0 -cd0,1"

(Dieser Befehl sollte in der Startup-Sequence stehen.)
```

Kommen wir nun dazu, wie man beim Debuggen mit dem SDB vorgeht:

Zunächst muß das zu debuggende Programm in der oben beschriebenen Form vorliegen (Compiler-Option -n und Linker-Option -g). Dann wird der SDB z.B. mit

```
SDB Print Hallo
```

gestartet. Der SDB wird geladen, der wiederum das Programm Print lädt (dem Programm Print wird der Parameter Hallo übergeben). Die ersten Zeilen des C-Programms werden im obersten Teil des SDB-Fensters angezeigt, wobei die Programmzeile, die als nächstes abgearbeitet werden soll, farbig unterlegt wird. Im Gegensatz zum DB brauchen Sie sich hier nicht erst mit Hilfe von Break-Points bis zur Routine main() vorzuarbeiten. Dies übernimmt der SDB für Sie (im C-Source-Modus).

Sie können nun wie beim DB mit den Kommandos t, s und g durch das Programm marschieren.

Zur Erinnerung:

Mit s wird jeweils der nächste Befehl abgearbeitet. Wenn als nächstes eine Routine angesprungen werden soll, wird nach s der erste Befehl dieser Routine dargestellt und für den nächsten Einzelschritt (Single-Step) bereitgestellt. Um zu vermeiden, daß Sie sich erst mühsam durch alle Befehle einer funktionsfähigen Routine "steppen" müssen, können Sie beim Aufruf von Routinen das t-Kommando verwenden. Hier wird die aufgerufene Routine vollständig abgearbeitet und der nach dem Funktionsaufruf folgende Befehl zum Debuggen bereitgestellt. Mit g wird das Programm gestartet (ähnlich dem RUN-Befehl des Amiga-BASIC) und erst gestoppt, wenn ein Break-Point erkannt wird oder das Programm vollständig abgearbeitet wurde. (Es gibt hier im Gegensatz zum DB keine Möglichkeit, die Abarbeitung des Programms durch einen beliebigen Tastendruck zu unterbrechen.)

Auch das Setzen von Break-Points geschieht beim SDB genauso wie beim DB. Mit Hilfe des Kommandos bs werden Break-Points gesetzt (z.B. bs Write). Wenn bei der Abarbeitung eines Programms auf solch einen Break-Point gestoßen wird, wird das Programm unterbrochen, und Sie können ab dieser Stelle das Programm debuggen.

Mit Hilfe dieser 4 Befehle kann man ein Programm schon sehr effektiv debuggen. Jedoch wollen wir Ihnen die restlichen (und sehr kraftvollen Befehle) des SDB nicht vorenthalten. Dazu muß jedoch bekannt sein, welches Format z.B. ein AUSDRUCK, BEREICH usw. hat:

AUSDR: (EXPR)

Ein Ausdruck ist jeder gültige Ausdruck der Sprache C. Z.B: c++, i = i*2, (i==2) ? k:l, printf("Hallo") usw.

ADDR:

Eine Adresse ist entweder der Name einer C-Variable (oder einer Routine), eine absolute Adresse (z.B. 0x123456) oder ein C-Ausdruck, von dem eine Adresse ermittelt werden kann. Beispiele:

.23: 23ste Zeile des aktuellen Programms

Modul.c.23: 23ste Zeile des Programms Modul.c. Mit Hilfe dieser Adressen ist es möglich, Programme zu debuggen, die aus mehreren Modulen bestehen.

main

Adresse der Routine main().

Array[k]

Adresse des k-ten Elements des Arrays.

i

Adresse der Variable i.

BEREICH: (RANGE)

Ein Bereich wird benötigt, um z.B. festzulegen, welchen Umfang eine Ausgabe annehmen soll. Der Bereich bestimmt z.B., welcher Teil eines C-Programms aufgelistet werden soll.

Definiert ist ein Bereich wie folgt:

a. ADDR, ZÄHLER

b. ADDR to ADDR (Schlüsselwort to muß angegeben werden)

c. ADDR

d. ZÄHLER

Mit a wird z.B. die erste Zeile festgelegt und mit ZÄHLER (ganze Zahl (Basis: dezimal)) die Anzahl der Zeilen, die ab der angegebenen Adresse ausgegeben werden sollen.

Mit b wird der Bereich durch zwei Adressen festgelegt.

Bei c wird der vorher verwendete ZÄHLER-Wert verwendet, während bei d ein neuer ZÄHLER definiert und die zuletzt erreichte Adresse (z.B. ADDR+ZÄHLER) verwendet wird.

CMDLIST:

Eine Kommandoliste ist eine Aneinanderreihung von Kommandos, die jeweils durch ein Semikolon getrennt sind. Solche Kommandolisten werden häufig zur Definition von Macros verwendet. Hieraus resultiert, daß in dem Fall, daß ein Macro (auch eine Ansammlung von Kommandos) Teil einer CMDLIST sein soll, dieses am Ende dieser steht. Mögliche Formate sind:

Kommando1 [;Kommando2 ...]

Nachdem diese Definitionen geklärt sind, wollen wir Ihnen nun die einzelnen Kommandos des SDB vorstellen. Dabei werden Sie feststellen, daß die Syntax vieler Befehle auf diese Definitionen (BEREICH, ADDR, AUSDR, CMDLIST) zurückgreift. Machen Sie sich diese deshalb zu eigen. Meistens kommt die Kenntnis dieser Definitionen aber durch das häufige Arbeiten mit dem

Debugger. Da man sehr schnell Vorlieben für bestimmte Erscheinungsformen dieser Definitionen entwickelt, fällt es nicht schwer, sich diese zu merken.)

Ein kleiner Hinweis noch:

Das Debugging-Fenster setzt sich ja aus drei Teilen zusammen. Mit Hilfe der Tastenkombinationen <Shift>+<Alt>+<Crsr Up> und <Shift>+<Alt>+<Crsr Down> können Sie die Position der Kommandozeile verändern. Dabei wird Platz für den oberen oder unteren Darstellungsbereich frei. Mit Hilfe der Tasten <Shift>+<Crsr Up> und <Shift>+<Crsr Down> wird im C-Source "geblättert", während mit <Alt>+<Crsr Up> und <Alt>+<Crsr Down> im unteren Darstellungsbereich geblättert werden kann. (Diese Funktionen können auch mit Hilfe der Maus und der Slider am rechten Fensterrand aufgerufen werden.) Wenn Sie nur die Cursor-Tasten verwenden, werden im Kommandobereich die vorigen Befehle angezeigt, bzw. der Cursor innerhalb der Kommandozeile bewegt (die Kommandozeile erlaubt volle Editierung der Befehle).

acc [U, T, H, P]

Mit Hilfe dieses Befehls setzen Sie entweder die Farben für die Kommandozeile (Umrandungs-, Text-, Hintergrund- und Promptfarbe) oder lassen sich die aktuelle Farbdefinition anzeigen. Die Defaultwerte sind 1,1,0,1. Beachten Sie, daß bei einer Änderung alle 4 Farbdefinitionen angegeben werden müssen.

acd [T, H]

Mit Hilfe dieses Befehls werden die Farben für den Datenbereich des SDB-Fensters festgelegt bzw. angezeigt (Text- und Hintergrundfarbe). Die Defaultwerte sind: 1, 2.

acs [T, H, hT hH]

Mit Hilfe dieses Kommandos werden die Farben für den oberen Bereich (C-Source-Darstellung) des SDB-Fensters festgelegt bzw. angezeigt (Text, Hintergrund, highlighted Text, highlighted Hintergrund). Die Defaultwerte sind: 1,0,1,3.

add
Alle im System momentan vorhandenen Devices anzeigen.

adi
Alle im System momentan vorhandenen Interrupts anzeigen.

adl
Alle im System momentan vorhandenen Librarys anzeigen.

adp
Mit Hilfe dieses Befehls werden alle im System momentan vorhandenen und mit Namen versehenen Message-Ports angezeigt.

adr
Alle im System momentan vorhandenen Resources anzeigen.

ae
Mit Hilfe dieses Befehls, der wie ein Schalter wirkt, können Sie dafür sorgen, daß alle eingegebenen Kommandos auch im Datenbereich (unterster Teil) des SDB-Fensters angezeigt werden (siehe Option -e beim Aufruf des SDB).

am
Dieser Befehl gibt Ihnen Auskunft über den momentan freien Speicher.

ax
Da man mit dem SDB auch eigene Librarys und Devices debuggen kann (siehe ll und ld), ist es zeitweise so, daß der SDB zwei Symbol-Tabellen verwalten muß. Zum einen wird die Tabelle der Library oder des Devices benötigt, und zum anderen die Tabelle des Test-Programms (ohne Programm kann man Librarys oder Devices nämlich nicht debuggen). Mit Hilfe dieses Kommandos kann man nun zwischen beiden Symbol-Tabellen hin- und herspringen.

bc ADDR

bC

Mit Hilfe dieses Befehls werden ein einzelner (bc) oder alle (bC) Break-Points gelöscht.

bd

Dieses Kommando listet alle Break-Points auf. Ebenso wie beim DB haben Sie auch beim SDB die Möglichkeit einen Skip-Count und ein auszuführendes Kommando anzugeben. Diese beiden Informationen werden auch hier bei der Ausgabe berücksichtigt. Jedoch weicht die Ausgabe der Break-Point-Adresse ein wenig von der des DB ab:

```
#  address                          hits  skip  command
1  413f6   alt.c.504                  0     0
```

Zunächst wird die laufende Nummer des Break-Points angegeben. Dann wird die Adresse des Break-Points ausgegeben. Wie Sie sehen, wird die Zeile innerhalb des Programms angegeben. Auch wenn ein Break-Point z.B. wie folgt gesetzt wurde: "bs main", so wird dennoch nur die Adresse (Zeile) innerhalb des C-Programms ausgegeben. Nach der Adresse wird die Anzahl der "Treffer" ausgegegen - also die Anzahl der bisherigen "Berührungen" dieses Break-Points. skip gibt an, nach wie vielen Treffern tatsächlich das Programm an dieser Stelle unterbrochen werden soll. command schließlich enthält das Kommando, das beim Auftreten dieses Break-Points ausgeführt werden soll.

be AUSDR

Mit Hilfe dieses Kommandos wird ein Expression-Change-Break-Point gesetzt. Wenn also eine Veränderung bei dem angegebenen Ausdruck auftritt (z.B. Veränderung einer Variablen), wird das Programm unterbrochen. Wird das g-Kommando für den Ablauf des Programms verwendet, wird der Ausdruck nur bei jedem Eintritt in bzw. Austritt aus einer Funktion überprüft (Zeitersparnis). Bei der Verwendung von s und t wird der Ausdruck nach jedem einzelnen dieser Kommandos überprüft.

br [ADDR]

Dieses Kommando löscht die hit- und skip-Zähler des angegebenen, bzw. aller Break-Points.

[#] bs ADDR [;CMDLIST]

Mit Hilfe dieser Routine wird ein Break-Point gesetzt. Wenn während des Programmablaufs der Programmcounter die angegebene Adresse erreicht, wird eine Unterbrechung ausgelöst. Wenn Sie beim Setzen eines Break-Points ein Kommando angegeben haben, wird dieses nach der Unterbrechung ausgeführt (z.B. Darstellung eines Speicherbereichs und anschließende Weiterbearbeitung des Programms). Wenn Sie vor dem Schlüsselwort bs noch eine Zahl angeben, so gibt diese Zahl Aufschluß darüber, nach dem wievielten Mal des Erreichens dieser Adresse durch den Programmcounter (Treffer) tatsächlich eine Unterbrechung ausgelöst werden soll.

bt, bT

Diese Kommandos erlauben Ihnen das Setzen des sogenannten Trace- bzw. Source-Line-Trace-Modus. Beim Trace-Modus wird bei der Abarbeitung des Programms (mit g) jede Funktion inklusive Parameter angezeigt, in die verzeigt wurde. Wird diese Routine verlassen und hat diese einen Rückgabewert, so wird auch dieser angezeigt. Normalerseis ist dieser Modus ausgeschaltet. Jedoch wirkt das bt-Kommando auf diesen Modus wie ein Schalter.

Der Source-Line-Trace-Modus äußert sich darin, daß jede Zeile des C-Programms, die gerade abgearbeitet wird, farbig hinterlegt (highlighted) wird. Auch hier wirkt das bS-Kommando wie ein Schalter, der diesen Modus an- bzw. ausschaltet.

c

Dieses Kommando sorgt dafür, daß die aktuelle C-Zeile im Ausgabefenster zentriert wird (Center-Funktion).

da, dA

Diese beiden Befehle sorgen dafür, daß die "automatischen" Variablen der aktuellen Routine angezeigt werden. Automatische

Variablen sind sowohl diese mit der Speicherklasse auto, als auch die, die innerhalb von Routinen definiert werden, und demgemäß ihren Speicher vom Stack beziehen. Nach da werden alle Auto-Variablen der aktuellen Routine, sowie deren augenblicklichen Werte ausgegeben. Nach dA werden nur die Adressen dieser Variablen preisgegeben.

db [BEREICH]
dw [BEREICH]
dl [BEREICH]
d

Mit Hilfe dieser Befehle ist es möglich, einen bestimmten Speicherbereich in Bytes (db), Words (dw) oder Longwords (dl) ausgeben zu lassen. Bei erneutem Aufruf eines dieser Befehle ohne Bereichsangabe wird mit der Darstellung ab der vorher zuletzt dargestellten Adresse begonnen, wobei die Anzahl anzuzeigender Bytes, Words oder Logwords vom vorigen Aufruf bestimmt wird. d verwendet das vorige Ausgabeformat, die zuletzt dargestellte Adresse, sowie die vorige Anzahl dargestellter Werte.

dc

Alle globalen Code-Symbole anzeigen (Routinen).

dd

Mit Hilfe dieses Kommandos werden alle globalen Daten-Symbole (Variablen) aufgezeigt.

df [FILENAME] BEREICH

Mit Hilfe dieses Befehls werden die durch BEREICH festgelegten C-Zeilen des aktuellen bzw. angegebenen Programms angezeigt.

dg

Alle globalen Variablen sowie deren Werte anzeigen.

[#] ds

[#] dS

Mit Hilfe dieser beiden Kommandos können Sie ermitteln, von welcher Routine die aktuelle Routine aufgerufen wurde, und von welcher dieser aufgerufen wurde, und von welcher diese aufgerufen wurde, ... Neben den Namen der Routinen werden auch ihre Parameter und Rückgabewerttypen angezeigt. Bei dS werden außerdem die Werte aller Auto-Variablen angezeigt.

fu, fd

Da es eigentlich nicht möglich ist, lokale Variablen anderer Routinen zu betrachten, wurden diese beiden Kommandos eingeführt. Mit fu (Frame Up) können Sie in die jeweils aufrufende Routine aufsteigen, um dort die lokalen Variablen zu betrachten. Mit fd steigen Sie dann wieder in die jeweils aufgerufene Routinen hinab.

[#] g [@ [Funktionsname]] [ADDR] [;CMDLIST]

[#] G [@ [Funktionsname]] [ADDR] [;CMDLIST]

Mit Hilfe dieser beiden Kommandos kann ein Programm ausgeführt werden. Dabei wird nach g beim Auftauchen eines Break-Points evt. (je nach skip# und hit) eine Unterbrechung hervorgerufen, während G alle vorher gesetzten Break-Points ignoriert. Sie haben beim Aufruf des g- bzw. G-Kommandos die Möglichkeit genauso wie bei bs einen Break-Point festzulegen (Inklusive Skip und Kommando). Beim G-Kommando wird allerdings nur dieser Break-Point akzeptiert. Eine besondere Art Break-Point wird mit g @ [Funktion] bzw. G @ [Funktion] gesetzt: Hier wird dafür gesorgt, daß der Programmablauf unterbrochen wird, wenn aus der angegeben bzw. aktuellen Funktion zurückgekehrt wird.

ld DeviceName

ll LibraryName

lp [[progfile] [arg1 arg2 ...]]

Mit Hilfe dieser Befehle wird ein Programm zum Debuggen in den Speicher des SDB geladen. Sie können das Programm genauso aufrufen wie im CLI - also inklusive aller Kommando-

Parameter. Wird der lp-Befehl ohne Parameter ausgeführt, werden die Parameter des letzten lp-Befehls verwendet. Dies ist z.B. dann recht nützlich, wenn ein vollständig abgearbeitetes Programm erneut debugged werden soll.

Sie haben allerdings auch die Möglichkeit, eigene Librarys oder Devices zu debuggen. Dazu braucht nur das Device oder die Library mit dem ld- bzw. ll-Kommando in den Speicher geladen zu werden (z.B. ll eigenelibrary ohne ".library"-Tag). Allerdings muß vorher ein Testprogramm, daß die Libary bzw. das Device testet, mit lp geladen worden sein. Mit ll bzw. ld werden dann die Symbole der Library bzw. des Device geladen.

mb ADDR WERT1 [WERT2 ..]

mw ADDR WERT1 [WERT2 ..]

ml ADDR WERT1 [WERT2 ..]

Mit Hilfe dieser Kommandos können Sie Werte in den Arbeitsspeicher übertragen. Dazu wird die Anfangsadresse des ersten zu ändernden Wertes übergeben (auf gerade Adressen bei mw und ml achten!). Die angegebenen Werte werden ab der angegebenen Adresse in den Speicher übertragen.

mc BEREICH = ADDR

Dieser Befehl erlaubt Ihnen zwei Speicherbereiche zu vergleichen. Die Adresse des ersten Speicherbereichs sowie die Anzahl zu vergleichender Bytes wird im BEREICH festgelegt. Der zweite Speicherbereich wird durch ADDR bestimmt.

mf BEREICH = WERT

Mit Hilfe dieses Kommandos ist es möglich, einen Speicherbereich mit einem bestimmten Wert zu füllen. Beachten Sie, daß der BEREICH korrekt ist, da sonst der Rechner abstürzen könnte.

mm BEREICH = ADDR

Mit Hilfe dieses Kommandos wird der angegebene Bereich an die angegebene Adresse kopiert.

ms BEREICH = WERT1 [WERT2 ..]

Mit Hilfe dieses Kommandos wird der angegebene Bereich byteweise nach der angegebenen Byte-Folge durchsucht. Es werden die Adressen aller entdeckten Byte-Folgen angezeigt.

q

Mit diesem Befehl wird der SDB verlassen (Quit).

r [REGISTER=Wert]

Mit Hilfe dieses Kommandos können Sie entweder die Inhalte aller Register auflisten lassen oder ein bestimmtes Register (A0-A7, D0-D7, SP, SR und PC) verändern.

[#] s

[#] S

[#] t

[#] T

Mit Hilfe dieser Kommandos können Sie durch ein Programm befehlsweise fortschreiten. Bei s wird ein einziger Befehl abgearbeitet. Nach s werden die Registerinhalte angezeigt. S funktioniert genauso wie s - bis auf die Tatsache, daß auf die Ausgabe der Registerinhalte verzichtet wird.

Mit Hilfe der Kommandos t und T (keine Registerausgabe) können ganze Unterroutinen wie ein Befehl behandelt werden. Wenn Sie also wissen, daß eine bestimmte Funktion fehlerfrei ist, müssen Sie sich nicht durch jeden einzelnen Befehl dieser Routine mit den s-Kommandos quälen, sondern können den gesamten Funktionsaufruf mit t bearbeiten.

Mit # bestimmen Sie die Anzahl aufeinanderfolgender s- bzw. t-Aufrufe.

u [BEREICH]

U [BEREICH]

Mit Hilfe dieser beiden Befehle können Sie den angegebenen BEREICH disassemblieren lassen. Bei u wird auf die Symbolta-

belle zurückgegriffen, während U nur mit absoluten Adressen arbeitet.

Wenn Sie den BEREICH weglassen, wird mit der Darstellung des disassemblierten Codes ab der letzten disassemblierten Zeile begonnen. Die Anzahl der dargestellten Zahlen richtet sich nach der letzten Anzahl dargestellter Zeilen.

x MacroName [CMDLIST]

X

Mit Hilfe dieser beiden Befehle können Macros definiert, angezeigt oder ausgeführt werden. Die Definition eines Macros erfolgt z.B. so:

```
x Macro lp;db main,23
```

Ausgeführt wird dieses Macro mit "x Macro". Beachten Sie, daß die Schreibweise (Groß-, Kleinschreibung) des Macro-Namens (maximal 40 Zeichen) keinen Einfluß hat.

Mit X werden alle definierten Macros aufgelistet.

= AUSDR

e AUSDR

Mit Hilfe dieses Kommandos werden C-Ausdrücke ausgewertet. Erlaubt sind solch einfache Ausdrücke wie 1+2. Aber auch ganze Funktionsaufrufe sind erlaubt, wobei hier der Rückgabewert angezeigt wird (z.B. = Write (OutputFile, "Hallo", 5L). Beachten Sie, daß die benötigten Librarys und Variablen ordnungsgemäß initialisiert sind.

z

Mit Hilfe dieses Kommandos wechseln Sie zwischen dem Source-Modus und dem Assembler-Modus des SDB. Der SDB verhält sich nach diesem Kommando nämlich wie der "einfache" DB. Nach erneutem Aufruf dieses Kommandos ist der SDB wieder ein Source-Level-Debugger.

/"STRING"
Mit Hilfe dieses Kommandos kann nach dem angegebenen String im C-Source gesucht werden. Mit der Suche wird dabei ab der augenblicklichen Zeile im C-Source-Code begonnen.

> File
< File
>> File
Mit Hilfe dieser Kommandos können Sie die Ein- und Ausgaben umlenken:

< File sorgt dafür, daß die Eingaben nicht mehr von der Tastatur, sondern aus dem angegebenen File erfolgen. Dies geschieht solange, bis das File vollständig gelesen wurde.

> File sorgt dafür, daß alle Ausgaben (Registerinhalte, etc.) in das angegebene File geschrieben werden. Dies geschieht solange, bis entweder ein neues Ausgabe-File festgelegt oder nur > aufgerufen wird. Während der Ausgabe auf ein File wird auch das Window wie gewohnt verwendet.

>> File sorgt dafür, daß nur Kommandos in das angegebene File geschrieben werden. Dies ist z.B. dann recht nützlich, wenn der Debugging-Vorgang sehr kompliziert ist und häufig wiederholt werden muß (siehe: < File). Diese Option ist zu vergleichen mit dem Spielstand-Speichern bei einem Adventure - nicht allein deshalb, weil das Debuggen zu einem aufregenden und nervenaufreibenden Abentuer werden kann.

?
Mit Hilfe dieses Kommandos können Sie sich eine Übersicht der SDB-Kommandos holen.

2.5 Programmiertechniken

Zu einem großen C-Buch gehört auch die Beschreibung der gängigsten Programmiertechniken. In diesem Kapitel wollen wir Ihnen zeigen, wie Sie auf absolute Speicherstellen zugreifen,

Funktionsarrays anlegen, Files öffnen und benutzen und Directorys anzeigen können.

2.5.1 Der Zugriff auf absolute Speicherstellen

Sicher haben auch Sie sich schon einmal gefragt, wie man in einem C-Programm absolute Speicherstellen abfragen kann, ohne erst Librarys, Devices oder Resources öffnen zu müssen.

Am einfachsten geschieht dieser Zugriff mit Hilfe von Zeigern. Wie Sie wissen, enthalten Zeigervariablen Adressen auf Speicherstellen, die wiederum Werte enthalten. Unsere Aufgabe besteht also nur darin, der Zeigervariable mitzuteilen, auf welche Speicherstelle diese zeigen soll.

Normalerweise wird dies ja symbolisch durch Ausdrücke wie Zeiger = &Variable erledigt. Aber man kann Zeiger ja auch löschen (Zeiger = 0L;). Was ist dies aber anderes als die Zuweisung der Adresse 0 an den Zeiger?

Wenn also dies funktioniert, dann auch Zeiger = 0xabcdef. Beachten muß man hierbei nur noch, daß die Zuweisung korrekt ist, daß also eine Typumwandlung (CAST) durchgeführt wird. So akzeptiert der Compiler solch eine Zuweisung ohne jegliche Meldung.

Folgendes Programm erzeugt einen Zeiger auf die Speicherstelle 0xbfe001. Dieses Hardwareregister des CIA-A gibt Ihnen neben dem Status der Power-LED auch Auskunft über den Status der Feuerknöpfe angeschlossener Joysticks oder Mäuse. Das Programm sorgt nun solange dafür, daß die LED dunkel bleibt, bis die linke Maustaste gedrückt wurde:

```
#define Absolute_Adresse  0xbfe001
char                     *CIAA_Reg1; /* Zeigerdefinition */

#define MausKnopfB 6                 /* Mausknopf Bit */
#define LEDB       1                 /* LED Bit       */

#define MausKnopfF (1<<MausKnopfB)   /* Flags (Masken) */
#define LEDF       (1<<LEDB)
```

```
main()
{
   int i;   /* Zählvariable */

   CIAA_Reg1 = (char *)Absolute_Adresse;   /* Adressenzuweisung */

   printf("BLINK läuft. Warte auf Mausklick\n");

   i=0;

   *CIAA_Reg1 |= LEDF;                    /* Bit setzen */

   while ((*CIAA_Reg1 & MausKnopfF) == MausKnopfF);
                                          /* Mausknopf gedrückt? */

   *CIAA_Reg1 &= (0xff-LEDF);             /* Bit löschen */
}
```

Programm Blink.c

Beachten Sie, daß die hier verwendeten Bits Low-Activ sind. Das heißt, daß bei gesetztem Bit (Bit 6 dieses Registers) die Maustaste nicht gedrückt ist, während bei gelöschtem Bit die linke Maustaste niedergedrückt wurde.

Ähnlich ist es mit dem Bit für die Power-LED (Bit 1). Ist dieses Bit gesetzt, leuchtet die LED dunkel. Bei gelöschtem Bit leuchtet die LED hell.

Bei der Verwendung von Zeigern auf absolute Speicheradressen gilt es zu beachten, daß die Adresse in Einklang mit dem Zeigertyp steht. Wenn Sie z.B. einem Zeiger auf ein Wort oder Langwort eine ungerade Adresse zuweisen und dann mit Hilfe dieses Zeigers versuchen, die Speicheradresse anzusprechen, führt dies zu einem Systemabsturz, da Wörter und Langwörter nur an geraden Adressen liegen dürfen.

2.5.2 Schalter in der Kommandozeile

"Was hat man sich unter dieser Überschrift vorzustellen?" werden Sie sich sicher fragen. Sie alle kennen doch Programme, denen man beim Aufruf mitteilen kann, was wie zu passieren hat. Der Compiler ist hierfür ein gutes Beispiel.

Wie Sie wissen, können Sie mit Hilfe der Option -o festlegen, wohin das Output-File (daher der Name dieser Option) geschrieben werden und welchen Namen es erhalten soll. Neben der Option -o existieren allerdings weitere Optionen. So z.B. die Option -s, die dafür sorgt, daß alle Warnings unterdrückt werden.

Wie programmiert man solche Schalter bzw. Optionen?

Betrachten wir dazu einmal folgendes Programmsegment:

```
BOOL NewOutputFlag      = FALSE;
BOOL NoWarnings         = FALSE;
char *NewOutputFileName = 0L;

main(argc, argv)
int  argc;
char      *argv[];
{
   int i;
   for (i=1;i<argc;i++) /* Programmnamen übergehen */
   {
      if (argv[i][0] == '-') /* Optionspräfix */
      {
         switch(argv[i][1])  /* Option */
         {
            case 'o': /* anderes als Default-Output-File */
               NewOutputFilename = &argv[i][2];
               NewOutputFlag = TRUE;
            break;

            case 's': /* Silent-Option (keine Warnings) */
               NoWarnings = TRUE;
            break;
            ...

            default:
              /* Option ignorieren */
              /* oder Benutzerhinweis auf unbekannte Option  */
           }
      }
   }

   ...
   /* Output-File öffnen */
   if (NewOutputFile)
   {
      /* Öffne neues File mit Namen *OutputFilename */
   }
   else /* Öffne neues File mit Default-Namen */
```

```
        /* (z.B. mit Extension .o) */

    ...
    /* Compilieren */
    if (Warning_aufgetreten)
       if(!NoWarnings)
       {
          /* Warnung ausgeben */
       }
    ...
}
```

Man untersucht zunächst jeden Kommandozeilen-Parameter darauf, ob der erste Buchstabe eines Parameters (argv[i][0]) ein Optionspräfix ist (-). Ist dies der Fall, wird der zweite Buchstabe (argv[i][1]) daraufhin untersucht, ob er der Einleitung einer gültigen Option dient. Ist dies nicht der Fall, so kann man je nach Bedarf den Benutzer darauf hinweisen, daß er eine unbekannte Option angegeben hat und dann evt. das Programm abbrechen.

Im Verlauf des Programms kann dann anhand der booleschen Variablen, die bei Erkennen einer Option gesetzt werden, der Ablauf beeinflußt werden. Allerdings gilt es zu beachten, bestimmte Optionskonstellationen auszuschließen. Stellen Sie sich vor, daß Sie eine Option zum Erzeugen eines neues Files und eine zum Erweitern eines bestehenden Files zur Verfügung stellen. Wählt der Benutzer nun beide Optionen aus, so muß hier doch wohl ein Irrtumm vorgefallen sein. Schließlich kann man ein File nicht neu anlegen und gleichzeitig erweitern. Der Programmierer muß also dafür sorgen, daß sich ausschließende Optionen nicht ins Gehege kommen und so evt. Daten verloren gehen.

2.5.3 Funktionstabellen

Nun wollen wir Ihnen einen kleinen Vorgeschmack auf das Thema Intuition geben. Wie Sie vielleicht wissen, haben Sie mit Hilfe von Intuition die Möglichkeit, Menüs zu programmieren.

Wenn Sie einen Menüpunkt ausgewählt haben, so erhalten Sie von Intuition die Nummer des Menüs sowie die Nummer des Menüpunktes.

Sie haben nun die Möglichkeit, die einzelnen auszuführenden Funktionen durch Switch-Anweisungen auszuwählen:

```
switch(Menu)
{
   case 0: /* Projekt Menü */
     switch(Menüpunkt)
     {
        case 0: /* Info */
         ...;
        break;

        case 1: /* Speichern */
         ...;
        break;

        case 2: /* Laden */
         ...;
        break;

     }
   break;

   case 1:  /* Bearbeiten */
     switch(Menüpunkt)
     {
        case 0: /* Neue Schriftart */
         ...;
        break;

        case 1: /* Funktionstasten-Belegung */
         ...;
        break;
        ...;
     }
   break;
   ...
}
```

Wie Sie sehen, ist die Selektion der Routinen, die nach dem Anwählen eines bestimmten Menüpunktes angesprungen werden sollen, recht aufwendig.

Viel einfacher ist es doch, eine Funktionstabelle anzulegen, die die Funktionen enthält, die je nach ausgewähltem Menü aufgerufen werden sollen:

```
VOID (*Funktionen[MaxMenues][MaxMenuePunkte])();
```

Jetzt brauchen wir nur die aufzurufenden Funktionen an das Array zu übergeben,

```
#define PROJEKT 0
#define INFO 0
#define SPEICHERN 1
#define LADEN 2

#define BEARBEITEN 1
#define NEUESCHRIFT 0
#define FTASTEN 1
...
   Funktionen[PROJEKT][INFO]       = Info;
   Funktionen[PROJEKT][SPEICHERN] = Speichern;
   Funktionen[PROJEKT][LADEN]      = Laden;

   Funktionen[BEARBEITEN][NEUESCHRIFT] = NeueSchrift;
   Funktionen[BEARBEITEN][FTASTEN]     = FTasten;
...
```

und dann die jeweilige Funktion je nach Menüpunkt aufrufen:

```
Funktionen[Menü][MenüPunkt]();
```

Hier wird allerdings davon ausgegangen, daß alle Funktionen keinen Rückgabewert besitzen, also vom Typ VOID sind. Außerdem müssen alle Funktionen (Info(), Speicher(), Laden(), usw.) vor der Zuweisung an das Array, also vor main() oder der Routine, innerhalb der diese Zuweisung stattfindet, definiert worden sein.

Ein Nachteil den Funktionstabellen haben ist die Tatsache, daß alle enthaltenen Funktionen den gleichen Rückgabewert haben. Allerdings ist dies auch wieder nicht so tragisch. Wenn man alle Funktionen des Funktionsarrays als Funktionen mit Rückgabewert des Typs ULONG deklariert, kann man dann, wenn man einen diese Routinen einen Rückgabewert liefert, diesen in den jeweiligen Typ wandeln.

Hier wäre es allerdings angebrachter, gobale Variablen zu benutzen, was allerdings mehr Übersicht bei der Programmierung verlangt.

2.5.4 File-Handling

Ein Problem, das sich dem Programmierer recht häufig stellt, ist die Datenein- und -ausgabe. Ausgaben auf den Bildschirm können noch recht einfach mit Hilfe des printf()-Befehls getätigt werden. Auch Eingaben lassen sich leicht mit scannf() realisieren. Um aber Daten abzuspeichern, sind einige Schritte nötig.

Leider hat sich gezeigt, daß viele Programmierer die Vorsichtsmaßnahmen, die hier geboten sind, vernachlässigen. Dies ist eine häufige Ursache für Systemabstürze. Um Ihnen hier zu helfen, wollen wir uns im nun folgenden Kapitel mit der Erzeugung von Dateien befassen. Darüber hinaus werden wir Ihnen auch ein paar Tips und Kniffe für das Umfeld der Dateiprogrammierung geben. Sie werden z.B. erfahren, wie man feststellt, ob ein zu öffnendes File schon existiert, welche Files auf der Diskette oder in einem Unterverzeichnis der Diskette enthalten sind, welche Informartionen über eine Datei (bzw. über ein File) vom System zur Verfügung gestellt werden und wie man an diese herankommt.

2.5.4.1 Das Öffnen und Schließen von Dateien

Wie nicht anders zu erwarten, müssen auch Dateien geöffnet werden. Es gibt nahezu nichts auf dem Amiga, was nicht vor Benutzung geöffnet werden muß. Man denke nur an Librarys und Devices.

Um eine Datei zu öffnen bzw. neu anzulegen, wird das Open()-Kommando der DOS-Library verwendet. Keine Angst! Die DOS-Library gehört zu den beiden Librarys, die dem Benutzer innerhalb eines C-Programms sofort zur Verfügung stehen. EXEC- und DOS-Library sind diejenigen Librarys, die innerhalb des Startup-Codes eines jeden C-Programms geöffnet werden. Dies entbindet Sie als C-Programmierer von der lästigen Library-Eröffnung.

Betrachten wir einmal einen typischen Open()-Aufruf:

```
#include "libraries/dos.h"
#include "libraries/dosextens.h"
....

struct FileHandle *DateiSchluessel = 0L;
       /* Siehe "libraries/dos.h" */

....

  DateiSchluessel = (struct FileHandle*)
     Open(FileName, (ULONG) MODE_NEWFILE);
  if (DateiSchluessel == 0L)
  {
    printf("Die Datei <%s> konnte nicht erzeugt werden !!!",
          FileName);
    /* exit((ULONG) RETURN_ERROR); */
    /* oder Aufruf einer Fehlerbehandlungsroutine */
  }
....
  Close(DateiSchluessel);
```

Wie Sie sehen, wird dem Open()-Befehl der Name der zu öffnenden Datei übergeben. Dies ist ein normaler - also mit \000 abgeschlossener - String. Das zweite Argument, das dem Open()-Befehl übergeben wird, ist eine Modus-Variable. Diese gibt an, ob ein bestehendes File geöffnet (MODE_OLDFILE) oder ein neues File erzeugt werden soll.

Betrachten wir diese beiden Modi einmal genauer:

```
#define MODE_OLDFILE 1005 (s. "libraries/dos.h")
```

Dieser Modus veranlaßt den Open()-Befehl, den Dateischlüssel auf ein schon bestehendes File zu erzeugen. Existiert allerdings das File, das man öffnen möchte, gar nicht, so erzeugt Open() hier ein neues, leeres File.

```
#define MODE_NEWFILE 1006
```

Dieser Modus veranlaßt den Open()-Befehl, in jedem Fall ein neues File zu erzeugen. Wenn noch kein File mit dem angegebenen Namen existierte, erzeugt Open() hier ein neues, leeres File. Was passiert aber, wenn Open() festgestellt hat, daß schon ein File mit dem angegebenen Namen existiert? Nun ganz einfach: Open() sorgt dafür, daß das existierende File gelöscht wird (!) und erzeugt im Anschluß daran ein neues File. Beachten Sie,

daß der Open()-Befehl davon ausgeht, daß Sie tatsächlich ein neues File anlegen wollen. Sie werden von Open() nicht befragt, ob das schon existierende File wirklich gelöscht werden soll oder nicht. (Sie können jedoch feststellen, ob das zu öffnende File schon existiert. Dazu später mehr.)

Was passiert jedoch, wenn die Diskette, auf der Sie das File unterbringen wollen, einen Schreib-/Lesefehler hat oder ganz einfach schon vollkommen gefüllt ist? Nun, Sie werden sich sicher sagen: Es erscheint einfach ein Systemrequester, der mich auf diesen Fehler hinweist und das war's.

Leider war es das nicht. Schließlich muß Ihr Programm darauf reagieren, daß solch ein Fehler aufgetreten ist. Erkannt wird ein Fehler beim Öffnen eines Files daran, daß der von Open() zurückgegebene Dateischlüssel den Wert 0 hat.

Ist dies der Fall, können Sie entweder das Programm mit exit() verlassen oder in eine eigene Fehlerbehandlungsroutine springen, die den Benutzer auf den Fehler hinweist und ihn z.B. auffordert, eine neue Diskette einzulegen. (Beachten Sie bitte, daß Sie zum Verlassen eines C-Programms den Befehl exit() verwenden. Dies ist ein Compiler-immanenter Befehl und darf nicht mit dem Exit()-Befehl der DOS-Library verwechselt werden! Der Unterschied zwischen Exit() und exit() liegt darin, daß exit() ganz genau "Bescheid weiß", welche Librarys im Startup-Code geöffnet wurden, wieviel vom Stack verwendet wurde, und wie mit den Kommando-Parametern umgegangen worden ist. Diese Informationen fehlen dem Exit()-Befehl natürlich, so daß der Aufruf von Exit() zu einem totalen Systemabsturz führen könnte.)

Dem exit()-Befehl kann - wie Sie oben sehen - ein Argument übergeben werden. Dieses Argument ist eigentlich nur dann interessant, wenn das Programm innerhalb eines Batches aufgerufen wurde. Dieser Return-Code bestimmt nämlich den weiteren Ablauf der Batch-Datei. Je nach FAILAT-Level wird der Batch-Job abgebrochen - ausgegeben wird dieser Wert allerdings immer. Im Include-File libraries/dos.h finden Sie einige "Exit-

Return-Value"-Definitionen, die Sie je nach Schwere des aufgetretenen Fehlers verwenden sollten:

RETURN_OK (0)
Kein Fehler aufgetreten (Ende des Programms).

RETURN_WARN (5)
Warnung an Benutzer (z.B. Backup konnte nicht erstellt werden).

RETURN_ERROR (10)
Fehler (z.B. File konnte nicht geöffnet werden).

RETURN_FAI (20)
Schwerwiegender Fehler (z.B. Library oder Speicher konnten nicht geöffnet bzw. allokiert werden).

Doch beschäftigen wir uns noch einmal mit dem von Open() zurückgegebenen Dateischlüssel:

Wie Sie sicher bemerkt haben werden, wird dieser schon bei der Deklaration auf 0 gesetzt. Wahrscheinlich werden Sie jetzt sagen, daß dies recht sinnlos ist - schließlich sorgt Open() doch dafür, daß beim erfolglosen Öffnen einer Datei automatisch der Wert 0 an den Dateischlüssel übergeben wird. Aber stellen Sie sich einmal folgende hypothetische Situation vor:

Neben einem File werden auch mehrere Librarys geöffnet und Speicherblöcke allokiert. Nun stellt sich heraus, daß nicht mehr genügend Speicher vorhanden war. Das Programm muß also vorzeitig verlassen werden. Natürlich haben Sie es sich einfach gemacht und eine Routine geschrieben, die alles, was innerhalb Ihres Programms geöffnet oder allokiert wurde, wieder freigibt:

```
CloseAll()
{
   if (Memory != 0L)  FreeMem(Memory, ANZBYTES);
   if (GfxBase != 0L) CloseLibrary(GfxBase);
   if (Datei != 0L)   Close(Datei);
   exit((ULONG)RETURN_FAIL);
}
```

Stellen Sie weiterhin sich vor, daß der Fehler vor dem Open()-Aufruf auftritt, und in die obige Routine verzweigt wird. Nun wird festgestellt, daß der Dateischlüssel ungleich 0 ist, weil die Variable, die die Adresse des Dateischlüssels enthält, in einem uninitialisierten Datenbereich liegt. Leider wird jetzt eine Datei geschlossen, die gar nicht geöffnet wurde - und der Systemabsturz ist programmiert. Um dies zu vermeiden, wird schon bei der Deklaration aller wichtigen Variablen (Library-Basen, Speicher-Adressen, Dateischlüssel bzw. File-Handles) dafür gesorgt, daß diese auf 0 gesetzt werden. In der CloseAll()-Routine werden dann nur die Librarys, Speicherbereiche und File-Handles freigegeben, die tatsächlich allokiert bzw. angelegt wurden.

Wie Sie aus dieser CloseAll()-Routine schon ersehen können, werden geöffnete Dateien durch Close() geschlossen. Sie übergeben dem Close()-Befehl dazu einfach den von Open() erzeugten Dateischlüssel (bzw. die Adresse dieses Schlüssels).

In dieser CloseAll()-Routine können Sie das Programm dann mittels exit() verlassen. Dazu sollten Sie wissen, daß exit() von jeder Stelle im Programm aus aufgerufen werden kann.

Aber das Verlassen von Programmen ist in diesem Kapitel nicht das Hauptthema. Viel wichtiger ist, wie man mit Dateien umgeht.

Nach dem Öffnen einer Datei muß man schließlich Daten dort ablegen bzw. von dort auslesen können.

2.5.4.2 Daten lesen und schreiben

Zum Schreiben von Daten in eine Datei wird der Write()-Befehl benutzt. Diesem Befehl wird der von Open() gelieferte Dateischlüssel übergeben sowie die Adresse des Speicherbereichs, in dem die zu schreibenden Daten stehen. Damit der Write()-Befehl "weiß", wie viele Bytes geschrieben werden sollen, wird außerdem die Anzahl der zu schreibenden Bytes übergeben.

Ein Schreibaufruf sieht also folgendermaßen aus:

```
Write (Handle, Datenbereich, AnzBytes);
```

Zu beachten ist hierbei, daß die Anzahl der zu schreibenden Bytes als LONG-Wert angegeben werden muß.

Weiterhin ist der Rückgabewert der Write()-Funktion zu beachten. Dieser Rückgabewert gibt nämlich die Anzahl der tatsächlich geschriebenen Bytes an. Wie Sie sich leicht denken können, bestehen beim Schreiben von Daten viele Fehlerquellen. Die unangenehmsten Fehler dürften Schreib-/Lesefehler auf der Diskette sein. Einen Hinweis auf diesen Fehler erhalten Sie durch einen Systemrequester. Auch recht häufig dürfte die Meldung sein, die Ihnen mitteilt, daß die Kapazität der Diskette voll ausgeschöpft ist.

Wie beim Open()-Befehl müssen Sie auch beim Write()-Befehl auf solch einen Fehler reagieren. Feststellen, daß solch ein Fehler aufgetreten ist, können Sie ganz einfach daran, daß eine Diskrepanz zwischen der Anzahl der tatsächlich geschriebenen Bytes (Rückgabewert von Write() vom Typ LONG) und der Angabe der zu schreibenden Bytes (Parameter des Write()-Aufrufs) besteht.

```
if ((Geschrieben = Write(Handle, Daten, AnzBytes)) != AnzBytes)
{
   printf("Übertragungsfehler !!!
           Konnte nur %ld Bytes schreiben!\n", Geschrieben);
  /* exit(RETURN_ERROR) */
  /* oder CloseAll()    */
}
```

Das Lesen von Daten geschieht analog zum Schreiben. Nur daß hierbei ein anderer Befehl - nämlich Read() - verwendet wird. Ein korrekter Read()-Aufruf sähe also so aus:

```
if ((Gelesen = Read(Handle, Daten, AnzBytes)) != AnzBytes)
{
   printf("Übertragungsfehler !!!
           Konnte nur %ld Bytes lesen!\n", Gelesen);
  /* exit(RETURN_ERROR) */
  /* oder CloseAll()    */
}
```

Beachten sollten Sie, daß die Anzahl der zu lesenden Bytes die Kapazität Ihres LesePuffers nicht übersteigt. Angenommen Sie

haben einen Datenbereich von 1024 Bytes allokiert und wollen, daß dieser Bereich mit Daten aus einem File gefüllt wird. Wenn Sie nun den Read()-Befehl veranlassen, mehr als 1024 Bytes zu lesen und in diesen Speicherbereich zu schreiben, kann es passieren, daß wichtige Variablen oder sogar Programmcode von den gelesenen Bytes überschrieben wird - was zum Programmabsturz führen kann. Der Read()-Befehl hat nämlich gar keine Ahnung davon, wie groß der Speicherbereich ist, der die gelesenen Daten aufnehmen soll.

Ebenso verhält es sich mit dem Write()-Befehl. Auch dieser "weiß" nicht, wie viele Daten-Bytes der angegebene Speicherbereich enthält. Wenn Sie also eine größere Anzahl zu schreibender Bytes angeben, als der tatsächlich abzuspeichernde Speicherbereich enthält, so werden mit Sicherheit sinnlose Daten (z.B. Variablen, Programmcode etc.) mit abgespeichert.

Doch kommen wir noch einmal zur Fehlerbehandlung zurück. Die Tatsache, daß ein Fehler aufgetreten ist, kann ja daran festgestellt werden, daß die Anzahl der übertragenen Bytes von der Anzahl der Bytes, die Sie übertragen wollten, abweicht. Allerdings gibt dies nur Auskunft darüber, daß ein Fehler aufgetreten ist, mehr nicht. Wer mehr über den aufgetretenen Fehler erfahren will, muß auf die IoErr()-Funktion zurückgreifen.

Diese Funktion liefert den zuletzt aufgetretenen Fehler zurück. Dabei ist zu beachten, daß der Rückgabewert dieser Funktion im Gegensatz zur üblichen Verfahrensweise vom Type WORD ist. Wenn der Rückgabewert von IoErr() den Wert -1 hat, so besagt dies, daß kein Fehler aufgetreten ist.

Ist der Rückgabewert von IoErr() aber größer als 0, so ist bei der letzten Datenübertragung ein Fehler aufgetreten. Dies trifft für folgende Rückgabewerte zu:

Open

103 (insufficient free store)

Sie können nahezu beliebig viele Files öffnen. Abhängig ist die Anzahl der Files, die eröffnet werden können, nur von der Größe des noch freien Speichers. Schließlich belegt der Open()-Befehl Speicherplätze, um den Dateischlüssel (dessen Adresse Sie ja geliefert bekommen) anzulegen. Wenn hierfür nicht mehr genug Speicher vorhanden ist, wird Open() abgebrochen und von IoErr() dieser Fehler gemeldet.

202 (object in use)

Die Datei, die versucht wurde zu öffnen, hat schon ein anderes Programm eröffnet.

206 (invalid window)

Wie Sie vielleicht wissen, ist es mit dem Open()-Befehl auch möglich, Fenster zu öffnen (Open("CON:0/0/640/256/ WindowName", (ULONG)MODE_OLDFILE);). Wenn die Parameter, die Sie hier angeben, dazu führen, daß das Window nicht eröffnet werden kann, erscheint dieser Fehler.

210 (invalid stream component name)

Der angegebene Dateiname ist ungültig.

212 (object not of required type)

Sie haben versucht, ein Directory zu eröffnen (Open("SYS1:c", (ULONG) MODE_OLDFILE);).

218 (device not mounted)

Die angegebene Diskette (z.B. SYS1:) wurde auch nach Aufforderung durch einen Requester nicht eingelegt.

225 (not a DOS disk)

Die Diskette, die man anzusprechen versucht, ist nicht im Amiga-Format formatiert.

226 (no disk in drive)

Es ist keine Diskette im angegebenen Laufwerk eingelegt. Sie werden zunächst durch einen Systemrequester darauf aufmerksam gemacht. Wenn Sie der Aufforderung dieses Requesters, eine Diskette einzulegen, allerdings nicht folgen, tritt dieser Fehler auf.

Read/Write

213 (disk not validated)

Die Diskette, auf der sich das geöffnete File befindet ist noch nicht vom System anerkannt worden. Dieser Fehler tritt besonders dann auf, wenn Sie eine Diskete aus dem Laufwerk entfernen, wenn die rote Kontrollampe noch leuchtet. Diese Disk-Validation (Gültigmachung) ist notwendig nachdem Files geschlossen wurden. Nachdem nämlich Daten auf die Diskette geschrieben wurden, müssen diskettenineterne Informationen (belegte Blocks usw.) erzeugt und abgespeichert werden.

214 (disk is write protected)

Die Diskette ist schreibgeschützt. Natürlich kann dann nicht auf die Diskette geschrieben werden. Das Lesen ist allerdings möglich. Wenn Sie nach dem Auftauchen des entsprechenden Systemrequesters den Schreibschutz entfernen, tritt dieser Fehler nicht auf. Der Systemrequester verschwindet wieder, und die Daten werden ordnungsgemäß abgespeichert.

221 (disk full)

Die Diskette ist voll. Es passen keine weiteren Daten mehr auf die Diskette.

223 (file is protected from writing)

In die geöffnete Datei kann nichts hineingeschrieben werden.

224 (file is protected from reading)

Aus der geöffneten Datei kann nichts gelesen werden. (Diese beiden Fehlermeldungen (224 und 223) werden Sie allerdings erst ab Workbench 1.3 erhalten können.)

Was passiert aber, wenn Sie zunächst eine Diskrepanz zwischen der Anzahl der zu lesenden Bytes und der Anzahl der tatsächlich gelesenen Bytes fesgestellt haben, und die IoErr()-Funktion den Rückgabewert -1 liefert?

Hier kann doch offensichtlich etwas nicht stimmen. Schließlich haben Sie einen Lesefehler festgestellt, der mit Hilfe der IoErr()-Funktion näher bestimmt werden soll. IoErr() "sagt" Ihnen aber, daß kein Fehler aufgetreten ist. Was ist hier also los?

Nun, Sie haben ganz einfach das Ende der Datei erreicht. Alle in dem File enthaltenen Daten haben Sie gelesen.

Wie Sie also sehen, kann es gefährlich sein, sich einfach nur auf den Diskrepanzunterschied zur Fehlererkennung zu verlassen. Wird solch ein Unterschied nämlich festgestellt, wenn das File leergelesen wurde und dieser Unterschied dann als Fehler interpretiert, verlassen Sie vielleicht Ihr Programm, obwohl alles ordnungsgemäß abgelaufen ist.

Wenn Sie sich allerdings darauf verlassen, daß der Diskrepanzunterschied nur beim File-Ende auftritt, kann auch dies recht gefährlich werdenm, weil hier dann ein aufgetretener Fehler als End of File interpretiert wird, und das File unvollständig gelesen wurde.

Das ganze verliert natürlich dann seine Gefährlichkeit, wenn man genau weiß, wie lang das zu lesende File ist. Als erste Möglichkeit, dies zu erfahren, bietet sich an, beim Abspeichern der Daten dafür zu sorgen, daß als allererstes die Anzahl der abgespeicherten Daten in das File geschrieben wird:

```
ULONG Anzahl;
...
if (Write(FileHandle, &Anzahl, 4L) != 4L)
{
   /* Fehler */
   printf("Fehlercode: %d\n",IoErr());}
```

Beim Lesen können Sie dann zunächst dieses abgespeicherte Langwort einlesen:

```
ULONG Anzahl;

if (Read(FileHandle, &Anzahl, 4L) != 4L)
{
   /* hier ist auf jeden Fall ein Fehler aufgetreten      */
   /* da in dem File ja vorher die Anzahl abgespeichert   */
   /* wurde. Ein Abfrage von IoErr() ist also nicht       */
   /* unbedingt notwendig und das Programm kann verlassen */
   /* werden.                                             */
}
```

Das funktioniert allerdings nur dann, wenn Sie absolut sicher sein können, daß das zu lesende File auch in Ihrem Format - also mit dieser Längenangabe direkt zu Anfang - abgespeichert wurde. Wenn zwischendurch jemand dieses File manipuliert hat oder es einfach mit anderen Daten überschrieben wurde, funktioniert das nicht mehr.

Es ist also auch hier geboten, nach dem Lesen den Fehlerstatus mit IoErr() zu erfragen.

```
ULONG Anzahl;

if (Read(FileHandle, &Anzahl, 4L) != 4L)
{
  if (IoErr() != (WORD)-1)
  {
      /* Tatsächlicher Fehler (nicht End of File) */
      /* z.B. return(FALSE) */
  }
  else /* OK, Ende des Files */
       /* z.B. Verlassen einer Leseroutine mit 'return(TRUE)' */
}
```

Man kann nur in Verbindung mit IoErr() absolut sicher feststellen, ob ein File bis zum Ende gelesen wurde oder ein Fehler aufgetreten ist.

2.5.4.3 File-Informationen erfahren

Wie kann man aber feststellen, wie lang ein File ist? Man kann nämlich nicht davon ausgehen, daß in allen Files eine Längenangabe enthalten ist. Außerdem ist solch eine zusätzliche Längenangabe falsch. Man denke da an die IFF-Files, die universell

austauschbar sind und keinerlei Formatänderungen unterzogen werden sollten.

Derjenige unter Ihnen, der sich ein wenig mit den IFF-Files auskennt, wird sich jetzt wahrscheinlich sagen, daß in den einzelnen Bestandteilen eines IFF-Files die Länge dieser Bestandteile enthalten ist (Hunk-Längen). Was passiert jedoch, wenn ein Benutzer das gesamte IFF-File in den Speicher einlesen will, um es dort zu verarbeiten?

Eine Möglichkeit, die Länge eines Files zu erhalten, ist folgende:

```
ULONG AnzByte = 0L;
BYTE Buffer;
...

while (Read(FileHandle, &Buffer, 1L) == 1L) AnzBytes++;

if (IoErr() != (WORD)-1) printf("Fehler !!!\n");
else printf("OK\n Das File enthält %ld Bytes\n",AnzBytes);

...
```

Natürlich werden Sie sich sagen, daß diese Art der Längenbestimmung umständlich und zeitaufwendig ist. Es muß doch noch eine andere Möglichkeit geben, die Länge eines Files festzustellen. Schließlich wird die Länge eines jeden Files ja auch beim Anzeigen des Disketteninhalts mit Hilfe des LIST-Kommandos angegeben. Der LIST-Befehl liest ganz bestimmt nicht jedes einzelne Byte eines jeden Files. Er greift auf andere Informationen zurück. Aber auf welche? Hier müssen wir auf die Verwaltung eines Files auf der Diskette zu sprechen kommen.

Zunächst einmal muß auf der Diskette irgendwo abgespeichert sein, welche Files auf ihr enthalten sind. Wenn dies nicht so wäre, wäre eine Aufteilung des Diskettenspeicherplatzes auf mehrere verschiedene Dateien nicht möglich. Man hat sich jedoch nicht nur darauf beschränkt, den Namen eines Files abzuspeichern. Hier werden auch die Informationen über die Länge eines Files, das Erstellungsdatum sowie die Erstellungszeit und weitere Informationen (die Verteilung des Files auf die einzelnen Disketten-Sektoren) abgespeichert.

Um an diese Informationen eines einzelnen Files heranzukommen, sind einige Schritte nötig:

Zunächst einmal muß ein LOCK auf das zu untersuchende File geholt werden. Dies geschieht mit der FileLock-Struktur und dem Lock()-Befehl:

```
#include "libraries/dos.h"
#include "libraries/dosextens.h"

struct FileLock *FL = 0L;

...

   FL = (struct FileLock*) Lock(FileName, (ULONG)Access_Modus);
  if (FL == 0L)
   {
       printf("LOCK auf das File <%s>
              kann nicht geholt werden!!!\n",FileName);
  }
```

Beim Aufruf des Lock()-Befehls gilt es, zwei Acces_Modi - also Zugriffsarten - zu unterscheiden:

ACCESS_READ (-2)
Wenn Sie einen solchen LOCK (Zugriffsschlüssel) auf ein File holen, wird immer eine FileLock-Struktur erzeugt werden können - sofern das zu untersuchende File auch tatsächlich vorhanden ist.

Anders sieht es bei der zweiten Zugriffsart aus:

ACCESS_WRITE (-1)
Hier kann der Lock()-Befehl nur dann erfolgreich sein, wenn noch kein anderer Lock auf das File geholt wurde. Wenn Sie also innerhalb Ihres Programms schon den Zugriff auf dieses File gesichert haben, bleibt der Lock()-Befehl erfolglos. Beachten Sie hierbei, daß auch der Open()-Befehl mit Locks arbeitet, und diese erst nach dem Schließen des Files mit Close() oder bei einem Neustart freigegeben werden.

Sollten Sie sich einmal den Zugriff auf ein File mit Hilfe des Lock()-Befehls sichern wollen, sollten Sie unbedingt darauf

achten, daß dieser Zugriff nach Benutzung wieder freigegeben wird. Dies geschieht mit Hilfe des UnLock()-Befehls, dem einfach die vom Lock()-Befehl erzeugte FileLock-Struktur als Parameter übergeben wird. Sollten Sie einmal vergessen, den Lock wieder freizugeben, so haben Sie keine Möglichkeit mehr, auf das File zuzugreifen. Je nach Zugriffsart läßt sich das File gar nicht mehr Öffnen (bei ACCES_WRITE), oder man kann keine Daten mehr hineinschreiben (ACCES_READ).

Leider gibt die von Lock() erzeugte FileLock-Struktur noch nicht die Informationen preis, die wir eigentlich erhalten wollten. Um endlich an diese heranzukommen, müssen wir mit dem Examine()-Befehl arbeiten.

Diesem Befehl wird die von Lock() erzeugte FileLock-Struktur sowie die Adresse des Speicherbereichs übergeben, der die Informationen aufnehmen soll.

Mit diesem Speicherbereich hat es allerdings eine besondere Bewandtnis. Er muß an einer durch 4 teilbaren Adresse beginnen. Da dieser Speicherbereich auch als Struktur definiert worden ist, könnte man eigentlich denken, daß es reicht, diese Struktur in seinem Programm zu definieren:

```
...
struct FileInfoBlock FIB;
...
```

Leider ist so nicht gewährleistet, daß diese Struktur an einer durch 4 teilbaren Adresse beginnt, obwohl das erste Element dieser Struktur ein Langwort ist:

```
Offset      Struktur
------      --------

            struct FileInfoBlock
            {
  0 0x000      LONG fib_DiskKey;
  4 0x004      LONG fib_DirEntryType; /* Directory oder File */
  8 0x008      char fib_Filename[108];
116 0x074      LONG fib_Protection;
120 0x078      LONG fib_EntryType;
124 0x07c      LONG fib_Size; /* Größe */
128 0x080      LONG fib_NumBlocks; /* vom File belgte Sektoren */
```

```
132 0x084       struct DateStamp fib_Date; /* Erstellungszeit */
                   /* struct DateStamp
                      {
   132 0x084             LONG ds_Days;
   136 0x088             LONG ds_Minute;
   140 0x08c             LONG ds_Tick;
                      }
                   */
144 0x090       char fib_Comment[116]; /* Kommentar */
260 0x104   }
```

Wie Sie wissen, müssen Wörter und Langwörter nur an geraden Adressen beginnen. Gerade Adressen sind aber auch die Adressen 2, 6, 10, usw. Diese sind aber nicht durch 4 teilbar. Was ist also zu tun?

Hier hilft uns der AllocMem()-Befehl der EXEC-Library weiter, mit dem es möglich ist, Speicherbereiche zu allokieren. AllocMem() achtet dabei von selbst darauf, daß der allokierte Speicherbereich an einer durch 4 teilbaren Adresse beginnt. Wir brauchen also in unserem Programm nur einen Zeiger auf die FileInfoBlock-Struktur zu definieren und den von dieser Struktur benötigten Speicher mittels AllocMem() zu allokieren:

```
struct FileInfoBlock *FIB = 0L;

...

if ((FIB = (struct FileInfoBlock*)
          AllocMem((ULONG)sizeof(struct FileInfoBlock),
          (ULONG)MEMF_CHIP)) == 0L)
{
    printf("Kein Speicher für FIB !!!\n");
}
```

Wenn AllocMem() keinen Speicher mehr zur Verfügung stellen konnte, wird dem Zeiger, der auf den allokierten Speicherbereich zeigt, der Wert 0 übergeben. Ansonsten enthält dieser Zeiger die Anfangsadresse des allokierten Speichers.

Nun können wir endlich an die gewünschten Informationen herankommen. Wir brauchen nur noch mit Examine() die FileInformationen in den FileInfoBlock übertragen lassen und können diese dann auswerten. Damit Examine() auch weiß, welches File untersucht werden soll, muß hier auch der von Lock() erzeugte FileLock übergeben werden:

```
        if (Examine (FL, FIB) == 0L)
        {
           printf ("Fehler bei Examine !!!\n");
        }
```

Folgendes Programm wertet alle wichtigen Informationen über ein File aus. Neben dem File-Namen werden auch das Erstellungsdatum und die Erstellungszeit ausgegeben.

```
#include "exec/types.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

struct FileLock       *FL = 0L;
struct FileInfoBlock *FIB = 0L;

VOID *Lock();  /* Diese Funktionen liefern normalerweise ULONGs */
VOID *AllocMem(); /* und keine Zeiger zurück.                   */

ShowTime(Minute, Tick)
ULONG    Minute, Tick;
{
  printf("%02ld:%02ld:%02ld\n", Minute/60L, Minute%60L, Tick/50L);
                               /* Stunden, Minuten, Sekunden */
}

BOOL Schaltjahr (Year)
ULONG            Year;
{
   if ((((Year/4L)*4L) == Year) &&
       (((Year/100L)*100L) != Year)) return((BOOL) TRUE);
   else return((BOOL)FALSE);
}

ShowDate(Days)
ULONG    Days;
{
   ULONG Year;     /* Jahr  */
   ULONG Month;    /* Monat */
   ULONG WochenTag;

   static char *DayTable[7] = {"Sonntag", "Montag", "Dienstag",
               "Mittwoch", "Donnerstag", "Freitag", "Samstag"};

   static char *MonthTable[12] = {"Januar", "Februar", "März",
                                  "April", "Mai", "Juni",
                                  "Juli", "August", "September",
                                  "Oktober", "November", "Dezember"};

   static ULONG DaysOfMonth[]  = {31L,28L,31L,
                                  30L,31L,30L,
```

```
                                    31L,31L,30L,
                                    31L,30L,31L};

   WochenTag = Days % 7;

   Year = 1978L;

   while (Days > 366L)  /* Jahr berechnen */
   {
      if(Schaltjahr(Year)) Days -= 366L;     /* Schaltjahr */
      else                 Days -= 365L;

      Year++;                                /* ein Jahr mehr */
   }

   if ((Days == 365L) && !Schaltjahr(Year)) Days -= 365L;
                                        /* noch ein Jahr übrig? */

   if (Schaltjahr(Year)) DaysOfMonth[1] = 29L; /* Febr. updaten */
   else                  DaysOfMonth[1] = 28L;

   Month = 0L;

   while (Days > DaysOfMonth[Month])
   {
      Days -= DaysOfMonth[Month];
      Month++;
   }

   /* Days enthält Monatstag */

   Days++; /* Damit Days == 0L als 1. d. Monats ausgegeben wird */

   printf("%s, den %ld. %s im Jahre des Herrn %ld\n",
               DayTable[WochenTag], Days,
               MonthTable[Month], Year);
}

main(argc, argv)
int  argc;
char      *argv[];
{
   if (argc != 2)
   {
      printf("AUFURF: FileInfo FILENAME !\n");
      exit (10L);
   }

   if ((FL = (struct FileLock *) Lock(argv[1],(ULONG)ACCESS_READ))
            == 0L)
   {
      printf("Das File <%s> existiert nicht !!!\n", argv[1]);
      exit(0L);
```

```
    }

    if ((FIB = (struct FileInfoBlock *)
              AllocMem((ULONG)sizeof(struct FileInfoBlock),
                       (ULONG)MEMF_CHIP)) == 0L)
    {
       printf("Kein Speicher mehr !!!\n");
       UnLock(FL);
       exit(10L);
    }

    if (Examine(FL, FIB) == 0L)
    {
       printf("Fehler bei EXAMINE !!!\n");
       UnLock(FL);
       FreeMem(FIB, (ULONG)sizeof(struct FileInfoBlock));
       exit(10L);
    }

    printf("Das File enthält  %ld Bytes\n", FIB->fib_Size);
    printf("Erstellungsdatum: ");
    ShowDate(FIB->fib_Date.ds_Days);
    printf("Erstellungszeit:  ");
    ShowTime(FIB->fib_Date.ds_Minute, FIB->fib_Date.ds_Tick);
    printf("Protectionbits:   ");

    printf("%ld\n",FIB->fib_Protection);

    if ((FIB->fib_Protection & (ULONG)FIBF_READ) > 0L)
       printf("-");
    else
       printf("r");

    if ((FIB->fib_Protection & (ULONG)FIBF_WRITE) > 0L)
       printf("-");
    else
       printf("w");

    if ((FIB->fib_Protection & (ULONG)FIBF_EXECUTE) > 0L)
       printf("-");
    else
       printf("e");

    if ((FIB->fib_Protection & (ULONG)FIBF_DELETE) > 0L)
       printf("-\n");
    else
       printf("d\n");

    UnLock(FL);
    FreeMem(FIB, (ULONG)sizeof(struct FileInfoBlock));
}
```

Programm FileInfo.c

Wie Sie sehen, ist die Auswertung des Datums und der Zeit nicht ganz so einfach, weil diese beiden Informationen in einem speziellen Format abgespeichert werden. In der Variable, die das Erstellungsdatum des Files enthält (FIB->fib_Date.ds_Days) ist die Anzahl der Tage abgespeichert, die seit dem 1.1.1978 vergangen sind.

Ein Wert von 0 bedeutet also, daß das File am 1.1.1978 erstellt wurde. Bei einem Wert von 1 wurde das File am 2.1.1978 erstellt usw.

Wenn wir diese Variable in ein für uns verständliches Format - also Wochentag, Tag.Monat.Jahr - umrechnen wollen, sind einige Schritte notwendig.

Zunächst wird von der Datumsvariable solange die Anzahl der Tage eines Jahres (366 oder 365) abgezogen, bis die verbleibenden Anzahl der Tage kleiner als ein Jahr ist. Dabei gilt es zu beachten, daß auch Schaltjahre korrekt erfaßt werden.

Ein Schaltjahr ist ein Jahr, dessen Jahreszahl ohne Rest durch 4 teilbar ist. Allerdings sind alle hunderter Jahre, also die Jahre 1900, 2000, 2100 keine Schaltjahre, obwohl diese durch vier teilbar sind.

Wurde das Jahr berechnet, kann daran gegangen werden, den Monat zu berechnen. Auch hier werden die Tage der Monate solange von der verbleibenden Zahl abgezogen, bis eine Subtraktion negativ ausfallen würde (Achtung! Der Februar hat 28 oder 29 Tage). Bei jeder Subtraktion wird hier wie bei der Berechnung des Jahres eine Variable hochgezählt, die den aktuellen Monat angibt.

Würde die Subtraktion der Anzahl der Tage eines Monats negativ ausfallen, bedeutet das, daß die noch verbleibende Anzahl der Tage den Monatstag angibt. Doch Vorsicht! Wenn hier der Wert 0 übrigbleibt, würde das ja bedeuten, daß der 0. des errechneten Monats der Monatstag ist. Dies ist aber falsch. Der errechnete Tag ist der erste des Monats. Deshalb muß nach die-

ser Berechnung der Wert 1 addiert werden, um z.B. bei einem Wert von 0 den Ersten des Monats zu erhalten.

Um noch den Wochentag zu erhalten, braucht man den originalen Datumswert (also die volle Anzahl der vergangenen Tage seit dem 1.1.1978) nur einer Modulodivision durch 7 zu unterziehen. Der hier erhaltene Wert gibt den Wochentag an. Da der 1.1.1978 ein Sonntag war, bedeutet der Wert 0, daß der errechnete Wochentag ein Sonntag ist. Der Wert 6 bedeutet, daß das File an einem Samstag erstellt wurde.

Die so errechneten Werte können bei der Ausgabe des Datums als Indizes für verschiedene String-Arrays dienen (siehe ShowDate()).

Auch die Berechnung der Erstellungszeit ist ein wenig kompliziert. Die Erstellungszeit wird nämlich als Anzahl der vergangenen Minuten seit Mitternacht angegeben. Die Anzahl der vergangenen Sekunden der aktuellen Minute wird auch angegeben - allerdings in Systemticks, also 50stel Sekunden. Dabei ist der hier angegebene Wert immer ein Vielfaches von 50. Wenn die "Sekundenvariable" also den Wert 250 hat, bedeutet das, daß 5 Sekunden vergangen sind.

Um die Anzahl der Stunden und Minuten zu erhalten, braucht man die "Minutenvariable" (FIB->DateStamp.ds_Minute) nur durch 60 zu teilen (Stunden) bzw. einer Modulodivision durch 60 zu unterziehen (Minuten). Die Routine ShowTime() zeigt Ihnen, wie einfach man die Erstellungszeit ausgeben kann.

Unser obiges Programm kann aber noch mehr, als einfach das Erstellungsdatum, die Erstellungszeit angeben. Es gibt auch die in dem untersuchten File enthaltenen Bytes an. Dazu wird die Variable FIB->fib_Size herangezogen.

Aber auch das ist noch nicht alles. Unser Programm gibt auch die Protection-Bits aus. Diese Bits bestimmen, was man mit einer Datei bzw. einem File alles machen darf. Ist die Variable FIB->fib_Protection gleich 0, bedeutet das, daß das File gelesen (r), geschrieben (w), ausgeführt (e) und gelöscht (d) werden

darf. Dabei wird durch verschiedene Bits bestimmt, was mit der Datei geschehn darf:

```
#define FIBB_READ    3  /* Bitnummern */
#define FIBB_WRITE   2
#define FIBB_EXECUTE 1
#define FIBB_DELETE  0

#define FIBF_READ    (1<<FIBB_READ) /* Masken */
#define FIBF_WRITE   (1<<FIBB_WRITE)
#define FIBF_EXECUTE (1<<FIBB_EXECUTE)
#define FIBF_DELETE  (1<<FIBB_READ) /* Siehe "libraries/dos.h" */
```

Ist eines dieser Bits gesetzt, darf die entsprechende Operation nicht ausgeführt werden. Wenn es jedoch gelöscht ist, darf man alles mit dem File machen.

Leider funktioniert bisher nur das DELETE-Flag. Wenn also dieses Bit in der Protection-Maske gesetzt ist, heißt das, daß das File nicht gelöscht werden darf. Auch ein Open()-Befehl - mit der MODE_NEWFILE-Option aufgerufen - bleibt erfolglos.

Bleibt nun noch die Frage, was man all diesen Informationen anfangen kann.

Wenn Sie mit dem Aztec-Compiler arbeiten, ist Ihnen sicher das Make-Untility bekannt (es wurde ja auch zu Beginn dieses Buches beschrieben). Dieses Utility arbeitet aufgrund der Erstellungszeiten von Files.

Wenn Make z.B. feststellt, daß das C-Cource-File eines Programms jünger als das vom Linker erzeugte Objekt-File ist, wird damit begonnen, das wahrscheinlich vor kurzem geänderte C-Source zu kompilieren. Im Anschluß daran wird dann meist festgestellt, daß das ablauffähige Programm älter als das Objekt-File ist, so daß hier das ablauffähige Programm vom Linker neu erzeugt wird.

Sollten Sie das Make-Utility verwenden, müssen Sie darauf achten, daß die Systemzeit immer stimmt. Nach dem Anschalten des Rechners ist es 00 Uhr 00 am 1.1.1978. Mit date können Sie aber dafür sorgen, daß die aktuelle Systemzeit korrekt gesetzt wird.

Daß die Systemzeit stimmt, ist für Make deshalb so wichtig, da bei unkorrekten Systemzeiten die Erstellungsdaten der Files durcheinandergeraten könnten. Stellen Sie sich vor, Sie erstellen ein File am 10.11.1988. Der Rechner stürzt beim Aufruf des erzeugten Programms jedoch ab und wird neu gebootet. Wenn Sie nun vergessen, die richtige Zeit einzustellen und dann den C-Source ändern, so stellt Make fest, daß der nun geänderte C-Source älter als das Objekt-File und das ablauffähige Programm ist. Es muß - laut Make - also nicht neu compiliert werden. Natürlich ist das falsch, schließlich haben Sie den Fehler behoben und wollen, daß das neue Programm erzeugt wird. Sie sehen also, daß das Setzen der Systemzeit insbesondere bei Datums- und zeitabhängigen Operationen von großer Wichtigkeit ist.

Doch haben wir weitaus mehr Informationen als nur das Erstellungsdatum erhalten, z.B. die Größe eines Files. Anhand dieser kann z.B. festgestellt werden, ob das File vollständig in den Speicher eingelesen werden kann, um dort bearbeitet zu werden. Der CLI-Editor ED geht z.B. so vor: ohne weitere Angaben stellt ED Ihnen 20000 Bytes Speicher für ein zu editierendes File zur Verfügung. Wenn Sie nun ein größeres File mit dem ED bearbeiten wollen, wird ED Sie darauf hinweisen, daß nicht soviel Speicher zur Verfügung steht. Sie müssen ED also "sagen", daß er mehr Speicher für Ihr File zur Verfügung stellen soll (z.B. ED File.c 30000).

Allerdings war auch dies noch nicht alles. Der FileInfoBlock enthält nämlich auch Informationen darüber, ob es sich bei dem mit Lock() anvisierten File-Namen wirklich um ein File oder ein Directory handelt. Ist die Variable FIB->fib_DirEntryType nämlich größer als 0, so handelt es sich hier um ein Directory!

Wie Sie sehen, enthält die FIB-Struktur auch ein Element namens fib_Comment. In diesem CHAR-Array ist der sogenannte File-Kommentar abgespeichert. Obwohl Ihnen über das CLI mit dem Kommando SetComment nur 80 dieser 116 Bytes zugängig gemacht woprden sind, können hier bis 116 Zeichen gespeichert werden. Sie können hier z.B. Informationen über das File wie "Autor: MEIN NAME (c) ICH" abspeichern.

Weniger interessant ist die Tatsache, daß der FileInfoBlock auch den File-Namen enthält. Obwohl dieser nur maximal 30 Zeichen enthalten darf, werden Ihnen hier 108 Zeichen (bzw. Bytes) zur Verfügung gestellt.

Aber ist diese Angabe wirklich so uninteressant? Nein, natürlich nicht. Mit Hilfe dieser Angaben kann das DIR-Kommando nämlich alle Files und Unterverzeichnisse einer Diskette angeben:

2.5.5 Directorys anzeigen lassen

Dazu wird allerdings ein anderer Befehl als Examine() verwendet - nämlich ExNext(). Dieser Befehl ist Examine() zwar sehr ähnlich, aber er übernimmt noch einige Vorarbeiten. Betrachten wir dazu einmal die FileLock-Struktur, die mit Hilfe des Lock()-Befehls erzeugt wird:

```
Offset  Struktur
------  --------
        struct FileLock
        {
 0  0x00    BPTR fl_Link;   /* Zeiger auf nächsten Lock */
 4  0x04    LONG fl_Key;
 8  0x08    LONG fl_Access; /* Zugriffsart */
12  0x0c    struct MsgPort *fl_Task;
16  0x10    BPTR fl_Volume;
20  0x14 };
```

Die meisten Informationen, die diese Struktur bereithält, sind für uns von geringem Interesse. Das Element fl_Link jedoch beinhaltet eine schwerwiegende Information - nämlich die, wo die nächste FileLock-Struktur zu finden ist.

Stellen Sie sich dazu alle Files und Unterdirectorys einer Diskette verkettet in einer einfach verketteten Liste vor. Man kann sich nun - ausgehend von einem beliebigen Punkt dieser Liste - bis zum Ende der Liste durchhangeln. Dies - und nichts anders - erledigt die ExNext()-Funktion. Sie ermittelt die nächste FileLock-Struktur und führt darauf hin Examine() aus. Nun haben Sie alle Informationen über das nächste File.

Das folgende Programm ermöglicht es Ihnen, alle nachfolgenden Files - ausgehend von einem, das Sie angegeben haben - auszugeben:

```
#include "exec/types.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

VOID *AllocMem();
VOID *Lock();
VOID *CurrentDir();
VOID *Output();

struct FileInfoBlock *FIB     = 0L;
struct FileHandle *OutputFile = 0L;
BOOL   Error                  = FALSE;

VOID GetDir     (FL)    /* Verzeichnis scannen */
struct FileLock *FL;
{
   if (!Error)
   {
      if (Examine(FL, FIB) == 0L)  /* Erst Examine, dann ExNext */
      {
         Write(OutputFile, "Directory zerstört !\n", 21L);
         Error = TRUE;
         goto End1;
      }

      while (ExNext(FL, FIB) != 0L)
      {
         if (FIB->fib_DirEntryType > 0)   /* Directory */
         {
            Write(OutputFile, "      ", 6L);
            Write(OutputFile, FIB->fib_FileName, (ULONG)
                        strlen(FIB->fib_FileName));
            Write(OutputFile, " (dir)\n", 7L);
         }
         else
         {
            Write(OutputFile, "  ", 2L);  /* File */
            Write(OutputFile, FIB->fib_FileName, (ULONG)
                        strlen(FIB->fib_FileName));
            Write(OutputFile, "\n", 1L);
         }
      }
   }
End1:   if (FL != 0L)  UnLock(FL);
}

main(argc, argv)
```

```
int  argc;
char      *argv[];
{
   struct FileLock *FL = 0L;
   OutputFile = Output();

   FL = (struct FileLock*) Lock(argv[1], (ULONG)ACCESS_READ);

   if (FL == 0L)  /* Lock auf Dir holen und testen */
   {
      exit(0L);
   }

   FIB = (struct FileInfoBlock*)
      AllocMem((ULONG) sizeof(struct FileInfoBlock), (ULONG)
                        (MEMF_CHIP|MEMF_CLEAR));

   if (FIB == 0L)
   {
      Write(OutputFile, "Kein Speicher mehr !!!\n",24L);
      UnLock(FL);
      exit((ULONG)RETURN_FAIL);
   }

   GetDir(FL);

   if (FL != 0L)  UnLock(FL);
   if (FIB != 0L) FreeMem(FIB,(ULONG)sizeof(struct
                           FileInfoBlock));
}
```

Programm FileDir.c

Wie Sie sehen, erfolgt der Aufruf von ExNext() mit denselben Übergabeparametern wie Examine(). Hat ExNext() den letzten File-Eintrag erreicht, sorgt ein erneuter Aufruf dieser Funktion für den Rückgabewert 0. Daran erkennen Sie, daß alle Files untersucht wurden. (ExNext() sollte übrigens erst nach einem vorherigen Examine() aufgerufen werden.)

Beachten Sie, daß diesem Programm der Pfadname eines auszugebenden Directorys übergeben werden muß! Wenn Sie z.B. FileDir df0:c/RUN aufrufen, stürzt Ihr Rechner ab. (In den nächsten beiden Unterkapiteln werden wir Ihnen jedoch ein Programm vorstellen, das alle Files - auch die in weiteren Unterverzeichnissen enthaltenen - ausgibt und testet, ob tatsächlich der Pfadname zu einem Directory angegeben wurde.)

Wie Sie sicher festgestellt haben, ist das, was dieses Programm erzeugt, ein wenig anders als das, was DIR zum Vorschein bringt. Dies liegt aber einfach daran, daß wir die Files so ausgeben, wie Sie auf der Diskette angelegt worden sind. Das DIR-Kommando sortiert alle Files und Verzeichnisse und gibt diese dann in alphabetischer Reihenfolge aus. Auf diesen Sortiervorgang haben wir hier verzichtet, so daß dieses Programm ein wenig schneller arbeitet als DIR.

Ein kleiner Hinweis: Sicher haben Sie durch die Geräusche Ihres Diskettenlaufwerkes festgestellt, daß der Schreib-/Lesekopf recht häufig neu positioniert werden muß. Dies liegt daran, daß bei der Neuanlage eines Files die Informationen dieses Files ans Ende der Disketten-internen Liste geschrieben werden. Wenn Sie ein File neu anlegen, das vorher unter gleichem Namen schon existierte, wird die neue FIB-Struktur ans Ende der Liste gesetzt und die alte gelöscht. Wenn Sie dann ein weiteres File anlegen, wird der vorher freigewordene Bereich mit dem FileInfoBlock gefüllt.

So entsteht nach einiger Zeit ein ziemliches Durcheinander auf Ihrer Diskette, was umfangreiche Schreib-/Lesekopf-Positionierungen erforderlich macht.

2.5.5.1 Directory anzeigen - inklusive aller Unterverzeichnisse, Wildcards und Unterbrechungen

Sie werden jetzt sicher sagen, daß das obige Programm (FileDir.c) nichts Besonderes ist. Es ist vielleicht interessant, eine gewisse Vorstellung davon zu bekommen, wie Files auf Diskette abgespeichert werden, welche Zusatzinformationen existieren und wie man diese auswertet. Der DIR-Befehl des CLI kann aber doch wesentlich mehr. So haben wir ja mit Hilfe von DIR opt a die Möglichkeit, alle Files eines Verzeichnisses anzeigen zu lassen. Hier werden auch die Files angezeigt, die sich in evt. vorhandenen Unterverzeichnissen befinden.

Auch können wir mit Wildcards arbeiten. Der Aufruf DIR c#? sorgt z.B. dafür, daß alle Files des aktuellen Verzeichnisses, de-

ren erster Buchstabe das c, bzw. C ist (Groß- und Kleinschreibung haben hier keinen Einfluß !), ausgegeben werden. Oder z.B Dir c?. Hier werden z.B. alle zweibuchstabigen Files ausgegeben, die mit c beginnen. Ausgegeben werden hier z.B. cd, cc usw. Anders gesagt: Wir können beliebige Buchstaben durch ein ? ersetzen (Joker), und beliebige Zeichenketten (auch leere) durch #? (Wildcard).

Was beim DIR-Kommando auch recht angenehm auffällt, ist die Tatsache, daß man die Abarbeitung des Befehls durch Ctrl-C unterbrechen kann. Auch dies ist bei unserem Programm FileDir nicht möglich.

So stellen sich uns also drei Probleme:

Problem 1:

Wie werden alle Files - auch die in anderen Unterverzeicnissen ausgegeben?

Problem 2:

Wie programmiert man die Ausgabe der File-Namen in Abhängigkeit von Patterns (Strings, die Wildcards und Joker enthalten)?

Problem 3:

Wie erreicht man, daß ein Programm durch Ctrl-C abgebrochen wird?

Zu Problem 1: Pfadangaben

Die Untersuchung aller Unterverzeichnisse einer Diskette erweist sich leider als nicht so einfach, wie der eine oder andere unter Ihnen denken könnte. Die Ausgabe aller Files beruht zwar auf einem rekursiven Algorithmus, aber es reicht leider nicht, die Routine, die die Files ausgibt (wir nennen sie von nun an GetDir(), siehe FileDir), beim Feststellen, daß der Name eines Unterverzeichnises ausgegeben werden soll, sich einfach selbst wieder aufrufen zu lassen.

Wenn wir also entdeckt haben, daß wir gerade den Namen eines Unterverzeichnisses ausgeben, müssen wir uns auf dieses Unterverzeichnis einen neuen Lock holen, weil erst dann auf die in diesem Verzeichnis enthaltenen Files zugegriffen werden kann. Dazu ist aber der Pfadname notwendig. Angenommen, Sie befinden sich im Verzeichniss df0:include und treffen bei der Ausgabe nun auf das Verzeichnis exec. Wenn Sie die in diesem Verzeichnis enthaltenen Files ausgeben wollen, müssen Sie sich einen Lock auf das Verzeichnis exec holen. Was passiert aber, wenn sich innerhalb dieses Unterverzeichnisses noch weitere Verzeichnisse befinden? Um die Files dieser Verzeichnisse auszugeben müssen Sie also wieder einen Lock auf das auszugebende Verzeichnis holen. Da Sie sich aber immer noch im Verzeichnis df0:include befinden, müssen Sie hier dem Lock()-Befehl z.B. schon den Pfadnamen exec/exec_weitere angeben. Sie müssen also immer dafür sorgen, daß der Pfadname aufgearbeitet wird.

In bezug auf eventuelle spätere Erweiterungen des DIR-Befehls (eingebaute Delete- und Rename-Funktionen, die Sie, wenn Sie Lust haben, selbst entwickeln können) wollen wir dafür sorgen, daß uns immer der vollständige Pfadname eines auszugebenden Files bekannt ist. Dies erfordert allerdings ein wenig Rechnerei mit Strings.

Auch muß schon zu Beginn der Pfadname erzeugt werden. Dazu ein Beispiel: Sie befinden sich wieder im Verzeichnis df0:include. Um die in diesem Verzeichnis enthaltenen Files ausgeben zu lassen, brauchen Sie bekanntlich nicht DIR df0:include aufzurufen. Es reicht der einfache Aufruf von DIR, der die Files des aktuellen Verzeichnisses ausgibt (hier eben df0:include). Auch unser Programm soll so komfortabel sein.

Dazu ist es aber unbedingt notwendig, auch beim Aufruf unseres Programms ohne Pfadangaben (DIR), den Pfadnamen des gerade aktuellen Verzeichnisses zu ermitteln.

Wenn wir diesen Pfadnamen ermittelt haben, müssen wir immer dann, wenn wir die Files eines Unterverzeichnisses ausgeben wollen, veranlassen, daß dieser Pfadname um den Namen des

auszugebenden Unterverzeichnisses erweitert wird. Nach der Ausgabe der Files muß der Name des Unterverzeichnisses wieder entfernt werden.

Sie werden sich sicher schon denken können, daß der String, der den Pfadnamen enthalten soll, ein globaler String ist. Auch werden wir eine globale Variable einführen müssen, die uns immer mitteilt, an welcher Stelle der letzte Buchstabe in diesem String bzw. CHAR-Array steht. Damit wir nach der Ausgabe der Unterverzeichnis-Files den Namen des jeweiligen Unterverzeichnisses wieder entfernen können, müssen wir eine Variable zur Verfügung stellen, die uns Auskunft darüber gibt, wie viele Buchstaben zum alten Pfadnamen hinzugekommen sind. Diese Variable muß natürlich lokal sein - also bei jedem Rekursionsaufruf neu angelegt werden.

Wenn bei der Ausgabe des Directorys also auf ein Unterverzeichnis gestoßen wird, das ausgegeben werden soll, so muß man den globalen Pfadnamen-String um den Namen des auszugebenden Unterverzeichnisses erweitern. Nachdem das Unterverzeichnis ausgegeben wurde, wird der vorher zugefügte Name wieder entfernt.

Zu Problem 2: Wildcards und Joker

Es ist mitunter sehr sinnvoll, eine Selektion der auszugebenden Files und Unterverzeichnisse zu treffen. Was ist z.B., wenn alle C-Programm-Sources aufgelistet werden sollen? Mit dem obigen Programm (FileDir) kann man nur alle Files eines Verzeichnisses anzeigen lassen.

Um solch eine Selektion vornehmen zu können, müssen wir also dafür sorgen, daß die File-Namen vor der Ausgabe untersucht werden, ob Sie auch ausgegeben werden sollen.

Wie wird aber erkannt, ob ein File-Name ausgegeben werden soll oder nicht? Zunächst einmal muß der String definiert werden, der bestimmt, nach welchem Muster (engl. Pattern) die Files ausgegeben werden sollen. Wenn dieser bekannt ist, können

die File-Namen darauf untersucht werden, ob sie auf das angegebene Pattern passen oder nicht.

Ein solcher Pattern-String kann folgende "Zeichen" enthalten:

#? steht für beliebige Zeichenketten

? steht für einen beliebigen Buchstaben.

Um z.B. alle C-Sources auszugeben, muß das Pattern #?.c angegeben werden. Bei der Ausgabe der File-Namen werden nur die Files berücksichtigt, die am Ende die Buchstabenkombination .c enthalten.

Um z.B. nur die Files Temp1.c, Temp2.c, ..., Temp9.c auszugeben, muß nur das Pattern Temp?.c angegeben werden.

Doch betrachten wir einmal, wie man vorgehen muß, um diese Wildcards zu programmieren:

Nehmen wir an, daß der Wildcard-String folgendes Aussehen hat:

```
c#?b?.c
```

Übersetzt für die Ausgabe des Directorys bedeutet dieser String:

> Gib alle File-Namen aus, die mit einem c beginnen, auf das ein beliebiger String (auch ein leerer String) folgt, auf den wiederum der Buchstabe b folgt, der wiederum von einem beliebigen Buchstaben und der Kombination .c gefolgt wird.

Um nun festzustellen, ob ein File-Name auf dieses Pattern paßt, muß der Wildcard-String nach folgenden Schema untersucht werden:

1. Ist das aktuelle Zeichen ein Fragezeichen?
2. Wenn ja, ignoriere das aktuelle Zeichen des File-Namens. Gehe zu 1.
3. Wenn Nein: Ist ein Wildcard #? das aktuelle Zeichen?

4. Wenn Ja: Ermittle den String, der zwischen dem aktuellen und einem evt. folgenden Wildcard steht, und suche nach diesem String. Wenn dieser String nicht gefunden werden konnte, erfüllt der File-Name nicht die Wildcard-Bedingungen. Wenn der String gefunden wurde, gehe zu 1.
5. Wenn das Zeichen nicht der Joker oder ein Wildcard ist, vergleiche das aktuelle Zeichen des Wildcard-Strings mit dem aktuellen Zeichen des File-Namens. Wenn beide gleich sind, gehe zu 1. Wenn nicht, paßt der File-Name nicht auf das Pattern.

Diese Schritte werden solange ausgeführt, bis entweder festgestellt wurde, daß der File-Name nicht auf das Pattern paßt, oder einer bzw. beide Strings vollständig abgearbeitet wurden.

Zu Problem 3: Ctrl-C abfragen

Kommen wir nach den Wildcards zu dem Abfangen von Ctrl-C. Nach dem Betätigen dieser Tastenkombination muß das Programm beendet werden. Das Feststellen dieser Tastenkombination geht recht einfach vonstatten. Dazu müssen wir nur das Signal-System der Amiga-Tasks für unsere Zwecke ausnutzen.

Wie Sie vielleicht wissen, werden zur Kommunikation zwischen Tasks sogenannte Signale verwendet, die dem Empfänger-Task mitteilen, was für eine Nachricht an ihn gesandt wurde.

Normalerweise wird beim Feststellen, daß ein Signal gesetzt wurde, eine Nachricht abgeholt. Dies ist bei der Abfrage von Ctrl-C allerdings nicht notwendig.

Hier reicht es, einfach nur festzustellen, ob ein bestimmtes Signal gesetzt wurde. Derjenige unter Ihnen, der sich ein wenig mit Tasks und Signalen auskennt, wird jetzt sicher fragen, woher wir wissen, welches der 32 Signale, die zur Verfügung stehen, benutzt wird. Nun, das ist recht einfach, denn die Signale 0 bis 15 werden vom System festgelegt. Diese Signale haben also immer dieselbe Bedeutung, während die Signale 16 bis 31 vom Benutzer ihre Bedeutung zugewiesen bekommen.

Wenn also Ctrl-C gedrückt wurde, wird an den Task der gerade abläuft, das Signal mit der Nummer 12 gesendet.

Wie stellt man aber fest, welches Signal gesendet wurde? Nun, dazu greifen wir auf eine Routine zurück, die eigentlich dazu gedacht ist, die aktuellen Signale zu verändern:

```
AlteSignale = SetSignal(NeueSignale, Maske);
```

Diese Routine überträgt den Zustand von NeueSignale in die Taskstruktur - allerdings nur dann, wenn auch in Maske das jeweilige Bit gesetzt ist. Um z.B. das Signal mit der Nummer 0 zu setzen, muß sowohl Bit 0 in NeueSignale als auch in Maske gesetzt sein. Um das Signal 0 zu löschen, muß das Bit 0 in NeueSignale gelöscht, in Maske aber gesetzt sein. Der Rückgabewert dieser Funktion schließlich gibt den Zustand der Signale vor dieser Veränderung zurück.

Um den Zustand der Signale zu erhalten, ohne diese zu verändern, müssen wir SetSignal() nur wie folgt aufrufen:

```
AktuelleSignale = SetSignal(0L, 0L);
```

Um nun festzustellen, ob Ctrl-C gedrückt wurde, muß nur das Bit 12 untersucht werden. Ist es gesetzt, wurde Ctrl-C gedrückt:

```
#define SIG_CTRL_CB 12L /* Bitnummer */
#define SIG_CTRL_C  (1<<12L)

...

AktuelleSignale = SetSignal(0L, 0L);

if ((AktuelleSignale & SIG_CTRL_C) == SIG_CTRL_C)
{
   /* BREAK */
}
```

Vor der Ausgabe eines File-Namens brauchen wir also nur zu testen, ob Ctrl-C gedrückt wurde, und müssen dann das Programm verlassen.

Da die Ausgabe der File-Namen aber rekursiv sein kann (wenn alle Unter-Directorys ausgegeben werden), können wir das Pro-

gramm nicht einfach mit exit(10L) o.ä. verlassen, da so Speicher, der für die FIBs allokiert wurde, verloren geht. Deshalb wird beim Erkennen von Ctrl-C eine Fehlervariable gesetzt und ans Ende der Ausgaberoutine gesprungen. Hier wird dafür gesorgt, daß die Speicherplätze wieder freigegeben werden.

Wenn während eines Rekursionsaufrufes Ctrl-C gedrückt wird (oder ein Fehler auftritt, was durch den gleichen Mechanismus zum Programmabbruch führt), wird dies nach dem rekursiven Aufruf der Funktion anhand der globalen Fehlervariablen festgestellt.

Da sich dies alles ein wenig verwirrend anhört, wollen wir Ihnen nun das Listing des Directory-Befehls zeigen, der mit Directory [Verzeichnis] [-a] aufgerufen wird.

```
#include "exec/types.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

VOID *Lock();
VOID *AllocMem();
VOID *ParentDir();
VOID *Open();
VOID *Output();

struct FileHandle *OutputFile = 0L;
                        /* Für Ausgabe */

char  Path[256];        /* Enthält vollständigen Path-Namen */
ULONG GlobalPos = 0L;   /* Enthält akt. Position in Path-String */
BOOL  Error     = (BOOL) FALSE;  /* Globale Fehlervariable */
ULONG Verschachtellung = 0L;   /* muß Global sein, da diese Variable */
                               /* innerhalb der rekursiven Funktion   */
                               /* GetDir() benutzt wird, und dort die */
                               /* aktuelle Verschachtellungstiefe der */
                               /* Directorys angibt.          */

char WildcardString[100]; /* Dieses Array enthält den Wildcard-String */
                          /* extrahiert aus Kommando-Parameter.       */

#define SIG_CTRL_CB  12L      /* Bitnummer */
#define SIG_CTRL_C  (1<<12L)  /* Maske     */

BOOL CTRL_C()
{
```

```
   /* Beim Betätigen von Ctrl-C wird an den Task das Signal */
   /* Nummer 12 gesandt (siehe #defines).                   */

   if ((SetSignal(0L,0L) & SIG_CTRL_C) == SIG_CTRL_C)
      return(TRUE);
   else
      return(FALSE);
}

GetWildcardString(Name)
char             *Name;
{
   int i;
   int Len = 0;

   BOOL WildcardFound = FALSE;

   i = strlen(Name);

   while((Name[i] != ':') && (Name[i] != '/') && (i>=0))
   {
      Len++;                  /* Länge des Wildcard-Strings */

      if (Name[i--] == '?') WildcardFound = TRUE;
   }

   /* Suche von hinten nach vorne, ob Wildcard oder Joker */
   /* enthalten ist. (Es braucht nur nach einem ? gesucht */
   /* werden, da dieses Zeichen Bestandteil beider ist)   */
   /* Wenn kein Wildcard oder Joker gefunden wird, ist    */
   /* der Teil vor : oder / Directoryname.                */

   if (WildcardFound)
   {
      i = strlen(Name);

      while(Len > 0)
      {
         WildcardString[Len-1] = Name[i--];
         Len--;
      }
      Name[i+1] = '\000';
   }
   else
   {
      WildcardString[0] = '#';
      WildcardString[1] = '?';
      WildcardString[2] = '\000';
   }
}

/* Such-String für Wildcard ermitteln */
```

```
GetSearchString(Pattern, SearchString)
char          *Pattern,*SearchString;
{
   int i = 0;
   int j = 0;

   while((Pattern[i] != (char)0)   &&
         (Pattern[i] != (char)'#') &&
         (Pattern[i] != (char)'?'))
                  /* Bis zu nächsten Wildcard suchen */
      SearchString[j++] = Pattern[i++];

   SearchString[j] = (char)0;
}

/* Nach Such-String suchen */

BOOL Search(Original, SearchIt)
char       *Original,*SearchIt;
{
   int i = 0;                  /* Suche im original String nach */
   int j;                      /* ermitteltem Such-String.      */

   while(Original[i] != (char)0)
   {
      if(Original[i] == SearchIt[0]) /* erstes Zeichen gefunden */
      {
         j=0;                        /* Stimmt Rest des Strings ? */

         while(SearchIt[j] !=(char)0)
         {
            if ((Original[i++]|0x20) !=
                (SearchIt[j++]|0x20))
                return((BOOL)FALSE);
         }

         return((BOOL)TRUE);
      }
      else i++;
   }
   return((BOOL)FALSE);
}

/* Paßt Pattern (#?, ?) auf String ? */

BOOL Wildcard(Pattern, Match)
char         *Pattern,*Match; /* Pattern: Wildcard String */
                              /* Match: String der untersucht werden
soll */
                              /* ob er auf Pattern |passt|
*/
{
   int i = 0;
   int j = 0;
```

```
   char SearchString[256];
   BOOL Longer;

   while (Pattern[i] != (char) 0)
   {
      if ((Pattern[i] == '#') && (Pattern[i+1] == '?'))
                              /* Wildcard universal gefunden */
      {
         if (Match[j] == (char) 0)  return((BOOL) TRUE);
                              /* Ende des Match- */
         i+=2;                /* Wildcard überlesen */
         GetSearchString(&Pattern[i],SearchString);

         if (SearchString[0] != (char)0) /* Leerer SearchString */
            if (Search(&Match[j],SearchString) == (BOOL)FALSE)
               return((BOOL)FALSE);  /* Search-String nicht gefunden! */

         i += strlen(SearchString);
                        /* Ab gefundenem Search-String weitermachen */
         j += strlen(SearchString);
         Longer = FALSE;
      }
      else
      if (Pattern[i] == '?')
                       /* Wildcard one shot (Joker) gefunden */
      {
         if (Match[j] == (char) 0)  return((BOOL) FALSE);
                                    /* Ende des Match- */
         i++;                       /* Strings ?       */
         j++;
         if (Match[j] != (char)0) Longer = TRUE;
                                 /* Nach ? folgt noch ein Zeichen */
         else Longer = FALSE;

         if (Match[j+1] != (char)0) Longer = TRUE;
      }
      else
      if ((Pattern[i++]|0x20) != (Match[j++]|0x20))
                                                  /* KLEINBUCHSTABEN */
            return((BOOL)FALSE);
         else Longer = FALSE;
   }

   return((BOOL)!Longer);
}

BOOL Directory(DirName)
char          *DirName;
{
   struct FileLock       *FL  = 0L;
   struct FileInfoBlock *FIB = 0L;

   BOOL   Result  = TRUE;
```

```
   FL = (struct FileLock*) Lock(DirName, (ULONG)ACCESS_READ);

   if (FL == 0L)  /* Lock auf Dir holen und testen */
   {
      Write(OutputFile, "Wrong path !!!\n",16L);
      Result = FALSE;
      goto DirEnd2;
   }

   FIB = (struct FileInfoBlock*)
      AllocMem((ULONG) sizeof(struct FileInfoBlock), (ULONG)
(MEMF_CHIP|MEMF_CLEAR));

   if (FIB == 0L)
   {
      Write(OutputFile, "Kein Speicher mehr !!!\n", 23L);
      Result = FALSE;
      goto DirEnd2;
   }

   if (Examine(FL, FIB) == 0L)
   {
      Write(OutputFile, "Directory zerstört !\n", 21L);
      Result = FALSE;
      goto DirEnd2;
   }

   if (FIB->fib_DirEntryType <= 0L)
   {
      Write (OutputFile, "Achtung! Kein Directory!\n", 26L);
      Result = FALSE;
   }


DirEnd2:
   if (FIB != 0L) FreeMem(FIB,(ULONG)sizeof(struct FileInfoBlock));
DirEnd1:
   if (FL != 0L)  UnLock(FL);

   return (Result);
}


/* Strings der Verzeichnisse aneinanderhängen, und */
/* in dem String Path abspeichern.                 */

Concat(String, FL)
char  *String;
struct FileLock
             *FL;
{
   ULONG LocalPos = 0L;    /* Position im anzuhängenden String */
```

```
   while (Path[GlobalPos++]=String[LocalPos++]); /* Kopieren */

   /* Die Variable GlobalPos zeigt auf das     */
   /* String-Endzeichen (\000) des Path-Strings */

   /* Je nach Verzeichnisstufe (Hauptverzeichnis (z.B. SYS1:) */
   /* Unterverzeichnis (z.B. include/) die Zeichen :, bzw. /  */
   /* zwischensetzen                                          */

   switch ((ULONG)FL)
   {
      case 0L:
         Path[--GlobalPos]=':';      /* Hauptverzeichnis erreicht */
         Path[++GlobalPos]=(BYTE)0;  /* String-Ende  */
      break;

      default:
         Path[--GlobalPos]='/';      /* Unterverzeichnis erreicht */
         Path[++GlobalPos]=(BYTE)0;  /* String-Ende  */
      break;
   }
}

VOID GetBackPath (FL)
struct FileLock  *FL;
{
   struct FileInfoBlock *FIB = 0L;
   struct FileLock      *TL  = 0L;
   struct FileLock      *GL  = 0L;

   if (!Error)    /* Noch kein Fehler             */
   {              /* Diese Routine ist REKURSIV ! */

      FIB = (struct FileInfoBlock*)
        AllocMem((ULONG) sizeof(struct FileInfoBlock),
                 (ULONG) MEMF_CHIP|MEMF_CLEAR);

      /* Muß jedesmal neu allokiert werden, da im FIB ja der jeweilige
*/
      /* Verzeichnisnamenthalten ist.
*/

      if (FIB == 0L)
      {
         Write(OutputFile, "Kein Speicher mehr !!!\n",18L);
         Error = TRUE; /* Global Error */
         goto BackEnd;
      }

      TL = (struct FileLock*)ParentDir(FL);

/* Lock auf übergeordnetes Verzeichnis holen */
```

```
/* Wenn TL != 0 ist, ist das Hauptverzeichnis noch nicht erreicht. */
/* In diesem Fall wird GetBackPath() solange Rekursiv aufgerufen,  */
/* bis das Hauptverzeichnis erreicht wurde. Dann werden vom Haupt- */
/* verzeichnis ausgehend alle Strings aneinandergehängt            */
/* (z.B: SYS1 + ':' + include + '/' + exec + '/').                 */

      if (TL != 0L)
      {
         GetBackPath(TL);

         GL = (struct FileLock*) ParentDir(TL);
         /* Jetzt übergeordnetes Verzeichnis erreicht ? */

         if (Examine(TL, FIB) == 0L)
         {
            Write(OutputFile, "Directory zerstört !\n",21L);
            goto BackEnd;
         }

         Concat(FIB->fib_FileName, GL);

         /* Wenn Hauptverzeichnis erreicht wurde, gibt ParentDir() */
         /* den Wert 0L zurück. Dann wird bei Concat() das Zeichen */
         /* : zwischen die anzuhängenden Strings gesetzt.          */

         UnLock(GL);
         GL = 0L;
      }
      /* else Hauptverzeichnis erreicht */
   }

BackEnd:

   if (GL != 0L)  UnLock (GL);
   if (TL != 0L)  UnLock (TL);
   if (FIB != 0L) FreeMem(FIB,(ULONG)sizeof(struct FileInfoBlock));
}

GetWholePath (Name)
char         *Name;
{
   struct FileLock      *FL  = 0L;
   struct FileLock      *PL  = 0L;
   struct FileInfoBlock *FIB = 0L;


   FIB = (struct FileInfoBlock*)
         AllocMem((ULONG) sizeof(struct FileInfoBlock),
                  (ULONG) MEMF_CHIP|MEMF_CLEAR);

   /* FIB allokieren */

   if (FIB == 0L)
   {
```

```
      Write(OutputFile, "Kein Speicher mehr !!!\n",18L);
      Error = TRUE;       /* Globale Fehlervariable */
      goto WholeEnd;
   }

   FL  = (struct FileLock*) Lock(Name,(ULONG)ACCESS_READ);

   /* Lock auf angegebenes Verzeichnis holen */

   if (Examine(FL,FIB) == 0L)
   {
      Write (OutputFile, "Directory zerstört !\n",21L);
      Error = TRUE;
      goto WholeEnd;
   }

   if (FIB->fib_DirEntryType==0L) /* Kein Directory !!!! */
   {
      Write(OutputFile, Name, (ULONG)strlen(Name));
      Write(" ist ein File und kein Verzeichnis !!!\n", 40L);
      Error = TRUE;
      goto WholeEnd;
   }

   GetBackPath(FL);

   if (Error)              /* Fehler aufgetreten ?                */
      goto WholeEnd;       /* ja                                  */

   /* In GetBackPath() wurde der Name des angegebenen Verzeichnisses */
   /* nicht an Path angehangen. Dies wird hier nachgeholt.           */
   /* Zur Erinnerung: Im FIB steht z.B. nur der Name "exec", obwohl  */
   /* "include/exec" angegeben wurde.                                */

   PL = ParentDir(FL);

   Concat(FIB->fib_FileName, PL);  /* Pfadnamen vervollständigen */
   UnLock(PL);
   PL = 0L;

WholeEnd:
   if (FL != 0L)  UnLock(FL);
   if (FIB != 0L) FreeMem(FIB,(ULONG)sizeof(struct FileInfoBlock));
}


VOID GetDir(DirName, All)
char       *DirName;
BOOL                All;
{
   struct FileInfoBlock *FIB = 0L;
   struct FileLock      *FL  = 0L;
   struct FileLock      *TL  = 0L;
   ULONG                LocalPos;
```

```
   ULONG                i;
   ULONG                Len = 0L;   /* Länge des File-Namens */

   if (!Error)
   {  /* Im Fehlerfall, oder nach Ctrl-C nur noch belegte Speicher */
      /* freigeben.                                                 */

      FL = (struct FileLock*) Lock(DirName, (ULONG)ACCESS_READ);

      if (FL == 0L)  /* Lock auf Dir holen und testen */
      {
         Write(OutputFile, "Wrong path !!!\n",16L);
         Error = TRUE;
         goto End2;
      }

      FIB = (struct FileInfoBlock*)
        AllocMem((ULONG) sizeof(struct FileInfoBlock), (ULONG)
(MEMF_CHIP|MEMF_CLEAR));

      if (FIB == 0L)
      {
         Write(OutputFile, "Kein Speicher mehr !!!\n", 23L);
         Error = TRUE;
         goto End2;
      }

      if (Examine(FL, FIB) == 0L)
      {
         Write(OutputFile, "Directory zerstört !\n", 21L);
         Error = TRUE;
         goto End2;
      }


      while (ExNext(FL, FIB) != 0L)
      {
         if (Wildcard(WildcardString, FIB->fib_FileName))
         {
            if (CTRL_C())
            {
               Write(OutputFile, "*** BREAK ***\n", 14L);
               Error = TRUE;
               goto End2;
            }

            for (i=0L;i<Verschachtellung;i++)
               Write(OutputFile, "      ", 6L);

            if (FIB->fib_DirEntryType > 0)   /* Directory */
            {
               Write(OutputFile, "      ", 6L);
               Write(OutputFile, FIB->fib_FileName,
                    (ULONG)strlen(FIB->fib_FileName));
```

```
            Write(OutputFile, " (dir)\n", 7L);

            if (All)
            {
               /* Inhalt eines weiteren Directory ausgeben */
               /* Dazu wird der Name des Directorys an den */
               /* von GetBackPath() erzeugten Pathnamen    */
               /* angehangen.                              */

               LocalPos = 0L;

               while (Path[GlobalPos++] =
                              FIB->fib_FileName[LocalPos++]);

               /* Anhängen */

               Path[--GlobalPos]=(char)'/';
               Path[++GlobalPos]=(char)0;

               /* Das Zeichen / anhängen, und String beenden */

               Verschachtellung++;

               GetDir(Path,All);

               /* REKURSION */

               if (Error)                  /* Fehler */
                  goto End2;

               Verschachtellung--;

               /* Keine Weiterbearbeitung nach Fehler oder Abbruch */

               GlobalPos -= LocalPos;
                              /* vorher angehangenen Namen */
                              /* entfernen, bzw. GlobalPos */
                              /* updaten, und \000 setzen  */
               Path[GlobalPos] = (char)0;
            }
         }
         else
         {
            Len = (ULONG)strlen(FIB->fib_FileName);

            Write(OutputFile, "  ", 2L);
            Write(OutputFile, FIB->fib_FileName, Len);
            Write(OutputFile, "\n", 1L);
         }
      }
   }
}
End2:   if (FIB != 0L) FreeMem(FIB,(ULONG)sizeof(struct FileInfoBlock));
End1:   if (FL != 0L)  UnLock(FL);
```

```
}


main(argc, argv)
int  argc;
char      *argv[];
{
   int i;
   char *Pfadname = "\000";

   BOOL ALL    = FALSE;

   for (i=1;i<argc;i++)
   {
      if (argv[i][0] != '-')
      {
         Pfadname = argv[i];
      }
      else
      switch(argv[i][1])
      {
         case 'a':
            ALL = TRUE;
         break;

         default:
            printf("Unbekannte Option <%s> !\n",argv[i]);
            exit(0L);
      }
   }


   OutputFile = (struct FileHandle *) Output();

   GetWildcardString(Pfadname);

   if (Directory(Pfadname))
   {
     GetWholePath(Pfadname);        /* gesamten Pfad ermitteln */
     GetDir(Path,(BOOL)ALL);        /* Directory ausgeben      */
   }
}
```

Programm Directory.c

Erläuterungen zu obigem Programm

In main() wird zunächst einmal festgestellt, ob beim Aufruf die Option -a angegeben wurde. Ist dies der Fall, wird die Variable ALL auf TRUE gesetzt, was dafür sorgt, daß alle Unterverzeichnisse ausgegeben werden.

Dann wird mit der Funktion Output() der Ausgabekanal ermittelt. Diesen Ausgabekanal kann man wie ein mit Open() geöffnetes File behandeln. Im Gegensatz zu printf() kann die Ausgabe über diesen Ausgabekanal nicht unterbrochen werden.

Wie Sie vielleicht festgestellt haben, führt die Tastenkombination Ctrl-C während der Ausgabe eines Textes mit printf() zum Programmabbruch. Dies wollen wir aber vermeiden, da so die von uns belegten Speicherplätze verloren gehen. Wir fragen ja deshalb selber die Tastenkombination Ctrl-C ab, um die belegten Speicherplätze freigeben zu können.

Danach wird die Routine GetWildcardString() aufgerufen. Diese Routine sorgt dafür, daß eine Wildcardangabe von den Pfadangaben des auszugebenden Unterverzeichnisses getrennt werden. Wenn Sie z.B. Directory df0:include/ex#? aufrufen, würde theoretisch ja versucht, das Verzeichnis df0:include/ex#? auszugeben. Dieses existiert aber gar nicht. Ausgegeben werden soll das Verzeichnis df0:include/. ex#? ist nur das Pattern (Muster), nach dem die File-Namen ausgegeben werden sollen. GetWildcardString() geht bei der Trennung von Pfadangaben und Wildcard-String so vor sich:

Der String (z.B. df0:include/ex#?) wird von hinten nach vorne nach dem Vorkommen von ? durchsucht. Dieses Zeichen ist ja Bestandteil von Wildcard(#?) und Joker (?). Es wird allerdings nur solange gesucht, bis das Zeichen / oder :, oder der Anfang des Strings gefunden wurde. Wurde das Vorkommen des ? festgestellt, so stellt der hintere Teil des Strings bis zum erstmaligen Auftauchen von / oder : den Wildcard-String dar. Dieser wird als globaler String verwaltet. Auch der Pfadname wird von dieser Routine bearbeitet. Nach dem Bestimmen und Kopieren des Wildcard-Strings wird dieser aus dem der Routine GetWildcard-String() übergebenen String entfernt. Dazu wird einfach das erste Zeichen des gefundenen Wildcard-Strings durch das String-Endezeichen \000 ersetzt.

Nach der Ermittlung von Wildcard-String und Pfadname kann nun daran gegangen werden, das Directory auszugeben. Dazu wird erst einmal geprüft, ob der angegebene Pfadname tatsäch-

lich der Pfadname zu einem Directory ist (Directory()). Ist dies nicht der Fall, wird das Programm verlassen.

Wenn aber der angegebene Pfadname zu einem Directory gehört, wird erst einmal dafür gesorgt, daß der Pfadname aufgearbeitet wird (GetBackPath()). Dazu wird ein Lock auf das angegebene Verzeichnis geholt. Nun wird solange mit ParentDir() in das übergeordnete Verzeichnis gestiegen, bis das Hauptverzeichnis erreicht wurde. Dies geschieht mit Hilfe eines rekursiven Algorithmus. Es wird übrigens bei jedem Aufstieg ein neuer FileInfoBlock allokiert.

Wenn das Hauptverzeichnis erreicht wurde, wird der im FIB enthaltene Name in einen globalen String kopiert und das Zeichen : angehängt. Da jetzt die einzelnen Rekursionstufen wieder verlassen werden, bildet sich der aufgerarbeitete File-Name, wobei zwischen zwei Verzeichnisnamen das /-Zeichen gesetzt wird. So entsteht aus df0:include/exec z.B. SYS1:include/exec/.

Bei der Ausgabe der File-Namen (GetDir()) wird dann, wenn auf ein auszugebendes Unterverzeichnis gestoßen wird, der Name dieses Unterverzeichnisses an den globalen Pfad-String angehängt und nach der Ausgabe wieder entfernt.

2.5.6 Spielereien mit der Systemzeit

Sicher haben auch Sie schon einmal die Fernsehwerbung einer deutschen Kaffeerösterei gesehen, in der ein Kaffeetrinkender Computerbenutzer auftaucht. Dieser schaltet gerade seinen Rechner ein, und auf dem Bildschirm erscheint die Meldung "GOOD MORNING". Wäre es nicht schön, wenn auch Ihr Amiga Sie so nett begrüßen würde?

Dazu muß nur die Systemzeit ermittelt werden, in Abhängigkeit derer Ihr Computer Sie mit "Guten Morgen", "Guten Tag" oder "Guten Abend" begrüßt. Da wir nicht die Möglichkeit haben, den Amiga direkt nach dem Anschalten mit diesem Gruß zu beauftragen, muß dies beim Booten des Systems in der Startup-Sequenz geschehen. Aber natürlich wäre es recht einfallslos, den

Amiga jedesmal, wenn der Rechner neu gebootet werden muß - was bei Programmentwicklern ja recht häufig vorkommen soll - einen Gruß von sich gibt. Der Amiga soll höchstens viermal am Tag grüßen.

Dazu muß das Datum sowie die Uhrzeit des letzten Grußes abgespeichert werden. Erkennt das Programm, daß es an einem Tag schon mal "Guten Morgen" gesagt hat, läßt es den Gruß und wartet bis zur Mittagszeit. Schließlich ist Ihr Amiga kein übertriebener Heuchler:

```
#include "exec/types.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

VOID *Lock();
VOID *Open();

#define MORGEN 0L    /* Zeitabschnitte */
#define TAG    1L
#define ABEND  2L
#define NACHT  3L

struct DateStamp  ActualDate, /* Systemzeit */
                  OldDate;    /* gespeichertes Datum */

#define READ_WRITE_LEN (ULONG)sizeof(struct DateStamp)

struct FileHandle *Handle     = 0L;
struct FileLock   *HandleLock = 0L;

char *FileName = "SYS:Systemzeit";

VOID Gruss(OldDate, Date, Name)
struct DateStamp
          *OldDate,*Date;
char                    *Name;
{
   LONG  Hours;
   LONG  OldHours;

   char *GrussString;

   LONG ZeitAbschnitt;


   Hours =    Date->ds_Minute/60L;  /* Stunden berechnen */
   OldHours = OldDate->ds_Minute/60L;
```

```
if ((Hours > 4L) &&  (Hours <= 11L))
/* von        5  bis           11 Uhr */
{
   GrussString = "Guten Morgen";
   ZeitAbschnitt = MORGEN;
}

if ((Hours > 11L) && (Hours <= 18L))
/* von       12  bis           18 Uhr */
{
   GrussString = "Guten Tag";
   ZeitAbschnitt = TAG;
}

if ((Hours > 18L) && (Hours <= 22L))
/* von       19  bis           22 Uhr */
{
   GrussString = "Guten Abend";
   ZeitAbschnitt = ABEND;
}

if ((Hours > 22L) || (Hours <= 4L))
/* von       23 bis 0 und 0 bis 4 Uhr */
{
   GrussString = "Gute Nacht";
   ZeitAbschnitt = NACHT;
}

if (Date->ds_Days ==  OldDate->ds_Days)
                          /* Wurde heute schon gegrüßt ? */
{
   if (((OldHours >  4L) && (OldHours <= 11L))
                         && (ZeitAbschnitt != MORGEN))
   /* Wurde heute schon GUTEN MORGEN gesagt ? */
      printf("%s %s\n", GrussString, Name); /* Nein */
   else
   if (((OldHours > 11L) && (OldHours <= 18L))
                         && (ZeitAbschnitt != TAG))
   /* Wurde heute schon GUTEN TAG gesagt ? */
      printf("%s %s\n", GrussString, Name); /* Nein */
   else
   if (((OldHours > 18L) && (OldHours <= 22L))
                         && (ZeitAbschnitt != ABEND))
   /* Wurde heute schon GUTEN ABEND gesagt ? */
      printf("%s %s\n", GrussString, Name); /* Nein */
   else
   if (((OldHours > 22L) || (OldHours <=  4L))
                         && (ZeitAbschnitt != NACHT))
   /* Wurde heute schon GUTE NACHT gesagt ? */
      printf("%s %s\n", GrussString, Name); /* Nein */
}
else /* nein */
{
   printf ("%s %s\n", GrussString, Name);
```

```
   }
}

main(argc, argv)
int  argc;
char      *argv[];
{
   DateStamp(&ActualDate); /* Systemzeit holen */

   HandleLock = (struct FileLock *)Lock(FileName, (ULONG)ACCESS_READ);
   if (HandleLock == 0L)
   {
      /* File "SYS:Systemzeit nicht vorhanden */
      Handle = (struct FileHandle *)Open(FileName, (ULONG)MODE_NEWFILE);
      if (Handle == 0L)
      {
         printf("Kann <%s> nicht erzeugen !!!\n", FileName);
         exit (0L);
         /* Kein UnLock() nötig, da FileLock nicht erzeugt wurde */
      }

      if (Write(Handle, &ActualDate, READ_WRITE_LEN) != READ_WRITE_LEN)
      {                  /* Datum abspeichern */
         printf ("Schreibfehler !!!\n");
         Close(Handle);
         exit(0L);
      }

      OldDate.ds_Days   = 0L; /* Immer grüssen */
      OldDate.ds_Minute = ActualDate.ds_Minute;
      OldDate.ds_Tick   = 0L;
   }
   else
   {
      UnLock(HandleLock);  /* Damit Open(),bzw. Write() funktionieren
kann */

      Handle = (struct FileHandle *)Open(FileName, (ULONG)MODE_OLDFILE);
      if (Handle == 0L)
      {
         printf("Kann <%s> nicht öffnen !!!\n", FileName);
         exit (0L);
         /* Kein UnLock() nötig, da schon ausgeführt */
      }

      if (Read(Handle, &OldDate, READ_WRITE_LEN) != READ_WRITE_LEN)
      {           /* Datum Lesen */
         printf("Lesefehler !!!\n");
         Close(Handle);
         exit(0L);
      }

      Seek(Handle, 0L, (ULONG)OFFSET_BEGINNING);
```

```
        /* An den Anfang des Files zurückspulen */

        if (Write(Handle, &ActualDate, READ_WRITE_LEN) != READ_WRITE_LEN)
        {           /* Datum Lesen */
           printf("Schreibfehler !!!\n");
           Close(Handle);
           exit(0L);
        }
     }

     Gruss(&OldDate, &ActualDate, argv[1]);
     Close(Handle); /* File muß hier immer geschlossen werden */
}
```

Programm Grüsse.c

Dieses Programm ermittelt zunächst die Systemzeit. Dies geht mit dem Befehl DateStamp() recht einfach. Dieser Befehl füllt eine DateStamp-Struktur (die auch im FileInfoBlock enthalten ist) mit dem aktuellen Datum und der aktuellen Uhrzeit. Diesem Befehl wird als einziger Parameter die Adresse solch einer Struktur übergeben (DateStamp(&DateStamp-Struktur)).

Daraufhin wird versucht, das File zu öffnen, das die Zeit des letzten Grußes enthält (SYS:Systemzeit). Ist dieses File nicht vorhanden, wird es erzeugt, und das aktuelle Datum wird hineingeschrieben. Außerdem wird in diesem Fall auf jeden Fall gegrüßt. Wenn es aber vorhanden ist, wird die darin enthaltene Systemzeit nach "OldDate" geladen, und diese Zeit durch die aktuelle Systemzeit ersetzt. Dazu wird der File-Positions-Zeiger mit Hilfe von Seek() zunächst an den Anfang des Files bewegt und dann das alte Datum vom aktuellen Datum überschrieben.

Dann wird die Routine Gruss() aufgerufen, der das alte und das aktuelle Datum sowie der erste Kommandoparamter (argv[1]) übergeben wird.

Gruss() erkennt nun den Morgen daran, daß es noch nicht 11 Uhr ist. Zwischen 11 Uhr und 18 Uhr wird "Guten Tag" gesagt, und ab 18 Uhr werden Sie mit "Guten Abend" begrüßt. Zwischen 22 und 5 Uhr erscheint "Gute Nacht". Um das ganze noch ein bißchen freundlicher zu gestalten können Sie dem Programm

auch noch Ihren Namen übergeben, so daß z.B. mit "Guten Abend Sabine" o.ä. gegrüßt wird (Komando-Parameter).

Wenn Sie den Kommando-Parameter weglassen, macht das gar nichts, weil argv[1] dann ein leerer String ist, bei dessen Ausgabe bekanntlich nichts passiert.

Wenn Gruss() erkannt hat, welche Tageszeit gerade ist, wird festgestellt, ob das alte Datum mit dem aktuellen Datum übereinstimmt. Ist dies nicht der Fall wird auf jeden Fall gegrüßt. Wenn das alte und das aktuelle Datum aber übereinstimmen heißt das, daß heute schon mindestens einmal ein Gruß ausgesprochen wurde. Um festzustellen, welcher, wird die alte Zeit zunächst auf die Tageszeit untersucht. Wenn zu dieser Tageszeit aber schon gegrüßt wurde - was daran erkenntlich ist, daß die Variable Zeitabschnitt den entsprechenden Wert (MORGEN, TAG, ABEND bzw NACHT) hat -, wird die Ausgabe des Grußes übersprungen und das Programm beendet.

Theoretisch läßt sich dieses Programm beliebig erweitern. Man kann z.B. mit Hilfe des Narrator-Devices auch veranlassen, daß der Amiga regelrecht zu Ihnen spricht. Dazu wird die systemeigene Stimme verwendet. Wenn Ihnen aber diese Stimme nicht gefällt, könnten Sie z.B. hingehen, und die Stimme Ihres Mannes, Freundes, Ihrer Freundin oder Frau mit einem Digitizer samplen und dann abspielen lassen.

3. Intuition und C für Fortgeschrittene

Dieses Kapitel beschäftigt sich mit der Programmierung des Amiga-Betriebssystems, und zwar speziell mit der Systemprogrammierung von Intuition. Sicherlich ist klar, daß mit C die nun weltbekannte Programmiersprache gemeint ist. Auch unter Systemprogrammierung kann man sich einiges vorstellen. Zum Beispiel:

> Wie sag' ich meinem Amiga-Betriebssystem, daß ich einen bestimmten grafischen Aufbau haben möchte?

oder ...

> Wie kann mir das System mitteilen, daß der User Eingaben vorgenommen hat (mit der Maus oder über die Tastatur)?

Dies sind zwei Fragen, die unter das Thema Systemprogrammierung fallen. Sicherlich gibt es noch weitere, und alle diese Fragen werden genau in diesem Kapitel beantwortet.

"Intuition" ist die Bezeichnung für die Kommunikationsart, mit der man auf dem Amiga arbeitet. Unter Kommunikation verstehen Sie bitte jeden Informationsaustausch, der direkt abläuft. Zu nennen sind da als Eingabemedium die Tastatur und die Maus - beide Einheiten sind ja im Lieferumfang des Rechners enthalten, doch bestimmt finden Sie noch mehr Medien.

Die Ausgabe geschieht im wesentlichen über den Bildschirm. Dabei darf nicht vergessen werden, daß es auch noch Drucker und Disketten gibt, aber diese werden nicht über Intuition angesteuert und können deshalb nicht unter dieser Thematik erläutert werden.

Welche Mittel und Wege bietet Intuition?

Beschäftigen wir uns zur Klärung dieser Frage zuerst mit den Ausgabewegen und -mitteln:

Beim Booten der Workbench-Diskette sehen wir gleich zwei Medien, die uns Intuition zur Verfügung stellt. Zuerst und auch als Erstrangiges den Screen. Er bildet den Untergrund für jede Ausgabe! Als wichtigstes Beispiel ist dazu gleich der Workbench-Screen zu nennen, der ja wie eben beschrieben gleich beim Booten erscheint. Er begleitet den Benutzer auf Schritt und Tritt und wird auch für viele Programme als Ausgabeuntergrund verwendet.

Für die Ausgabe selbst wird sehr selten ein Screen verwendet, dessen Handhabung nicht sehr einfach gestaltet ist. Um sich die Arbeit zu erleichtern, verwendet man Windows, die zweite Ausgabeeinrichtung Intuitions. Die Windows werden vollständig von Intuition verwaltet und sind somit in den glücklicherweise weit gesteckten Grenzen nutzbar. Aber auch das sollte Ihnen bekannt sein, denn schließlich arbeitet man ja tagtäglich mit ihnen.

Kommen wir nun zur Eingabe von Informationen. Das einfachste Eingabemedium sind die Gadgets! Sie befinden sich sowohl in Screens als auch in Windows und geben im einfachsten Fall nur einen Impuls weiter. Natürlich gibt es auch komplexere Typen, mit denen man schon ganze Texte eingeben kann, doch dazu muß auch wieder die Tastatur benutzt werden, und wir wollen uns zuerst auf die Mauseingabe konzentrieren.

Gadgets eignen sich zwar gut zur Informationseingabe, doch lassen sich komplexe Prozesse nicht so einfach mit ihnen auslösen. Für elementare Funktionen, die programmspezifisch sind, wurden die Menüs entwickelt. Auch sie werden über die Maus angesteuert, aber hier liegt der Schwerpunkt eindeutig auf der Textauswahl, während die Gadgets eher mit Grafiken ausgedrückt werden. Ausnahmen und Übergänge bestätigen die Regel! Hier haben Sie einen ersten Eindruck von den Fähigkeiten, die Ihnen Intuition bietet. Die Arbeit mit allen diesen Funktionen ist Ihnen bestimmt schon geläufig. Wir wollen uns hier vorrangig um folgende Frage kümmern:

"Wie programmiere ich das?"

Lassen Sie sich überraschen, welche Fähigkeiten in diesem Supercomputer stecken!

3.1 Windows, die Grundlage jeder Ein- und Ausgabe

Wie schon in der Einleitung des Intuition-Kapitels gesagt, sind die Windows das wichtigste Ein- und Ausgabemedium!

Wir haben uns zur Aufgabe gemacht, in diesem Buch die Beschreibung zu liefern, wie Sie selbst eine Anwendung programmieren können - von uns wurde ein Texteditor für C-Programme ausgesucht. Deshalb soll dieses Unterkapitel darüber Auskunft geben, wie Sie die Grundlagen für die Ein- und Ausgabe programmieren und aus den vielen Möglichkeiten zusammenstellen. Sie werden gleich erkennen, welche Vielfalt dem Programmierer von Intuition geboten wird.

Nach der Aufzählung von Einstellungsmöglichkeiten werden wir uns an die Einrichtung eines von unserem Programm verwalteten Windows machen. Dabei lernen Sie kennen, wie man sich alle Befehle Intuitions zunutze macht, denn so einfach vorhanden sind sie nicht!

Weiterhin erstellen wir zusammen ein allgemeines Window-Programm, das Sie immer wieder mit geringfügigen Änderungen benutzen können. Sie sollten dafür eine Diskette anlegen, denn diese Modulsystematik werden wir bis zum Ende des Buches weiterverwenden, um Ihnen viel Tipparbeit abzunehmen.

3.1.1 Die Window-Parameter und wie man sie auswählt

Für die Einstellungen eines Windows benötigen wir viele Parameter, mit denen jede nur erdenkliche Größe festgesetzt werden kann. Um diese einzeln aufzuführen, rufen wir uns dafür die Eigenschaften, die uns von normalen Workbench-Windows bekannt sind, in Erinnerung:

Äußere Merkmale

Jedes Window läßt sich in seiner Größe und Position verändern! Damit können wir folgende Werte einstellen: X-Position, Y-Position, Breite, Höhe.

Wenn sich ein Window in seiner Größe verändern läßt, so ist es in vielen Fällen wünschenswert, wenn man als Programmierer ein Maximum und Minimum angeben kann, um nicht am Ende vielleicht böse Überraschungen erleben zu müssen. Bei der Arbeit mit der Workbench wird man es selten erleben, daß einem verboten wird, ein Inhaltsverzeichnis-Window weiter zu vergrößern oder zu verkleinern, doch vergessen wir nicht die Requester. Bei ihnen ist es nur erlaubt, sie zu verkleinern, ein Vergrößern ist vollkommen ausgeschlossen. Bei welchen Werten nun nicht mehr vergrößert oder verkleinert werden darf, kann man als Programmierer mit MaxBreite, MaxHöhe, MinBreite und Minhöhe einstellen.

Außerdem, und das wissen vielleicht nicht alle, kann man die Farben bestimmen, natürlich in Abhängigkeit des Screens, mit denen der Window-Rand und die dort befindlichen Gadgets gezeichnet werden. Die zwei Farben werden als Zeichenstift und Blockstift bezeichnet. Ganz sicher gehört zu jedem Window auch eine Titelleiste mit ihrem Text. Auch dieser kann beliebig definiert werden.

Für die, denen die wenigen System-Gadgets, so bezeichnet man die Gadgets, die von Intuition angeboten und teilweise auch verwaltet werden, nicht reichen, ist es vorgesehen, noch mehr Gadgets in ein Window einzubinden. So werden z.B. die Schiebebalken in den Verzeichnisfenstern verwaltet, denn sie sind nicht als System-Gadget definiert. Als Programmierer können wir jedes erdenkliche Gadget selbst konstruieren.

Als letzte äußere Eigenschaft, die wir an jedem selbst erzeugten Window einstellen können, ist die Möglichkeit zu nennen, daß die System-Gadgets beliebig eingesetzt oder weggelassen werden können, je nachdem, was für eine Anwendung sinnvoll ist.

Weitere Einstellungen (innere Merkmale)

Nachdem wir die offensichtlichen Einstellungen an einem Window durchgesprochen haben, kommen jetzt die Einstellungen, bei denen nicht immer gleich grafische Veränderungen erkennbar sind.

Eine Möglichkeit, die ziemlich selten genutzt wird, ist es, für den Haken, das sog. CheckMark, der Menüpunkte als abgehakt kennzeichnet, ein neues Aussehen zu definieren. Ich will Sie ermutigen, dieses ruhig einmal auszuprobieren!

Sie haben bestimmt schon von der wunderbaren Errungenschaft gehört, daß man einen eigenen Screen von Intuition verwalten lassen kann. Gehen wir von der Tatsache aus, daß wir einen eigenen Screen besitzen und auf ihm unser neues Window einrichten wollen. Dazu teilen wir dem Betriebssystem in einem Flag mit, daß es sich um ein Window handelt, welches nicht auf der Workbench erscheinen soll, sondern auf dem neuen Screen eingerichtet werden muß. Weiterhin benötigt Intuition aber noch die Information, auf welchem eigenen Screen, denn es ist durchaus denkbar, daß mehr als ein CustomScreen geöffnet wurde. Möchten wir also ein Window auf einem selbstdefinierten Screen öffnen, so kennzeichnen wir das durch ein Flag und übergeben der Struktur den Pointer auf den Screen.

Programminterne Einstellungen (besondere Merkmale)

Die Anwendungszwecke, mit denen Windows beauftragt werden, sind so vielfältig, daß man mit einem Standardtyp nicht jeder Anwendung gerecht werden kann. Aus diesem Grund wurden verschiedene Wege bereitgestellt, von denen sich der Programmierer einen (oder mehrere) aussuchen kann.

Erhaltung und Wiederherstellung des Window-Inhalts

Der Inhalt eines Windows - das kann sowohl Text als auch Grafik sein - sollte möglichst unabhängig von anderen erhalten bleiben, auch wenn diese zeitweise das eigene überlagern und somit Teile verdecken. Doch dazu würde Intuition, wenn es einfach

die verdeckten Bereiche zwischenspeichern würde, man nennt das Puffern, sehr viel Speicherplatz benötigen. Aber in einigen Fällen ist das überhaupt nicht nötig. Weiß man nämlich, was sich im Window befindet und ist es ziemlich einfach wiederherzustellen, dann könnte man alles vom Programm aus vornehmen. Das bekannteste Beispiel ist die Workbench mit den Inhaltsverzeichnissen. Dort werden die Icons einfach wieder neugezeichnet, wenn sie durch etwas anderes verdeckt wurden. Die Bilder werden also vom Programm Workbench restauriert und nicht über Intuition gepuffert. Dadurch erreicht man, daß sehr viel weniger Speicherplatz verbraucht wird, wenn sich viele Windows überlagern!

Bei unserem eigenen Programm müssen wir deshalb zuvor überlegen, welchen Anwendungsbereich das Window abdeckt. Dabei sind drei grundlegend verschiedene Gebiete möglich:

1. Programme, denen der Inhalt des Windows vollkommen bekannt ist und deren Daten sowieso in anderer Form gespeichert sind. Ein Beispiel wäre da die Textverarbeitung, die den Text zusätzlich zur grafischen Information auf dem Bildschirm auch noch in ihrem Textpuffer speichert. Bei solchen Anwendungen empfiehlt es sich, die Wiederherstellung selbst vorzunehmen, um anderen gleichzeitig laufenden Programmen möglichst wenig Speicherplatz wegzunehmen.

2. Programme, die zeitlich nicht in der Lage sind, einen zerstörten Bereich wieder neu zu konstruieren. Hier würde man Intuition mit der Sicherung der überlagerten Bereiche beauftragen, um möglichst wenig Arbeit zu haben. Das Problem tritt häufig bei Grafiken auf, deren Neuzeichnen viel länger dauern würde als eine Zwischenspeicherung. Wichtig ist, daß der Speicherbedarf in einem richtigen Verhältnis zur Zeitersparnis steht, denn auch bei einem Computer mit 1 MByte RAM sollte der Speicherplatz sorgfältig verplant werden.

3. Programme, die mehr Grafik-Display benötigen als überhaupt darstellbar ist. Für diese Kategorie ist eine vollständig neue Methode entwickelt und bereitgestellt worden. Das Programm gibt seine Grafik in eine vollkommen ei-

genständige BitMap (so bezeichnet man den Gafikspeicher), und Intuition holt sich daraus den gewünschten Ausschnitt für das Window. So kann bei jeder Wiederherstellung auf den Grafikspeicher zurückgegriffen werden.

Fassen wir noch einmal die drei Modi in einer Tabelle zusammen, damit Sie einen kurzen prägnanten Überblick gewinnen können. Die Tabelle enthält weiterhin die Namen der Flags, die für den entsprechenden Modus gesetzt werden müssen. Es ist darauf zu achten, daß jeder Modus nur alleine lauffähig ist!

Refresh-Art	Flag	Speicherverbrauch
Einfach	SIMPLE_REFRESH	Keiner, denn Intuition kümmert sich nicht um das Window.
Unterstützt	SMART_REFRESH	Entsprechend der verdeckten Window-Bereiche.
Vollständig	SUPER_BITMAP	Die gesamte Grafik des Windows wird zwischengepuffert.

Tabelle 3.1: Window-Refresh

Spezielle Window-Typen

Besondere Anwendungen erfordern es auch, besondere Window-Typen zur Verfügung gestellt zu bekommen. Dies wurde hauptsächlich aus dem Grund in Erwägung gezogen, da mit einem Standard-Window-Typ sicherlich die eine oder andere Anwendung benachteiligt gewesen wäre. So ist es aber gelungen, sowohl unterstützende Anwender-Software leicht zu programmieren als auch ausgefallene Problemlösungen vorbildlich zu unterstützen.

Randlose Windows

Es ist keineswegs so, daß Ihr Window immer mit dem bekannten Rand, der aus Titelleiste, Gadgets und Begrenzungslinien besteht, abgebildet werden muß.

Wenn man von der primitivsten Möglichkeit ausgeht, so besteht ein Window nur aus einer Fläche, deren Inhalt getrennt von anderen vielleicht vorhandenen Flächen behandelt wird. Der Rand (engl. Border), den Intuition darum zieht, dient nur der Ver-

ständlichkeit. Wären auf der Workbench alle Windows ohne diesen Rand, dann würde sicherlich das totale Chaos herrschen.

Es hat also einen tieferen Sinn, wenn man diese grafische Unterteilung vornimmt. Aber sicherlich wird man auch eine Anwendung finden, bei der ein Rand unpassend wäre. Dafür kennzeichnen Sie das beim Einrichten eines neuen Windows mit einem Flag namens BORDERLESS. Doch achten Sie darauf, daß der Benutzer nicht durch ein solches Window verwirrt wird, denn man kann es überhaupt nicht erkennen, wenn es z.B. alleine auf einem leeren Screen liegt.

Benachteiligte Windows

Der Titel lenkt hier etwas von dem eigentlichen Zweck ab, man sollte ihn deshalb nicht so wörtlich nehmen. Gemeint ist, daß solch ein Window nicht mehr die gesamten Eigenschaften normaler Windows hat. Es ist dazu "verdammt", immer das hinterste Window zu sein und somit von allen neu hinzukommenden überlagert zu werden. Es gibt keine Möglichkeit für den Benutzer, dieses Window in den Vordergrund zu bringen. Deshalb kann man auch nicht mehr alle System-Gadgets verwenden: Nur noch das Close-Gadget ist erlaubt. Das hat durchaus Vorteile:

Sie können diesen Window-Typ als Untergrund für jede beliebige Anwendung benutzen. Ob Sie nun einen Zeichentrickfilm ablaufen lassen, der durch das BACKDROP-Window einen Hintergrund erhält, oder ob Sie ein Zeichen-Tool programmieren, bei dem auf dem BACKDROP-Window gezeichnet wird und bei dem jedes weitere Window, das irgendwelche unterstützenden Informationen oder Funktionen enthält, über diesem geöffnet wird. Alles dies sind Anwendungen, die durch die BACKDROP-Eigenschaft unterstützt werden, denn kein neues Window kann hinter das so bezeichnete gelegt werden. Das Programm braucht sich nicht darum zu kümmern, denn Intuition verwaltet alles. Wenn Sie diesen Typ bei der Initialisierung kennzeichnen wollen, so verwenden Sie das Flag mit dem Namen BACKDROP.

Eine interessante Kombination wäre z.B. ein BORDERLESS-BACKDROP-Window, das sich wunderbar für die oben genannten Beispiele eignet. Aber Sie können nicht nur die ersten beiden Typen miteinander kombinieren. Auch die nächsten beiden lassen sich hinzufügen:

Ein extra Window-Rand

Im vorhergehenden Text wurde beschrieben, wie man ein Window erzeugt, das keinen Rand hat. Hier ist nun die genau entgegengesetzte Einstellung beschrieben. Sie wissen ja, daß ein Window eigentlich nur die rechteckige Fläche ist, die in den meisten Fällen noch grafisch durch den Rand von anderen abgehoben wird. Allerdings ergeben sich dadurch auch ein paar Probleme: Zeichnet man nämlich in ein solches Window, so kann es leicht passieren, daß beim Zeichnen die Titelleiste, die Gadgets oder der Rand übermalt werden, denn es ist nur eine Sperre eingebaut, die verhindert, daß man aus dem Window "rauszeichnet".

Im Normalfall liegt es also beim Programm, sprich Programmierer, nicht in den Rand eines Windows zu zeichnen. Aber wer möchte schon dieses alles überprüfen und dann in ausführliche Abfragen umsetzen? Diese Voraussicht hatten auch die Entwickler von Intuition, und deshalb wurde der Modus GIMMEZEROZERO implementiert. Hier verändern sich einige bekannte Einstellungen. Das Window besteht nun aus zwei Feldern. Einmal aus dem Feld, in dem sich alle Gadgets, der Window-Titel und der Window-Rand befinden, und zusätzlich aus dem inneren Feld, in dem das Programm zeichnen kann, ohne darauf zu achten, ob es irgend etwas beschädigt. Der Name rührt daher, daß dieses innere Feld, es wird als "inner window" bezeichnet, in der linken oberen Ecke immer die Koordinaten 0,0 hat, unabhängig von der Größe und dem Ausschnitt, der gezeigt wird.

Für den zusätzlich betriebenen Aufwand wird auch mehr Speicher und Verarbeitungszeit in Anspruch genommen. Außerdem hat der Window-Teil, in den gezeichnet werden kann, ja nicht die bei der Definition angegebene Breite und Höhe. Dafür wurden bei diesem Window-Typ extra zwei weitere Variablen ein-

gerichtet, die man abfragen kann, um sich über die Größe der wirklichen Zeichenfläche zu informieren. Es sind diese: GZZ-Height und GZZWidth. Dieser Bereich liegt zwischen den folgenden Werten: BorderLeft, BorderTop, BorderRight, Border-Bottom.

Da die Mauskoordinaten eines Windows immer auf die gesamte Fläche berechnet werden, kann dies hier zu einigen Problemen führen. Deshalb hält Intuition gleichzeitig noch Mauskoordinaten bereit, die sich auf das GIMMEZEROZERO-Window beziehen: GZZMouseX und GZZMouseY.

Windows, die größer sind als sie selbst

Intuition bietet dem Programmierer die Möglichkeit, eine viel größere Grafikfläche zu verwalten, als auf dem Screen abgebildet werden kann. Gleichzeitig ist der sog. SUPER_BITMAP-Modus auch noch ein Window-Typ, bei dem auf eine ganz bestimmte Art der Inhalt gerettet wird, doch dazu haben Sie ja schon oben unter dem Themengebiet Refresh etwas gelesen. Das Interessante an diesem Window ist, daß man eine fast beliebig große BitMap verwalten kann und nur einen Teil in seinem Window darstellt. Möchte man einen anderen Ausschnitt sehen, so wird einfach dieser abgebildet. Ganz wunderbar eignet sich diese Methode zum Scrollen (hin- und herrollen) einer Grafik.

Wenn Sie diesen Window-Typ in ein Programm einbauen wollen, so setzen Sie einfach das SUPER_BITMAP-Flag. Eine Kombination mit GIMMEZEROZERO oder BACKDROP ist durchaus möglich und kann teilweise auch sehr hilfreich sein.

Die System-Gadgets im Window

Gadgets sind Klickfelder, die mit der linken Maustaste aktiviert werden können. Intuition bietet vier Gadgets quasi als Startset an. Diese vier Gadgets unterstützen Funktionen, die normalerweise nicht vom Programm bewältigt werden können. Ein weiterer wichtiger Punkt sind ihre Eigenschaften! Alle System-Gadgets sind in ihrer Position festgelegt. Ebenso ist die grafische

Gestaltung einheitlich und höchstens in der Farbe veränderbar. Man erkennt also immer die Bedeutung gleich wieder und braucht nicht lange zu raten. Sehen wir sie uns einzeln an:

Das Close-Gadget

Es dient zum Schließen des Windows. Der Vorgang ist Ihnen bestimmt von der Workbench vertraut: Sie wollen ein Verzeichnisfenster nicht mehr haben und betätigen das Gadget in der linken oberen Ecke Ihres Windows. Und schon ist es vom Screen verschwunden! Sie erkennen das Gadget an dem Kasten, in dem sich ein dicker Punkt befindet.

Das Drag-Gadget

Sie erkennen es an den drei fetten Streifen in der Titelleiste des Windows. Mit ihm können Sie, wenn es im Window vorhanden ist, dessen Position ändern. Dazu brauchen Sie nur die Titelleiste anzuklicken, so lange den Button gedrückt zu halten und die Maus zu verschieben, bis die neue gewünschte Position erreicht ist.

Die Depth-Arrangement-Gadgets

Beide treten immer nur zusammen auf und dienen zum Über- oder Hinterlagern der Windows. Man findet sie in der rechten oberen Ecke. Wichtig ist immer die helle rechteckige Fläche. Sie stellt das eigene Window dar und somit auch die neue Position.

Das Sizing-Gadget

Wenn es vom Programm zugelassen ist, die Größe des Windows zu verändern, so finden Sie in der rechten unteren Ecke das Sizing-Gadget. Es stellt ein kleines und ein großes Window durch zwei Rechtecke dar. Klicken Sie es einfach an, und halten Sie den Button gedrückt. Jetzt kann die Maus solange bewegt werden, bis die neue Größe eingestellt ist. Diese liegt aber nur in dem Bereich, der vorher durch die Minimal- und Maximalwerte bestimmt worden ist.

Da durch das Sizing-Gadget der Rand des Windows, der Bereich, in dem sich Intuition-Grafiken befinden, vergrößert wird, muß weiterhin noch angegeben werden, wo dieser Rand vom

wirklichen Ausgabefenster genommen werden soll. Als Möglichkeiten werden der rechte oder der untere Rand angeboten. Je nachdem, ob Sie mehr Zeilen oder mehr Spalten benötigen, entscheiden Sie diese Wahl. Es ist auch möglich, beide Bereiche für Intuition zu reservieren.

Weiterhin muß beachtet werden, daß Intuition zwar eine Information durch das Close-Gadget erhält, diese jedoch nicht selbst verarbeitet. Wie darauf reagiert wird, hängt ganz von dem Programm ab, das das Window steuert. Intuition schließt das Fenster also nicht selbständig. Genauso wie das Programm eine Nachricht erhält, wenn das Fenster geschlossen werden soll, bekommt es auch eine Nachricht, wenn die Größe des Fensters beeinflußt wurde, da dieses vom Programm sicherlich berücksichtigt werden muß.

3.1.2 Wie verwaltet Intuition unser Window?

Sie sind jetzt über die groben Einstellungen unterrichtet. Bevor wir uns ans Ausprobieren dieser Möglichkeiten machen können, müssen wir untersuchen, wie man überhaupt an die Intuition-Befehle herankommt und welchen Prinzipien diese unterliegen. Auch die interne Datenübermittlung soll uns interessieren.

3.1.2.1 Der Zugang zur Intuition-Library

Das ganze Betriebssystem des Amiga ist aus Befehlsgruppen zusammengesetzt. Diese nennt man Librarys (engl. für Bibliothek). Eine Library besteht aus verschiedenen Befehlen, denen alle Parameter übergeben werden und die auch Werte, z.B. Ergebnisse, zurückgeben können.

Allerdings sind nicht alle Befehlsgruppen gleich beim Einschalten vorhanden. Selbst wenn sie schon geladen sind, so muß man noch ankündigen, daß man eine bestimmte Befehlsgruppe auch benutzen will, denn der Zugriff kann nur mit Hilfe eines Zeigers geschehen. Dafür gibt es einen EXEC-Befehl (die Library,

die schon nach Einschalten über das Kickstart aktiv ist), der, wie alle EXEC-Befehle, jedem Programm zugänglich ist.

```
library = OpenLibrary(LibraryName, Version)
            D0          A1          D0
```

Sie übergeben der Funktion den Namen der Library, in unserem Fall natürlich "intuition.library", und die Versionsnummer. Denn in neueren Versionen könnten ja Befehle sein, die eine alte Library nicht enthält, und so können Sie sich davor schützen, eine Library zu laden, die gar nicht alle benötigten Befehle enthält.

Wir wollen aber nur auf die Grundbefehle zurückgreifen, denn Windows gab es schon immer in Intuition. Die ersten Zeilen sehen so aus:

```
struct IntuitionBase *IntuitionBase;
void                 *OpenLibrary();

if (!(IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", NULL)))
   {
   printf ("Keine Intuition Library gefunden!\n");
   exit (FALSE);
   }
```

Programmteil 3.1: Open-Intuition

Dokumentation

In der ersten Strukturdefinition führen wir einen Zeiger ein, der auf die Befehlsliste von Intuition zeigt. Er wird vom Compiler benötigt, denn alle Befehle werden relativ zu diesem Zeiger angesprungen. Lassen Sie ihn ruhig einmal weg, die Fehlermeldung wird mir recht geben.

Für die OpenLibrary()-Funktion haben wir in diesem Fall auch einen Zeiger von unbekanntem, d.h. gar keinem Typ definiert. Denn es kann durchaus sein, daß wir nicht nur die Intuition-Library öffnen und dann gäbe es Probleme in der Typenumwandlung!

In der If-Überprüfung wird zuerst die Funktion selbst aufgerufen und dann ihr Rückgabewert, der Zeiger auf die Intuition-

Befehle, auf Gültigkeit überprüft. Dieser Wert muß ungleich null sein, denn dann stellt er die Speicheradresse dar, an der die Library beginnt. Ansonsten bedeutet Null, daß ein Fehler aufgetreten ist. Im Befehlsteil von if wird dann ein Text in ein mögliches DOS-Window ausgegeben. Danach steigt das Programm über EXIT() aus.

Jetzt fehlt nur noch die Anwort auf die Frage: "Wie beende ich den Zugriff auf eine Library?" Wenn wir wirklich den Zugriff bekommen haben, und nur dann, können wir ganz einfach die Library wieder schließen, und zwar mit dem EXEC-Befehl CloseLibrary():

```
CloseLibrary(LibraryZeiger)
     -414          A1
```

In unserer Anwendung hieße das dann:

```
CloseLibrary(IntuitionBase);
```

3.1.2.2 Die NewWindow-Struktur

Sie haben hiermit den prinzipiellen Aufruf der Library-Funktionen kennengelernt und auch, wie man sie wieder schließt. Dies läßt sich auf jede andere Library übertragen, was für Sie später bestimmt noch interessant sein wird. Beschäftigen wir uns zunächst wieder mit den Windows. Alle am Anfang dieses Kapitels besprochenen Eigenschaften eines Windows lassen sich natürlich beim Öffnen einstellen und teilweise auch nachträglich verändern. Uns kümmert zuerst aber eine neue Frage: "Wie teile ich Intuition meine Startwerte mit?" Die Antwort lautet ganz einfach: "Mit Hilfe einer Struktur!"

Wir benutzen für unser neues Window eine sog. NewWindow-Struktur, die alle Werte enthält, die Intuition benötigt, um ein neues Window einzurichten. Die Struktur hat folgendes Aussehen:

```
struct NewWindow
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
0x08  08  UBYTE DetailPen;
0x09  09  UBYTE BlockPen;
0x0a  10  ULONG IDCMPFlags;
0x0e  14  ULONG Flags;
0x12  18  struct Gadget *FirstGadget;
0x16  22  struct Image *CheckMark;
0x1a  26  UBYTE *Title;
0x1e  30  struct Screen *Screen;
0x22  34  struct BitMap *BitMap;
0x26  38  SHORT MinWidth;
0x28  40  SHORT MinHeight;
0x2a  42  USHORT MaxWidth;
0x2c  44  USHORT MaxHeight;
0x2e  46  USHORT Type;
0x30  48
}
```

Wie Sie ja wissen, besteht eine Struktur aus einer Ansammlung vieler zum Teil verschiedener Variablen. Sie werden sich vielleicht wundern, warum vor den Variablentypen-Deklarationen mit ihren Namen auch noch Zahlen stehen, die im normalen C-Programm nicht zu finden sind. Diese Zahlen zeigen Ihnen den *Offset* an, unter dem die Variablen später im Speicher aufgefunden werden können. Für die, die manchmal mit dem Debugger arbeiten oder sich etwas in Assembler auskennen, kann es eine große Hilfe sein. Wie Sie damit richtig umgehen, können Sie im zweiten Kapitel nachlesen.

Die einzige wichtige Schlußfolgerung, die Sie aus dem obigen Absatz ziehen müssen, ist, daß Sie nie, nie, nie die Zahlen mit abtippen dürfen!

Als letztes sei noch angemerkt, daß die erste Spalte hexadezimal und die zweite den Funktions-Offset dezimal ausdrückt. Die letzte Zahl beziffert die Gesamtlänge der Struktur. Das kann für die Speicherplanung nicht unwesentlich sein!

Bis hierhin haben wir geklärt, auf welcher Grundlage unsere Kommunikation mit Intuition abläuft. Man kann dies als allgemeingültig für alle Librarys festhalten. Jeder Kontakt zu den

Unterprogrammen läuft dann über vordefinierte Funktionen ab, denen wir Werte übergeben, wie z.B. der OpenLibrary()- oder CloseLibrary()-Funktion. Intuition wird sogar, wie auch die meisten anderen Librarys, zu seinen Verwaltungszwecken eigene Strukturen anlegen, in denen wir nachschauen können, um uns Daten, d.h. Informationen zu verschaffen. Kommen wir jetzt zur aktiven Arbeit mit Intuition:

Einrichtung einer NewWindow-Struktur

Die Überschrift gibt zwar den Hauptgedanken wieder, doch rankt sich noch viel mehr drumherum. Wir werden die Grundbausteine der weiteren Entwicklungsarbeit legen und dazu gleich zwei Funktionen entwerfen, die dann in das erste Window-Programm eingebaut werden. Aber jetzt zu den beiden Funktionen, die, wie Sie sofort erkennen werden, einen guten Zweck erfüllen und helfen, später vielleicht auftretende Probleme mit Leichtigkeit zu lösen.

Die erste Funktion soll alles das beinhalten, was am Anfang des Programms geöffnet, intitialisiert, besorgt oder organisiert werden muß. Dazu zählt z.B.: Speicher besorgen, Librarys öffnen, Felder löschen, Zugriffe sichern ...

Die zweite Funktion soll entsprechend alles das beinhalten, was am Ende wieder geschlossen werden muß. Denn würden wir z.B. eine Library nicht mehr schließen, so würde der zur Verwaltung benötigte Speicherplatz dem System nie wieder zurückgegeben werden. Daß so etwas nicht passieren darf, liegt auf der Hand!

Die Lösung dieser Aufgaben sieht zwar einfach aus, doch haben wir bis jetzt ein klitzekleines Problem außer acht gelassen: Es könnte ja beim Öffnen der Fall sein, daß aus irgendeinem Grund etwas fehlschlägt (ein Speicherbereich ließ sich nicht bereitstellen...). In dieser Situation sollte ein Fehlertext ausgegeben werden, und das Programm müßte abgebrochen werden. Wir dürfen aber nicht vergessen, vorher alles schon Geöffnete wieder zu schließen. Aber gerade hier liegt die Fehlerstelle!

Wenn man nämlich einfach zur Schließroutine springt, so würden vielleicht auch Sachen geschlossen werden, die gar nicht geöffnet wurden. Aber dann "stürzt" die Betriebssystemroutine unter Garantie ab, denn keine ist gegen diese Möglichkeit gesichert. Wir schreiben deshalb eine Schließfunktion, die in Abhängigkeit von Pointern, die ja bei jedem Öffnen als Resultat gewonnen werden, alles Geöffnete wieder schließt, alles andere aber aus dem Spiel läßt. Sehen wir uns die Funktionen an:

```
/*************************************
 *                                   *
 * Funktion: Alles Nötige öffnen     *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur Intuition *
 *                                   *
 *************************************/
  {
  void          *OpenLibrary();

  if (!(IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", NULL)))
     {
     printf("Keine Intuition Library gefunden!\n");
     Close_All();
     exit(FALSE);
     }
  }
```

Funktion 3.1: Open_All

```
/*************************************
 *                                   *
 * Funktion: Alles Geöffnete schließen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur Intuition *
 *                                   *
 *************************************/
  {
  if (IntuitionBase)   CloseLibrary(IntuitionBase);
  }
```

Funktion 3.2: Close_All

Dokumentation

Die Arbeit der ersten Funktion ist uns schon bekannt. Der Inhalt der zweiten ist fast lächerlich, doch liegt der Hauptteil der Überlegung auf der If-Abfrage, denn ohne sie würde das Programm "abstürzen", sobald durch irgendeinen Umstand die Intuition-Library nicht erreicht und somit nicht geöffnet werden konnte.

Beide Funktionen sehen im ersten Moment zwar einfach und kurz aus, doch haben sie den Sinn und die Aufgabe, ständig ergänzt zu werden! Tippen Sie sie deshalb sorgfältig ab, und speichern Sie beide auf Ihrer Programmdiskette in einem Unterverzeichnis für Funktionsblöcke. Wenn eine von beiden benötigt wird, finden Sie nicht mehr das gesamte Listing vor, sondern nur noch einen Hinweis auf den schon bekannten Teil und die für das Programm benötigten Ergänzungen.

Nach der Erarbeitung unseres elementaren Grundinventars können wir endlich mit dem ersten Programm beginnen, das die Aufgabe hat, ein einfaches Window zu öffnen und nach einiger Zeit wieder zu schließen. Wie schon im vorhergehenden Unterkapitel angesprochen wurde, benötigt Intuition für die Einrichtung eines Windows einige Informationen, die wir über eine Struktur übermitteln. Das Aussehen dieser NewWindow-Struktur ist bekannt. Legen wir jetzt die Eigenschaften des ersten Windows fest:

Im Prinzip ist die Position und Größe für unseren Test egal, doch bitte wählen Sie Breite und Höhe nicht zu klein, denn für die Gadgets und den Window-Rand muß ja noch Platz sein. Ich habe mich für die Position 160, 50 mit der Ausdehnung 320, 200 entschieden. Schreiben wir die Werte in die Struktur, die übrigens FirstNewWindow heißt:

```
FirstNewWindow.LeftEdge   = 160;
FirstNewWindow.TopEdge    = 50;
FirstNewWindow.Width      = 320;
FirstNewWindow.Height     = 200;
```

Da ich das Sizing-Gadget nicht weglassen will, entscheide ich mich für die Screen-Ausdehnung als Maximum. Das Minimum sollte nicht zu klein gewählt werden!

```
FirstNewWindow.MinWidth   = 100;
FirstNewWindow.MinHeight  = 50;
FirstNewWindow.MaxWidth   = 640;
FirstNewWindow.MaxHeight  = 256;
```

Bevor wir uns mit den etwas komplexeren Einstellungen in der NewWindow-Struktur beschäftigen, sollen noch die Parameter abgehandelt werden, mit denen wir uns erst später auseinandersetzen können:

Weil wir noch keinen Screen selbst verwalten können, bleibt nichts anderes übrig, als die Workbench zu benutzen. Der Typ des Windows steht dadurch fest, und ein Zeiger muß nicht gesucht werden.

```
FirstNewWindow.Type       = WBENCHSCREEN;
FirstNewWindow.Screen     = NULL;
```

Eigene Gadget-Definitionen folgen in einem Extrakapitel, und auch die Menüs sollen nicht sofort angesprochen werden:

```
FirstNewWindow.FirstGadget = NULL;
FirstNewWindow.CheckMark   = NULL;
```

Gehen wir jetzt über zu den grafischen Details. Da haben wir zuerst den Window-Titel! Entweder man gibt einen Pointer an, der auf eine mit Null beendete Zeichenkette zeigt - die im weiteren Text als String bezeichnet wird, oder wir übergeben einen Null-Pointer, d.h. wir wollen keinen Text in der Titelzeile. Ist dann gleichzeitig auch kein System-Gadget zugelassen, so wird überhaupt keine Titelleiste dargestellt!

Bei der Arbeit mit dem Aztec-Compiler sollte vor dem String noch eine Cast-Anweisung stehen. Nur so vermeidet man die altbekannten und lästigen Warnings.

```
FirstNewWindow.Title   = (UBYTE *)"Systemprogrammierung Test";
```

Eine interessante Einstellung, deren Möglichkeiten aber nur selten genutzt werden, sind die beiden Farbstifte. Entsprechend des Screens können sie einen Wert zwischen 0 und 31 annehmen, der natürlich von der Anzahl der Bitplanes abhängig ist. Einen besonderen Wert stellt -1 (0xFF) dar! Bei dieser Einstellung wird nicht, wie man vielleicht denken könnte, zufällig eine Farbe ausgesucht, sondern die Farbauswahl richtet sich jetzt nach den Dafault-Werten des Screens. Mit der Implementierung dieser Methode hat man es erreicht, alle Windows eines Screens im gleichen Farbstil auszugeben, und gleichzeitig kann so global die Farbe geändert werden. In der Beispieleinstellung sind zwar auch die Farbwerte der Workbench gewählt, trotzdem wird nicht auf die Default-Werte zurückgegriffen. Dies macht das neue Window unabhängig von den Einstellungen des übergeordneten Screens, auch wenn dieser im Moment die gleichen Werte enthält.

```
FirstNewWindow.DetailPen  = 0;
FirstNewWindow.BlockPen   = 1;
```

Von allen Parametern, die wir bei der Initialisierung beeinflussen können, fehlen nur noch zwei. Es sind zwei Flag-Parameter, deren Möglichkeitenspektrum so komplex ist, daß wir uns etwas genauer damit auseinandersetzen sollten. Richten wir dazu unseren Blick zuerst auf die einfachen Flags. Sie lassen sich grob in fünf Gruppen unterteilen:

1. Die Gadget-Flags

Je nachdem, welches System-Gadget im Window vorhanden sein soll, müssen die Flags gesetzt werden. Folgende stehen zur Auswahl:

WINDOWDRAG

Das Window läßt sich mit Hilfe der Titelleiste im Screen positionieren.

WINDOWDEPTH

Das Window läßt sich durch die beiden Gadgets hinter oder vor andere Windows bringen.

WINDOWCLOSE

Versieht das Window mit dem Schließ-Gadget, das vom Programm aus abgefragt werden kann.

WINDOWSIZING

Versieht das Window mit dem Sizing-Gadget, damit die Größe vom Benutzer geändert werden kann.

2. Die Sizing-Gadget-Position

Aufgrund des letzten Gadgets mußten noch zwei weitere Flags integriert werden, da alle System-Gadgets in den sog. Window-Borders untergebracht werden. Es erscheint klar, daß für die ersten drei der obere Window-Rand genommen wird. Doch beim Sizing-Gadget stehen zwei Möglichkeiten zur Diskussion.

SIZEBRIGHT (SIZ(E)ing-Gadget im RIGHT Border)

Der rechte Window-Rand wird für das Sizing-Gadget verwendet.

SIZEBBOTTOM (SIZ(E)ing-Gadget im BOTTOM Border)

Der untere Window-Rand wird für das Sizing-Gadget verwendet.

3. Die Refresh-Flags

Der Inhalt eines Windows kann leicht durch Überlagerung anderer zerstört werden. Natürlich muß man dem nicht tatenlos zusehen! Intuition bietet verschiedene Möglichkeiten, der Situation zu begegnen:

SIMPLE_REFRESH

Die einfachste Art des Refreshes. Dieser Refresh-Status bedeutet, daß Intuition überhaupt keine Anstrengungen unternimmt, das Window wiederherzustellen. Jede Restauration muß vom Programm bzw. dem Programmierer organisiert werden.

SMART_REFRESH

Hier hat Intuition schon einiges zu tun! Jeder verdeckte Bereich des Windows wird von Intuition zwischengespeichert, d.h. ge-

puffert, und nach optischer Freigabe wieder dargestellt. Probleme bringt diese Methode nur, wenn das Window durch das Sizing-Gadget beeinflußt wurde! Durch Verkleinerungen verschluckte Teile des Windows werden nicht zwischengespeichert und auch nicht wiederhergestellt.

SUPER_BITMAP

Die speicheraufwendigste Art der Window-Sicherung. In einigen Fällen kann es vorkommen, daß eine wesentlich größere Grafik gespeichert werden muß, als mit dem Window dargestellt wird. Dafür gibt es diesen Window-Typ. Er dient also einmal als Refresh-Modus, denn aus der eigenen BitMap der Window-Grafik kann ja immer wieder jede beliebige Information für den Refresh geholt werden, andererseits wird es gleichzeitig unterstützt, Grafiken mit übergroßen Dimensionen zu verwalten.

NOCAREREFRESH

Ruhe auf der Datenleitung! Die Window-Typen SMART_REFRESH und SIMPLE_REFRESH erfordern teilweise oder auch größtenteils die Mitarbeit des Programms, denn bei SMART_REFRESH weiß Intuition beim Sizing-Vorgang nicht mehr weiter, und bei SIMPLE_REFRESH muß bei jeder Überlagerung neu gezeichnet werden, wenn einem am Inhalt etwas liegt. Das Programm erhält deshalb über eine Datenleitung die Nachricht, daß etwas getan werden muß. Die Datenleitung und ihre Handhabung wird später erklärt. Will man aber überhaupt nichts tun, so blockieren ständige Nachrichten sicherlich diese Leitung. Mit diesem Flag kann man das unterbinden und sagen, daß keine Information erwünscht ist.

4. Die Flags für die Window-Typen

Im Abschnitt über die verschiedenen Window-Typen wurden schon die Haupteigenschaften und Anwendungsgebiete aufgezählt. Hier nur noch einmal die Flag-Namen mit kurzem Kommentar:

BACKDROP

Das Window wird immer hinter allen anderen Windows liegen. Hier ist es nur sinnvoll, höchstens ein solches Window pro Screen einzurichten.

SUPER_BITMAP

Obwohl SUPER_BITMAP auch ein Refresh-Typ ist, gilt es gleichzeitig auch als Window-Typ. Es gibt trotzdem nur ein Flag.

BORDERLESS

Intuition unterläßt es, den Window-Rand zu zeichnen (s.u.).

GIMMEZEROZERO

Intuition verwaltet den Rand des Windows, da wo alle System-Gadgets untergebracht werden und auch eigene Gadgets liegen können, getrennt vom Inhalt. Das erleichtert die grafische Handhabung.

5. Die Maus-Flags und der Rest

Wir kommen jetzt zu den drei letzten Flags. Zwei davon beziehen sich auf die Maus. Mit der anderen wird einfach nur eingestellt, ob das Window beim Öffnen gleich aktiv sein soll.

ACTIVATE

Bei gesetztem Flag wird, wie gerade erwähnt, das Window sofort beim Öffnen aktiv. Dies ist sicherlich oft wichtig, denn die Eingabe wird immer in das aktive Window geleitet. Das kann Konsequenzen haben, wenn der Benutzer gerade etwas eingibt, wenn ein neues Window mit diesem Flag aufgemacht wurde. Benutzen Sie solche Windows mit Vorsicht!

REPORTMOUSE

Mit dem Setzen dieses Flags erhält man ständig eine Nachricht über die aktuelle Mausposition.

RMBTRAP (Right Mouse Button TRAP)
Alle Maus-Signale des rechten Maus-Buttons werden in Signale des linken Buttons umgewandelt. Das ist anwendbar, wenn keine Menüs vorhanden sind o.ä.

Alle aufgeführten Flags werden als gesetzte oder gelöschte Bits des Flag-Eintrags in der NewWindow-Struktur dargestellt. In der C-Programmierung können Sie natürlich den bekannten und wesentlich leichter zu merkenden Namen verwenden. Aber für die Assembler-Freunde und alle, die mehr Informationen wünschen, hier eine Tabelle aller Flags und der dazugehörigen Werte:

Flag-Name	Hex-Wert	Gruppe
WINDOWSIZING	0x00000001L	System-Gadgets
WINDOWDRAG	0x00000002L	
WINDOWDEPTH	0x00000004L	
WINDOWCLOSE	0x00000008L	
SIZEBRIGHT	0x00000010L	Gadget-Position
SIZEBOTTOM	0x00000020L	
NOCAREREFRESH	0x00020000L	Refresh-Typen
SIMPLE_REFRESH	0x00000040L	
SMART_REFRESH	0x00000000L	
SUPER_BITMAP	0x00000080L	
BACKDROP	0x00000100L	Window-Typen
GIMMEZEROZERO	0x00000400L	
BORDERLESS	0x00000800L	
ACTIVATE	0x00001000L	
REPORTMOUSE	0x00000200L	Maus-Flags
RMBTRAP	0x00010000L	

Tabelle 3.2: Window-Flags

Als Ergebnis der obigen Analyse aller normalen Window-Flags ergibt sich folgende Definition in der NewWindow-Struktur:

```
FirstNewWindow.Flags  = WINDOWDEPTH | WINDOWSIZING |
                        WINDOWDRAG | WINDOWCLOSE | SMART_REFRESH;
```

Nachdem wir die einfachen Flags betrachtet haben, kommen wir nicht um die IDCMP-Flags herum! Dazu sollten Sie zuerst wissen, was unter "IDCMP" zu verstehen ist. Der Begriff stellt eine Abkürzung aus dem Englischen dar: "Intuition Direct Communication Message Ports", was zu deutsch etwa heißt "Intuitions direkter Kommunikations-Nachrichten-Kanal". Diese wörtliche Übersetzung bringt uns aber nur weiter, wenn wir verstehen, zu welcher Kommunikation dieser direkte Nachrichtenkanal genutzt wird.

Der IDCMP-Kanal ist eine sehr komplexe Datenleitung, die Informationen zu allen auf Intuition bezogenen Bereichen liefert. Hier soll eine Aufzählung ausreichen, denn das Thema ist so umfangreich, daß ihm ein ganzes Kapitel gewidmet wurde. Allerdings werde ich schon hier einige Informationen mit einfließen lassen, denn ganz ohne diesen IDCMP-Kanal kommen Sie bei Intuition nicht aus, ohne irgendwelche großen Einschränkungen hinnehmen zu müssen.

Zu folgenden Themengebieten können Sie Informationen erlangen: Windows, Gadgets, Menüs, Maus, Tastatur, Disk, Preferences, Uhr.

Da wir aber noch überhaupt nichts darüber wissen, behelfen wir uns auf eine trickreiche Art. Anstatt das Window, das ja von unserem Programm geöffnet werden soll, durch ein Close-Gadget zu schließen, wird es nach einer bestimmten Zeit wieder entfernt. Somit haben wir uns wenigstens zu Anfang von dem Vorgriff befreit:

```
FirstNewWindow.IDCMPFlags = NULL;
```

Damit wäre die Strukturdefinition von FirstNewWindow abgeschlossen. Sehen Sie sich einmal das Programm an. Es ist aus den beiden Funktionen Open_All() und Close_All(), der Struktur und einem Hauptprogramm zusammengesetzt :

```
/*************************************
 *                                   *
 * Programm: Window lokale Definition *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  erstes Test-  *
 *                     Window        *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow;

   void          *OpenLibrary();
   struct Window *OpenWindow();

main()
   {

   FirstNewWindow.LeftEdge   = 160;
   FirstNewWindow.TopEdge    = 50;
   FirstNewWindow.Width      = 320;
   FirstNewWindow.Height     = 200;
   FirstNewWindow.DetailPen  = 0;
   FirstNewWindow.BlockPen   = 1;
   FirstNewWindow.IDCMPFlags = NULL;
   FirstNewWindow.Flags      = WINDOWDEPTH ¦ WINDOWSIZING ¦
                               WINDOWDRAG ¦ WINDOWCLOSE ¦
                               SMART_REFRESH;
   FirstNewWindow.FirstGadget= NULL;
   FirstNewWindow.CheckMark  = NULL;
   FirstNewWindow.Title    = (UBYTE *)"Systemprogrammierung Test";
   FirstNewWindow.Screen     = NULL;
   FirstNewWindow.BitMap     = NULL;
   FirstNewWindow.MinWidth   = 100;
   FirstNewWindow.MinHeight  = 50;
   FirstNewWindow.MaxWidth   = 640;
   FirstNewWindow.MaxHeight  = 256;
   FirstNewWindow.Type       = WBENCHSCREEN;

   Open_All();

   Delay(180L);
```

```
   Close_All();
   }


/*************************************
 *                                   *
 * Funktion: Library und Window öffnen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987                *
 *                                   *
 *                                   *
 *************************************/
   {

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will nicht aufgehen!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Funktion: Alles Geöffnete schließen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur Intuition *
 *                     und Window    *
 *                                   *
 *************************************/
   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

Programm 3.1: "Erstes Window"

Dokumentation

Das Listing bringt trotz der bekannten Teile doch noch einige wichtige Neuigkeiten. Zuerst einmal ist die Open_All()-Funktion um das Öffnen des Windows ergänzt worden, dementsprechend natürlich auch die Close_All()-Funktion um das Schließen des Windows. Weitere Erläuterungen zu anderen Intuition-Funktionen bezüglich der Windows kommen im Anschluß an diese Dokumentation.

Wie geschieht nun das Öffnen eines Intuition-Windows, wenn man zuvor seine NewWindow-Struktur definiert hat? Zuerst einmal ist es sehr wichtig, daß die Struktur vor der main()-Funktion definiert wurde, da sonst die Struktur-Werte den anderen Funktionen nicht zugänglich sind. Der OpenWindow-Funktion wird dann einfach die Adresse unserer Definitionsstruktur (NewWindow) übergeben. Als Wert erhalten wir einen Zeiger auf die Window-Struktur zurück, die aufgrund unserer Daten angelegt wurde. Sie enthält noch einige weitere Verwaltungsinformationen. Genauso wie auch bei OpenLibrary überprüft werden mußte, ob kein Fehler aufgetreten ist, darf auch bei OpenWindow() nicht immer damit gerechnet werden, daß alles glücklich abgelaufen ist. Es gibt so viele Gründe, die OpenWindow() verhindern können, z.B. zu wenig Speicherplatz. Die Schutzabfrage ist der schon bekannten angeglichen. Die NewWindow-Struktur kann jetzt wieder gelöscht werden. Sie wurde nur zum Einrichten benötigt!

```
WindowPointer = OpenWindow(NewWindowStruktur)
     D0            -204           A0
```

Aber auch das Schließen des Windows wurde in der Close_All()-Funktion ergänzt. Der Aufbau gleicht auch hier wieder dem schon Bekannten. Wichtig ist, daß erst das Window geschlossen wird und dann die Intuition-Library. Würde man umgekehrt verfahren, so wäre dem Betriebssystem die Funktion CloseWindow() gar nicht mehr bekannt, und ein Absturz wäre sicher!

```
CloseWindow(WindowPointer)
   -72           A0
```

Im Hauptprogramm, in der Funktion main(), wurde zwischen Open_All() und Close_All() eine Verzögerung durch Delay() eingebaut. Diese Betriebssystemfunktion legt den Task unseres Programms so lange in den Wait-Status, bis die angegebene Zeit verstrichen ist (die Zeit wird in "Ticks" angegeben, 50 Ticks entsprechen einer Sekunde), da wir, wie schon angesprochen, noch nicht die Möglichkeit haben, das Close-Gadget abzufragen, um das Window wieder schließen zu lassen. Anstatt der Delay()-Funktion hätten wir natürlich auch eine For-Schleife zur Verzögerung heranziehen können. Doch dann hätten wir das Multitasking verlangsamt, obwohl wir nur warten wollten.

Betrachten wir zum Abschluß noch den Anfang des Programms und seine Kommentarblöcke. Vor der main()-Funktion werden zwei Strukturen definiert, die für unser Window unentbehrlich sind. Zuerst die NewWindow-Struktur mit dem Namen First-NewWindow. Damit reservieren wir Speicherplatz für die Informationen, die Intuition zur Einrichtung unseres Windows benötigt. Die zweite Struktur wird mit einem Pointer auf sie erreicht. Er zeigt auf eine Window-Struktur. Diese ist von der NewWindow-Struktur dadurch zu unterscheiden, daß sie viel mehr Informationen enthält und außerdem mit weiteren Windows verbunden ist, man bezeichnet dies als Linken. Der Aufbau im System, das von Intuition eingerichtet und verwaltet wird, sieht wie folgt aus:

Den Grundstock bildet ein Screen, meistens der Workbench-Screen. In dieser Screen-Struktur, sie wird noch ausführlich im nächsten Kapitel erläutert, wird ein Zeiger aufbewahrt, der auf das erste Window des Screens zeigt. Die Window-Struktur selbst enthält dann wiederum Zeiger auf die folgenden Windows. Existieren mehrere Screens, dann bewahrt der erste Screen auch noch einen Zeiger auf den nächsten Screen auf, bei dem dann die gleiche Window-Verknüpfung abläuft wie eben aufgeführt. Zur besseren Übersicht hier noch eine Grafik:

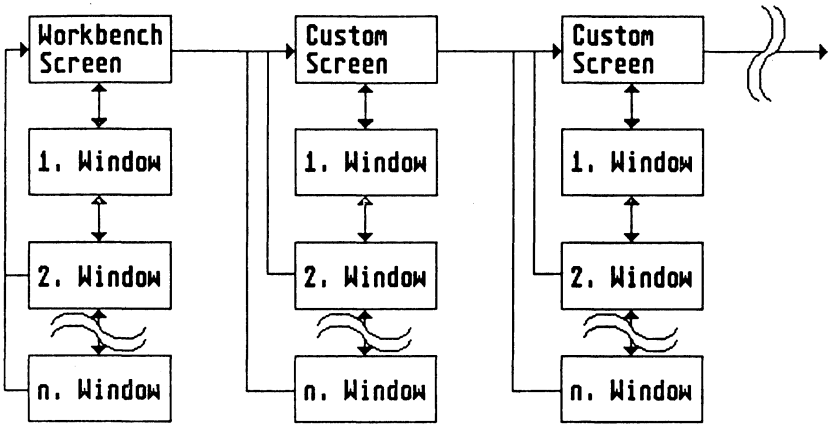


Abbildung 3.1: Screen-/Window-Verknüpfung

Die letzte anzusprechende Struktur-Definition mußte vorgenommen werden, damit das Betriebssystem unsere neuen Funktionen, die Intuition-Funktionen, versteht. Diese sind nur durch die geöffnete Library zugänglich, von der wir die Adresse kennen müssen, die sich aber gerade im Pointer IntuitionBase befindet.

Als letztes sollte etwas zu den Kommentaren gesagt werden. Sie enthalten immer Angaben zum Autor und das Entstehungsdatum. Damit ist der Tag gemeint, an dem die ersten Zeilen geschrieben wurden. Es wird nicht bei jeder minimalen Veränderung gleich ein weiteres Datum hinzugefügt, es sei denn, daß eine elementare Umgestaltung erforderlich war. In der Überschrift wird mit wenigen Worten der Sinn des Programms oder der Funktion dokumentiert. Es empfiehlt sich, diese Methode zu übernehmen, denn Sie werden viele hilfreiche Funktionen darin erkennen.

3.1.2.3 Die Window-Struktur

Nachdem wir das Listing besprochen haben, gilt das größte Interesse der Window-Struktur, die das Betriebssystem für uns eingerichtet hat. Sie enthält wirklich alle benötigten Informationen. Hier ist sie:

```
struct Window
{
0x00  00  struct Window *NextWindow;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0a  10  SHORT Height;
0x0c  12  SHORT MouseY;
0x0e  14  SHORT MouseX;
0x10  16  SHORT MinWidth;
0x12  18  SHORT MinHeight;
0x14  20  USHORT MaxWidth;
0x16  22  USHORT MaXHeight;
0x18  24  ULONG Flags;
0x1c  28  struct Menu *MenuStrip;
0x20  32  UBYTE *Title;
0x24  36  struct Requester *FirstRequest;
0x28  40  struct Requester *DMRequest;
0x2C  44  SHORT ReqCount;
0x2E  46  struct Screen *WScreen;
0x32  50  struct RastPort *RPort;
0x36  54  BYTE BorderLeft;
0x37  55  BYTE BorderTop;
0x38  56  BYTE BorderRight;
0x39  57  BYTE BorderBottom;
0x3A  58  struct RastPort *BorderRPort;
0x3E  62  struct Gadget *FirstGadget;
0x42  66  struct Window *Parent
0x46  70  struct Window *Descendant;
0x4A  74  USHORT *Pointer;
0x4E  78  BYTE PtrHeight;
0x4F  79  BYTE PtrWidth;
0x50  80  BYTE XOffset;
0x51  81  BYTE YOffset;
0x52  82  ULONG IDCMPFlags;
0x56  86  struct MsgPort *UserPort;
0x5A  90  struct MsgPort *WindowPort;
0x5E  94  struct IntuiMessage *MessageKey;
0x62  98  UBYTE DetailPen;
0x63  99  UBYTE BlockPen;
0x64 100  struct Image *CheckMark;
0x68 104  UBYTE *ScreenTitle;
0x6C 108  SHORT GZZMouseX;
0x6E 110  SHORT GZZMouseY;
```

```
0x70 112  SHORT GZZWidth;
0x72 114  SHORT GZZHeight;
0x74 116  UBYTE *ExtData;
0x78 120  BYTE *UserData;
0x7C 124  struct Layer *WLayer;
0x80 128  struct TextFont *IFont;
0x84 132
);
```

Wie auch bei der NewWindow-Struktur wurden vor die einzelnen Variablen Offsets geschrieben, mit denen der Assembler-Programmierer oder auch ein debuggender C-Programmierer eine große Hilfe hat.

Gehen wir nun die Parameter einzeln durch:

**NextWindow*

Ein Zeiger auf die nächste Window-Struktur des Screens. Mit diesem Pointer wird die oben erklärte Verkettung bewerkstelligt.

LeftEdge, TopEdge

Position des Windows auf dem Screen, wie in NewWindow definiert.

Width, Height

Ausdehnung des Fensters, wie es in der NewWindow-Struktur definiert wurde.

MouseX, MouseY

Mauskoordinaten relativ zur linken oberen Ecke des Windows.

MinWidth, MinHeigth

Über die NewWindow-Struktur initialisierte Minimalwerte.

MaxWidth, MaxHeight

Ebenfals initialisierte Maximalwerte.

Flags

Flag-Liste, wie sie in NewWindow initialisiert wurde.

**MenuStrip*

Zeiger auf die Menü-Struktur dieses Windows. Siehe Kapitel 3.8 für nähere Informationen. Über NewWindow kann kein Menü eingesetzt werden. Dies geschieht nur über eine Funktion.

**Titel*

Zeiger auf String, der den Titeltext des Windows enthält.

**FirstRequester*

Zeiger auf den ersten Requester, der in dieses Window gesetzt wurde. Siehe dazu Kapitel Requester.

**DMRequest*

Zeiger auf Double-Menu-Requester. Siehe "Menüs/Requester".

ReqCount

Zähler für Requester, die in diesem Window geöffnet wurden.

**WScreen*

Zeiger auf den Screen, in dem dieses Window geöffnet wurde.

**RPort*

Zeiger auf RastPort, der für dieses Window eingerichtet wurde.

BorderLeft, BorderTop, BorderRight, BorderBottom

Die Werte beschreiben die jeweilige Breite des Randes.

**BorderRPort*

Zeiger auf den RastPort des Borders. Diese Variable wird bei GIMMEZEROZERO-Windows benutzt.

**FirstGadget*

Zeiger auf das erste Gadget einer gelinkten Liste aller Gadgets dieses Windows.

**Parent, *Descendant*

Verkettung mit vorhergehendem und nachfolgendem Window zur Erleichterung der Arbeit beim Öffnen und Schließen.

**Pointer*

Zeiger auf die Grafik des Maus-Cursors dieses Fensters. Er kann für jedes Fenster beliebig definiert werden.

PtrHeight, PtrWidth

Größe des Maus-Cursors (X-Wert darf 16 nicht überschreiten).

XOffset, YOffset

Offsets, die den Punkt des Sprites angeben, an dem der Klickpunkt liegt.

IDCMPFlags

Die vom Programm gesetzten Flags, wie sie auch in der NewWindow-Struktur standen.

**UserPort*

Message-Port zur Übermittlung von Daten.

**WindowPort*

Message-Port zur Übermittlung von Daten.

**MessageKey*

Message-Port für Intuitions-Nachrichten.

DetailPen, BlockPen

Farbstifte, wie sie in NewWindow definiert wurden.

**CheckMark*

Zeiger auf den für die Menüs definierten Haken.

**ScreenTitle*

Zeiger auf den String, der den Screen-Titel darstellt. Er kann nur über die Window-Funktion SetWindowTitles() eingestellt werden. Erklärung siehe weiter unten.

GZZMouseX, GZZMouseY

Mauskoordinaten für GZZ-Window. Bei ihnen wird automatisch der Rand berücksichtigt und vom normalen Wert abgezogen.

GZZWidth, GZZHeight

Da auch die Breite und Höhe des Windows vom Rand des GZZ-Window abhängt, erfährt man die richtigen Maße am besten aus diesen Variablen.

**ExtData*

Zeiger auf externe Datenstrukturen. Bisher wird er nicht genutzt.

**UserData*

Zeiger auf eine Datenstruktur, die vom Programmierer eingerichtet werden kann.

**WLayer*

Zeiger auf die Layer-Struktur dieses Windows. Der gleiche Wert ist auch über RPort->Layer zu erreichen.

**IFont*

Zeiger auf den Font, mit dem in diesem Window Intuition-Text ausgegeben werden.

Die Window-Struktur enthält einige Zeiger auf sehr wichtige Intuition- und Grafik-Elemente.

Die folgende Grafik soll das noch einmal verdeutlichen:

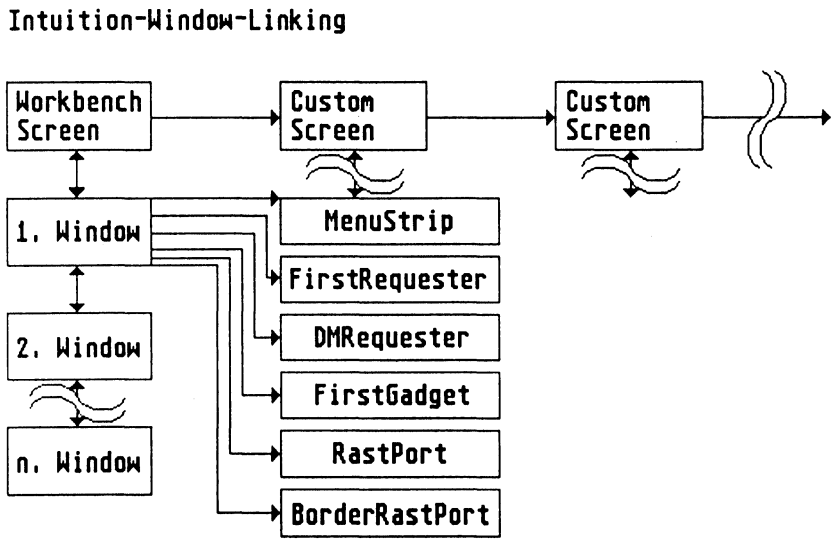


Abbildung 3.2: Intuition-Window-Linking

An alle diese Werte kommt man ganz leicht heran! Wollen Sie z.B. die Breite Ihres GIMMEZEROZERO-Windows wissen, so schreiben Sie einfach:

```
Breite = FirstWindow->GZZWidth;
```

Umgekehrt ist es natürlich auch durchaus möglich, irgendeinen gewünschten Wert zu verändern. Sie wollen unbedingt eine neue Grafik für das CheckMark einsetzen. Dann brauchen Sie nur folgendes zu schreiben:

```
FirstWindow->CheckMark = NeuesCheckMark;
```

Später wird es sehr wichtig sein, daß Sie leicht auf die Werte unseres Windows zurückgreifen können, denn sonst könnte man nie die herrschenden Voraussetzungen überprüfen, die sicherlich eine nicht unwichtige Bedeutung für Reaktionen des Programms haben.

Doch kehren wir mit unseren Betrachtungen zu dem ersten Programm zurück. Sie werden an dem ersten Aufruf unseres Windows erkannt haben, wie einfach die Programmierung von Intuition sein kann. Daran wird sich prinzipiell nichts ändern. Es bleibt nur die Frage, ob man sich noch durch das Listing arbeiten kann, wenn wir vielleicht fünf Windows definieren werden. Abgesehen von der Länge steht einem sicherlich viel Arbeit ins Haus. Wir müßten eine Methode finden, mit der gerade die Struktur-Definition einfacher und nicht so schreibintensiv gemacht wird. Dafür habe ich folgende Definition vorbereitet:

```
struct NewWindow FirstNewWindow =
   {
   160, 50,                  /* LeftEdge, TopEdge   */
   320, 200,                 /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   NULL,                     /* IDCMP Flags         */
   WINDOWDEPTH |             /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                     /* First Gadget        */
   NULL,                     /* CheckMark           */
   (UBYTE *)"Systemprogrammierung Test",
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   100, 50,                  /* Min Width, Height   */
   640, 256,                 /* Max Width, Height   */
   WBENCHSCREEN              /* Typ                 */
   };
```

... was auch noch bis auf die nackten Informationen gekürzt werden kann:

```
struct NewWindow FirstNewWindow =
   {
   160, 50, 320, 200, 0, 1, NULL,
   WINDOWDEPTH|WINDOWSIZING|WINDOWDRAG|WINDOWCLOSE|SMART_REFRESH,
   NULL, NULL, (UBYTE *)"Systemprogrammierung Test",
   NULL, NULL, 100, 50, 640, 256, WBENCHSCREEN
   };
```

Der Vorteil liegt klar auf der Hand: Sie haben wesentlich weniger zu schreiben, weil der Struktur-Name nur einmal aufgeführt werden muß. Ob Sie die Kommentare dahinter schreiben, bleibt

Ihnen überlassen, es empfiehlt sich aber wenigstens bei der ersten NewWindow-Zuweisung.

So wie Sie die Struktur hier gesehen haben, gehört sie natürlich vor die main()-Funktion. Damit ist sie allgemeingültig und jedem Programmteil bekannt. Wollen Sie vielleicht einzelne Werte verändern, so bleibt Ihnen immer noch die alte Methode mit Zugriff auf jedes einzelne Element über dessen Namen. Sie müssen bei der Kurzschreibweise beachten, daß die Reihenfolge entscheidet, was Sie definieren! Durch Verwechslungen können leicht Programmabstürze entstehen.

3.1.2.4 Eine Übersicht aller Window-Funktionen

Die Vorarbeit ist endlich geleistet, auf das Erreichte können wir nun aufbauen! Für die kleineren nun folgenden Demonstrationen bitte ich Sie, als Standard das Programm 1 zu verwenden. Dabei ist es unwesentlich, ob Sie nun die Struktur in der main()-Funktion oder davor definiert haben. Behalten Sie aber bitte zuerst die Parameter bei, damit wichtige Grundvoraussetzungen herrschen. Es ist außerdem üblich und wesentlich einfacher zu handhaben, wenn Sie eine globale Strukturdefinition nach der zweiten Methode vornehmen.

Die ersten beiden Intuition-Funktionen bezüglich der Windows haben wir schon kennengelernt. Zum Öffnen benutzten wir die OpenWindow()- und zum Schließen die CloseWindow()-Funktion. Ihre Anwendung ist Ihnen bestimmt klar geworden, und auch die erwarteten Parameter sind nicht schwer zu behalten.

Achten Sie in allen weiteren Programmen nur darauf, daß alle Windows, die einmal geöffnet wurden, auch wieder geschlossen werden. Denn endet ein Programm, ohne alle Windows zu schließen, so ist der Zeiger auf das verbleibende Window verloren, und es wird bis zum Ausschalten des Computers (oder einem Reset) bestehen bleiben. Ein eventueller Screen, auf dem es sich befindet, ließe sich dann auch nicht mehr schließen!

Sehen Sie sich einmal die weiteren unterstützenden Funktionen an. Da haben wir zuerst SetWindowTitles(). Mit ihr ist es möglich, den Titel eines Windows nach seinem Einrichten zu verändern. Um Ihnen das zu demonstrieren, fügen Sie bitte folgende Zeilen in Ihr Programm (das Standardprogramm 1) ein:

```
(nach Delay(180L)!)
SetWindowTitles(FirstWindow,
  "Neuer Window Titel", "Auch der Screen hat jetzt einen Titel!");
Delay(180L);
```

Wie Sie bestimmt schon an den Texten erkannt haben, wird zuerst dem Window eine neue Überschrift gegeben. Der zweite macht es möglich, daß auch der Screen eine Überschrift bekommt. Wenn Sie nämlich ein Window anklicken, so erhält der Screen-Titel-Balken ein neues Aussehen. In den meisten Fällen, das Programm wurde von der Workbench gestartet, bekommt der Screen keinen besonderen Titel, so auch beim CLI nicht. Es wird einfach der Text "Workbench Screen" in die Leiste gesetzt. Als Programmierer steht es Ihnen aber frei, einen anderen Text zu wählen, der erscheinen soll, wenn Ihr Window aktiv ist. Das Format dieses Befehls sieht wie folgt aus:

```
SetWindowTitles(Window, WindowTitel, ScreenTitel);
    -276          A0         A1           A2
```

Ein selbstverständlicher Parameter, der Pointer auf die Window-Struktur des bezeichneten Windows, wurde noch nicht näher ausgeführt. Ich meine aber, daß es selbstverständlich ist, anzugeben, auf welches der vielen möglichen Windows man sich bezieht. Dieser wichtige Parameter darf bei keiner Window-Funktion fehlen!

Die Texte werden natürlich nicht in ihrer Zeichenkettenform übergeben, sondern sind als Pointer auf einen String bereitzustellen. Doch diese Arbeit nimmt einem wie so vieles der Compiler ab.

Nach dem trockenen Durchspielen von SetWindowTitles() ist es Zeit, sich zu überlegen, bei welchen Anwendungen der Gebrauch nützlich sein könnte. Denken Sie vielleicht einmal an eine Textverarbeitung. Sicherlich bearbeiten Sie dort mehrere

Texte hintereinander oder gleichzeitig. Es ist jedoch immer wichtig zu wissen, an welchem Dokument man gerade arbeitet. Es ist deshalb schon eine Selbstverständlichkeit, daß in der Titelleiste des Editor-Windows der Name des Textes erscheint, damit man immer darüber informiert ist. Als Zugabe schreiben manche Programmierer dort sogar noch den Pfad hinein. So haben Sie Überblick, mit welcher Textdiskette Sie arbeiten. Einen Text in die Titelleiste des Screens zu schreiben, ist nur mit Hilfe von SetWindowTitles() möglich. Über die NewWindow-Struktur wird es nicht unterstützt.

Die Funktion SetWindowTitles() kennt für beide Texte noch zwei Sonderangaben. Stellen Sie sich folgende Situation vor: Die Screen-Titelleiste wird von Ihrem Programm mit seinem Namen und einem Copyright-Vermerk versehen. Haben wir eine Textverarbeitung vor uns, kommt es bestimmt mehrmals vor, daß ein anderer Text geladen wird, und so muß auch der Titel des Windows geändert werden, den des Screens wollen Sie aber gleich behalten. Dafür können Sie ganz einfach -1 anstelle des Pointers einsetzen. So sparen Sie die Arbeit, den Pointer, es könnte ja auch immer ein wechselnder Text sein, immer wieder rauszusuchen.

Die zweite Sondereinstellung gibt dem Programmierer die Möglichkeit, den Text einer der beiden Titelleisten oder auch beider zu löschen. Dies kann der Fall sein, um bei unserem Beispiel zu bleiben, wenn der Textspeicher geleert wurde und noch nichts Neues eingegeben oder geladen wurde. Setzen Sie dafür einfach den oder die Parameter gleich Null.

Ich möchte Sie ermutigen, beide Einstellungen ruhig einmal auszuprobieren. Schreiben Sie dazu einige gewünschte Versionen in Ihr Programm, und verzögern Sie die Ausführung zwischen zwei Befehlen mit der Delay()-Funktion.

Die nächste die Window-Handhabung unterstützende Funktion wurde erst in der Version 1.2 der "intuition.library" ergänzt. Sie macht es dem Programmierer möglich, zu gewünschten Zeitpunkten des Programmablaufs ein Window zu aktivieren. Die Funktion heißt ActivateWindow() (eine plausible englische

Übersetzung). Ihr wird als einziger Parameter der Zeiger auf die Window-Struktur übergeben. Probieren Sie es gleich einmal aus. Fügen Sie bitte folgende Zeilen in das Standardprogramm 1 ein (wir gehen wieder von der Orginalfassung aus, nicht von den Ergänzungen zum SetWindowTitles(), die sie aber troztdem abspeichern sollten):

```
(nach Delay(180L)!)
ActivateWindow(FirstWindow);
Delay(180L);
```

Damit Sie überhaupt ein Ergebnis erkennen, wenn Sie das Programm ablaufen lassen, müssen Sie nach dem Erscheinen des Windows sofort irgendwo in den Screen oder auf ein anderes Window mit der linken Maustaste klicken. Jetzt ist Ihr Window deaktiviert, und nach einiger Zeit wird es automatisch aktiv gesetzt.

```
ActivateWindow(Window);
    -450        A0
```

Der Befehl hat eine große Bedeutung für die Eingabe! Dazu müssen Sie wissen, daß Eingaben immer nur in das aktive Window möglich sind. Fragt ein Programm ab, ob ein Window aktiv ist, das nicht aktiv sein soll, weil keine Eingaben gemacht werden dürfen, dann setzt es ein anderes aktiv. In diesem Fall ist es Ihnen nicht möglich, irgendwelche Informationen oder Anweisungen zu geben. Eine passende Anwendung finden Sie im Kapitel zur Informationsübermittlung "Abfrage der IDCMP-Flags".

Aus welchem Grund auch immer könnte es einmal notwendig sein, die Position eines Windows zu verändern. Vielleicht muß ganz dringend ein neues Fenster an der gleichen Position geöffnet werden, und das alte soll noch vom Inhalt her erkennbar sein. Dann darf man die Arbeit nicht dem Benutzer überlassen, denn dieser Fall könnte öfter auftreten, und das würde ihn verärgern. Wenn Sie also als Programmierer schon wissen, wann eine Positionsänderung unbedingt nötig ist, dann sollten Sie auf die MoveWindow()-Funktion zurückgreifen. Ihr übergeben wir zusätzlich zum Window-Pointer noch zwei relative Werte, die die Verschiebung in x- und y-Richtung angeben.

Aber gerade die Verschiebewerte, im weiteren Text mit Delta-Werten bezeichnet, bergen eine große Gefahr in sich. Wird nämlich durch die Verschiebung das Window auch nur einen Pixel aus dem Screen geschoben, so stürzt das System ab. Wir müssen deshalb unser Programm beauftragen, die Delta-Werte zu überprüfen. Dabei greifen wir auf die Informationen unserer Window-Struktur zurück, in der ja die aktuelle Position gespeichert ist. Fügen Sie bitte folgende Zeilen in das Standardprogramm 1 ein:

```
(nach Delay(180L)!)
DeltaX = FirstWindow.LeftEdge;
DeltaY = FirstWindow.TopEdge;
MoveWindow(FirstWindow, -1*DeltaX, -1*DeltaY);
Delay(180L);
```

Bevor Sie dieses Programm starten können, müssen Sie noch die beiden Delta-Variablen als SHORT definieren! Würde das Programm dann endlich ausgeführt, so brächte es das neu geöffnete Window nach kurzer Zeit in die linke obere Ecke des Screens, denn es verschiebt es genau um die aktuelle Position zum Nullpunkt.

Sie können natürlich auch noch aufwendigere Verschiebungen programmieren, doch diese bot sich an, da bei ihr keine Abfragen zur Delta-Wert-Prüfung nötig sind. Achten Sie auch darauf, daß Sie nicht mit zu vielen Verschiebungen arbeiten, da diese sehr viel Zeit in Anspruch nehmen, besonders, wenn mehrere Windows in einem Screen offen sind.

Zum Schluß noch das allgemeine Format der neuen Funktion:

```
MoveWindow(MeinWindow, DeltaX, DeltaY);
   -168          A0          D0       D1
```

Ein Befehl in ähnlicher Kategorie ist SizeWindow(). Es kann durchaus vom Programm erwünscht sein, das Format eines Windows zu ändern. Einmal in den Fällen, in denen es dem Benutzer nicht gestattet worden ist, dieses selbst vorzunehmen, aber auch, wenn eine größere Arbeit vermieden werden soll. So bieten es gute Programme an, mit einem Gadget-Klick das ganze Arbeits-Window auf kleinste Größe zu bringen. Damit erspart

sich der Benutzer die Arbeit. Umgekehrt geht es natürlich auch. Dann wird zumeist in Verbindung mit MoveWindow() das ganze Fenster in die linke obere Ecke gebracht und auf die Größe des Screens vergrößert.

Ich möchte hier nur das erste Beispiel vorstellen, denn für die letzten beiden ist schon im letzten Teil dieses Kapitels ein Beispielprogramm vorhanden. Ergänzen Sie deshalb Ihr Standard-Listing wie folgt:

```
(nach Delay(180L)!)
DeltaX = LeftEdge - MinWidth;
DeltaY = TopEdge  - MinHeight;
SizeWindow(FirstWindow, -1*DeltaX, -1*DeltaY);
Delay(180L);
```

Auch hier müssen Sie zuerst die Variablen als SHORT definieren, da Sie sonst einen Error erhalten würden.

Wichtig für diesen Window-Typ ist, daß Sie die MinWidth- und MinHeight-Variablen mit Werten belegt haben. Ansonsten tritt das Problem auf, daß das Window auf Pixelgröße verkleinert wird. Das ist aber nicht erwünscht. Dieser Fehler kann nur auftreten, wenn Sie kein Sizing-Gadget gewählt haben und damit auch nicht die Minimum- und Maximum-Werte einstellen brauchen. Stellen Sie diese aber bitte dann trotzdem ein, denn wir möchten ja keinen Absturz provozieren.

Das Format ähnelt sehr stark dem des MoveWindow()-Befehls:

```
SizeWindow(MeinWindow, DeltaX, DeltaY);
   -288        A0          D0       D1
```

Achten Sie auch hier darauf, daß keine unmöglichen Werte für die Deltas eingesetzt werden. Das System stürzt bei jeder Übertretung ab! Das Window darf weder kleiner gemacht werden als möglich noch größer, als der Screen es erlaubt.

Da wir gerade bei der Größenveränderung sind, bietet es sich an, gleich noch die nächste Funktion anzusprechen. Mit WindowLimits() können Sie nachträglich die Vergrößerungs- und Verkleinerungsgrenzen, die Limits, setzen.

Erkennt Ihr Programm z.B., daß es aus Speicherplatzgründen nicht mehr möglich sein kann, das Grafikfenster zu vergrößern, so korrigieren Sie einfach die Maximalwerte. Der Benutzer hat dann nur noch die Möglichkeit, sein Window zu verkleinern. Das Beispiel, das Sie wieder in unser Standardprogramm 1 einfügen können, läßt zuerst die über die NewWindow-Struktur eingestellten Maximal- und Minimalwerte zu. Nach unserer bekannten Verzögerung, wird dieses Maximum verkleinert. Das Fenster läßt sich dann nicht mehr größer einstellen. Am besten sehen Sie sich das Beispiel an und probieren es aus:

```
(nach Delay(180L)!)
MinWidth    = FirstWindow->MinWidth;
MinHeight   = FirstWindow->MinHeight;
NewMaxWidth = 200;
NewMaxHeight= 100;
Erfolg = WindowLimits(FirstWindow, MinWidth, MinHeight,
                      NewMaxWidth, NewMaxHeight);
Delay(180L);
```

Bevor Sie diese Ergänzung ausprobieren können, definieren Sie die Grenzwerte als USHORT und den Ergebniswert als BOOL. Sie erhalten TRUE zurück, wenn kein Wert außerhalb der erlaubten Grenzen lag.

Werden allerdings die Maximalwerte neu gesetzt, und, wenn das Fenster größer ist, als die neuen Werte erlauben, dann erhält das Programm den Wert FALSE als Ergebnis. Zum Abschluß hier noch das Befehlsformat in der allgemeinen Schreibweise:

```
Erfolg = WindowLimits(MeinWindow, MinBreite, MinHöhe,
  D0       -318           A0         D0         D1

                      MaxBreite, MaxHöhe);
                         D2         D3
```

Ein neu geöffnetes Window wird immer vor allen anderen plaziert. Daß dadurch andere verdeckt werden können, ist Ihnen sicherlich aus der täglichen Arbeit mit Intuition bekannt. Für den Programmierer stellt sich aber die Frage, wie er über sein Programm ein Fenster in den Hintergrund bringen kann, wenn es andere stören könnte, und wie er es wieder in den Vordergrund bekommt. Der Benutzer regelt diesen Vorgang einfach

über die Depth-Arrangement-Gadgets. Dafür gibt es zwei Intuitions-Funktionen: WindowToBack() und WindowToFront().

Beide Funktionen benötigen nur den Zeiger auf die Window-Struktur und führen dann ihre Aufgabe durch. Da Sie beim Starten aller Programme sicherlich das DOS-Window offen haben, können wir gleich einmal ausprobieren, ob die Befehle funktionieren. Schreiben Sie folgende Zeilen zu unserem Standardprogramm 1:

```
(nach Delay(180L)!)
WindowToBack(FirstWindow);
Delay(180L);
WindowToFront(FirstWindow);
Delay(180L);
```

Nach kurzer Zeit wird sich das neue Window hinter dem großen DOS-Fenster verstecken und dann wieder auftauchen. Hier noch das allgemeine Format:

```
WindowToBack(Window);
     -306      A0


WindowToFront(Window);
     -312      A0
```

3.1.3 Programmsammlung "Anwendungen mit Windows"

Im ersten Teil dieses Kapitels haben wir uns darauf beschränkt, ausführlich auf die Strukturen und Window-unterstützenden Funktionen einzugehen. Dieser abschließende Teil will nun nicht mehr Lehrbuchcharakter haben, sondern soll anhand von kleinen Beispielprogrammen demonstrieren, wie einfach es durch Intuition wird, seine Ausgabeeinheit Window zu bedienen.

Es ist geplant, zu Anfang einige allgemeine Beispiele zu bringen, die sich überall einsetzen lassen. Wir bieten dadurch die Möglichkeit, noch einmal abzuwägen, wann welcher Window-Typ angebracht ist.

Im zweiten Teil sind Window-Strukturen und Programmteile zu finden, die für das geplante Großprojekt unseres Buches entworfen wurden. Alle werden in der Textverarbeitung, einem Editor, der hauptsächlich für C-Programme geschrieben wurde, benötigt. Haben Sie also Interesse an diesem Spitzeneditor, dann empfiehlt es sich, den Grundstock, der in diesem Kapitel zu finden ist, abzutippen. In den folgenden Unterkapiteln über Screens, Ausgabe, Gadgets usw. finden Sie dann Ergänzungen mit Beschreibung, wie Sie diese in das schon Vorhandene einfügen. Das ganze Intuition-Kapitel kann aber nicht einen fertigen Editor ausarbeiten, wir brauchen noch einiges an Programmierkunst. Dazu finden Sie im dritten Teil, aufbauend auf unser Intuition-Gerüst, Funktionen und Routinen, die sich mit der Textverarbeitung beschäftigen.

Der dritte Abschnitt befaßt sich dann mit den restlichen kleinen Programmbeispielen, die schon im Intuition-Teil vollständig fertiggestellt werden können. Lassen Sie sich überraschen.

3.1.3.1 Windows für jeden Zweck

Wie schon erwähnt, finden Sie hier Beispiellistings, die Ihnen Anregungen für die eigene Programmierung geben sollen. Jedes Listing eignet sich für einen bestimmen Anwendungsbereich. Sie können dann ganz einfach, durch leichte Veränderung der Einstellungen, das Programm Ihren Bedürfnissen anpassen.

Zeichenprogramm (allg. Grafikprogramme)

Gehen wir für das Zeichenprogramm davon aus, daß der Screen, auf dem gezeichnet werden soll, die gewünschten Farbeinstellungen hat. Wir nehmen jetzt dazu die Workbench. Wenn Sie das zweite Kapitel durchgearbeitet haben, können Sie aber auch jeden anderen Screen öffnen.

Da es von keiner Betriebssystemfunktion unterstützt wird, auf einen Screen zu zeichnen, müssen wir den Weg über die Windows gehen. Dabei soll das Window möglichst Screen-ähnlichen

Charakter haben: maximale Größe, vollkommen leere Fläche und verschiebbar in vertikaler Richtung.

Unser Window kann natürlich die maximale Größe des Screens annehmen. Für die leere Fläche müssen wir dann einfach die Randbemalung abstellen. Dafür gibt es das BORDERLESS-Flag! Falls das Programm (Zeichenprogramm) aber andere Windows öffnet, kommen wir in Schwierigkeiten. Denn diese könnten ja hinter unser Window gelegt werden, womit sie dann nicht mehr erreichbar sind, weil unser Fenster keinen Rand mit Gadgets hat. Überall die Depth-Arrangement-Gadgets wegzulassen, widerspricht jedoch jeder guten Programmierkunst und wäre auch nicht benutzerfreundlich. Wir schalten einfach gleichzeitig noch BACKDROP ein, und schon haben wir ein Window mit den Eigenschaften eines Screens!

```
struct NewWindow FirstNewWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge    */
   640, 256,                /* Width, Height        */
   0, 1,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags          */
   BACKDROP |               /* Flags                */
   BORDERLESS |
   SMART_REFRESH,
   NULL,                    /* First Gadget         */
   NULL,                    /* CheckMark            */
   NULL,                    /* Window Title         */
   NULL,                    /* Screen               */
   NULL,                    /* BitMap               */
   0, 0,                    /* Min Width, Height    */
   0, 0,                    /* Max Width, Height    */
   WBENCHSCREEN             /* Type                 */
   };
```

Programm 3.2: Zeichenprogramm

Sie finden hier kein ganzes Listing abgedruckt, weil es gar nicht nötig war, dieses zu tun. Setzen Sie einfach diese Struktur in unser Standardprogramm 1 ein. Dadurch, daß dieses Programm allgemein zusammengebaut wurde, ersparen wir uns Druckarbeit und somit Platz. Außerdem können Sie, wenn Sie das Programm extra abgespeichert haben, es einfach laden und die Struktur neu einsetzen. So müssen Sie nicht immer wieder alles abtippen, denn wir werden es sicherlich noch öfter gebrauchen.

Betrachten wir jetzt die Zusammensetzung der Struktur: Die Position und Ausdehnung wurden entsprechend einem normalen Workbench-Screen gewählt. Das Window enthält kein einziges Gadget, ist dafür aber als BACKDROP und BORDERLESS definiert. Der Refresh-Mode wurde mit SMART_REFRESH genau richtig ausgesucht, denn SUPER_BITMAP ist nur nötig, wenn Grafiken behandelt werden sollen, die größer als die Ausmaße des Windows sind. Auf einen Titel wurde diesmal ganz verzichtet, damit keine Titelleiste erscheint. Somit haben wir, da ja auch keine Gadgets angewählt wurden, einen vollkommen leeren Bildschirm als Zeichenfläche. In diesem Fall ist es sogar gar nicht möglich, das Schließen vom Benutzer anzuweisen. Deshalb eignet sich die Methode mit der Verzögerung über Delay() wunderbar.

Als einziges sichtbares Ärgernis tritt bloß die Titelleiste des Screens in Erscheinung. Sie überdeckt im Normalfall jedes BACKDROP-Window, was aber bei unserem Zeichenprogramm gar nicht erwünscht ist. Wir wollen eine vollkommen leere Zeichenfläche durch unser Window bekommen. Dafür benutzen wir die Funktion ShowTitel(), mit der eingestellt werden kann, ob die Titelleiste des Screens gezeigt oder nicht gezeigt werden soll. Als Argument benötigen wir einen Zeiger auf den betreffenden Screen und den Aussagewert. Das folgende Hauptprogramm löst die Aufgabe, wenn Sie die beiden Funktionen hinzufügen:

```
/****************************************
 *                                      *
 * Programm: Window für Grafikprogramm  *
 * ==================================== *
 *                                      *
 * Autor:  Datum:      Kommentar:       *
 * ------  ----------  ----------       *
 * Wgb     06.12.1987  BACKDROP         *
 *                     BORDERLESS       *
 *                                      *
 ****************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
```

```
struct IntuiMessage  *message;


struct NewWindow FirstNewWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 256,                /* Width, Height       */
   0, 1,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags         */
   WINDOWCLOSE |            /* Flags               */
   BACKDROP |
   BORDERLESS |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   NULL,                    /* Window Title        */
   NULL,                    /* Screen              */
   NULL,                    /* BitMap              */
   0, 0,                    /* Min Width, Height   */
   0, 0,                    /* Max Width, Height   */
   WBENCHSCREEN             /* Type                */
   };


main()
   {

   Open_All();

   ShowTitle(FirstWindow->WScreen, FALSE);

   Delay(180L);

   Close_All();

   exit(TRUE);
   }
```

Programm 3.3: Zeichenprogramm II

Als nächstes möchte ich Sie mit dem GIMMEZEROZERO-Typ der Windows bekannt machen. Wir verwenden dazu ein Programm, das zuerst ein ganz einfaches Window öffnet. Dieses Window wird aber erst dann geschlossen, wenn Sie das Close-Gadget betätigen. Damit kommen Sie endlich in den Genuß der Gadget-Abfrage. Wir erweitern dann das Programm, indem es Ihnen im DOS-Window, über das Sie es ja gestartet haben, die Koordinaten der Maus ausgibt. Dann verändern Sie die Window-Struktur dahingehend, daß Sie jetzt einen GIMMEZEROZERO-

Typ initialisieren. Auch jetzt lassen Sie sich wieder nach dem Starten die Mauskoordinaten ausgeben. Siehe da, der Nullpunkt liegt jetzt unter der Titelleiste!

```
/*************************************
 *                                   *
 * Programm: Window GIMMEZEROZERO Test *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  mit Gadget-   *
 *                     Abfrage       *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewWindow FirstNewWindow =
   {
   160, 50,                 /* LeftEdge, TopEdge   */
   320, 200,                /* Width, Height       */
   0, 1,                    /* DetailPen, BlockPen */
   CLOSEWINDOW,             /* IDCMP Flags         */
   WINDOWDEPTH |            /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   GIMMEZEROZERO |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   (UBYTE *)"GIMMEZEROZERO Test",
   NULL,                    /* Screen              */
   NULL,                    /* BitMap              */
   100, 50,                 /* Min Width, Height   */
   640, 256,                /* Max Width, Height   */
   WBENCHSCREEN             /* Type                */
   };


main()
   {
   ULONG  MessageClass;
```

```
USHORT code;

struct Message *GetMsg();

Open_All();

FOREVER
   {
   if (message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort))
      {
      MessageClass = message->Class;
      code = message->Code;
      ReplyMsg(message);
      switch (MessageClass)
         {
         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;
         }
      }
   }
}
```

Programm 3.4: Gadget-Abfrage

Auch an dieses Listing fügen Sie bitte, wie bereits bekannt, die beiden allgemein definierten Open_All()- und Close_All()-Funktionen an.

Bevor Sie das Programm compilieren und starten, möchte ich noch kurz auf die Abfrage des System-Gadgets eingehen. Zur Nachrichtenübermittlung benötigen wir eine Message-Struktur, die wir am Anfang definieren. Zusätzlich benötigen wir noch eine entsprechende Funktion, mit der wir uns Nachrichten holen können. Die Funktion GetMsg() liefert uns laut Definition den Pointer auf eine solche Datenstruktur. Es ist dann allgemein üblich, diese Nachricht zu unterteilen in MessageClass, die Herkunft der Nachricht, und Code. Dieser Code wird gleich mitberechnet, ist aber für unsere Bedürfnisse noch nicht erforderlich. In einer SWITCH-Abfrage wird dann schließlich getestet, ob es sich um eine Nachricht des Typs CLOSEWINDOW handet - der Wert wurde entsprechend im include-File definiert - und wenn ja, wird das Entsprechende ausgelöst.

Diese Methode läßt sich noch viel weiter ausnutzen, doch dazu gibt es mehrere Kapitel: "Gadgets", "Abfrage der IDCMP-Flags" und "Abfrage der Menüs".

Um noch, wie es ja eigentlich unser Ziel war, die Mauskoordinaten auszulesen, definieren wir zwei weitere Variablen:

```
SHORT Mx, My;
```

Beide erhalten in der FOREVER-Schleife, noch bevor über IF abgefragt wird, ihren Wert zugewiesen durch:

```
Mx = FirstWindow->MouseX;
My = FirstWindow->MouseY;
```

Und werden mit:

```
printf("X: %d Y: %d\n", Mx, My);
```

ausgegeben! Wenn Sie das Ergebnis unserer kleinen Exkursion betrachtet haben, schreiten wir zu den letzten Änderungen.

Zuerst ergänzen Sie die NewWindow-Struktur um das Flag GIMMEZEROZERO. Dann definieren Sie zusätzlich die Variablen Gx und Gy als SHORT und weisen ihnen nach Mx und My folgendermaßen die Werte zu:

```
Gx = FirstWindow->GZZMouseX;
Gy = FirstWindow->GZZMouseY;
```

Auch die printf-Zeile muß ergänzt werden!

```
printf("X: %d Y: %d ; Gx: %d Gy: %d\n", Mx, My, Gx, Gy);
```

Wenn Sie jetzt das Programm laufen lassen, werden Sie einen Unterschied zwischen normalen und GIMMEZEROZERO-Windows erkannt haben!

Aber betrachten Sie doch einmal Ihren Speicherplatz über die Workbench-Anzeige, wenn keines der beiden Programme von eben läuft. Notieren Sie sich den Wert! Starten Sie jetzt das erste Programm - die einfache Koordinatenausgabe - und schreiben Sie sich auch diese Speicheranzeige auf. (Sie erhalten diese,

wenn Sie irgendwo auf die Workbench-Fläche mit der linken Maustaste klicken.) Danach beenden Sie das erste Programm, indem Sie das Close-Gadget betätigen. Nach dem Starten des zweiten Compilats dürfte einiges weniger an Speicher vorhanden sein!

3.1.3.2 Programmteile für Textverarbeitung

Machen Sie sich bereit für das Spitzenprodukt deutscher Programmierkunst, und folgen Sie mir in das Reich der Windows:

Um uns die Arbeit für den Quelltext-Editor so einfach wie möglich zu machen, bereiten wir jetzt schon alle Teile vor, die gebraucht werden. Dafür müssen wir aber zuerst überlegen, was wir brauchen! Keine Angst, ich habe Ihnen die Arbeit abgenommen, denn schließlich haben Sie sich ja deswegen das Buch gekauft.

Für den Editor benötigen wir mindestens drei Windows. Das erste benötigen wir für die Textausgabe. Es wird als Editor-Window bezeichnet.

Das zweite soll uns helfen, die Speicher- und Ladeaktionen leichter zu gestalten. Wir richten darin eine File-Select-Box ein. In solch einer Box finden Sie alle File-Namen eines Inhaltsverzeichnisses und können über die Maus den gewünschten auswählen. Eigentlich handelt es sich dabei um einen Requester, doch die Ergänzungen mit Gadgets und der Requester-Struktur nehmen wir erst später vor. Es ist nur wichtig, daß Sie schon das File haben, damit wir darauf zurückgreifen können.

Das letzte Window heben wir uns als einfache Struktur auf, die wir für irgendwelche Nachrichten-Windows oder Requester benutzen. Hier ist es auch wieder nur wichtig, daß Sie sich ein entsprechendes File einrichten.

Welche Einstellungen soll nun das Editor-Window haben? Um zu Anfang möglichst viel Text darzustellen, empfiehlt es sich, den ganzen Screen, hier den Workbench-Screen, für das Window zu

verwenden. Dieses sind dann auch gleich die Maximalwerte. Für das Minimum kann man sich eigentlich eigene Werte aussuchen, die nur so bedacht sein sollen, daß wenigstens noch das Sizing-Gadget zu erreichen ist. Als letztes sind noch die Flags interessant. Wählen wir am besten alle Gadgets für eine hohe Bedienerfreundlichkeit. Als Refresh-Mode würde ich SMART_REFRESH empfehlen, da SUPER_BITMAP übertrieben ist und das Programm mit SIMPLE_REFRESH zu viel Arbeit hätte. Hier ist also die Window-Struktur:

```
struct NewWindow EditorWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 256,                /* Width, Height       */
   0, 1,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags         */
   WINDOWDEPTH |            /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   (UBYTE *)"Systemprogrammierung C-Editor",
   NULL,                    /* Screen              */
   NULL,                    /* BitMap              */
   100, 50,                 /* Min Width, Height   */
   640, 256,                /* Max Width, Height   */
   WBENCHSCREEN,            /* Type                */
   };
```

Struktur 3.1: Editor-Window

Für die File-Select-Box gilt eigentlich das gleiche wie auch für das Editor-Window. Allerdings soll sie nicht den ganzen Screen einnehmen, um auch noch etwas vom Text sichtbar zu lassen. Als System-Gadgets sind nur noch WINDOWDEPTH und WINDOWDRAG erlaubt, weil WINDOWCLOSE durch ein anderes Gadget ersetzt wird und Sizing die ganze Programmierung wesentlich komplizierter machen würde. Hier also diese Struktur:

```
struct NewWindow FileWindow =
   {
   180, 50,                 /* LeftEdge, TopEdge   */
   250, 150,                /* Width, Height       */
   0, 1,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags         */
```

```
   WINDOWDEPTH |          /* Flags                */
   WINDOWDRAG |
   SMART_REFRESH,
   NULL,                 /* First Gadget         */
   NULL,                 /* CheckMark            */
   (UBYTE *)"File-Select-Box",
   NULL,                 /* Screen               */
   NULL,                 /* BitMap               */
   0, 0,                 /* Min Width, Height    */
   0, 0,                 /* Max Width, Height    */
   WBENCHSCREEN,         /* Type                 */
   };
```

Struktur 3.2: File-Select-Box

Das letzte Window, das wir auf jeden Fall für die Textverarbeitung benötigen, ist ein Nachrichten-Window. Es soll hauptsächlich für Fehlermeldungen oder ähnliches benutzt werden. Deshalb ist es noch etwas kleiner als unsere File-Select-Box. Außerdem fehlen ihm alle System-Gadgets, da es nur gelesen und bestätigt, aber nicht verschoben werden soll. Die Window-Struktur ist ziemlich einfach:

```
struct NewWindow NachrichtenWindow =
   {
   250, 100,             /* LeftEdge, TopEdge    */
   140, 80,              /* Width, Height        */
   0, 1,                 /* DetailPen, BlockPen */
   NULL,                 /* IDCMP Flags          */
   NULL,                 /* Flags                */
   NULL,                 /* First Gadget         */
   NULL,                 /* CheckMark            */
   NULL,
   NULL,                 /* Screen               */
   NULL,                 /* BitMap               */
   0, 0,                 /* Min Width, Height    */
   0, 0,                 /* Max Width, Height    */
   WBENCHSCREEN,         /* Type                 */
   };
```

Struktur 3.3: Nachrichten-Window

3.1.3.3 Window für ein neues CLI

Da es noch mehr Gründe gibt, weshalb man ein Window benötigt, ist dieses Unterkapitel entstanden.

Es ist in Planung, einen kleinen Editor zu schreiben, mit dem man im CLI arbeiten kann. Somit entfällt die lästige und sehr störende Arbeit mit dem alten Editor. Dafür brauchen wir natürlich auch ein Window! Das Hauptprogramm mit der Window-Struktur finden Sie gleich im Anschluß.

```
/*************************************
 *                                   *
 * Programm: Aufbau eines neuen CLI-Ed *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     20.10.1987  nur das Window *
 *                     mit Abfrage   *
 *                                   *
 *************************************/


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

struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewWindow ConsoleWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 101,                /* Width, Height       */
   2, 3,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags         */
   WINDOWDEPTH |            /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   ACTIVATE |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   (UBYTE *)"Wgb Prod. presents  BECKERshell",
   NULL,                    /* Screen              */
   NULL,                    /* BitMap              */
   640, 50,                 /* Min Width, Height   */
   640, 256,                /* Max Width, Height   */
   WBENCHSCREEN,            /* Type                */
   };


main()
   {
```

```
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   FOREVER
      {
      if (message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort))
         {
         MessageClass = message->Class;
         code = message->Code;
         ReplyMsg(message);
         switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
         }
      }
   }


/*************************************
 *                                   *
 * Funktion: Alles öffnen            *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     20.10.1987                *
 *                                   *
 * Keine Parameter                   *
 *                                   *
 *************************************/


Open_All()

   {
   struct Library *OpenLibrary();
   struct Window  *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      Close_All();
      exit(FALSE);
      }
```

```
   if (!(FirstWindow = (struct Window *)
       OpenWindow(&ConsoleWindow)))
      {
      printf("Window will nicht aufgehen!\n");
      Close_All();
      exit(FALSE);
      }
   }

/*************************************
 *                                   *
 * Funktion: Alles schließen         *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     20.10.1987                *
 *                                   *
 * Keine Parameter                   *
 *                                   *
 *************************************/

Close_All()

   {
   if (FirstWindow)
      CloseWindow(FirstWindow);

   if (IntuitionBase)
      CloseLibrary(IntuitionBase);
   }
```

Programm 3.5: Neues CLI

3.2 Screens, eine elementare Darstellungsgrundlage

Bei den Windows haben wir viele Einstellungsfaktoren kennengelernt, mit denen das Aussehen wesentlich beeinflußt werden kann. Eine viel wichtigere Rolle spielen Screens. Jede Einstellung eines Windows steht in gewisser Abhängigkeit zu denen des Screens. Die Farbe, die Auflösung und auch die Lage lassen sich durch den Screen beeinflussen und sind nur durch ihn vollkommen eindeutig definiert.

Wozu dient nun der Screen bei der Arbeit unter Intuition? Da der Screen elementare Parameter in sich birgt, wird er auch elementare Bedeutung haben. Der Screen dient, um gleich auf die Frage zu antworten, im wesentlichen als Untergrund für visuelle Ausgaben aller Art. Doch die Ausgaben haben verschiedene Zwecke und unterliegen demnach auch ganz verschiedenen Grundvoraussetzungen. Nehmen wir dafür zwei krasse Beispiele, an denen es sicher deutlich wird:

Zuerst eine Textverarbeitung. Der Hauptzweck der Bildschirmausgabe ist das lesbare Darstellen des Textes. Doch dazu benötigt man, wie es auch bei Büchern allgemein üblich ist, nur zwei Farben. Wichtig mag es aber sein, daß die Schrift mit einer hohen Punktauflösung ausgegeben wird, damit der Leser am Bildschirm keine Augenschmerzen bekommt.

Als zweites Beispiel soll eine vielfarbige Grafikausgabe dienen. Diese benötigt hauptsächlich, wie ja schon der erste Satz sagte, viele Farben. Die Auflösung ist zwar kein unwichtiges Faktum, doch gibt es Modi (z.B. HAM), bei denen sie gar nicht ins Gewicht fällt.

Schon diese beiden Beispiele machen klar, daß es nötig ist, mehrere Grundvoraussetzungen für die Programme zu bieten. Der Amiga geht hier sogar noch einen Schritt weiter. Er erlaubt es dem Programmierer, durch Screens mehrere verschiedene Modi auf dem Bildschirme gleichzeitig auszugeben, wohingegen es üblich ist, nur einen darzustellen.

Intuition unterstützt es nun, fast beliebig viele dieser Screens mit beliebig vielen verschiedenen Modi einzurichten. Prinzipiell läuft der Vorgang ähnlich wie bei den Windows ab. Die folgenden Seiten werden Ihnen darüber Auskunft geben.

3.2.1 Die Möglichkeiten beim Öffnen eigener Screens

Bei der Analyse der Screen-Eigenschaften wollen wir einen anderen Weg gehen, als wir es bei den Windows gemacht haben. Hier war es nämlich durchaus möglich, daß Sie mehrere Typen

schon über die Workbench kennengelernt haben (Beispiele siehe Demo-Schublade). Screens werden aber im allgemeinen nicht besonders üppig von der Workbench angeboten, und so ist der Amiga-Besitzer leicht in dem Glauben, es gäbe nur seinen Workbench-Screen. Wir wissen aber nun, daß dem nicht so ist.

Die Informationsübergabe läuft bei allen Betriebssystemroutinen über Strukturen. Deshalb gibt es für die Einrichtung eines neuen Screens, genauso wie es für die Einrichtung eines neuen Windows eine NewWindow-Struktur gab, eine NewScreen-Struktur.

3.2.1.1 Die NewScreen-Struktur

Wir wollen nun diese NewScreen-Struktur betrachten und daraus die einzelnen Einstellungen erarbeiten. Hier ist sie:

```
struct NewScreen
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
0x08  08  SHORT Depth;
0x0a  10  UBYTE DetailPen;
0x00  11  UBYTE BlockPen;
0x0c  12  USHORT ViewModes;
0x0e  14  USHORT Type;
0x10  16  struct TextAttr *Font;
0x14  20  UBYTE *DefaultTitle;
0x18  24  struct Gadget *Gadgets;
0x1c  28  struct BitMap *CustomBitMap;
0x20  32
};
```

Auffallend sind, wie auch bei der NewWindow-Struktur, die ersten vier Parameter. Sie haben genau die gleiche Bedeutung wie bei den Windows. Mit LeftEdge und TopEdge wird die Lage des Screens auf dem Bildschirm des Monitors bezeichnet. Bisher ist es aber nur möglich, die Lage vertikal zu verändern. LeftEdge wurde also nur aus Kompatibilitätsgründen implementiert, denn es könnte ja sein, daß Kickstart 1.6 auch das ermöglicht. Doch bisher ist der Video-Chip dazu einfach nicht in der Lage.

Mit Width und Height können wir angeben, wie viele Pixel der Screen breit und hoch sein soll. Dabei ist man durchaus nicht an die bekannten Auflösungswerte gebunden. Ein Screen kann auch nur 100 Pixel breit sein, dafür aber 300 Pixel hoch (im Non-Interlace-Modus). Als Maximalwerte für jeden Screen sind 720*452 Punkte erlaubt, die an den Rändern aus bildschirmtechnischen Gründen dann aber nicht mehr scharf abgebildet werden können.

Ein vollkommen neuer Parameter ist die Einstellung der "Tiefe" eines Screens. Hier stellen Sie ein, wie viele BitPlanes für einen Screen eingerichtet werden sollen. Damit wird gleichzeitig der Speicherverbrauch und die Farbanzahl bestimmt. Uns interessiert natürlich hauptsächlich die Farbanzahl, die in bestimmter Abhängigkeit zum Wert von Depth steht. Nach der folgenden Formel läßt sich die endgültige Anzahl von Farben berechnen: Farben=2^Depth.

Wir müssen dabei aber noch einige Einschränkungen berücksichtigen, auf die ich bei ViewModes zurückkommen werde.

Vorher sehen wir uns erst einmal die bekannten Einstellungen DetailPen und BlockPen an. Beim Window wurden damit die Farben bezeichnet, mit denen Intuition die grafische Gestaltung des Windows vornahm. Übertragen auf den Screen heißt es, daß Intuition durch die beiden Parameter mitgeteilt wird, mit welchen Farben die Titelleiste und die Depth-Arrangement-Gadgets gezeichnet werden sollen.

Zu beachten ist, daß nur Farbnummern erlaubt sind, die durch Depth eingestellt wurden. Haben Sie z.B. eine Tiefe von 3 BitPlanes zugelassen, so ist eine maximale Anzahl von 8 Farben möglich. Diese sind von 0 bis 7 durchnumeriert. Also kann der höchste Wert bei DetailPen oder BlockPen 7 betragen. Genauso verhält es sich auch bei den Windows, die ja vollkommen von den Screen-Einstellungen abhängig sind.

Nach den Farbstiften folgt in der NewScreen-Struktur ein Flag namens ViewModes. Mit ihm können wir die wichtigste Einstellung vornehmen. Wie schon zu Anfang angesprochen wurde, ist

es durchaus normal, wenn verschiedene Programmtypen verschiedene Anforderungen stellen. So benötigt man für die Schrift eine relativ hohe Auflösung und kann bei Grafiken teilweise darauf verzichten. Dafür benutzen wir dieses Flag, mit dem wir noch einiges mehr einstellen können.

Der Normalfall ist, daß Intuition eine Auflösung von maximal 320 Punkten in der X-Richtung annimmt. Damit ist gemeint, daß in dem viereckigen Bereich unseres Bildschirms 320 Punkte auf der Horizontalen Platz finden. Diese Punkte sind relativ breit und auch gut mit dem Auge zu erkennen. Bei hoher Auflösung werden die Punktbreiten dann halbiert, und wir haben 640 Punkte auf der Horizontalen.

Ähnlich läuft es auf der vertikalen Ebene. Normalerweise werden hier bei der PAL-Version des Amiga 256 Zeilen dargestellt. Auch hier läßt sich eine Verdoppelung einschalten, die aber durch ein anderes Verfahren realisiert wird. Der Elektronenstrahl der Bildröhre schreibt einfach noch einmal eine halbe Zeile versetzt über das Bild, diesmal aber mit anderen Bildinformationen. Da das Verfahren hauptsächlich für Grafiken interessant ist, möchte ich Sie hier auf das Supergrafik-Buch von DATA BECKER aufmerksam machen, in dem Sie ausführliche Informationen finden. Dort werden auch die anderen Modi beschrieben, auf die ich hier nicht weiter eingehen will, da sie den Rahmen des Buches sprengen würden.

Als Abschluß hier noch Liste aller möglichen Einstellungen:

Flag-Name	Hex-Wert	Beschreibung
HIRES	0x00008000L	Screen mit doppelter X-Auflösung.
LACE	0x00000004L	Screen mit doppelter Y-Auflösung.
HAM	0x00000800L	4096-Farben-Modus.
EXTRA_HALFBRITE	0x00000080L	64-Farben-Modus bei 5 BitPlanes.
PFBA	0x00000040L	

Flag-Name	Hex-Wert	Beschreibung
DUALPF	0x00000400L	Dual-Playfield-Modus.
SPRITES	0x00004000L	Sprites können verwendet werden.
VP_HIDE	0x00002000L	
GENLOCK_AUDIO	0x00000100L	
GENLOCK_VIDEO	0x00000002L	Bindet über die Hardware ein Video-Bild in die Grafik ein.

Tabelle 3.3: ViewModes

Bevor wir nun zum nächsten Wert der Struktur kommen, möchte ich noch den Zusammenhang zwischen Depth und ViewModes erklären: Da der Video-Chip große Speicherbereiche für unsere Screens verwalten muß, ist er eigentlich schon überlastet. Deshalb darf man es ihm nicht übelnehmen, wenn er bei der Auflösung von 640 Punkten streikt und nicht mehr als 16 Farben zuläßt. Dafür können Sie aber mit 320 Punkten alle maximalen 32 Farben darstellen. Wollen Sie noch mehr Farben, so ist dies nicht über die BitMap-Methode möglich, dafür gibt es den HAM-Modus.

Als nächster Wert wird der Screen-Typ eingestellt. Darunter sind nur zwei Möglichkeiten zu verstehen: als erstes der Workbench-Screen, der nur vom System aus geöffnet wird, und als zweites der CustomScreen (Benutzer-Screen). Zu diesen zwei Flags gesellen sich noch drei weitere, mit denen wir Besonderheiten vermerken können. Intuition muß ja für die BitMaps Speicher organisieren. Dies können wir aber auch selber machen und mit einem später folgenden Parameter einstellen. Damit Intuition weiß, daß wir schon Grafikspeicher besorgt haben, müssen wir dafür das dritte Flag setzen.

Mit dem vierten Flag ist es möglich, den neuen Screen hinter allen anderen zu öffnen. Somit kann man im Verborgenen z.B. eine Grafik aufbauen und braucht den Screen nicht erst im Vordergrund zu öffnen und dann in den Hintergrund zu bringen.

Über das fünfte Flag können wir den Screen "lahmlegen". Das heißt, wir können es Intuition untersagen, die System-Gadgets oder die Titelleiste zu zeichnen. Allerdings muß dann vom Programm aus dafür gesorgt werden, daß der rechte Maus-Button

nicht benutzt werden kann. Dies ist über ein bestimmtes Flag möglich. Ansonsten würde die Titelleiste mit den Menüs gezeichnet werden, da aber kein Löschen erlaubt ist, würde die Leiste nicht wieder entfernt werden.

Hier alle Flags im Überblick:

Flag-Name	Hex-Wert	Beschreibung
WBENCHSCREEN	0x00000001L	Screen-Typ.
CUSTOMSCREEN	0x0000000FL	
CUSTOMBITMAP	0x00000040L	Eigene BitMap.
SCREENBEHIND	0x00000080L	Wird im Hintergrund geöffnet.
SCREENQUIET	0x00000100L	Ohne Gadgets.
SHOWTITLE	0x00000010L	Die beiden letzten Flags werden von Intuition selbst gesetzt.
BEEPING	0x00000020L	Screen blinkt.

Tabelle 3.4: Screen-Typ

Der Screen hat, genau wie auch jedes Window, eine Titelleiste, in der ein Text steht, den wir auch über das Window einstellen können. Ist aber kein Window aktiv, so muß die Leiste nicht unbedingt leer sein! Die nächsten beiden Parameter erlauben es, elementare Texteinstellungen vorzunehmen.

Zuerst können Sie mit Font auf eine TextAttr-Struktur zeigen, mit der Sie den Zeichensatz bestimmen, der für den Screen-Titel und alle anderen Texte von Intuition verwendet wird. Möchten Sie aber den Zeichensatz übernehmen, den der Benutzer mit Preferences ausgewählt hat, so schreiben Sie anstatt des Zeigers den Wert Null an diese Stelle.

Der zweite Wert ist ein einfacher Zeiger auf einen String. Schreiben Sie hier Null, wenn Sie keinen Titel-Text ausgedruckt haben möchten.

Zu dem Pointer auf die Gadget-Struktur kann leider nur gesagt werden, daß er aus Kompatibilitätsgründen mit in die New-Screen-Struktur übernommen wurde. Leider ist es bisher nicht möglich, Custom-Gadgets an einen Screen zu hängen. Sie sollten den Wert deshalb immer mit Null initialisieren, denn sollte es

einmal möglich sein, dann enthielte die Struktur einen Wert, der im schlimmsten Falle zum Absturz führt.

Den letzten Pointer hatte ich teilweise schon unter Type angesprochen. Er zeigt im Normalfall auf eine BitMap-Struktur, die vom Programm selbst initialisiert wurde. Wenn Sie diese Funktion nicht nutzen, so sollte der Wert Null eingetragen werden.

3.2.1.2 Vorab das erste Screen-Listing

Bevor wir uns die Screen-Struktur ansehen, die ja genauso wie auch bei den Windows von Intuition aus der NewScreen-Struktur eingerichtet wird, sollten Sie sich das folgende Listing ansehen. Damit gehen wir von der Theorie wenigstens für einen kleinen Augenblick zur Praxis über:

```
/*************************************
 *                                   *
 * Programm: Window auf Custom Screen *
 * ================================= *
 *                                   *
 * Autor: Datum:      Kommentar:     *
 * -----  ----------  ----------     *
 * Wgb    16.10.1987  einfacher Screen *
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewScreen FirstNewScreen =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 256,                /* Width, Height       */
   1,                       /* Depth               */
   0, 1,                    /* DetailPen, BlockPen */
   HIRES,                   /* ViewModes           */
   CUSTOMSCREEN,            /* Type                */
```

```
    NULL,                       /* Font                 */
    (UBYTE *)"Screen Test",
    NULL,                       /* Gadgets              */
    NULL,                       /* CustomBitMap         */
    };

struct NewWindow FirstNewWindow =
    {
    160, 50,                    /* LeftEdge, TopEdge    */
    320, 200,                   /* Width, Height        */
    0, 1,                       /* DetailPen, BlockPen  */
    CLOSEWINDOW,                /* IDCMP Flags          */
    WINDOWDEPTH |               /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    NULL,                       /* First Gadget         */
    NULL,                       /* CheckMark            */
    (UBYTE *)"Test Custom-Screen",
    NULL,                       /* Screen (kommt noch)  */
    NULL,                       /* BitMap               */
    100, 50,                    /* Min Width, Height    */
    640, 256,                   /* Max Width, Height    */
    CUSTOMSCREEN,               /* Type                 */
    };


main()
    {
    ULONG MessageClass;
    USHORT code;

    struct Message *GetMsg();

    Open_All();

    FOREVER
        {
        if (message = (struct IntuiMessage *)
            GetMsg(FirstWindow->UserPort))
            {
            MessageClass = message->Class;
            code = message->Code;
            ReplyMsg(message);
            switch (MessageClass)
                {
                case CLOSEWINDOW : Close_All();
                                   exit(TRUE);
                                   break;
                }
            }
```

```
    }
  }

/***********************************************
 *                                             *
 * Funktion: Library, Screen und Window öffnen *
 * =========================================== *
 *                                             *
 * Autor:  Datum:      Kommentar:              *
 * ------  ----------  ----------              *
 * Wgb     16.10.1987                          *
 *                                             *
 *                                             *
 ***********************************************/

Open_All()
  {
  void          *OpenLibrary();
  struct Window *OpenWindow();
  struct Screen *OpenScreen();

  if (!(IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library", 0L)))
     {
     printf("Keine Intuition Library gefunden!\n");
     Close_All();
     exit(FALSE);
     }

  if (!(FirstScreen = (struct Screen *)
      OpenScreen(&FirstNewScreen)))
     {
     printf("Screen hat keine Zeit!\n");
     Close_All();
     exit(FALSE);
     }

  FirstNewWindow.Screen = FirstScreen;

  if (!(FirstWindow = (struct Window *)
      OpenWindow(&FirstNewWindow)))
     {
     printf("Window will nicht aufgehen!\n");
     Close_All();
     exit(FALSE);
     }
  }
```

```
/*************************************
 *                                   *
 * Funktion: Alles Geöffnete schließen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  Window, Screen *
 *                     und Intuition *
 *                                   *
 *************************************/

Close_All()
   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (FirstScreen)     CloseScreen(FirstScreen);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

Programm 3.5: Window auf Custom-Screen

Dokumentation

Der Aufbau dieses Listings sollte Ihnen bekannt vorkommen. Nach der allgemeinen Pointer- und Strukturendefinition folgt das Hauptprogramm, das am Anfang die Open_All()-Funktion aufruft und danach in eine Warteschleife verzweigt, die erst dann verlassen wird, wenn Sie das Close-Gadget betätigen. Dann wird nämlich das Programm über Close_All() beendet. Betrachten wir nun einzeln die vier großen Bereiche:

Unter den Pointern finden wir jetzt auch einen für die Screen-Struktur. Er wird benötigt, um später dem Window "eingepflanzt" zu werden. Vorher brauchen wir aber die NewScreen-Struktur, die einen einfachen Screen öffnet, der die gleichen Eigenschaften hat wie auch die Workbench, allerdings mit der Ausnahme, daß wir nur zwei Farben, also eine BitPlane, angefordert haben.

Das Hauptprogramm ist Ihnen so schon bekannt, denn die Gadget-Abfrage des Close-Gadgets wurde am Ende des Window-Kapitels erklärt.

Zu der Open_All()-Funktion ist zusätzlich noch das Öffnen des Screens gekommen, das ja vor dem des Windows und nach dem

der Library abgewickelt werden muß, denn ohne die Library gibt es gar keine OpenScreen()-Funktion, und ohne Screen könnte das Window nicht geöffnet werden. In das Window wird nämlich vorher noch der Zeiger auf den CustomScreen eingesetzt.

Die Close_All()-Routine schließt natürlich erst das Window, dann den Screen und erst dann die Library. Alles läuft in umgekehrter Reihenfolge zum Öffnen ab.

Wenn Sie Lust haben, sollten Sie ruhig einmal einen oder zwei Parameter der NewScreen- oder NewWindow-Struktur verändern. Machen Sie aber zu Anfang nicht zu viele Änderungen, es geschieht dann oft, daß man einen entstandenen Fehler übersieht. Wie wäre es mit zwei weiteren BitPlanes für unseren Screen und gleichzeitig mit anderen Farbstiften im Window? Sie können alle Farben von 0 bis 7 verwenden.

3.2.1.3 Die Screen-Struktur

Nach diesem Abstecher in die Programmierkunst kommen wir aber nicht daran vorbei, uns auch die von Intuition eingerichtete Screen-Struktur anzusehen. Sie enthält viele Werte, Zeiger und Flags, mit denen sich wunderbar arbeiten läßt. Hier deshalb die Struktur mit den Offsets:

```
struct Screen
{
0x00  00  struct Screen *NextScreen;
0x04  04  struct Window *FirstWindow;
0x08  08  SHORT LeftEdge;
0x0A  10  SHORT TopEdge;
0x0C  12  SHORT Width;
0x0E  14  SHORT Height;
0x10  16  SHORT MouseY;
0x12  18  SHORT MouseX;
0x14  20  USHORT Flags;
0x16  22  UBYTE *Title;
0x1A  26  UBYTE *DefaultTitle;
0x1E  30  BYTE BarHeight;
0x1F  31  BYTE BarVBorder;
0x20  32  BYTE BarHBorder;
0x21  33  BYTE MenuVBorder;
0x22  34  BYTE MenuHBorder;
```

```
0x23  35  BYTE WBorTop;
0x24  36  BYTE WBorLeft;
0x25  37  BYTE WBorRight;
0x26  38  BYTE WBorBottom;
0x28  40  struct TextAttr *Font;
0x2C  44  struct ViewPort ViewPort;
    {
    0x00  00  struct ViewPort *Next;
    0x04  04  struct ColorMap *ColorMap;
    0x08  08  struct CopList *DspIns;
    0x0C  12  struct CopList *SprIns;
    0x10  16  struct CopList *ClrIns;
    0x14  20  struct UCopList *UCopIns;
    0x18  24  SHORT DWidth;
    0x1A  26  SHORT DHeight;
    0x1C  28  SHORT DxOffset;
    0x1E  30  SHORT DyOffset;
    0x20  32  UWORD Modes;
    0x22  34  UBYTE SpritePriorities;
    0x23  35  UBYTE reserved;
    0x24  36  struct RasInfo *RasInfo;
    0x28  40
    };
0x54  84  struct RastPort RastPort;
    {
    0x00  00  struct Layer *Layer;
    0x04  04  struct BitMap *BitMap;
    0x08  08  USHORT *AreaPtrn;
    0x0C  12  struct TmpRas *TmpRas;
    0x10  16  struct AreaInfo *AreaInfo;
    0x14  20  struct GelsInfo *GelsInfo;
    0x18  24  UBYTE Mask;
    0x19  25  BYTE FgPen;
    0x1A  26  BYTE BgPen;
    0x1B  27  BYTE AOlPen;
    0x1C  28  BYTE DrawMode;
    0x1D  29  BYTE AreaPtSz;
    0x1E  30  BYTE linpatcnt;
    0x1F  31  BYTE dummy;
    0x20  32  USHORT Flags;
    0x22  34  USHORT LinePtrn;
    0x24  36  SHORT cp_x;
    0x26  38  SHORT cp_y;
    0x28  40  UBYTE minterms[8];
    0x30  48  SHORT PenWidth;
    0x32  50  SHORT PenHeight;
    0x34  52  struct TextFont *Font;
    0x38  56  UBYTE AlgoStyle;
    0x39  57  UBYTE TxFlags;
    0x3A  58  UWORD TxHeight;
    0x3C  60  UWORD TxWidth;
    0x3E  62  UWORD TxBaseline;
    0x40  64  WORD TxSpacing;
    0x42  66  APTR *RP_User;
```

```
    0x46  70  ULONG longreserved[2];
    0x4E  78  UWORD wordreserved[7];
    0x5C  92  UBYTE reserved[8];
    0x64  100
    };
0xB8 184  struct BitMap BitMap;
    {
    0x00  00  UWORD BytesPerRow;
    0x02  02  UWORD Rows;
    0x04  04  UBYTE Flags;
    0x05  05  UBYTE Depth;
    0x06  06  UWORD pad;
    0x08  08  PLANEPTR Planes[8];
    0x28  40
    };
0xE0 224  struct Layer_Info LayerInfo;
    {
    0x00  00  struct Layer *top_layer;
    0x04  04  struct Layer *check_lp;
    0x08  08  struct Layer *obs;
    0x0C  12  struct MinList FreeClipRects;
    0x18  24  struct SignalSemaphore Lock;
    0x3C  60  struct List gs_Head;
    0x4A  74  LONG longreserved;
    0x4E  78  UWORD Flags;
    0x50  80  BYTE fatten_count;
    0x51  81  BYTE LockLayersCount;
    0x52  82  UWORD LayerInfo_extra_size;
    0x54  84  WORD *blitbuff;
    0x58  88  struct LayerInfo_extra *LayerInfo_extra;
    0x5C  92
    };
0x13C 316 struct Gadget *FirstGadget;
0x140 320 UBYTE DetailPen
0x141 321 UBYTE BlockPen;
0x142 322 USHORT SaveColor0;
0x144 324 struct Layer *BarLayer;
0x148 328 UBYTE *ExtData;
0x14C 332 UBYTE *UserData;
0x150 330
};
```

Auch wenn dies die längste Intuition-Struktur ist, kommen wir doch nicht an ihr vorbei. Wenn man bedenkt, daß sich an dieser Struktur alle anderen orientieren, so sollten wir besondere Sorgfalt auf ihre Betrachtung legen.

**NextScreen*

Wie die Windows werden auch alle geöffneten Screens miteinander verbunden. Der Linkpointer steht in dieser Variablen.

**FirstWindow*

Jeder Screen enthält hier den Pointer auf das erste Window, das in ihm geöffnet wurde. Die Windows haben dann unter sich wieder einen Linkpointer.

LeftEdge, TopEdge, Width und Height

Diese vier Werte sind schon bei der NewScreen-Struktur erklärt.

MouseX, MouseY

Wie auch bei den Windows finden Sie hier die Mauskoordinaten, aber natürlich in bezug auf unseren Screen.

Flags

Dieser Wert wird initialisiert, wie wir es in unserer NewScreen-Struktur gemacht haben. Im Verlauf der weiteren Anwendungen findet man hier aber noch zwei neue Flags:

SHOWTITLE

Dieses Flag wird gesetzt, wenn durch ShowTitle() die Titelleiste des Screens gezeigt werden soll.

BEEPING

Dieses Flag wird in der Zeit gesetzt, in der der Screen durch DisplayBeep() aufblinkt.

**Title*

Genau wie bei den Windows steht hier ein Pointer auf den Titeltext-String.

**DefaultTitle*

Dieser Pointer tritt in Aktion, wenn ein Window keinen Screen-Titel bestimmt hat. Dann verwendet Intuition den DefaultTitle.

Die folgenden neun Variablen der Screen-Struktur gelten allgemein auf alle Windows des Screens und den Screen selbst!

BarHeight

Gibt die Höhe einer Titelleiste in Bildschirmzeilen an.

BarVBorder
Breite des vertikalen Randes.

BarHBorder
Breite des horizontalen Randes.

MenuVBorder
Breite des vertikalen Randes der Menüs.

MenuHBorder
Breite des horizontalen Menürandes.

WBorTop
Breite des oberen Window-Randes.

WBorLeft
Breite des linken Window-Randes.

WBorRight
Breite des rechten Window-Randes.

WBorBottom
Breite des unteren Window-Randes.

**Font*
Zeiger auf den allgemein von Intuition zu benutzenden Font. Initialisiert durch NewScreen-Struktur.

ViewPort
Hier wird im Gegensatz zu allen anderen Struktureinbindungen nicht ein Pointer auf die Struktur aufgenommen, sondern die Struktur selbst. Der ViewPort enthält Informationen zur Darstellung, die der Copper, der Video-Chip, benötigt.

RastPort
Die RastPort-Struktur, die auch wieder vollkommen in die Screen-Struktur übernommmen wurde, dient zur Verwaltung des

Screens und enthält alle Informationen, die für ein Zeichnen auf dem Screen benötigt würden. Weitere Informationen zu diesen elementaren Grafikstrukturen im Supergrafik-Buch.

BitMap

Die dritte eingebundene Struktur enthält genauere Angaben zu der Zeichenfläche, der BitMap. Weiterhin findet man hier Pointer auf jede einzelne BitMap.

LayerInfo

Layers sind die Grundlage der Window-Verwaltung. Diese Info-Struktur bildet nur den Anfang.

**FirstGadget*

Als nächstes finden wir den Pseudo-Pointer auf die Custom-Gadgets des Screens wieder. Natürlich ist er hier genauso wenig unterstützt, wie er es bei der NewScreen-Struktur war. Warten wir dafür einfach Version 1.8 oder höhere ab!

DetailPen, BlockPen

Die zwei Zeichenstifte, die schon unter NewScreen und NewWindow erklärt wurden.

SaveColor0

Da bei einem Screen-Blinken die Hintergrundfarbe gewechselt wird, also Farbe 0, muß diese irgendwo zwischengespeichert werden. Das geschieht in dieser Variablen.

**BarLayer*

Hier haben wir den Zeiger auf den Layer, Grafikspeicherbereich, in dem die Titelleiste des Screens aufbewahrt wird.

**ExtData*

Zeiger auf externe Daten, die zusätzlich angelegt werden können.

**UserData*
Zeiger auf Daten, die vollständig vom Benutzer verwaltet werden können.

3.2.1.4 Die Screen-Funktionen

Wie Sie ja schon in dem Beispielprogramm kennengelernt haben, gibt es für Screens auch eine Open()- und eine CloseScreen()-Funktion. Die Parameter sind der Window-Funktion ähnlich: zuerst die New-Struktur und danach die von Intuition angelegte.

Wie Sie aber bestimmt richtig vermutet haben, gibt es nicht nur eine Funktion zum Öffnen und eine zum Schließen. Viele andere helfen dem Programmierer bei seiner Arbeit und machen die Gestaltung einfacher. Betrachten wir deshalb jede einzeln.

So wird z.B. auch das Anklicken der Depth-Arrangement-Gadgets über Befehle unterstützt. Mit ScreenToFront() oder ScreenToBack() können Sie jeden Screen, zu dem Sie einen Pointer besitzen, in den Vorder- oder Hintergrund bringen. Um Ihnen dies einmal demonstrieren zu können, entfernen Sie bitte aus unserem Screen-Beispielprogramm die Gadget-Abfrage. Das Hauptprogramm sollte dann so aussehen:

```
main()
   {

   Open_All();

   Delay(180L);

   Close_All();
   exit(TRUE);
   }
```

Die Delay()-Zeile mußte noch zusätzlich eingefügt werden, damit Sie auch etwas davon merken, daß ein neuer Screen mit Window geöffnet wurde. Alle Ergänzungen fügen Sie dann bitte nach dem Delay(180L); ein, wenn es nicht anders gesagt wird. Als erstes hier ein Beispiel für ScreenToBack() und ScreenToFront():

```
ScreenToBack(FirstScreen);
Delay(180L);
ScreenToFront(FirstScreen);
```

Es handelt sich hier zwar um ein ganz einfaches Beispiel, denn der Screen wird nur einmal in den Hintergrund gebracht und dann nach kurzer Zeit wieder nach vorne, doch Sie sollen ja nur die Funktionen kennenlernen. Hier erst einmal die allgemeine Funktionsschreibweise mit Parametern, Registerangaben und Funktions-Offset:

```
ScreenToBack(Screen);
     -246       a0

ScreenToFront(Screen);
     -252       A0
```

Die gleichen Möglichkeiten, die für jeden CustomScreen vorgesehen sind, lassen sich natürlich auch auf die Workbench übertragen. Meistens hat man aber das Problem, daß man nicht im Besitz des Pointers auf den Workbench-Screen ist. Deswegen gibt es zwei weitere Funktionen, die das auch ohne Pointer erledigen:

```
WBenchToBack();
    -336

WBenchToFront();
    -342
```

Fügen Sie diese drei Zeilen ein:

```
WBenchToFront();
Delay(180L);
WBenchToBack();
```

Von BASIC ist es Ihnen vielleicht bekannt: Sie versuchen, im Editor mit dem Cursor aus dem Textbereich "herauszulaufen", und der Bildschirm blinkt. Dieses Blinken ist eine gekoppelte Warnaktion, die von Intuition ausgegeben wird. Sowohl ein optisches Signal, das Bildschirmblinken, als auch ein akustisches Signal, der Piepton, wird von diesem Befehl ausgesendet. Die folgende Beispielroutine liefert fünf Warnsignale hintereinander:

```
for (i=1; i<6; i++)

   for (j=0; j<10000; j++);
   DisplayBeep(FirstScreen);
```

Zum Starten des neuen Programms definieren Sie vorher noch die beiden Laufvariablen i und j als integer. Das Programm läßt danach fünfmal den neuen CustomScreen aufblinken.

Möchte man alle Screens aufblinken lassen, weil z.B. dem Programm nicht bekannt ist, ob der betreffende Screen überhaupt im Vordergrund ist, dann kann man als Pointer auch NULL angeben, um Intuition mitzuteilen, daß alle Screens blinken sollen.

```
DisplayBeep(NULL);
```

Zum Abschluß dieser Funktion noch das allgemeine Format:

```
DisplayBeep(Screen);
     -96       A0
```

Bei manchen Anwendungen kann es durchaus vorkommen, daß nicht genügend Grafikspeicher zur Verfügung steht. Dies läßt sich auch durch eine Speichererweiterung nicht beheben, denn der Chip-RAM-Bereich kann nicht über 512 KByte ausgebaut werden.

Nachdem man sich bei Commodore einige Gedanken zu diesem Thema gemacht hat, ist man zu dem Ergebnis gekommen, daß während eines Programmablaufs die Workbench nicht gebraucht wird, wenn das Programm einen eigenen Screen geöffnet. In solchen Fällen kann man den Workbench-Screen schließen. Die einzige Bedingung dafür ist, daß kein weiteres Programm läuft, das seine Ausgabe in einem Window der Workbench vornimmt. Ist diese Bedingung erfüllt, dann erlaubt es Intuition mit CloseWorkBench(), den Arbeitstisch zu schließen.

Wie schaffen wir es nun, ein Programm zu starten, ohne daß sich ein anderes auf der Workbench "breitmacht"? Leider können wir unser Programm nicht von dem CLI-Window aus starten, denn dieses CLI ist ein eigenständiges Programm, welches seine Ausgabe auf unserer Workbench macht. Deshalb besorgen wir

uns für unser Testprogramm ein Icon, so daß es auch über einen Doppelklick aufzurufen ist.

Compilieren Sie bitte das Standardprogramm 2, nachdem Sie es um die folgenden Zeilen ergänzt haben, und sichern Sie es auf Diskette. Denken Sie aber daran, mit "startup.o" zu linken(!):

Am Ende der Open_All()-Funktion nach Öffnen des Windows:

```
CloseWorkBench();
```

Am Anfang der Close_All()-Funktion:

```
OpenWorkBench();
```

Jeder Screen, dessen Titelleiste nicht durch ein BACKDROP-Window verdeckt ist, kann vom User über die linke Maustaste vertikal verschoben werden. Genauso, wie man es auch bei Windows vom Programm aus übernehmen kann, erlaubt eine weitere Intuition-Funktion es auch, den Screen zu verschieben. Mit MoveScreen() und den entsprechenden Delta-Werten ist das kein Problem.

Ich habe für Sie zwei Zeilen vorbereitet, die den Screen immer wieder nach oben schieben, wenn Sie ihn nach unten gezogen haben. Fügen Sie die Zeilen in das Standardprogramm 2 in die FOREVER-Schleife noch vor der Abfrage ein:

```
if (FirstScreen->TopEdge != 0)
   MoveScreen(FirstScreen, 0L, -1L);
```

Das allgemeine Format von MoveScreen() habe ich auch gleich parat:

```
MoveScreen(Screen, DeltaX, DeltaY);
    -162     A0      D0      D1
```

Da der ShowTitle()-Befehl schon im Window-Kapitel erklärt worden ist, bleiben noch zwei Funktionen, die sich auf Screens beziehen. Da haben wir zuerst die Funktion GetScreenData(), durch sie erhalten wir einen Datenüberblick zum angegebenen Screen.

Sicherlich lassen sich alle Werte auch über einen einfachen Strukturzugriff abfragen, doch hat man dann nicht den Überblick zu einem bestimmten Zeitpunkt, denn alle anderen Parameter können sich ja währenddessen ändern! Also richten wir uns für diese Funktion einen Puffer ein, in dem später alle Daten untergebracht werden. Dann rufen wir die Funktion auf, und alle Parameter sind in unseren Speicherbereich übertragen. Besonders sinnvoll ist diese Anwendung, wenn man die Daten des Workbench-Screen besorgen möchte. Da wir in einem Flag den Typ des Screen angeben können, ist es ein einfaches, dort Workbench einzusetzen. So kommen wir auch an diese Daten, die sonst nicht so einfach zu erreichen sind. Da ein Beispiel ziemlich umfangreich ist, hier nur die allgemeinen Parameterangaben:

```
Succes = GetScreenData(Buffer, Size, Type, Screen);
  D0          -426       A0     D0    D1     A1
```

Über MakeScreen() können wir einen von Intuition verwalteten Screen manipulieren, ohne Konflikte zu bekommen. Nach dem Aufruf von MakeScreen() und der Übergabe eines Screen-Pointers wartet er, bis der Intuition-View nicht mehr verwendet wird, erledigt währenddessen über MakeVPort() die Arbeit und gibt danach den Intuition-View wieder frei.

Sicherlich haben Sie die obige Aufführung der Tätigkeit von MakeScreen nicht gleich verstanden. Das dafür benötigte Grundwissen mag Ihnen vielleicht fehlen. Wenn Sie das Thema interessiert, so muß ich Sie auf das Supergrafik-Buch verweisen, in dem unter dem Stichwort MakeVPort() einige wissenswerte Informationen stehen, die einfach den Rahmen dieses Kapitels sprengen würden. Bitte haben Sie dafür Verständnis. Ich möchte nur der Vollständigkeit halber hier trotzdem das allgemeine Format liefern:

```
MakeScreen(Screen);
   -378       A0
```

Damit sind alle Funktionen, die sich auf die Screens von Intuition beziehen, beschrieben. Ich hoffe, daß Ihnen die kleinen Beispiele zum Verständnis geholfen haben. Im letzten Abschnitt dieses Screen-Kapitels finden Sie noch einige Beispielpro-

gramme, die einerseits nur die Programmierung noch verdeutlichen sollen und andererseits auch für unser Großprojekt benutzt werden können. Bei den allgemeinen Programmen finden Sie auch noch eines, mit dem Sie Ihre Workbench manipulieren können. Ich wünsche schon jetzt viel Spaß!

3.2.2 Anwendungsbeispiele für Screens

Abschließend möchte ich einige kleine Programme und ergänzende Programmteile vorstellen, die Sie in Ihre schon bestehenden Programme einfügen können.

Leider haben wir hier lange nicht so viele Beispiele. Das liegt daran, daß Screens nur für spezielle Anwendungen benötigt werden. Da wir uns hier aber nur auf den Intuition-Aspekt konzentrieren, ist es nicht ganz so wichtig.

3.2.2.1 Allgemeine Beispiele

Unter dieser Überschrift möchte ich zu dem bekannten Thema Grafikprogramme zwei Screen-Beispiele liefern, die Ihnen vielleicht Anregungen zu eigenen Programmen geben.

Zuerst einen superhochauflösenden Screen mit nur einer BitPlane. Superhochauflösend heißt, daß er mit 640*512 Punkten auf einem PAL-Fernseher läuft. Weiterhin habe ich nur eine BitPlane angewählt, weil schon genügend Speicher durch die hohe Auflösung verlorengeht. Es ist nämlich immer zu bedenken, daß ja maximal nur 512 KByte für Grafikspeicher zur Verfügung stehen, da nur dieser Teil von den Chips angesprochen werden kann (CHIP_MEMORY).

```
struct NewScreen SuperScreen =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 512,                /* Width, Height       */
   1,                       /* Depth               */
   0, 1,                    /* DetailPen, BlockPen */
   HIRES |                  /* ViewModes           */
   HAM,
```

```
    CUSTOMSCREEN |          /* Type              */
    SCREENQUIET,
    NULL,                   /* Font              */
    NULL,
    NULL,                   /* Gadgets           */
    NULL                    /* CustomBitMap      */
    };
```

Struktur 3.3: Super-Screen

Dieser Modus eignet sich außer für digitalisierte Bilder auch noch für Grafiken, die Sie in Ihrem Programm verwenden wollen. Natürlich können Sie damit ganz besondere hochauflösende und komplexe Bilder darstellen.

Im zweiten Beispiel möchte ich darauf eingehen, daß es ja nicht unbedingt nötig ist, auch wirklich die ganze Auflösung auszunutzen. So kann auch ein Interlace-Screen ganz normal mit 256 Zeilen initialisiert werden. Er nimmt dann eben nur die Hälfte des Bildschirms ein. Unter diesem Gesichtspunkt sei noch erwähnt, daß bei Verwendung eines Interlace-Screens jeder andere Screen, der dargestellt wird, mitflackert. Ein nicht immer wünschenswerter Nebeneffekt.

```
struct NewScreen Utility =
    {
    0, 200,                 /* LeftEdge, TopEdge   */
    640, 56,                /* Width, Height       */
    2,                      /* Depth               */
    2, 1,                   /* DetailPen, BlockPen */
    HIRES,                  /* ViewModes           */
    CUSTOMSCREEN,           /* Type                */
    NULL,                   /* Font                */
    (UBYTE *)"Utility-Screen",
    NULL,                   /* Gadgets             */
    NULL                    /* CustomBitMap        */
    };
```

Struktur 3.4: Utility-Screen

Die hier vorgestellte Struktur öffnet in den unteren 56 Zeilen des Workbench-Screen einen weiteren. Hier können Sie vielleicht ein BORDERLESS-Window hineinlegen, in dem mit Gadgets Hilfsfunktionen eines Utilities unterstützt werden.

3.2.2.2 Programmteil "Screen" für Textverarbeitung

Wenn Sie unseren C-Quelltext-Editor während der Arbeit mit dem Compiler laufen lassen, so kann es wünschenswert sein, daß er nach dem Editieren vollkommen vom Screen entfernt wird, weil z.B. auf der Workbench schon zu viele Windows geöffnet wurden und die Verarbeitungszeit somit ziemlich lange dauert. In solch einem Fall ist es geeignet, für ein neues Programm einen neuen Screen zu öffnen. Dann können Sie sich bei Bedarf vom Programm trennen und auf der Workbench ungestört weiterarbeiten.

Die vorliegende Struktur ist besonders für eine Textverarbeitung eingerichtet. Sie nimmt nur eine BitPlane in Anspruch, damit möglichst wenig Speicher verbraucht wird. Dies müßte ausreichen, denn Farben sind im Text ja nicht unbedingt nötig. Allerdings wird die hohe Auflösung angewählt, damit man einen gut leserlichen Text bekommt.

```
struct NewScreen EditorScreen =
   {
   0, 0,                        /* LeftEdge, TopEdge   */
   640, 256,                    /* Width, Height       */
   1,                           /* Depth               */
   1, 0,                        /* DetailPen, BlockPen */
   HIRES,                       /* ViewModes           */
   CUSTOMSCREEN,                /* Type                */
   NULL,                        /* Font                */
   (UBYTE *)"DATA BECKER C-Editor Screen",
   NULL,                        /* Gadgets             */
   NULL,                        /* CustomBitMap        */
   };
```

Struktur 3.5: Editor-Screen

3.3 Ausgabe - denn ohne geht es nicht

Sie sind nach dem Abschluß des 2. Unterkapitels über alle Fähigkeiten der Intuition-Screens und -Windows informiert. Wie man sie öffnet und schließt und auch, wie man über andere Funktionen mit ihnen arbeitet.

Es ist aber nicht unbedingt der Sinn der Screens und Windows, einfach auf dem Bildschirm zu verbleiben und auf nichts weiter zu warten, als vom Benutzer hin- und hergeschoben zu werden.

Ganz am Anfang hatte ich schon erwähnt, daß die Windows die Grundlage für jede Ein- und Ausgabe sind. Die Screens dienen nur dazu, eine Unterteilung vorzunehmen und festzulegen, welche elementaren Eigenschaften die Windows haben. Diese haben wir festgelegt, und wir können einen Screen einrichten. Wie man mit Windows umgeht, ist uns auch hinreichend bekannt. Aber wie steuert man die Ein- und Ausgabe?

Wir werden uns in diesem Unterkapitel mit der Ausgabe beschäftigen. Der Amiga bietet zwei Arten der Ausgabe. Am grundlegendsten sind die Ausgabefunktionen der "graphics.library". Sie bietet den gesamten Standard der Ausgabefunktionen.

Die zweite Möglichkeit wird von der "intuition.library" unterstützt. Sie bietet nur drei verschiedene Ausgabefunktionen, die aber ausreichend sind, um alle weiteren Elemente darauf aufzubauen. Alle Kommunikationselemente wie Gadget, Requester und Menüs greifen auf die Hilfe der drei Grafikfunktionen von Intuition zurück.

Sie sehen daran, wie wichtig es ist, daß wir uns jetzt damit beschäftigen. Sie werden später begeistert sein, wie einfach sich durch das Baukastensystem von Intuition die komplexesten Objekte aufbauen lassen.

Außerdem möchten Sie doch sicherlich endlich einmal die Windows benutzen, die wir immer geöffnet haben, aber nie für etwas verwendeten.

3.3.1 Textausgabe, unsere wichtigste Kommunikation

Wir werden jetzt Schritt für Schritt die einzelnen Ausgabemöglichkeiten durchgehen. Dazu nehmen wir zuerst die Textausgabe, weil sie die wichtigste Kommunikation unter den Menschen darstellt.

Es ist zu Anfang wichtig zu wissen, welche Einstellungen wir an unserem Text vornehmen können. Sie sind leicht zu verstehen.

3.3.1.1 Position, Farbe, Zeichenmodus...

Zuerst einmal die Position. Wir können in bezug auf unser Fenster punktgenau festlegen, wo der Text geschrieben werden soll. Weiterhin können wir sowohl für den Hintergrund als auch für den Vordergrund (die Textlinien selbst) die Zeichenfarbe bestimmen. Hier greifen wir zurück auf eine bestimmte Anzahl von Farbstiften, die uns durch den Screen bereitgestellt werden. Eine Sonderrolle spielt der Farbstift mit der Nummer -1 (0xFF)! Bei seiner Auswahl wird mit der Default-Farbe des Windows gezeichnet.

Wie es Ihnen vielleicht schon bekannt ist, gibt es verschiedene Modi, mit denen das Betriebssystem seine Ausgabe vornehmen kann.

Der erste Modus betrachtet nur das zu Zeichnende selbst und nicht den Hintergrund. Auf unseren Text bezogen hieße das, daß nur die Textlinien selbst, nicht aber der Schrifthintergrund ausgegeben wird. Dadurch kann man seine Zerstörung vermeiden. Den Modus bezeichnet man als JAM1. Nur die Farbe 1 wird in die Grafik geJAMt.

Im zweiten Modus benutzt Intuition beide Stifte, um sowohl den Hintergrund als auch die Schrift auszugeben. Der Modus heißt JAM2, weil beide Farben gezeichnet werden.

Der dritte Modus gleicht im großen und ganzen dem zweiten, mit dem Unterschied, daß beim Zeichnen die Farbstifte vertauscht werden. Eigentlich könnten Sie natürlich auch die Nummern der beiden Farbstifte selbst vertauschen. Wir benötigen INVERSEVID aber für die Default-Farben, die mit -1 (0xFF) gekennzeichnet werden und bei denen ein Vertauschen dann nicht mehr möglich ist.

Modus Nummer drei arbeitet ähnlich wie eins, doch wird entsprechend des Untergrundes gezeichnet. Jeder schon gesetzte Punkt wird gelöscht und jeder gelöschte Punkt wird gesetzt.

Hier ist noch einmal eine zusammenfassende Tabelle:

Zeichenmodus	Beschreibung
JAM1	Nur Farbstift 1 wird benutzt.
JAM2	Beide Farbstifte werden verwendet.
INVERSEVID	Wie JAM2, mit vertauschten Stiften.
COMPLEMENT	Wie JAM1, gelöscht wird gesetzt und umgekehrt.

Tabelle 3.5: DrawModes

Mit der letzten Texteinstellung können Sie sich einen beliebigen Zeichensatz aussuchen, der Ihnen als TextAttr-Struktur vorliegt.

3.3.1.2 Die IntuiText-Struktur

Wie üblich, werden bei Intuition die Einstellungen in einer Struktur zusammengefaßt, unsere sieht dann wie folgt aus:

```
struct IntuiText
{
0x00  00  UBYTE FrontPen;
0x01  01  UBYTE BackPen;
0x02  02  UBYTE DrawMode;
0x03  03  SHORT LeftEdge;
0x05  05  SHORT TopEdge;
0x07  07  struct TextAttr *ITextFont;
0x0B  11  UBYTE *IText;
0x0F  15  struct IntuiText *NextText;
0x13  19
};
```

Noch nicht besprochen sind die letzten beiden Elemente der Struktur. *IText* stellt einen Pointer auf einen String dar. Sie wissen ja, daß man unter einem String eine Zeichenkette versteht, die mit einem Null-Byte abgeschlossen ist.

Mit dem letzten Pointer *NextText* bietet einem das System die Möglichkeit, mehrere solcher IntuiText-Strukturen zu einer lan-

gen Kette zusammenzufügen. Der Vorteil liegt dann im Aufruf der Ausgabe. Diese beschränkt sich dann nämlich nicht mehr auf eine Zeile oder eine Farbeinstellung, sondern ist für jeden Einzeltext vollständig neu einstellbar. Somit erreicht man höchste Flexibilität auch bei nur einem Funktionsaufruf.

3.3.1.3 Der Zeichensatz macht alles interessanter

Der einzige Parameter, den wir noch nicht genauer besprochen haben, ist der Zeichensatz. Dafür läßt sich jeder auf der Workbench-Diskette vorhandene nehmen. Allerdings bietet Intuition eine Sondereinstellung.

Sie sollten es noch aus dem Screen- und Window-Kapitel in Erinnerung haben, daß man für jeden Screen und für jedes Window einen sog. Default-Font einstellen kann. Wenn wir nun diesen Font verwenden möchten - meistens wird er dann ja durch Preferences eingestellt - so schreiben wir statt des Pointers einfach den Wert Null dort hinein.

Es ist natürlich auch möglich und eigentlich vorgesehen, daß man sich über die "diskfont.library" einen eigenen Zeichensatz besorgt, der dann hier verwendet wird. Verwenden Sie aber nur die beiden ROM-Fonts, damit erstens die Leserlichkeit und zweitens ein allgemeines Bild erhalten bleibt. Die Zeichensätze sind außerdem immer zu erreichen und haben Standard-Maße, auf die Sie sich im Programm verlassen können (z.B. bei den Menütexten).

Für die Auswahl der beiden ROM-Fonts, aber auch jedes anderen, benötigen wir zuerst einmal die schon erwähnte TextAttr-Struktur, mit deren Hilfe sich sowohl der Font als auch seine Eigenschaften bestimmen lassen. Diese Struktur sieht wie folgt aus:

```
struct TextAttr
{
0x00  00  STRPTR ta_Name;
0x04  04  UWORD ta_YSize;
0x06  06  UBYTE ta_Style;
0x07  07  UBYTE ta_Flags;
```

```
    0x08  08
    };
```

In *ta_Name* wird ein Pointer auf einen String festgehalten, der den Namen unseres Fonts enthält. Wenn wir mit den ROM-Fonts arbeiten, so setzen wir "topaz.font" ein. Achten Sie bitte darauf, daß der Name vollständig in kleinen Buchstaben geschrieben wird, damit ihn das Betriebssystem erkennt, denn das achtet sehr genau auf die "Rechtschreibung". Haben Sie mit der "diskfont.library" zusätzlich noch andere Fonts geöffnet, so können Sie natürlich auch auf diese zurückgreifen.

Mit dem nächsten Wert stellt man die Höhe des Fonts in Pixel ein. Sie bestimmen damit die Größe des Fonts und auch die Anzahl der Zeichen pro Zeile. Für unsere ROM-Fonts gibt es da zwei Möglichkeiten:

Font-Size	Zeichen	Höhe
TOPAZ_SIXTY	Sechzig	Zeichen bei 640 Punkten = 9 Pixel.
TOPAZ_EIGHTY	Achtzig	Zeichen bei 640 Punkten = 8 Pixel.

Tabelle 3.6: Standardzeichensätze

Als nächsten Parameter enthält die TextAttr-Struktur *ta_Style*, mit dem der sog. Softstyle eingestellt werden kann. Darunter versteht man die Möglichkeit, einen vorhandenen Normalzeichensatz über Rechenalgorithmen zu beeinflussen. Dadurch läßt sich zusätzlich Unterstreichen, Kursivschrift, Fettschrift und Breitschrift realisieren. Folgende Flags stehen Ihnen zur Verfügung und sind auch untereinander kombinierbar:

Flag-Name	Hex-Wert	Bedeutung
FS_NORMAL	0x00	Keine Beeinflussung
FSF_ITALIC	0x04	Kursivschrift
FSF_BOLD	0x02	Fettschrift
FSF_UNDERLINED	0x01	Unterstrichen
FSF_EXTENDED	0x08	Breitschrift

Tabelle 3.7: Arithmetische Textveränderung

Im letzten Wert unserer Struktur geben wir mit weiteren Flags noch zusätzliche Informationen zu unserem Zeichensatz. Diese

Flags tragen den Namen "Font Preference Flags" und lassen noch einige grundsätzliche Einstellungen zu. Uns interessieren davon nur zwei. In ihnen wird nämlich angegeben, von wo Intuition sich den Font besorgen soll. Zur Auswahl stehen da die Disk-Fonts und die ROM-Fonts. Da der "topaz.font" im ROM liegt, stellen wir natürlich auch das ein:

Flag-Name	Hex-Wert	Bedeutung
FPF_ROMFONT	0x01	Font ist im ROM zu suchen.
FPF_DISKFONT	0x02	Font wurde von Disk geladen.
FPF_REVPATH	0x04	
FPF_TALLDOT	0x08	
FPF_WIDEDOT	0x10	
FPF_PROPORTIONAL	0x20	
FPF_DESIGNED	0x40	
FPF_REMOVED	0x80	

Tabelle 3.8: FontTyp

Möchten wir nun ganz bestimmt auf den "topaz.font" zurückgreifen, mit einer Zeichenzahl von 80 pro Zeile und einem ganz normalen Aussehen, dann sähe unsere Struktur so aus:

```
struct TextAttr TestFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_EIGHTY,
   FS_NORMAL,
   FPF_ROMFONT
   };
```

3.3.1.4 PrintIText, die Ausgabe kann beginnen

Nachdem schon fast zu ausführlich auf die einzelnen Parameter eingegangen worden ist, ist es endlich Zeit, auch mit der IntuiText-Struktur zu arbeiten.

Sie erkennen, daß für die Ausgabe des IntuiTextes die Funktion PrintIText gebraucht wird. Wir benötigen weiterhin noch ein Argument, welches wir uns aus dem Window, in das die Ausgabe erfolgen soll, besorgen müssen. Unser Argument ist der

RastPort, ohne den es nicht möglich ist, irgendwelche Ausgaben zu tätigen.

Mit der folgenden Definition und Deklarierungszeile ist es aber kein Problem:

```
struct RastPort *MeinesWindowsRastPort;

MeinesWindowsRastPort = MeinWindow->RPort;
```

Jetzt fehlt uns nur noch eine entsprechende IntuitText-Struktur, und die Ausgabe kann losgehen.

Ich habe deshalb eine für Sie vorbereitet:

```
struct IntuiText ErsterText =

   {
   1, 0,                        /* FrontPen, BackPen   */
   JAM2,                        /* DrawMode            */
   15, 0,                       /* LeftEdge, TopEdge   */
   NULL,                        /* Font (Standard)     */
   (UBYTE *) "Systemprogrammierung auf dem Amiga!",
   NULL                         /* NextText            */
   };
```

Diese Definition fügen Sie bitte in unser Standardprogramm 1 ein. Ich hoffe, Sie erinnern sich noch daran und haben es auch sorgfältig abgetippt und abgespeichert.

Denken Sie daran, daß dieser Definitionstyp vor der main()-Funktion stehen sollte.

Im Hauptprogramm kann dann, nachdem über Open_All() alles gelaufen ist, mit der vorher erläuterten Anweisung der RastPort des Windows bestimmt werden.

Nun haben wir fast alle Parameter beisammen, mit denen wir PrintIText() aufrufen können. Es fehlen nur noch die Koordinatenangaben.

Jetzt werden Sie sich wundern, wofür wir denn diese benötigen, aber schauen Sie sich dazu einmal die folgende Grafik an:

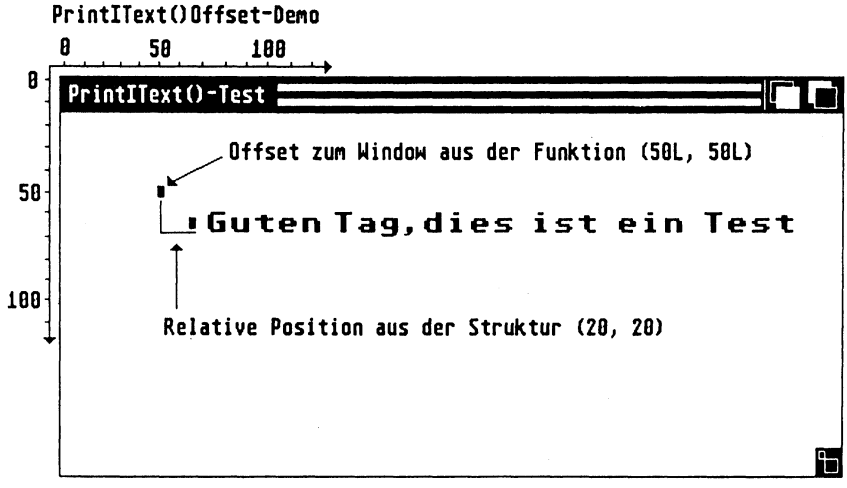


Abbildung 3.3: PrintIText()-Offsets

Beim Aufruf der PrintIText()-Funktion wird entsprechend zum RastPort der Punkt bestimmt, auf den sich alle weiteren Angaben in den Strukturen beziehen werden. Dieser Punkt wird praktisch zum "Nullpunkt". Die Positionsangabe in der IntuiText-Struktur ist dann relativ dazu zu sehen.

Fügen Sie zum Abschluß folgende Zeile in das Programm ein:

```
PrintIText(MeinesWindowsRastPort, ErsterText, 10L, 20L);
```

Hinweis: Denken Sie daran, daß die Offsets von PrintIText() als LONG-Werte angegeben werden, sonst könnte der Aztec-Compiler Schwierigkeiten machen.

Für alle Interessierten als "Bonbon" das Format der PrintIText-Funktion:

```
PrintIText(RastPort, IText, LeftOffset, TopOffset);
   -216      A0       A1       D0          D1
```

Um das Ganze noch abzurunden, biete ich Ihnen zuletzt noch eine verkettete (gelinkte) IntuiText-Struktur, die Sie natürlich

mit der gleichen Zeile wie oben aufrufen können. Das System kümmert sich von alleine darum, ob es mehrere Texte sein könnten, und gibt sie alle aus. Es ist darauf zu achten, daß immer der Text zuerst definiert wird, der als letzter in der Liste steht, denn sonst erkennt der Compiler die Pointer nicht als definiert!

```
struct IntuiText VierterText =
   {
   1, 0,                    /* FrontPen, BackPen   */
   JAM2,                    /* DrawMode            */
   15, 30,                  /* LeftEdge, TopEdge   */
   NULL,                    /* Font (Standard)     */
   (UBYTE *) "Dieser Text dient als Beispiel",
   NULL                     /* NextText            */
   };

struct IntuiText DritterText =
   {
   0, 1,                    /* FrontPen, BackPen   */
   JAM2,                    /* DrawMode            */
   15, 20,                  /* LeftEdge, TopEdge   */
   NULL,                    /* Font (Standard)     */
   (UBYTE *) "für die Verkettung",
   &Vierter Text            /* NextText            */
   };

struct IntuiText ZweiterText =
   {
   2, 0,                    /* FrontPen, BackPen   */
   JAM2,                    /* DrawMode            */
   15, 10,                  /* LeftEdge, TopEdge   */
   NULL,                    /* Font (Standard)     */
   (UBYTE *) "mehrere Text",
   &Dritter Text            /* NextText            */
   };

struct IntuiText ErsterText =
   {
   3, 0,                    /* FrontPen, BackPen   */
   JAM2,                    /* DrawMode            */
   15, 0,                   /* LeftEdge, TopEdge   */
   NULL,                    /* Font (Standard)     */
   (UBYTE *) "über IntuiText.",
   &Zweiter Text            /* NextText            */
   };
```

Die IntuiText-Struktur

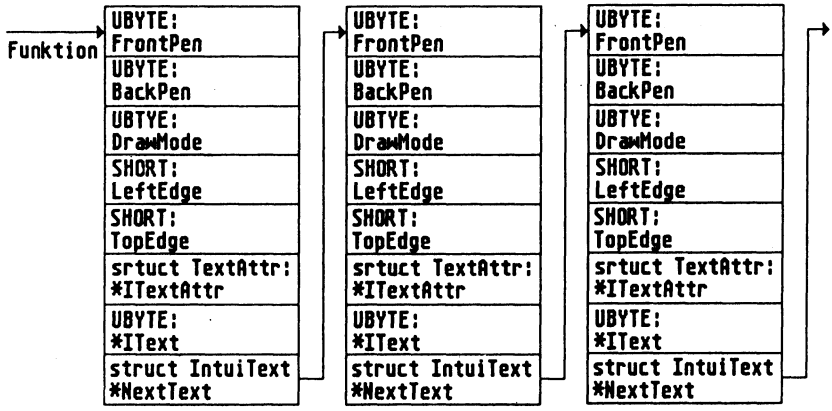


Abbildung 3.4: IntuiText-Linking

3.3.2 Linienausgabe und was man damit anfängt

Als zweites Element unterstützt Intuition die Linienausgabe. Die Wichtigkeit dieser Linien wird Ihnen vielleicht bewußt, wenn Sie sich z.B. die Workbench-Requester ansehen. Jedes der beiden Klickfelder, Gadgets, ist mit einem Doppelkasten umrahmt. Aber auch die Schieberegler in den Inhaltsverzeichnisfenstern sind mit einem Kasten umrahmt, der auf diese Linien zurückgeht. Es gibt noch einige weitere Beispiele, an denen wir uns aber nicht aufhalten wollen.

Zuerst soll es, wie auch bei der IntuiText-Struktur, unser Ziel sein, die Parameter der Border-Struktur zu untersuchen und zu definieren, damit wir im Anschluß daran einfache Linien zeichnen können. Da wir dafür etwas mehr Aufwand betreiben müssen, als bei den Texten nötig war, heißt es, sich genauestens mit der Materie zu beschäftigen.

3.3.2.1 Farbe, Position und und und

Genau wie auch bei der IntuiText-Struktur können wir auch bei der Border-Struktur die Startposition einstellen, an der die erste Linie gezeichnet werden soll. Diese wird, wie Sie ja schon bei PrintIText() kennengelernt haben, als Offset zu der beim Funktionsaufruf angegebenen Position betrachtet.

Auch Front- und BackPen lassen sich einstellen, obwohl momentan BackPen überhaupt nicht genutzt wird, denn eine Linie hat ja gar keinen näher definierten Hintergrund.

Aus dem gleichen Grund können unter DrawMode nicht mehr alle vier Modi benutzt werden. Nur JAM1 und INVERSEVID sind sinnvoll, weil sich die anderen beiden ja auf die zweite Farbe beziehen, die nicht benutzt wird.

Bevor wir nun zu den neuen Einstellungen kommen, betrachten wir zuerst die Border-Struktur:

3.3.2.2 Die Border-Struktur

```
struct Border
{
0x00  00  SHORT LeftEdge
0x02  02  SHORT TopEdge;
0x04  04  UBYTE FrontPen
0x05  05  UBYTE BackPen;
0x06  06  UBYTE DrawMode;
0x07  07  BYTE Count;
0x08  08  SHORT *XY;
0x0C  12  struct Border *NextBorder;
0x10  16
};
```

Im Gegensatz zur IntuiText-Struktur läßt sich bei den Linien kein Zeichensatz aussuchen. Dafür benötigen wir einen anderen Wert, und zwar eine Angabe, wie viele Eckpunkte unser Liniengebilde haben soll.

Dementsprechend muß der Zeiger nicht auf einen String, sondern auf eine Tabelle mit lauter Koordinatenpaaren zeigen.

Wie viele es sind, wird durch *Count* angegeben.

Wie auch bei den Texten können mehrere Border-Strukturen untereinander verknüpft werden, um z.B. die Farbe zu wechseln.

Es ergibt sich jetzt das neue Problem, daß wir uns noch genauer mit der Koordinatentabelle beschäftigen müssen:

3.3.2.3 Koordinatentabelle zum Linienzeichnen

Wie viele Koordinaten unsere Tabelle enthalten soll, wird in der Struktur über die Variable *Count* festgelegt. Wir müssen lediglich einen Zeiger auf das erste Element übergeben.

Eine Koordinatentabelle besteht aus lauter SHORT-Wertepaaren, jeweils einen für die X-Position und einen für die Y-Position. Die Angaben werden als Offsets zu LeftEdge und TopEdge gesehen, die Sie in der Border-Struktur initialisiert haben.

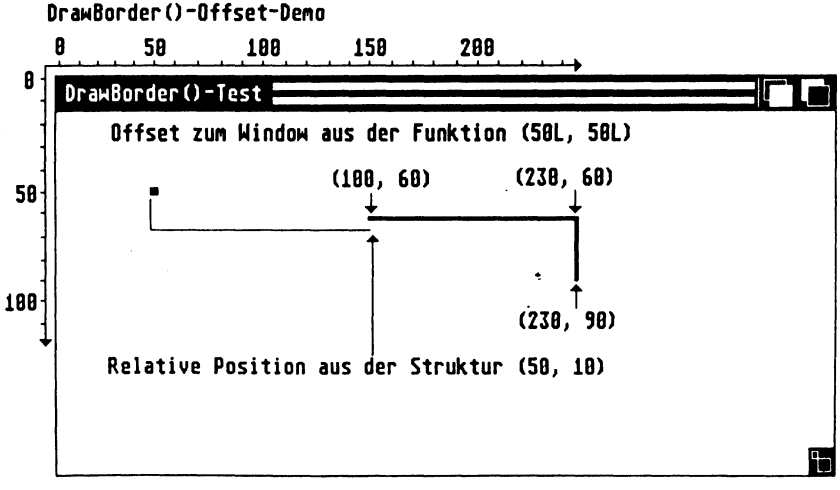


Abbildung 3.5: DrawBorder()-Offsets

Um die Sache noch komplizierter zu machen, kann ich noch nebenbei erwähnen, daß diese Koodinaten natürlich auch nur einen Offset zu denen des DrawBorder()-Kommandos darstellen.

Eine Koordinatentabelle könnte z.B. folgendes Aussehen haben:

```
SHORT TestWerte[] =
   {
    0,  0,
   50,  0,
   50, 12,
    0, 12,
    0,  0
   };
```

Damit haben wir einen rechteckigen Kasten beschrieben, der die Breite von 50 Pixeln und die Höhe von 12 Pixeln hat. Achten Sie darauf, daß wir für einen Kasten mit vier Ecken fünf Werte brauchen: den Startwert und vier Zielwerte für die Linien.

3.3.2.4 DrawBorder-Funktion mit Anwendungsbeispielen

Abschließend zur Bestandsaufnahme gehört es natürlich auch, endlich mit der Funktion, die uns Intuition zur Verfügung stellt, zu arbeiten. Dazu brauchen wir zuerst einmal eine vollständig mit Werten besetzte Border-Struktur und eine Koordinatentabelle:

```
struct Border TestBorder =
   {
   50, 20,
   2, 0,
   JAM1,
   5,
   TestWerte,
   NULL
   };
```

Nehmen wir als Koordinaten die schon oben abgedruckte Tabelle. Zusätzlich brauchen wir aber noch den RastPort (ich hoffe Sie, erinnern sich daran noch von der PrintIText()-Funktion) und die Offsets für die Funktion:

```
struct RastPort *MeinesWindowsRastPort;

MeinesWindowsRastPort = MeinWindow->RPort;

DrawBorder(MeinesWindowsRastPort, TestBorder, 10L, 10L);
```

Bauen Sie jetzt die Border-Struktur, die Koordinatentabelle, den RastPort und den Funktionsaufruf in das Standardprogramm 1 ein. Nach dem Starten sollte ein kleiner Kasten im Fenster erscheinen.

Als zusätzliches Info hier noch das Format der neuen Funktion:

```
DrawBorder(RastPort, Border, LeftOffset, RightOffset);
   -108       A0       A1        D0          D1
```

Zuletzt auch noch ein Beispiel für die Verkettung mehrerer Border-Strukturen. So wurden z.B. die mehrfarbigen Kästen der System-Requester realisiert.

```
SHORT WeißeWerte[] =
   {
    0, 0,
   50, 0,
   50,12,
    0,12,
    0, 0
   };

SHORT RoteWerte[] =
   {
    0, 0,
   54, 0,
   54,16,
    0,16,
    0, 0
   };

struct Border WeißerRand =
   {
   20, 20,
   1, 0,
   JAM1,
   5,
   WeißeWerte,
   NULL
   };

struct Border RoterRand =
   {
```

```
    18, 18,
    2, 0,
    JAM1,
    5,
    RoteWerte,
    WeißerRand
    );

DrawBorder(MeinesWindowsRastPort, RoterRand, 10L, 10L);
```

Die Border-Struktur

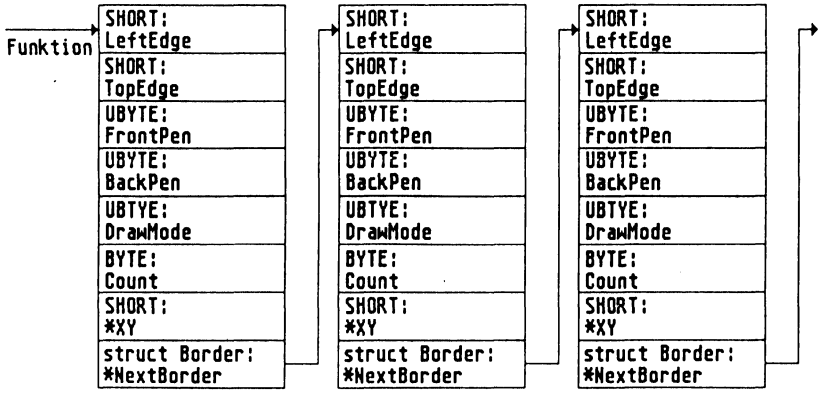


Abbildung 3.6: Border-Linking

3.3.3 Grafikausgabe, jetzt wird's interessant

Beschäftigen wir uns nun mit der wichtigsten Ausgabefunktion Intuitions! Ich meine damit die Grafiken, die Sie eigentlich überall wiederfinden. Sehen Sie sich die Workbench mit ihren Icons an! Jedes Icon ist eine kleine Grafik, die wir auch bald programmieren können. Auch jedes System-Gadget ist eine kleine Grafik. Es ist, wie man sieht, nicht weit hergeholt, wenn man sagt, daß Grafiken in der Programmierung eine wichtige Rolle spielen.

Der langen Vorrede ist jetzt ein Ende, wir beginnen wieder mit den allgemeinen Einstellungen.

3.3.3.1 Größe, Position, Farben und vieles mehr

Genau wie die Texte und Linien hat natürlich jede Grafik eine bestimmte Position im Window, die, wie Sie ja schon wissen, immer als Offset zu den Koordinaten des Funktionsaufrufes angesehen wird.

Zu der Positionsangabe kommen noch zwei Werte, die die Größe der Grafik bestimmen. Jede Grafik wird als eine rechteckige Fläche gesehen, von der wir die Breite und die Höhe angeben müssen.

Eine Neuigkeit finden wir hier aber auch. Es ist Ihnen sicherlich bekannt, daß jede Grafik des Amiga immer in mehrere BitPlanes zerlegt wird. Wenn Sie doch nicht so ganz sicher sind, möchte ich Sie auf ein weiteres DATA BECKER Buch verweisen. Im Supergrafik-Buch werden alle Grafikmöglichkeiten des Amiga ausführlich mit vielen Beispielprogrammen beschrieben - hier bleibt leider nicht genug Zeit. Wir müssen zu unserer Grafik nun angeben, aus wie vielen BitPlanes sie besteht, damit sie entsprechend verwaltet werden kann.

3.3.3.2 Die Image-Struktur

```
struct Image
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
0x08  08  SHORT Depth;
0x0A  10  USHORT *ImageData;
0x0E  14  UBYTE PlanePick
0x0F  15  UBYTE PlaneOnOff;
0x10  16  struct Image *NextImage;
0x14  20
};
```

Die ersten fünf Werte sind oben ja gerade erklärt worden. Sehen wir uns deshalb die restlichen vier an!

Der Pointer *ImageData* zeigt auf ein oder mehrere Datenfelder, die die Grafikdaten enthalten.

Mit dem letzten Pointer können wir - wie üblich - mehrere Image-Strukturen verbinden. So kann man ganz leicht komplexe Grafiken gestalten.

Die DrawImage()-Funktion von Intuition wird bei ihrem Aufruf die Grafikdaten in die BitPlanes unseres Windows oder unseres Screen schreiben. Mit dem Flag *PlanePick* teilen wir der Funktion mit, welche Planes unserer Grafik auch in die des Windows übertragen werden sollen. Jedes Bit steht für eine Plane:

PlanePick-Wert (binär)	Benutzte Planes
00000000	Keine
00000001	Plane 0
00000010	Plane 1
00000100	Plane 2
	usw.
00000011	Plane 0 und 1
00000101	Plane 0 und 2
00000111	Plane 0, 1 und 2
	usw.

Tabelle 3.9: PlanePick, PlaneOnOff

Mit der Tabelle sollte es für Sie ganz einfach sein, zu bestimmen, welche BitPlanes der Grafik in das Window übertragen werden sollen. Wofür wurde diese Methode gewählt? Über sie kann man als Programmierer bestimmte BitPlanes seiner Grafik ausschalten. Dies wird oft dazu benutzt, eine Grafik in verschiedenen Farben auszugeben. Versuchen Sie es ruhig einmal.

Das andere Flag *PlaneOnOff* wurde eingeführt, um die nicht benutzten BitPlanes ebenfalls zu verwenden. Über die gleichen Werte, wie sie in der obigen Tabelle stehen, können Sie ganz einfach die BitPlanes angeben, die mit 1ern gefüllt werden sollen. So kann eine Grafik leicht mit Farbe hinterlegt werden.

Sehen wir uns nun als letztes den Aufbau der Grafikdaten an:

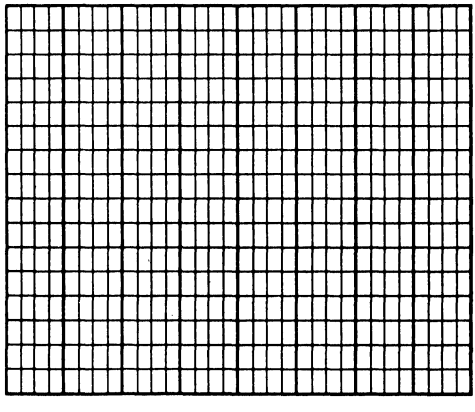
3.3.3.3 Die Grafikdaten in der Image-Struktur

Jede Grafik sollte man sich zur Entwicklung auf Millimeterpapier zeichnen. Jedes Kästchen gilt dann als ein Pixel. Diese Arbeit müssen Sie für alle BitPlanes machen. Danach erfolgt das Ausrechnen der Werte für unsere Daten.

Gehen Sie dafür wie folgt vor: Zuerst unterteilen Sie das Rechteck in Spalten à vier Pixel. Diese sehen Sie als eine Binär-Zahl an, wobei jeder gesetzte Punkt eine 1 bedeutet, und rechnen den Wert aus. Das machen Sie eine Zeile lang für jede Spalte und notieren sich von links nach rechts die hexadezimalen Ziffern. Danach gehen Sie in der nächsten Zeile genauso vor.

Als kleine Hilfe habe ich Ihnen dafür ein Gitter vorbereitet:

```
Für den eigenen Gebrauch !
--------------------------
```



```
1. BitPlane      2. BitPlane
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
0x    , 0x       0x    , 0x
```

Abbildung 3.7: Gitter für eigene Grafiken

Eine andere Möglichkeit gibt es, wenn Sie binäre Zahlen verwenden. Dann können Sie diese einfach aus der Grafik übernehmen, indem Sie jeden gesetzten Punkt durch eine 1 und jeden ungesetzten Punkt durch eine 0 darstellen. Es ist nur die Frage, ob Ihr Compiler binäre Zahlen verarbeitet.

Besteht Ihre Grafik aus mehreren BitPlanes, dann müssen zuerst die Daten für die erste BitPlane, anschließend die für die zweite BitPlane usw. in der Werteliste folgen.

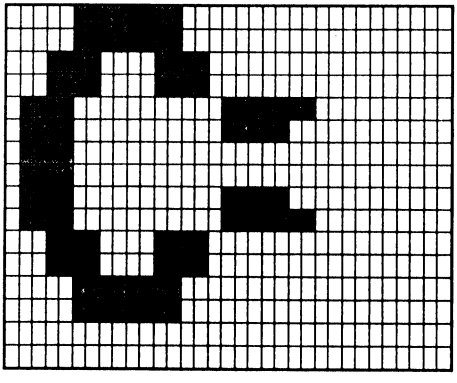
3.3.3.4 Die DrawImage-Funktion und ihre Anwendung

Nach so viel Theorie gibt es nun endlich mehr Beispiele! Stürzen wir uns also ins Geschehen und betrachten die erste Image-Struktur:

```
struct Image Beispiel =
   {
   20, 10,
   16, 16,
   2,
   &BeispielDaten[0],
   3, 0,
   NULL
   };
```

Dafür brauchen wir natürlich auch die dazugehörige Zeichnung, die ich schon vorbereitet habe:

Image-Data Beispiel



```
 1. BitPlane        2. BitPlane
0x07F8, 0x0000    0x0000, 0x0000
0x07F0, 0x0000    0x0000, 0x0000
0x1E1E, 0x0000    0x0000, 0x0000
0x1E1E, 0x0000    0x0000, 0x0000
0x7800, 0xFE00    0x0000, 0xFE00
0x7800, 0xF800    0x0000, 0xF800
0x7800, 0x0000    0x0000, 0x0000
0x7800, 0x0000    0x0000, 0x0000
0x7800, 0xF800    0x0000, 0xF800
0x7800, 0xFE00    0x0000, 0xFE00
0x1E1E, 0x0000    0x0000, 0x0000
0x1E1E, 0x0000    0x0000, 0x0000
0x07F8, 0x0000    0x0000, 0x0000
0x07F8, 0x0000    0x0000, 0x0000
```

Abbildung 3.8: ImageData

Das Datenfeld sieht dafür folgendermaßen aus (denken Sie beim Einsetzen in das Standardprogramm 1 daran, daß Sie es vor der Image-Struktur definieren):

```
USHORT BeispielDaten[] =
   {
   /* erste BitPlane */
   0x07F8, 0x0000,
   0x07F0, 0x0000,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x7800, 0xFE00,
   0x7800, 0xF800,
   0x7800, 0x0000,
   0x7800, 0x0000,
   0x7800, 0xF800,
   0x7800, 0xFE00,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x07F8, 0x0000,
   0x07F8, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   /* zweite BitPlane */
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0xFE00,
   0x0000, 0xF800,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0xF800,
   0x0000, 0xFE00,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000
   };
```

Als letztes müssen wir noch den RastPort des Windows ermitteln und die Ausgabefunktion DrawImage() aufrufen:

```
struct RastPort *MeinesWindowsRastPort;

MeinesWindowsRastPort = MeinWindow->RPort;

DrawImage(MeinesWindowsRastPort, Beispiel, 10L, 10L);
```

Als zusätzliche Information noch das allgemeine Format:

```
DrawImage(RastPort, Image, LeftOffset, TopOffset);
   -114      A0       A1        D0          D1
```

Bedenken Sie, daß unser Grafik-Chip nur Speicher ansprechen kann, der im sog. Chip-Memory liegt. Das ist der Speicherbereich bis 512 KByte. Es gibt nun zwei Möglichkeiten, dafür Sorge zu tragen, daß die Grafikdaten in diesen Speicherbereich kommen. Die Compiler bieten eine Option, mit der alle Daten in den Chip-Memory transferiert werden können. Das ist wohl die einfachste Methode. Es läßt sich aber auch im Programm durchführen. Dazu müssen Sie sich Speicher im Chip-Memory besorgen und die Grafikdaten in diesen kopieren. Erst jetzt können Sie die Grafik initialisieren und ausgeben.

3.3.4 Beispiele für jede Grafikanwendung

Abschließend zum Grafik-Kapitel im Intuition-Teil möchte ich Ihnen ein paar Anwendungen vorstellen, für die Sie sicherlich Interesse haben werden.

Es war nicht schwer zu erkennen, wie einfach die Bestimmung der Strukturenwerte ist. Viele Einstellungen machen es möglich, auf jeden Wunsch des Programmierers einzugehen. Vielleicht liegt gerade dort das Problem. Hat man z.B. sehr viele gleiche Texte, so kann es schon ärgerlich werden, wenn man immer wieder die IntuiText-Struktur mit den gleichen Werten ausfüllen muß. Dafür möchte ich in diesem Kapitel eine selbst programmierte Funktion vorstellen, die Abhilfe schafft.

Zusätzlich finden Sie hier einige kleine Grafiken, Texte und Linien, die wir für die Gadgets im nächsten Kapitel gut gebrauchen können. Wir erweitern unseren C-Quelltext-Editor wieder um ein kleines Stück.

Natürlich soll das CLI-Fenster nicht vergessen werden. Ich habe dafür ein Gadget vorbereitet, mit dem endlich das lästige Sizen auf bestimmte Werte entfällt.

Stürzen wir uns nun in das Abenteuer...

3.3.4.1 Auf einfache Weise viele Texte definieren

Es tritt öfter das Problem auf, daß man in seinem Programm mit vielen Texten arbeiten muß, die alle einzeln über eine IntuiText-Struktur definiert werden müssen. Da wir nicht immer eine neue Zeichenfarbe auswählen und uns meistens auch auf einen Zeichensatz beschränken, kann man als Programmierer gelegentlich ins Schwitzen kommen, wenn man 20 solcher Strukturen definieren muß.

Natürlich gibt es auch dafür eine Lösung, und die sieht folgendermaßen aus: Zuerst definieren wir eine sog. Default-Struktur, in der nur alle Parameter eingetragen sind, nicht aber der Zeiger auf den String. Hier ein konkretes Beispiel:

```
struct TextAttr DefaultFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_EIGHTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct IntuiText DefaultText =
   {
   1, 0,                    /* FrontPen, BackPen   */
   JAM2,                    /* DrawMode            */
   1, 1,                    /* LeftEdge, TopEdge   */
   &DefaultFont,            /* Font                */
   NULL,                    /* Textpointer         */
   NULL                     /* NextText            */
   };
```

Struktur 3.6: DefaultText

Gleichzeitig habe ich noch eine TextAttr-Struktur initialisiert, damit auch immer der richtige Zeichensatz benutzt wird.

Für die Texte, die später in diese Struktur eingesetzt werden sollen, definieren wir ein Pointerfeld des Typs char, so kommen wir an jeden String leicht heran:

```
char *Textfeld[] =
   {
   "Erster Testtext",
   "hier ist der zweite",
   "und noch einer",
   "hier haben wir schon den letzten und längsten!"
   };
```

Nun fehlt nur noch ein Feld von IntuiText-Strukturen, in das wir über ein Unterprogramm die Default-Struktur-Werte eintragen lassen und dann den Pointer ergänzen.

```
struct IntuiText AlleTexte[4];
```

Sehen wir uns schließlich das Unterprogramm an, welches die eben genannte Arbeit übernimmt:

```
Make_Text(Anzahl, Texte, Struktur)

int              Anzahl;
char            *Texte[];
struct IntuiText  Struktur[];

   {
   int i;

   for (i=0; i<Anzahl; i++)
      {
      Struktur[i]          = DefaultText;
      Struktur[i].IText    = (UBYTE *) Texte[i];
      Struktur[i].NextText = &Struktur[i+1];
      }
   Struktur[Anzahl-1].NextText = NULL;
   }
```

Funktion 3.3: Make_Text

Dokumentation

Nachdem auch für das Unterprogramm die übergebenen Variablen deklariert worden sind, beginnt es mit seiner Arbeit in einer Schleife. Hier wird der als Feld definierten Struktur zuerst immer die Default-Struktur zugewiesen. Damit sind die allgemeinen Werte initialisiert! Danach ergänzen wir den Pointer auf den ersten String und verketten die erste IntuitText-Struktur mit der zweiten. Die Schleife wird je nach Anzahl mit den nächsten Werten durchlaufen. Damit die letzte Struktur nicht auf eine

nicht vorhandene zeigt, muß der überflüssig gesetzte Zeiger wieder gelöscht werden.

Wenn Sie die Texte nicht miteinander verketten möchten, so können die zwei dafür vorgesehenen Zeilen natürlich entfallen. Sie dürfen ebenfalls jederzeit irgendwelche Ergänzungen machen. Zum Beispiel eine automatische Zeilenberechnung als Koordinateninitialisierung.

3.3.4.2 Border-Hilfen, die man immer braucht!

In diesem Abschnitt wollen wir schon etwas Vorarbeit für das nächste Kapitel leisten. Dort werden wir uns mit den Gadgets beschäftigen. Diese Gadgets benutzen nun auch alle drei Ausgabeelemente, die Sie hier kennengelernt haben. Wie es ja oft üblich ist, wird ein Kasten zur Begrenzung des Klickfeldes eines Gadgets benutzt. Deshalb wollen wir eine Border-Struktur entwickeln, mit der wir später weiterarbeiten können.

Außerdem kann man sich wie auch bei Texten einige Arbeit abnehmen lassen. Denken Sie z.B. nur an Textumrandungen, die immer unterschiedliche Länge haben. Warum lassen wir das nicht einfach von einer Funktion berechnen?

Nehmen wir als erstes Beispiel die einfach und zweifach umrandeten Texte. Sie werden z.B. bei BECKERtext und TEXTOMAT von DATA BECKER dazu verwendet, zu kennzeichnen, welches Gadget auch durch die Return-Taste ersetzt werden kann.

Hier ist der einfache Rand (er ist für 8 Buchstaben gedacht):

```
SHORT einfachWerte[] =
   {
    0, 0,
   66, 0,
   66,10,
    0,10,
    0, 0
   };

struct Border einfachRand =
   {
```

```
    0, 0,
    1, 0,
    JAM1,
    5,
    einfachWerte,
    NULL
    };
```

Struktur 3.7: Einfacher Rand

... und passend dazu der zweite:

```
SHORT doppeltWerte[] =
    {
     0, 0,
    70, 0,
    70,14,
     0,14,
     0, 0
    };

struct Border doppeltRand =
    {
    -2, -2,
    2, 0,
    JAM1,
    5,
    doppeltWerte,
    NULL
    };
```

Struktur 3.8: Doppelter Rand

... er kann leicht über den Pointer angefügt werden!

```
einfachRand.NextBorder = &doppeltRand;
```

Als nächstes biete ich eine Lösung für das Problem an, einen Rand für einen Text, dessen Länge das Programm erst beim Ablauf kennt, zu definieren.

Als allgemein müssen vor dem Hauptprogramm eine Border-Struktur und ein Wertefeld definiert werden. Das sollte etwa so aussehen:

```
SHORT *Werte[] =
   {
    0, 0,
  999, 0,
  999,10,
    0,10,
    0, 0
   };

struct Border Rand =
   {
   0, 0,
   1, 0,
   JAM1,
   5,
   &Werte[0],
   NULL
   };
```

Struktur 3.9: Allgemeine Border

Außerdem benötigen wir eine IntuiText-Struktur, in die nur noch der Pointer für den Text eingesetzt wird:

```
struct IntuiText Text =
   {
   1, 0,                  /* FrontPen, BackPen   */
   JAM2,                  /* DrawMode            */
   2, 2,                  /* LeftEdge, TopEdge   */
   NULL,                  /* Font (Standard)     */
   NULL,                  /* Textpointer         */
   NULL                   /* NextText            */
   };
```

Struktur 3.10: Allgemeiner IntuiText

Sowohl Text als auch Border werden mit den gleichen Offsets in den RastPort geschrieben:

```
DrawBorder(RastPort, Rand, 10L, 10L);
PrintIText(RastPort, Text, 10L, 10L);
```

Bevor dies gemacht werden kann, muß noch die Funktion aufgerufen werden, die die Border-Werte richtig berechnet. Sie ist ganz einfach zu verstehen:

```
Berechne(Werte, Text)

USHORT          *Werte[];
struct IntuiText  Text;

   {
   int Breite = 0;

   Breite = IntuiTextLenght(Text);

   Werte[3] = Breite+4;
   Werte[5] = Breite+4;
   }
```

Funktion 3.4: Textlänge berechnen

Das einzige Unbekannte ist die IntuiTextLength()-Funktion, die uns als Werte die Breite des Textes in Pixeln zurückgibt. Dabei berücksichtigt sie auch den Zeichensatz!

Haben wir den Wert, dann setzen wir ihn einfach in das Feld ein.

3.3.4.3 Symbole sagen mehr als Worte

Als Ergänzung zum Abschnitt über die Images möchte ich hier eine Grafik liefern, die später als Symbol für ein Gadget dienen wird.

Damit soll ein großer Bogen zum Kapitel gespannt werden, in dem ich das CLI-Editor-Fenster vorgestellt habe. Im nächsten Kapitel möchte ich als erste wirklich nützliche Anwendung das Gadget präsentieren, zu dem wir eben dieses Symbol benötigen.

Die Aufgabe, die beim Anklicken bewältigt werden soll, ist denkbar einfach. Für den Anwender ist es immer lästig, wenn er das Fenster für viel Text auf maximale Größe ziehen muß. Genauso umständlich ist es, wenn man es ganz klein machen möchte.

Da wir dies auch über Window-Funktionen erledigen können, lösen wir die Aktion einfach über das angesprochene Gadget aus. Dafür brauchen wir ein Symbol.

Das Symbol soll ein kleines Fenster und ein großes Fenster darstellen. Zwischen diesen beiden Fenstern besteht der Größenwandel, der durch die Linie gekennzeichnet wird.

```
Image-Data CLI-Wide

                                  1. BitPlane
                                  0xFF00, 0x0000
                                  0x8100, 0x0000
                                  0x91FF, 0x0000
                                  0x8C01, 0x0000
                                  0xFF01, 0x0000
                                  0x10C1, 0x0000
                                  0x1021, 0x0000
                                  0x1001, 0x0000
                                  0x1001, 0x0000
                                  0x1FFF, 0x0000
                                  0x0000, 0x0000
                                  0x0000, 0x0000
                                  0x0000, 0x0000
                                  0x0000, 0x0000
                                  0x0000, 0x0000
                                  0x0000, 0x0000
```

Abbildung 3.9: Resize-Gadget

Es dürfte für Sie wohl kein Problem sein, aus dem Schaubild und den Daten eine Image-Struktur mit Grafikdaten aufzubauen.

3.4 Gadgets, einfacher Informationsaustausch

Anwenderprogramme benötigen allgemeine Informationen wie Text oder Daten, um arbeiten zu können. Außerdem brauchen sie Anweisungen, mit deren Hilfe sie in ihrer Ausführung gelenkt werden. In diesem Kapitel klären wir die Frage: "Wie übermittle ich dem Programm einfach Informationen und auf welche Weise kann ich das tun?"

Als versierter Amiga-Besitzer wissen Sie natürlich, daß die leichteste Informationsübermittlung über die sog. Gadgets abläuft. Das sind rechteckige Klickfelder in Windows oder auf Screens, die beim Anklicken mit der linken Maustaste reagieren.

Der Amiga kennt aber nicht nur einfache Gadgets, wie das, bei dem nur ein Klickimpuls weitergeleitet wird. Es gibt auch Gadgets, die man ein- oder ausschalten kann, die man für Texteingabe benutzen kann, und solche, die man für Positionsangaben verwendet. Alle diese vollkommen verschiedenen Typen sind auch noch in Untergruppen unterteilt. Sie haben also die vielfältigsten Möglichkeiten, Informationen vom Benutzer in das Programm zu holen.

Weil man aber diese Klickfelder nicht erkennen würde, wenn sie einfach nur "da wären", unterstützt es Intuition, die rechteckigen Felder mit Rändern, Grafiken und Texten "auszuschmücken". Deswegen sind die Ausgabetechniken, die ich im vorhergehenden Kapitel beschrieben habe, so besonders wichtig.

Sie haben jetzt einen Vorgeschmack von dem bekommen, was alles möglich ist. Auch das klare Konzept von Intuition läßt immer mehr erkennen, wie einfach und doch komplex es aufgebaut ist: Einfache Screens dienen als elementare Einstellung des Grafikmodus. Windows stellen eine einfache Ein- und Ausgabeeinheit dar. Über die Ausgabefunktionen lassen sich mit wenigen Mitteln durchdachte Strukturen ausgeben. Jetzt werden alle diese Mittel kombiniert, damit der Programmierer für seine Anwendung wie aus einem Baukastensystem alles Nötige "basteln" kann.

3.4.1 Verschiedene Gadgets, verschiedene Anwendungen

Intuition bietet ihnen zwei große Gadget-Kategorien zur Arbeit an. Da sind zuerst die System-Gadgets. Sie werden von Intuition angeboten und sind in ihrem Aussehen und in ihrer Wirkung festgelegt. Dazu lesen Sie mehr im Window-Kapitel.

Die zweite Gadget-Gruppe ist einzig für den Programmierer gedacht und unterstützt das eigene Anfertigen. Sie können aus drei

verschiedenen Gadget-Typen wählen, die vollkommen unterschiedliche Eigenschaften haben.

Boolean-Gadgets

Für ganz einfache Informationen, die nur aus einem Wahrheitswert bestehen (Ja oder Nein), sind Boolean-Gadgets vorgesehen. Sie bieten zwei Modi:

Der erste Modus heißt "hit select" und bedeutet, daß Intuition eine Information sendet, wenn das Gadget über die linke Maustaste, sie wird im folgenden Action-Taste genannt, angeklickt wird. Dabei verändert sich für die Zeit, in der sich der Maus-Cursor im Klickfeld befindet, das Aussehen des Gadgets. Bewegt man die Maus wieder weg oder läßt man den Button los, ist das Aussehen wieder wie vorher. Über diesen Modus können Sie ganz einfach bestimmte Aktionen auslösen.

Mit dem zweiten Modus, "toggle select", schalten Sie mit dem ersten Anklicken das Gadget an. Es verändert dann sein Aussehen so, als wäre es ständig angeklickt. Mit einem zweiten Anklicken wird das Gadget wieder in den Urzustand gebracht. Diese Methode eignet sich besonders gut für das Ein- und Ausschalten von irgendwelchen Funktionen.

Proportional-Gadgets

Möchten Sie Bereiche oder Positionen darstellen, dann ist dieser Gadget-Typ geeignet. Mit ihm können Sie eindimensional oder zweidimensional bestimmte Proportionen ausgeben und vom Benutzer Ihres Programms verändern lassen.

Dieses Gadget besteht aus einem rechteckigen Kasten, in dem sich ein beliebiges Objekt befindet. Ich halte den Begriff hier absichtlich allgemein, weil Sie dafür jede beliebige Grafik definieren können. Das Objekt stellt z.B. bei Preferences die Kante des Workbench-Screens dar, die Sie über ein Proportional-Gadget einstellen können. In diesem Fall ist es sogar ein zweidimensionales. Ein Beispiel für eindimensionale Proportional-Gadgets

sind die Schieber der Inhaltsverzeichnis-Windows. Der Balken, unser "Objekt", stellt die Größe des Ausschnitts dar und kann im gesamten Bereich hin- und hergeschoben werden.

Für jedes Proportional-Gadget können Sie die Größe des Schiebekastens (container), das verwendete Objekt (knob) und die Größe des Objektes einstellen.

String-Gadget

Um einfache Texteingabe realisieren zu können, ist dieser Typ implementiert worden. Sie haben wie beim Proportional-Gadget einen "container", in dem der Text einzeilig eingegeben werden kann. Dabei spielt die Länge überhaupt keine Rolle: Intuition scrollt den Text, wenn nötig.

Zusätzlich werden noch einige Funktionen wie UNDO unterstützt, so daß die Eingabe komfortabel getätigt werden kann.

Jetzt kennen Sie alle Gadget-Typen mit ihren grundsätzlichen Eigenschaften. Sie sollten sich die Einzelheiten genau ansehen, damit Sie, nach Auswahl eines Typs, diesen auch wirklich gut programmieren können.

3.4.1.1 Das Boolean-Gadget

Einer der am meisten verwendeten Gadget-Typen ist das Boolean-Gadget. Wir finden es schon bei den System-Gadgets dreimal wieder: Im Close-Gadget und in den beiden Depth-Arrangement-Gadgets.

Boolean-Gadgets stellen den Grundtyp aller Gadgets dar, deshalb behandeln wir diesen auch besonders ausführlich.

Jedes Gadget kann in seiner Position, Höhe und Breite festgelegt werden - eine selbstverständliche Einstellung. Sehen Sie sich doch die Struktur an, bevor wir auf die anderen Werte eingehen:

```
struct Gadget
{
0x00  00  struct Gadget *NextGadget;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  USHORT Flags;
0x0E  14  USHORT Activation;
0x10  16  USHORT GadgetType;
0x12  18  APTR GadgetRender;
0x16  22  APTR SelectRender;
0x1A  26  struct IntuiText *GadgetText;
0x1E  30  LONG MutualExclude;
0x22  34  APTR SpecialInfo;
0x26  38  USHORT GadgetID;
0x28  40  APTR UserData;
0x2C  44
};
```

Genau wie bei allen anderen bisher behandelten Intuition-Strukturen, können auch Gadgets gelinkt werden. Dafür steht der erste Pointer. Nach den Werten für linke und rechte Ecke, Höhe und Breite finden wir ein Flag, das mehrere Aufgaben erfüllt.

In *Flags* finden wir zuerst einmal die Einstellungen zur Grafik des Gadgets. Sie können angeben, auf welche Weise das Gadget-Aussehen verändert werden soll, wenn es angeklickt wird. Dafür stehen Ihnen vier GADGHIGHBITS zur Verfügung:

GADGHCOMP

Alle Punkte im Klickbereich werden *komplementiert* dargestellt.

GADGHBOX

Um den Klickbereich wird ein Kasten gezeichnet.

GADHIMAGE

Es wird ein anderes Image oder eine andere Border dargestellt.

GADGHNONE

Keine Veränderung tritt ein. Weiterhin beinhaltet das Flag Positionsangaben zum Gadget:

GRELBOTTOM

Wenn Sie dieses Flag setzen, beschreibt die TopEdge-Variable der Struktur einen Offset relativ zum Fuß des Fensters. Ist das Flag nicht gesetzt, dann ist *TopEdge* eine relative Angabe zum oberen Rand des Fensters.

GRELRIGHT

Wenn Sie dieses Flag setzen, beschreibt die LeftEdge-Variable der Struktur einen Offset relativ zum rechten Rand des Fensters. Ist das Flag gelöscht, dann ist *LeftEdge* relativ zum linken Rand zu sehen.

Auch die Größe des Gadgets können Sie in Abhängigkeit zum Fenster setzen:

GRELWIDTH

Wenn Sie dieses Flag setzen, wird der Wert der Width-Variable von der Fensterbreite abgezogen, und das Ergebnis stellt dann die wirkliche Breite des Gadgets dar. Setzen Sie das Flag nicht, dann steht in *Width* ein absoluter Wert!

GRELHEIGHT

Wenn Sie dieses Flag setzen, wird der Wert der *Height*-Variablen von der Fensterhöhe abgezogen, und das Ergebnis stellt die wirkliche Höhe des Gadgets dar. Löschen Sie die Flags, wenn der Wert absolut gesehen werden soll.

Für den Status des Gadgets sind drei Flags vorgesehen:

GADGIMAGE

Wenn Sie dieses Flag setzen, wird anstelle einer Border ein Image in der Variablen *GadgetRender* vermutet. Die Bedeutung erstreckt sich auch auf die Variable *SelectRender*!

GADGDISABLED

Dieses Flag ist gesetzt (es kann nur zu Anfang gesetzt und später nur abgefragt werden), wenn das Gadget nicht anwählbar ist.

SELECTED

Dieses Flag ist gesetzt - es kann nur zu Anfang gesetzt und später nur abgefragt werden -, wenn das Gadget selected wurde.

Zum Schluß noch einmal alle Flags mit ihren Hex-Werten:

Flags	Hex-Wert
GADGHIGHBITS	0x0003L
GADGHCOMP	0x0000L
GADGHIMAGE	0x0002L
GADGHBOX	0x0001L
GADGHNONE	0x0003L
GRELBOTTOM	0x0008L
GRELRIGHT	0x0010L
GRELWIDTH	0x0020L
GRELHEIGHT	0x0040L
GADGIMAGE	0x0004L
SELECTED	0x0080L
GADGDISABLED	0x0100L

Tabelle 3.10: Gadget-Flags

Kommen wir jetzt zum nächsten Wert unserer Gadget-Struktur. Es ist wiederum ein Flag. Es enthält diesmal Informationen darüber, wie das Gadget aktiviert werden soll und ob es sich im Window-Rand befinden soll:

Betrachten wir zuerst die Flags, die sich mit dem Aktivierungsstatus beschäftigen:

TOGGLESELECT

Wie oben schon angesprochen wurde, gibt es einen besonderen Typ des Boolean-Gadgets. Dieser wird durch dieses Flag bestimmt. Danach wird das Gadget wie ein Schalter behandelt: Beim ersten Anklicken wird es eingeschaltet und bleibt solange in diesem Status, bis es wieder angeklickt wird.

GADGIMMEDIATE

Bei einfachen Boolean-Gadgets kann der Programmierer noch zwischen zwei Select-Modi wählen: Einmal angewählt erhält das Programm sofort eine Nachricht. Das geschieht, wenn dieses Flag gesetzt ist.

RELVERIFY

Wurde dieses Flag gesetzt, wartet Intuition mit der Nachricht, bis der Benutzer auch den Action-Button über dem Klickbereich losläßt. Man gesteht dem Anwender damit noch eine Schreckseskunde zu.

Für die Position des Gadgets gibt es noch mehr Variationen.

Diese beziehen sich aber nicht auf die Lage, sondern mehr auf die Verwaltung. Sie wissen ja, daß über den Window-Typ GIMMEZEROZERO der Window-Rand extra verwaltet wird, damit dessen Grafik nicht beschädigt werden kann.

Unsere eigenen Gadgets können wir auch darin unterbringen! Setzen Sie dafür eins der folgenden Flags.

Beachten Sie aber, daß dadurch die Breite des Randes beeinflußt wird:

RIGTHBORDER

Das Gadget wird im rechten Window-Rand untergebracht.

LEFTBORDER

Das Gadget wird im linken Window-Rand untergebracht.

TOPBORDER

Das Gadget wird im oberen Window-Rand untergebracht.

BOTTOMBORDER

Das Gadget wird im unteren Window-Rand untergebracht.

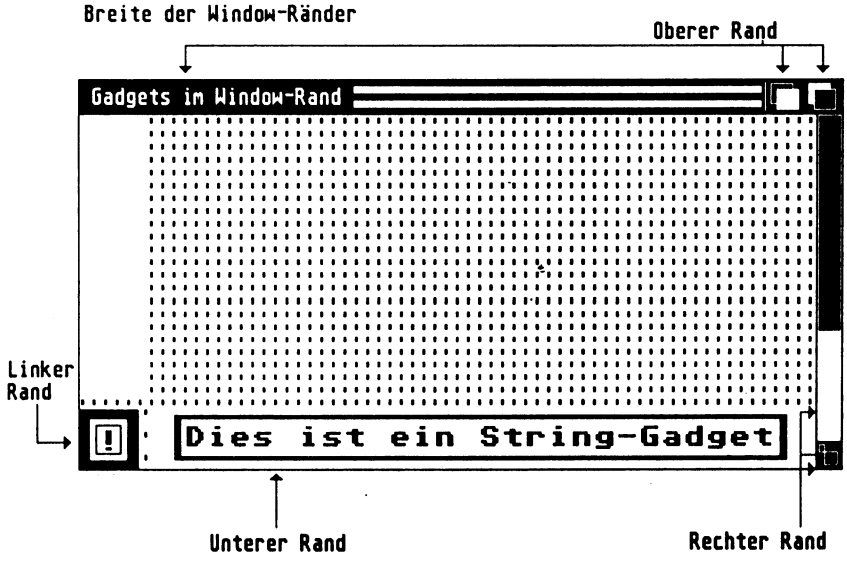


Abbildung 3.10: Gadgets im Window-Rand

Die folgende Tabelle faßt alle Activation-Flags zusammen:

Activation-Flag	Hex-Wert	Bemerkung
TOGGLESELECT	0x0100L	Schalter-Gadget
RELVERIFY	0x0001L	Vergewisserung
GADGIMMEDIATE	0x0002L	Sofort
RIGHTBORDER	0x0010L	Position in GZZ-Border.
LEFTBORDER	0x0020L	
TOPBORDER	0x0040L	
BOTTOMBORDER	0x0080L	
STRINGCENTER	0x0200L	Später für String-Gadgets.
STRINGRIGHT	0x0400L	
LONGINT	0x0800L	Für Integer-Gadgets.
ALTKEYMAP	0x1000L	Eigene Tastaturtabelle.

Activation-Flag	Hex-Wert	Bemerkung
BOOLEXTEND	0x2000L	
ENDGADGET	0x0004L	Für Requester.
FOLLOWMOUSE	0x0008L	Für IDCMP.

Tabelle 3.11: Activation-Flags

Kehren wir wieder zurück zu der Gadget-Struktur, die es immer noch zu untersuchen gilt! Wir kommen jetzt zum letzten Flag-Wert, der etwas über den Typ des Gadgets aussagt. *GadgetType* enthält aber noch weitere Informationen:

BOOLGADGET
Ein Boolean-Gadget, das wir auch in diesem Kapitel programmieren werden.

PROPGADGET
Ein Proportional-Gadget, zu dem Sie später noch mehr erfahren werden.

STRGADGET
Ein String-Gadget, mit dem Sie einfache Texteingabe realisieren können. Für ein Integer-Gadget setzen Sie bitte zusätzlich noch das LONGINT-Flag in der Flags-Variablen der Struktur.

Mit diesen drei Flags bestimmen Sie den Gadget-Typ. Über zwei weitere Flags, die vom System gesetzt werden, erfährt man etwas darüber, vom wen das Gadget stammt und wo es liegt.

SCRGADGET sagt, daß das Gadget zu einem Screen gehört, und *SYSGADGET* identifiziert die System-Gadgets in einer Liste.

Soll das Gadget in einem GZZ-Window in dessen Rand untergebracht werden, so ist zusätzlich zum Bestimmungs-Flag des Randes auch noch GZZGADGET zu setzen!

GadgetType	Hex-Wert	Bedeutung
BOOLGADGET	0x0001L	
GADGET0002	0x0002L	Unbenutzt
PROPGADGET	0x0003L	
STRGADGET	0x0004L	
SYSGADGET	0x8000L	System-Gadget
SCRGADGET	0x4000L	Screen-Gadget
GZZGADGET	0x2000L	GIMMEZEROZERO-Window-Gadget
REQGADGET	0x1000L	Requester-Gadget

Tabelle 3.12 Gadget-Typen

Endlich kommen wir, nach so vielen Flags und Einstellungen, zu den sichtbaren Merkmalen der Gadgets. Wir beschäftigen uns nun mit der Grafik, die wir zur Kenntlichmachung mit dem Gadget verknüpfen werden. Über die Höhe, Breite und Position haben wir nur die Lage und Ausdehnung des Klickbereiches definiert. Nun ist es an der Zeit, diesen auch für den Benutzer unseres Programms "sichtbar" zu machen. Dafür verwenden wir eine Border- oder eine Image-Struktur. Ein Zeiger auf eine der beiden Strukturen-Typen wird in der Variablen *GadgetRender* untergebracht. Diese Grafik wird dann beim Darstellen des Gadgets ausgegeben.

Haben wir zusätzlich noch festgelegt, daß beim Anklicken ein anderes Aussehen erwünscht ist, dann muß in der Variablen *SelectRender* ein weiterer Zeiger auf eine Border- oder Image-Struktur stehen.

Allerdings können wir nicht einfach einmal eine Border-Struktur und dann wieder eine Image-Struktur verwenden. Wie Sie oben sehen konnten, gibt es für jeden Modus natürlich ein Flag. Das heißt, daß für eine Image-Struktur das Flag GADGIMAGE gesetzt werden muß und für eine Border-Struktur kein Flag.

Für *SelectRender* legen wir uns mit GADGIMAGE gleich mit fest, jedoch ist es nicht gesagt, daß dort auch ein Pointer stehen muß. Nur wenn auch das Flag GADGHIMAGE gesetzt ist, sucht Intuition auch dort nach einem Struktur-Pointer.

Für weitere Gestaltungen des Gadgets bietet Intuition einen Zeiger auf eine IntuiText-Struktur an, über den wir gleichzeitig noch einen Text unterbringen können. Dieser Text hat aber nichts mit dem Anwählen des Gadgets zu tun!

Die letzten vier Werte der Struktur sind für komplexere Anwendungen gedacht. Damit wir aber endlich zu praktischen Anwendungen kommen können, möchte ich sie zuerst außer acht lassen. Später werde ich auf sie zurückkommen.

Das erste Gadget

Wenden wir uns nun der Programmierung zu. Ich habe die folgende Gadget-Struktur für das erste Beispiel vorbereitet:

```
struct Gadget BoolGadget =
   {
   NULL,                  /* NextGadget          */
   10, 40,                /* LeftEdge, TopEdge   */
   60, 20,                /* Width, Height       */
   GADGHBOX,              /* Flags               */
   RELVERIFY,             /* Activation          */
   BOOLGADGET,            /* Gadget Type         */
   (APTR)&GadgetBorder,   /* GadgetRender        */
   NULL,                  /* Select Render       */
   &GadgetText,           /* GadgetText          */
   NULL,                  /* MutualExclude       */
   NULL,                  /* SpecialInfo         */
   1,                     /* GadgetID            */
   NULL,                  /* UserData            */
   };
```

Strukturbeschreibung

Es handelt sich hier um ein Gadget des Typs Boolean, welches einen rechteckigen Klickbereich in der oberen linken Ecke des Windows haben wird. Das Gadget wird durch Umrandung des Bereiches selektiert dargestellt und ist erst nach einer Überprüfung aktiviert. Das Gadget besitzt zur Kennzeichnung des Klickbereiches eine Border und einen Text, um die Bedeutung für den Benutzer deutlich zu machen.

Neu ist der Wert GadgetID, unter dem wir eine Zahl von 0 bis 65535 eintragen können. Diese Methode dient der Unterschei-

dung mehrerer Gadgets. Zur Gadget-Struktur fehlen noch die Ergänzungen, die Border-Struktur, die Koordinatentabelle und der Text. Hier sind sie:

```
struct IntuiText GadgetText =
   {
   2, 0, JAM2, 4, 7, NULL, (UBYTE *)"Gadget" ,NULL,
   };

SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };
```

Fügen Sie alle neuen Programmteile in das Standardprogramm 1 aus dem Kapitel "Windows" ein. Denken Sie daran, daß zuerst die IntuiText- und die Border-Struktur definiert werden müssen, bevor Sie die Gadget-Struktur einrichten können. Als letztes ergänzen wir in der Window-Struktur einen Pointer, der auf das erste User-Gadget weist. Die ganze Window-Struktur sollte dann etwa so aussehen:

```
struct NewWindow FirstNewWindow =
   {
   160, 50,                  /* LeftEdge, TopEdge   */
   320, 200,                 /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   CLOSEWINDOW ¦             /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH ¦             /* Flags               */
   WINDOWSIZING ¦
   WINDOWDRAG ¦ WINDOWCLOSE ¦
   SMART_REFRESH,
   &BoolGadget,              /* First Gadget        */
   NULL,                     /* CheckMark           */
   (UBYTE *)"Gadget-Programmierung im Test",
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   100, 50,                  /* Min Width, Height   */
   640, 256,                 /* Max Width, Height   */

   WBENCHSCREEN,             /* Type                */
   };
```

Richten Sie Ihr Augenmerk auch auf das neue IDCMP-Flag, das uns über die Datenleitung immer Nachrichten senden wird, wenn ein Gadget angewählt wurde!

Aus diesem Grund muß auch die Abfrage der IDCMP-Flags für diesen neuen Fall ergänzt werden. Nicht nur das Close-Gadget wird bei einer Nachricht untersucht, sondern auch die Möglichkeit, daß ein selbstdefiniertes Gadget angewählt wurde:

```
FOREVER
   {
   if ((message = (struct IntuiMessage *)
        GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
   MessageClass = message->Class;
   code = message->Code;
   ReplyMsg(message);
   switch (MessageClass)
      {
      case GADGETUP    : nr += 1;
                         printf("Gadget zum %d. Mal
                                        aktiviert!\n", nr);
                         break;

      case CLOSEWINDOW : Close_All();
                         exit(TRUE);
                         break;

      }
   }
```

Programmteil 3.2: Abfrageschleife Boolean-Gadget

Die hier vorliegende Abfrageschleife bietet, im Gegensatz zu der bisher bekannten, noch einen weiteren großen Vorteil:

Durch das Wait()-Kommando wird der Task in den Wait-Status gesetzt, bis ein Signal aufgetreten ist. Damit verbraucht die Schleife keine Prozessorzeit, wenn sie nichts zu tun hat.

Die Abfrage des neu definierten Gadgets ist ziemlich einfach:

In der Switch()-Abfrage wird geprüft, ob ein selbstdefiniertes Gadget angewählt wurde. Da wir nur eines benutzen entfällt jede weitere Aussortierung, und wir können einen Zähler ausgeben, der die Anzahl des Anklickens notiert. Vergessen Sie nicht, diese Variable vorher zu definieren!

Programmbeschreibung
Nach dem Starten des Programms erscheint ein Window auf dem Workbench-Screen, in dem sich ein Boolean-Gadget befindet. Das Programm bekommt nun jedesmal eine Nachricht, wenn Sie dieses Gadget mit der Action-Taste anklicken. Weil wir RELVERIFY gesetzt haben, muß der Button auch im Klickbereich wieder losgelassen werden. Im DOS-Window wird dann ein Text mit der Anzahl des Anklickens ausgegeben. Das Programm können Sie mit dem Close-Gadget beenden.

Abschließend läßt sich zu den Boolean-Gadgets sagen, daß sie immer gut für einfache Prozesse zu gebrauchen sind. Hiermit kann der Benutzer leicht etwas auslösen, bestätigen oder abbrechen.

3.4.1.2 Das Proportional-Gadget

Alle weiteren Gadget-Typen setzen sich aus den Boolean-Gadgets und einer kleinen Erweiterung zusammen, so auch das Proportional-Gadget. Wie Sie es von der Workbench her kennen, wird es dazu benutzt, Schiebebalken oder Objekte in einem Kasten einzurichten. Meistens hilft man sich damit bei Darstellungen, die größer sind, als es die Ausgabe erlaubt. Dann wird ein Schiebebalken benutzt, um die Größe des Ausschnitts wiederzugeben und es dem Benutzer zu ermöglichen, einen anderen Ausschnitt zu wählen. Preferences benutzt ein zweidimensionales Proportional-Gadget, damit Sie die Lage des Ausgabebereiches festlegen können. Auch dafür eignet sich dieses Gadget.

Nehmen wir also für unser erstes Proportional-Gadget eine einfache Gadget-Grundstruktur, die einen länglichen senkrechten Klickbereich hat:

```
struct Gadget PropGadget =
   {
   NULL,                    /* NextGadget          */
   100, 40,                 /* LeftEdge, TopEdge   */
   20, 80,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   PROPGADGET,              /* Gadget Type         */
   NULL,                    /* GadgetRender        */
   NULL,                    /* Select Render       */
   NULL,                    /* GadgetText          */
   NULL,                    /* MutualExclude       */
   &BeispielProp,           /* SpecialInfo         */
   2,                       /* GadgetID            */
   NULL                     /* UserData            */
   };
```

Wie Sie unschwer erkennen, ist diese Struktur an Einstellungen wesentlich unterbelegt! Wir brauchen aber auch nicht so viele: Ein Text ist bei diesem Beispiel nicht nötig, und ein Rand um den "container" wird bei Proportional-Gadgets von selbst gezeichnet, solange wir es nicht unterbinden.

Allerdings haben wir nun den Wert *SpecialInfo* belegt! Dieser Zeiger wird nämlich immer benutzt, wenn ein Boolean-Gadget nicht ausreicht. Dann zeigt er auf eine ergänzende Struktur. In unserem Fall benötigen wir eine PropInfo-Struktur, die folgendes Aussehen hat:

```
struct PropInfo
{
0x00  00  USHORT Flags;
0x02  02  USHORT HorizPot;
0x04  04  USHORT VertPot;
0x06  06  USHORT HorizBody;
0x08  08  USHORT VertBody;
0x0A  10  USHORT CWidth;
0x0C  12  USHORT CHeight;
0x0E  14  USHORT HPotRes;
0x10  16  USHORT VPotRes;
0x12  18  USHORT LeftBorder;
0x14  20  USHORT TopBorder;
0x16  22
};
```

Sie sehen, für ein Proportional-Gadget brauchen wir noch einige Daten mehr. Diese enthalten hauptsächlich die Ausmaße des

"containers" und die des Schiebesymbols. Aber raten wir nicht lange, hier sind die Beschreibungen zu jedem Parameter:

Flags

Wieder ein Flag-Wert! Für das Proportional-Gadget brauchen wir noch ein paar Werte. Sie können wählen zwischen den folgenden Einstellungen:

AUTOKNOB

Mit diesem Flag bestimmen Sie, daß Sie selbst kein Symbol für den Schieber definiert haben und Intuition ein Rechteck verwenden soll. Dieses Rechteck wird von Intuition erstellt und vollständig an jede Situation angepaßt. Die Größe des "knobs" können Sie über die nächsten beiden Werte in der Struktur bestimmen.

FREEHORIZ

Wenn Sie dieses Flag setzen, kann der Schieber horizontal bewegt werden.

FREEVERT

Wenn Sie dieses Flag setzen, kann der Schieber vertikal bewegt werden. Der Schieber kann in beide Richtungen bewegt werden, wenn Sie auch das FREEHORIZ-Flag setzen.

KNOBHIT

Dieses Flag können Sie nicht selber setzen. Es wird von Intuition gesetzt, wenn der Schieber gerade über die Maus angeklickt ist.

PROPBORDERLESS

Im Normalfall zeichnet Intuition einen Rand um den "container". Durch dieses Flag wird das unterbunden.

Nachdem Sie die gewünschten Flags gesetzt haben, dürfen Sie nicht vergessen, die anderen Werte der Struktur mit Angaben zu versehen.

HorizPot
Gibt die horizontale Position des Schiebers an.

VertPot
Gibt die vertikale Position des Schiebers an.

HorizBody
Hiermit bestimmen Sie die "Sprungstärke" des Gadget-Schiebers. Damit wird die Weite bezeichnet, die der Schieber springt, wenn Sie in den "container" klicken, aber nicht auf den Schieber selbst.

VertBody
Die gleiche Einstellung wie bei HorizBody, nur bezogen auf die vertikale Linie.

Alle folgenden Werte brauchen nicht vom Programmierer eingestellt zu werden. Sie werden von Intuition berechnet und können nach der Initialisierung des Gadgets abgefragt werden.

CWidth
Die Breite des "containers", des Klickbereiches.

CHeight
Die Höhe des "containers".

HPotRes
Auflösung des Schiebers in horizontaler Richtung.

VPotRes
Auflösung des Schiebers in vertikaler Richtung.

LeftBorder
Tatsächliche Position des linken Randes unseres "containers".

TopBorder
Tatsächliche Position des oberen Randes unseres "containers".

Für das Beispiel-Gadget sähe die PropInfo-Struktur etwa so aus:

```
struct PropInfo BeispielProp =
{
   AUTOKNOB | FREEVERT,     /*    Flags          */
   0x8000,                  /*    HorizPot       */
   0x8000,                  /*    VertPot        */
   0x0800,                  /*    HorizBody      */
   0x0800,                  /*    VertBody       */
   0,                       /*    CWidth         */
   0,                       /*    CHeight        */
   0,                       /*    HPotRes        */
   0,                       /*    VPotRes        */
   0,                       /*    LeftBorder     */
   0                        /*    TopBorder      */
};
```

Alle Strukturen sind beisammen! Wir können sie in unser Programm einsetzen. Aber denken Sie daran, daß natürlich zuerst die PropInfo-Struktur definiert werden muß, damit der Gadget-Struktur der Wert von BeispielProp bekannt ist. Gleiches gilt natürlich für die Window-Struktur, in die die Adresse der Gadget-Struktur "eingepflanzt" wird.

Wenn Sie dies alles befolgt haben, ist die Arbeit leider immer noch nicht abgeschlossen! Wir bekommen bei einem Proportional-Gadget keine Nachricht, daß ein Gadget "niedergedrückt" wurde (GADGETDOWN), sondern erhalten die Information, wenn das Gadget losgelassen wurde (GADGETUP). Dies wird ab jetzt in der Schleife auch noch berücksichtigt:

```
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
   MessageClass = message->Class;
   code = message->Code;
   ReplyMsg(message);
   switch (MessageClass)
      {
      case GADGETUP    : printf("Position: u%\n",
                                 BeispielProp.VertPot);
                         break;
```

```
        case CLOSEWINDOW : Close_All();
                           exit(TRUE);
                           break;

        case GADGETDOWN  : nr += 1;
                           printf("Gadget zum %d. Mal
                                          aktiviert!\n", nr);
                           break;

        }
    }
```

Programmteil 3.3: Abfrageschleife Proportional-Gadget

3.4.1.3 Das String-Gadget

String-Gadgets sind, genau wie Proportional-Gadgets, eine Weiterentwicklung der Boolean-Gadgets. Wie man sich leicht denken kann, fügen wir nicht eine PropInfo-Struktur an, sondern eine StringInfo-Struktur. Diese ist etwa gleich lang und enthält alle zusätzlichen Informationen, die ein String-Gadget benötigt.

Dieses String-Gadget haben Sie bestimmt schon einmal benutzt! Selbst die Workbench bietet eines an. Wenn Sie z.B. den Namen eines Files oder einer Diskette ändern, fahren Sie den Punkt Rename im Workbench-Menü an. Dann erscheint ein Fenster, in dem Sie den Namen korrigieren können. Das ist ein String-Gadget. In diesem Fall hat das Fenster auch die Größe des Gadgets, aber das ist nicht so wichtig.

Viel wichtiger ist, daß wir für ein String-Gadget ebenso eine Gadget-Struktur einrichten müssen wie für jedes andere Gadget auch. Dafür müssen Sie einen waagerechten Klickbereich definieren, denn die Texteingabe ist nur einzeilig möglich.

```
struct Gadget StringGadget =
   {
   NULL,                  /* NextGadget          */
   50, 40,                /* LeftEdge, TopEdge   */
   120, 10,               /* Width, Height       */
   GADGHCOMP,             /* Flags               */
   RELVERIFY |            /* Activation          */
   STRINGCENTER,
   STRGADGET,             /* Gadget Type         */
```

```
(APTR)&GadgetBorder,    /* GadgetRender      */
NULL,                   /* Select Render     */
&GadgetText,            /* GadgetText        */
NULL,                   /* MutualExclude     */
(APTR)&StringInfo,      /* SpecialInfo       */
3,                      /* GadgetID          */
NULL                    /* UserData          */
};
```

Einige Variablen bekommen bei String-Gadgets noch eine zusätzliche oder andere Bedeutung. Da ist zuerst das Flags-Flag. Es beschreibt nun das Aussehen des Cursors und nicht mehr das Aufblinken des ganzen Klickbereiches.

Unter dem Punkt *Activation* sind jetzt vier weitere Flags sinnvoll:

STRINGCENTER

Setzen Sie dieses Flag, wenn Sie möchten, daß der Text im Eingabefeld immer zentriert ausgegeben werden soll.

STRINGRIGHT

Setzen Sie dieses Flag, wenn Sie möchten, daß der Text immer zum rechten Rand ausgerichtet werden soll.

Hinweis: Die normale Einstellung bewirkt, daß der Text immer zum linken Rand ausgerichtet wird. Dafür müssen Sie kein Flag setzen.

LONGINT

Dieses Flag kennzeichnet ein Integer-Gadget. Dieser Typ erlaubt es dem Benutzer nur noch, Zahlenwerte einzugeben.

ALTKEYMAP

Für die Eingabe wird die allgemein definierte Tastaturtabelle verwendet. Wollen Sie eine andere zur Verfügung haben, so setzen Sie dieses Flag, und ergänzen Sie den Wert *AltKeyMap* in der StringInfo-Struktur.

Nachdem wir den Gadget-Typ mit STRGADGET spezifiziert haben, dürfen wir nicht vergessen, den Klickbereich durch eine

Grafik anzudeuten. Denn nur bei Proportional-Gadgets wird automatisch ein Rand gezeichnet. Ich verwende hier eine einfache Border-Struktur ohne viel Aufwand:

```
SHORT GadgetPairs[] =
   {
   0, 0, 122, 0, 122, 12, 0, 12, 0, 0,
   };

struct Border GadgetBorder =
   {
   -2, -2, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };
```

Sie können die Textzeile aber auch einfach durch einen Unterstrich kennzeichnen und müssen nicht einen ganzen Kasten nehmen. Der Gadget-Text ist zwar nicht nötig, Sie sollten ihn aber verwenden, damit der Benutzer immer weiß, was er dort eingeben soll:

```
struct IntuiText GadgetText =
   {
   2, 0, JAM2, -40, 1, NULL, (UBYTE *)"Text" ,NULL,
   };
```

Als letzte erwähnenswerte Ergänzung enthält unsere Struktur den Zeiger auf das *StringInfo*. Diese neue Struktur enthält folgende Parameter:

```
struct StringInfo
{
0x00  00  UBYTE *Buffer;
0x04  04  UBYTE *UndoBuffer;
0x08  08  SHORT BufferPos;
0x0A  10  SHORT MaxChars;
0x0C  12  SHORT DispPos;
0x0E  14  SHORT UndoPos;
0x10  16  SHORT NumChars;
0x12  18  SHORT DispCount;
0x14  20  SHORT CLeft;
0x16  22  SHORT CTop;
0x18  24  struct Layer *LayerPtr;
0x1C  28  LONG LongInt;
0x20  32  struct KeyMap *AltKeyMap;
0x24  36
};
```

Gleich die beiden ersten Werte sind eigentlich die wichtigsten. Sie zeigen jeweils auf einen Puffer, in dem die Zeichenketten untergebracht werden. Der erste Puffer enthält den Text, der wirklich eingegeben wurde. Der zweite Puffer ist für den Benutzer als Komfort gedacht. Dadurch ist es mit einem Tastendruck möglich, die Veränderung, die man am Text vorgenommen hat, rückgängig zu machen.

Solch einen Puffer können wir uns ganz schnell besorgen. Dafür definieren wir einfach ein Feld mit der entsprechenden Anzahl an Zeichen. Das sieht dann so aus:

```
#define STRINGSIZE 80
unsigned char StringBuffer[STRINGSIZE] = "Hallo Amiga!";
unsigned char UndoBuffer  [STRINGSIZE];
```

Es ist nicht unbedingt vorgeschrieben, einen *UndoBuffer* einzurichten. Allerdings hat der Benutzer dann auch nicht die Möglichkeit, seinen Text über diese Funktion zu korrigieren.

Mit *BufferPos* können Sie bei Initialisierung der Struktur den Cursor an eine bestimmte Stelle setzen. Dies ist interessant, wenn beim Aufruf schon ein Text im Puffer steht, wie im obigen Beispiel.

Da Intuition natürlich nicht weiß, wie groß der Textpuffer eingerichtet wurde, gibt man unter *MaxChars* an, wie viele Zeichen eingegeben werden dürfen. Es ist darauf zu achten, daß bei der Länge der Zeichenkette auch noch das abschließende NULL-Byte mitgezählt werden muß, denn es handelt sich hier um einen String!

Der letzte vom Programmierer einzustellende Wert ist *DispPos*. Die Nummer, die Intuition dort wiederfindet, kennzeichnet das erste Zeichen, das im Display-Feld ausgegeben wird.

Alle weiteren Werte werden von Intuition selbst verwaltet und berechnet. Sie können diese für Informationen abfragen.

UndoPos
Die Position des Cursors im *UndoBuffer*.

NumChars
Die Anzahl der Zeichen, die sich momentan im Puffer befinden.

DispCount
Die Anzahl der Zeichen, die im Klickbereich dargestellt werden können.

CLeft
Der Offset des "containers" zum linken Fensterrand.

CTop
Der Offset des "containers" zum oberen Fensterrand.

LayerPtr
Ein Zeiger auf den Layer, in dem dieses Gadget untergebracht ist.

LongInt
Hier steht bei einem Integer-Gadget - nach dem Return-Druck - die eingegebene Zahl. Sie können sie auslesen und dann weiterverarbeiten.

AltKeyMap
Wie oben schon erwähnt, können Sie hier einen Zeiger auf eine eigene Tastaturtabelle einsetzen, die dann für die Eingabe verwendet wird.

Wenden wir uns wieder der praktischen Anwendung zu. Bauen Sie dafür in unser Standardprogramm die Gadget-, die StringInfo- und die restlichen Strukturen ein.

Alle Definitionen müßten zusammen folgendermaßen aussehen:

```
#define STRINGSIZE 80
unsigned char StringBuffer[STRINGSIZE] = "Hallo Amiga!";
unsigned char UndoBuffer  [STRINGSIZE];

struct StringInfo StringInfo =
   {
   &StringBuffer[0],        /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   0,                       /* Buffer Position     */
   STRINGSIZE,              /* MaxChars            */
   0,                       /* Display Positoin    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL                     /* AltKeyMap           */
   };

SHORT GadgetPairs[] =
   {
   0, 0, 122, 0, 122, 12, 0, 12, 0, 0
   };

struct Border GadgetBorder =
   {
   -2, -2, 1, 0, JAM1, 5, GadgetPairs, NULL
   };

struct IntuiText GadgetText =
   {
   2, 0, JAM2, -40, 1, NULL, (UBYTE *)"Text" ,NULL
   };

struct Gadget StringGadget =
   {
   NULL,                    /* NextGadget          */
   50, 40,                  /* LeftEdge, TopEdge   */
   120, 10,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY |              /* Activation          */
   STRINGCENTER,
   STRGADGET,               /* Gadget Type         */
   (APTR)&GadgetBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &GadgetText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   (APTR)&StringInfo,       /* SpecialInfo         */
   1,                       /* GadgetID            */
   NULL                     /* UserData            */
   };
```

```
struct NewWindow FirstNewWindow =
   {
   160, 50,                  /* LeftEdge, TopEdge   */
   320, 200,                 /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   CLOSEWINDOW ¦             /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH ¦             /* Flags               */
   WINDOWSIZING ¦
   WINDOWDRAG ¦
   WINDOWCLOSE ¦
   SMART_REFRESH,
   &StringGadget,            /* First Gadget        */
   NULL,                     /* CheckMark           */
   (UBYTE *)"String-Gadget Test",
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   100, 50,                  /* Min Width, Height   */
   640, 256,                 /* Max Width, Height   */
   WBENCHSCREEN              /* Type                */
   };
```

Für das Programm verwenden wir eine etwas abgeänderte Abfrage-Schleife der Gadgets:

```
switch (MessageClass)
   {
   case GADGETUP    : printf("Der Text heißt: %s\n",
                              &StringBuffer[0]);
                      break;

   case CLOSEWINDOW : Close_All();
                      exit(TRUE);
                      break;
   }
```

Programmbeschreibung

In das Fenster wurde ein String-Gadget eingebaut, in dem Sie nach dem Anklicken den Text editieren können. Nach dem Druck auf die Return-Taste gibt das Programm den eingegebenen Text im DOS-Window aus.

Für jeden Return-Druck in einem String-Gadget erhält das Programm eine GADGETUP-Nachricht auf der Datenleitung. Diese kann als Bestätigung angesehen werden, denn die Eingabe ist dann abgeschlossen.

3.4.2 Gadgets in Aktion

Mit den drei eben beschriebenen Gadget-Typen lassen sich nun die vielfältigsten Anwendungen gestalten.

Einmal sind Gadgets durch ihre Symbolik besonders wirkungsvoll. Wir können damit ganz einfach die Prozeßauslösung steuern. Aber auch String- und Proportional-Gadgets sind nicht ohne. Wir wollen uns deshalb mit einigen Anwendungs-Beispielen beschäftigen, die sich einerseits in jedes andere Programm einbauen lassen, die aber auch im besonderen für unseren Editor geeignet sind.

Bisher wurden die Gadgets immer in die NewWindow-Struktur mit eingebunden, und das Programm hatte deshalb keine Sorgen mit der Verarbeitung. Dieses kann sich aber auch ändern, denn nicht immer ist es der Fall, daß gleich beim Öffnen des Fensters alle Gadgets gebraucht werden. Oft kommt es vor, daß nach und nach einige Gadgets in das Fenster eingefügt werden. Dafür und für vieles mehr stehen die Gadget-Funktionen zur Verfügung. Sie sind ein weiteres Themengebiet wert.

Das wohl größte Anwendungsgebiet der Gadgets sind die Requester. Das sind Ansammlungen vieler Gadgets in einem Sinnzusammenhang. Da wir dafür viele Gadgets brauchen, will ich Ihnen dazu schon einige Anwendungen vorstellen, die sich dann im Requester-Kapitel vervollständigen werden.

3.4.2.1 Die Gadget-Funktionen

Wir wollen uns zuerst mit dem oben angesprochenen Problem beschäftigen, daß die Gadgets nicht immer gleich beim Öffnen des Fensters erwünscht sind. In manchen Programmen kann es wünschenswert sein, daß ein Gadget bei Bedarf in das Window eingefügt wird. Auch dafür stellt Intuition viele Funktionen zur Verfügung.

Betrachten wir zuerst den Fall, daß ein Window geöffnet wird, wie es auch das Standardprogramm 1 macht. Dann, nachdem et-

was Zeit vergangen ist, denn wir haben noch keine Programmausführung, wird ein Gadget in die Window-Struktur eingebunden und auf dem Bildschirm dargestellt.

Ich beschreibe den Vorgang in Einzelschritten, weil wir in den gleichen Einzelschritten in der Programmierung vorgehen müssen. Als Grundprogramm verwenden Sie bitte das Standardprogramm 1 mit der globalen NewWindow-Struktur. Definieren Sie mit der nachfolgenden Gadget-Struktur ein Toggle-Gadget, das Sie aber nicht in die NewWindow-Struktur einbinden.

```
SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL
   };

struct IntuiText ToggleText =
   {
   3, 0, JAM2, 4, 7, NULL, (UBYTE *)"Toggle" ,NULL
   };

struct Gadget ToggleGadget =
   {
   NULL,                    /* NextGadget          */
   120, 40,                 /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHBOX,                /* Flags               */
   RELVERIFY |              /* Activation          */
   TOGGLESELECT,
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&GadgetBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &ToggleText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   1,                       /* GadgetID            */
   NULL                     /* UserData            */
   };
```

Struktur 3.11: Toggle-Gadget

Weil eventuell einige Leser noch kein Toggle-Gadget kennen, habe ich hier diesen Typ gewählt. Es ist ein Gadget, das bei

einmaligem Anklicken in den "selected"-Status übergeht und bei erneutem Anklicken wieder "unselected" wird.

Wollen wir jetzt dieses Gadget nachträglich in das Window einbinden, müssen wir eine Intuition-Funktion dafür benutzen. Sie trägt den Namen AddGadget() und hat folgende Syntax:

```
RealPosition = AddGadget(Window, Gadget, Position);
     D0           -42       A0      A1       D0
```

Wir übermitteln zuerst das Window, in das das Gadget eingefügt werden soll, und natürlich die Adresse des Gadgets selbst. Als nächstes können wir, aufgrund großer Freizügigkeit im Betriebssystem, sogar noch die Position bestimmen, die es in der Liste einnehmen soll. Als Ergebnis bekommen wir die wirklich erlangte Position zurück.

Für unser Toggle-Gadget würde das Kommando so aussehen:

```
Pos = AddGadget(FirstWindow, ToggleGadget, -1L);
```

Mit der -1 erklären wir, daß das Gadget als letztes in die Liste eingefügt werden soll.

Mit diesem Funktionsaufruf ist das Gadget zwar in die Liste des Windows übernommen, sehen kann es der Programmbenutzer aber immer noch nicht. Dafür bedarf es einer zusätzlichen Funktion, die sonst beim Einrichten eines neuen Fensters automatisch aufgerufen wird. Wir rufen sie ganz einfach unter ihrem Namen auf:

```
RefreshGadgets(Gadgets, Window, Requester);
     -222          A0        A1         A2
```

Diese Funktion zeichnet alle angegebenen Gadgets in der Window-Liste neu. Damit nicht alle neu ausgegeben werden - das ist ja nicht immer nötig -, bestimmen wir mit Gadgets, ab welchem Gadget die Routine beginnen soll. Deswegen haben wir unser neues Gadget auch ans Ende der Liste gehängt. Geben wir seine Nummer an, wird nur dieses neu gezeichnet und kein anderes. Man muß nämlich wissen, daß das Neuzeichnen Zeit in An-

spruch nimmt und außerdem sichtbar ist, weil die Grafik vorher gelöscht wird.

Fügen wir hinter den AddGadget()-Befehl RefreshGadgets() ein:

```
RefreshGadgets(Pos, FirstWindow, NULL);
```

Damit ist auch das Gadget sichtbar, das wir erst nachträglich eingefügt haben. Nun sehen wir uns noch die Abfrage eines Toggle-Gadgets an:

```
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
   MessageClass = message->Class;
   code = message->Code;
   GadgetPtr = (struct Gadget *) message->IAddress;
   SelectMode= GadgetPtr->Flags & SELECTED;

   ReplyMsg(message);
   switch (MessageClass)
      {
      case GADGETUP    : if (SelectMode)
                            {
                            printf("aktiviert ");
                            }
                         else
                            {
                            printf("deaktivier\n");
                            }
                         break;

      case CLOSEWINDOW : Close_All();
                         exit(TRUE);
                         break;
      }
   }
```

Programmteil 3.4: Abfrageschleife Toggle-Gadgets/Select-Mode

Die Überprüfung, ob ein Gadget niedergedrückt wurde, ist noch um eine Unterscheidung reicher geworden. Auch der Status des Gadgets wird jetzt untersucht, und zwar auf "selected" oder

"unselected". Durch eine einfache UND-Verknüpfung ließ sich das realisieren. Jetzt gibt das Programm in das DOS-Window aus, welchen Status das Gadget hat.

Im Moment hat dieses neue Toggle-Gadget noch keine weitere Funktion. Was könnte dieses Gadget ein- und ausschalten? Die Antwort ist ganz einfach. Ein anderes Gadget! Wir fügen ein neues Gadget hinzu und lassen es vom Toggle-Gadget ein- und ausschalten. Damit lernen wir sogar noch zwei neue Funktionen kennen. Als zweites Gadget wählen wir ein ganz normales Boolean-Gadget, um möglichst wenig Aufwand zu betreiben. Hier ist auch gleich die Struktur:

```
SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border SelectBorder =
   {
   -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL,
   };

struct IntuiText SelectText =
   {
   1, 0, JAM2, 4, 7, NULL, (UBYTE *)"Select" ,NULL,
   };

struct Gadget SelectGadget =
   {
   NULL,                    /* NextGadget          */
   10, 40,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP |              /* Flags               */
   GADGDISABLED,
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &SelectText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   2,                       /* GadgetID            */
   NULL                     /* UserData            */
   };
```

Struktur 3.12: Select-Gadget

(Wenn Sie wollen, können Sie jedes andere Gadget nehmen!)

Strukturbeschreibung

Wie gewohnt, habe ich Ihnen auch die Border- und die Intui-Text-Struktur gleich mitgeliefert, damit Sie in den weniger wichtigen Gebieten keine Arbeit haben. Richten wir unser Augenmerk jetzt auf das Select-Gadget.

Es handelt sich hierbei um ein ganz normales Boolean-Gadget mit der Besonderheit, daß es "disabled" ist. Es wurde durch das Flag GADGDISABLED in einen Zustand versetzt, der es für den Anwender unmöglich macht, es anzuwählen.

Dabei wird der ganze Klickbereich mit einem Raster überdeckt. Man nennt diesen grafischen Zustand auch "ghosted".

Auch dieses zweite Gadget muß dem Window "angehängt" werden. Dazu könnten wir noch einen weiteren AddGadget()-Befehl einbauen. Aber Intuition bietet auch für den Fall, daß mehrere Gadgets eingebunden werden müssen, eine weitere Funktion. Mit AddGList() können wir ganze Listen von Gadgets, die einfach gelinkt sind, in die Window-Liste einbinden. Setzen Sie dafür in das erste Gadget einen Zeiger auf das zweite. Jetzt können Sie mit

```
RealPosition = AddGList(Window, Gadget, Position, Numgad,
    D0          -438     A0      A1       D0       D1

                        Requester);
                          D2
```

die beiden Gadgets in unsere Window-Liste einfügen. Dafür sind die ersten drei Parameter bereits von der einfachen Funktion bekannt. Duch *Numgad* geben Sie an, wie viele Gadgets die Liste enthält, und der *Requester* muß auf den Requester zeigen, wenn wir einen benutzen. Das machen wir aber nicht! Also den Wert mit Null belegen.

Das gleiche Problem, das wir mit AddGadget() hatten, tritt auch bei RefreshGadgets() auf. Über AddGList() erfahren wir die Position unseres ersten Gadgets. Diese geben wir auch bei der Refresh-Funktion an, jedoch sollen ja insgesamt nur zwei Gadgets neu gezeichnet werden. Deshalb gibt es auch einen Listen-Befehl zum Refresh:

```
RefreshGList(Gadgets, Window, Requester, Numgad);
     -432      A0       A1         A2       D0
```

Analog zur AddGList-Funktion wird auch hier mit *Numgad* eingestellt, wie viele Gadgets einem Refresh unterworfen werden sollen.

Kehren wir jetzt zurück zu unserem Programm. Es soll nun das Boolean-Gadget wieder freigegeben werden, wenn das Toggle-Gadget aktiviert wurde und umgekehrt. Dafür müssen wir zuerst die beiden selbstdefinierten unterscheiden können. Hier hilft uns die *GadgetID*! Mit der Nummer, die wir dort eintragen können, ist es uns möglich, jedes Gadget zu Identifizieren. Wählen Sie deshalb in einem Programm möglichst verschiedene Nummern.

Dementsprechend muß jetzt auch die Abfrage geändert werden. Doch zuvor sollen Sie die beiden Funktionen kennenlernen, mit denen wir den Status beeinflussen können. Es sind dies:

```
OffGadget(Gadget, Window, Requester);
   -174     A0      A1         A2
```

und

```
OnGadget(Gadget, Window, Requester);
   -186     A0      A1         A2
```

Beide Funktionen benötigen die gleichen Parameter: einen Zeiger auf das Gadget, dessen Status geändert werden soll, den Zeiger auf das Window, in dem sich das Gadget befindet, und einen Zeiger auf einen Requester, wenn er vorhanden ist. Die letzte Variable ist für uns momentan noch uninteressant.

Wir brauchen die Abfrage nur dahingehend zu verändern, daß wir bei einem positiven Test des SELECT-Flags OnGadget()

aufrufen und bei ungesetztem Flag OffGadget(). Jedesmal übergeben wir einen Zeiger auf das Select-Gadget:

```
switch (GadgetNr)
   {
   case 2     : printf("selected!\n");
                break;

   case 1     : if (selectmode)
                   {
                   printf("aktiviert ");
                   OnGadget(&SelectGadget, FirstWindow,NULL);
                   RefreshGadgets(&SelectGadget,
                                        FirstWindow,NULL);
                   }
                 else
                   {
                   printf("deaktivier\n");
                   OffGadget(&SelectGadget, FirstWindow,NULL);
                 }
              break;
}
break;
```

Programmteil 3.5: Abfrageteil GadgetOn/Off

Diese Ergänzung fügen Sie bitte anstatt der If-Abfrage beim case GADGETUP ein. Zusätzlich müssen Sie die USHORT-Variable *GadgetNr* vorher berechnen lassen:

```
GadgetNr  = GadgetPtr->GadgetID;
```

Programmbeschreibung

Wenn Sie alle neuen Teile im ersten Programm ergänzt haben, sollte sich Ihnen folgendes Bild bieten:

Ein Fenster mit zwei Gadgets: Das rechte Gadget zum Ein- und Ausschalten und ein weiteres Gadget, das zuerst "ghosted" dargestellt ist. Klicken Sie nun auf das rechte Gadget, so sollte wenigstens die Schrift des anderen wieder leserlich sein. Jetzt können Sie auch dieses Gadget anklicken. Das Programm zeigt seine Reaktionen über Texte im DOS-Window. Für das Toggle-Gadget erscheinen Texte wie "aktiviert" und "deaktiviert". Nur wenn Sie das linke Gadget anklicken können, sendet das Programm den Text "selected!".

Nachdem wir uns so ausführlich mit den Boolean-Gadgets beschäftigt haben, werden nun auch die anderen beiden Typen beschrieben:

Zuerst soll unsere Aufmerksamkeit dem Proportional-Gadget gelten, für das es sogar zwei spezielle Funktionen gibt. In diesem Beispiel soll das Gadget nicht mit dem Autoknob versehen werden, sondern mit einer eigenen Grafik. Hier ist die dazugehörige Image-Struktur mit den Grafikdaten:

```
USHORT KnobGrafik[] =
   {
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999
   };

struct Image Knob =
   {
   0, 0,
   16, 16,
   1,
   &KnobGrafik[0],
   1, 0,
   NULL
   };
```

Zu dieser Grafik gehört natürlich die Gadget-Struktur mit der Erweiterung durch PropInfo:

```
struct PropInfo SchieberProp =
{
   FREEVERT,              /*   Flags          */
   0x8000,                /*   HorizPot       */
   0x8000,                /*   VertPot        */
   0x0800,                /*   HorizBody      */
```

```
    0x1000,                 /*    VertBody          */
    0,                      /*    CWidth            */
    0,                      /*    CHeight           */
    0,                      /*    HPotRes           */
    0,                      /*    VPotRes           */
    0,                      /*    LeftBorder        */
    0                       /*    TopBorder         */
};

struct Gadget Schieber =
    {
    NULL,                   /* NextGadget           */
    260, 40,                /* LeftEdge, TopEdge    */
    24, 120,                /* Width, Height        */
    GADGHNONE |             /* Flags                */
    GADGIMAGE,
    RELVERIFY,              /* Activation           */
    PROPGADGET,             /* Gadget Type          */
    (APTR)&Knob,            /* GadgetRender         */
    NULL,                   /* Select Render        */
    NULL,                   /* GadgetText           */
    NULL,                   /* MutualExclude        */
    (APTR)&SchieberProp,    /* SpecialInfo          */
    1,                      /* GadgetID             */
    NULL                    /* UserData             */
    };
```

Struktur 3.12: Proportional-Gadget: Schieber

Hinweis: Beachten Sie, daß die Grafikdaten des Schiebers im Chip-Memory abgelegt werden müssen.

Bevor das Programm startfertig ist, sollten wir noch eine kleine Ergänzung machen, um wenigstens die Position des Schiebers untersuchen und auswerten zu können. Setzen Sie diese zwei Zeilen in die Gadget-Abfrage ein:

```
    case GADGETUP      : printf("Schieber-Position: %x\n",
                                    SchieberProp. VertPot);
                         break;
```

Wir können nun in einer Anwendung die Position des Schiebers im Programm abfragen und auswerten. Allerdings gibt es auch Situationen, in denen es wichtig ist, die Lage des Schiebers durch das Programm zu verändern, z.B. die Größe oder ähnliches. Grundsätzlich ist das auch alles zugelassen. Zuerst möchte ich eine Methode beschreiben, die sich auf jedes Gadget bezieht und die Sie auch bei jedem Typ einmal ausprobieren sollten!

Danach erkläre ich noch zwei Funktionen, die, wie oben schon angedeutet, extra für Proportional-Gadgets eingerichtet wurden.

Zur allgemeinen Methode: Um irgendwelche Einstellungen an einem Gadget zu ändern, ist es zuerst nötig, dieses Gadget wieder aus der Liste des Windows zu entfernen. Wird dies nicht gemacht, dann treten Fehler auf! Demnach benötigen wir als erstes eine Funktion, die es erlaubt, ein Gadget wieder aus der Liste zu nehmen. Mit RemoveGadget() ist das kein Problem:

```
Position = RemoveGadget(Window, Gadget);
   D0         -228          A0       A1
```

Nehmen wir als Beispiel das Proportional-Gadget. Es läßt sich ganz einfach mit folgendem Befehl wieder entfernen:

```
Pos = RemoveGadget(FirstWindow, Schieber);
```

Jetzt können wir an der Gadget-Struktur, an der PropInfo-Struktur und allen angebundenen Strukturen Veränderungen vornehmen. Danach läßt es sich wieder über AddGadget() in die Liste eingliedern. Zum Schluß müssen Sie noch über RefreshGadgets() das Aussehen des Gadgets aktualisieren, damit die Ausgabe auf dem neuesten Stand ist.

In anderen Fällen, in denen ganze Mengen von Gadgets entfernt, geändert und wieder in die Liste eingetragen werden, gibt es noch einen zusätzlichen Befehl:

```
Position = RemoveGList(Window, Gadget, Numgad);
   D0           -444       A0       A1       D0
```

Hier geben Sie zusätzlich die Anzahl der Gadgets an, die entfernt werden sollen. Das Gegenstück, AddGList(), ist Ihnen bereits bekannt. Vergessen Sie auch hier nicht, mit RefreshGList() die Grafik zu aktualisieren.

Der zweite Weg, der ausschließlich für Proportional-Gadgets gedacht ist, erspart uns den Aufwand, jedes Gadget aus der Liste zu lösen und dann wieder einzufügen. Mit der Funktion ModifyProp() können wir, nach Angabe des Gadgets, alle vom Benutzer möglichen Einstellungen der PropInfo-Struktur wieder

verändern. Das Gadget wird danach automatisch wieder neu gezeichnet.

```
ModifyProp(Gadget, Window, Requester, Flags, HorizPot, VertPot,
   -156      A0      A1        A2        D0      D1       D2

           HorizBody, VertBody);
              D3        D4
```

Leider hat diese Funktion einen kleinen Nachteil: In manchen Anwendungen ist es ziemlich oft nötig, über ModifyProp() die Einstellungen zu ändern. Da die Funktion aber auf RefreshGadgets() zurückgreift, werden alle nachfolgenden Gadgets in der Liste auch "refreshed". Das ist störend, und deshalb wurde NewModifyProp() zu den Intuition-Funktionen aufgenommen.

```
NewModifyProp(Gadget, Window, Requester, Flags, HorizPot, VertPot,
     -468        A0       A1        A2        D0      D1       D2

                HorizBody, VertBody, Numgad);
                   D3        D4        D5
```

Bei der neuen Funktion, die genau die gleiche Aufgabe hat, können Sie noch wählen, wie viele Gadgets in der Liste "refreshed" werden sollen. Als geeignetsten Wert empfehle ich hier 1, es sei denn, Sie haben noch einige Gadgets, die von der Veränderung betroffen sind und gleich mitbearbeitet werden sollen.

Sehen Sie sich als nächstes folgendes Problem an: Wir haben im Window eine Liste von Namen, die untereinander geschrieben wurden. Sie möchten nun, daß der Benutzer einen dieser Namen aussuchen kann. Wie könnten wir dieses Problem lösen?

Zuerst fragen Sie sich natürlich, wann so ein Fall überhaupt auftreten könnte. Denken Sie doch einmal an eine File-Select-Box. Da haben wir eine Liste von File-Namen, aus denen wir uns einen aussuchen sollen. Das gleiche Problem haben wir auch.

Ich möchte gleich allen den Wind aus den Segeln nehmen, die gedacht haben, daß man einfach über den IntuiText, der in die Gadget-Struktur eingebunden werden kann, die Namen ausgibt. Dem möchte ich entgegenhalten, daß diese Methode ziemlich umständlich wird, wenn erst eine File-Liste vorhanden ist, die

mehr Namen enthält als in das Window passen. Dann müssen die Texte beim Scrollen immer wieder geändert werden. Dazu eignet sich diese Methode aber überhaupt nicht. Sie ist viel zu umständlich.

Am besten wäre es, eine Routine damit zu beauftragen, die Liste der Namen, beginnend bei einem bestimmten Namen, in festgelegter Anzahl auszugeben. Vorher richten wir in diesem Feld die festgelegte Anzahl von Gadgets ein, die gar keine Grafik besitzen. Nur durch den Klickbereich sind sie zu erkennen. Dieser darf natürlich nicht höher als eine Zeile, also 8 bzw. 9 Punkte, sein und muß die maximal zugelassene Zeichenzahl eines File-Namens umfassen. Da dies alles Boolean-Gadgets sind, können wir mit der einfachen Gadget-Struktur arbeiten:

```
struct Gadget FileGadget =
   {
   NULL,                  /* NextGadget          */
   80, 0,                 /* LeftEdge, TopEdge   */
   200, 8,                /* Width, Height       */
   GADGHCOMP,             /* Flags               */
   RELVERIFY,             /* Activation          */
   BOOLGADGET,            /* Gadget Type         */
   NULL,                  /* GadgetRender        */
   NULL,                  /* Select Render       */
   NULL,                  /* GadgetText          */
   NULL,                  /* MutualExclude       */
   NULL,                  /* SpecialInfo         */
   0,                     /* GadgetID            */
   NULL                   /* UserData            */
   };
```

Struktur 3.13: FileGadget

Wegen der großen Breite unserer Gadgets möchte ich Sie bitten, jetzt erneut das Standardprogramm 1 zu laden und dort die Window-Struktur dahingehend zu ändern, daß das Window als *LeftEdge* 120 und als *Width* 360 hat. Wir wollen nämlich jetzt schon mit der Entwicklung einer File-Select-Box beginnen, und dazu brauchen wir ein so breites Window. Sehen Sie sich vielleicht schon mal die Entwurfszeichnung an:

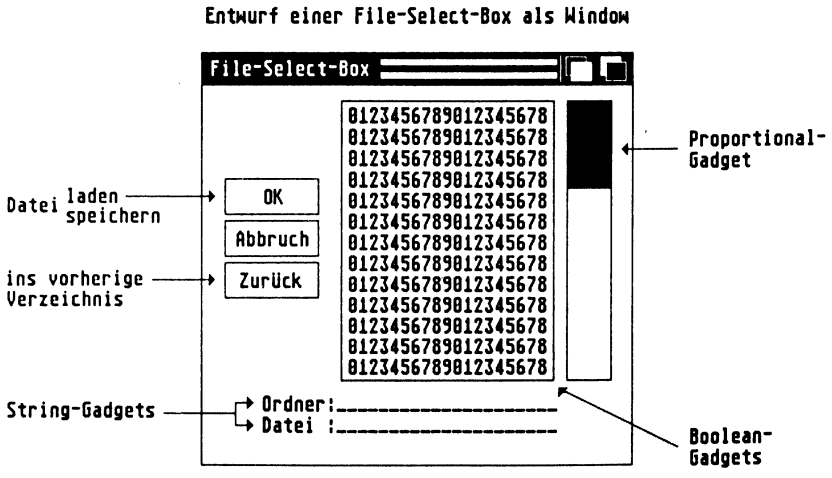


Abbildung 3.11: Entwurf: Aussehen File-Select-Box

Wir brauchen also ein großes Feld für die File-Namen, einen Schiebebalken, den wir ja oben bereits programmiert haben, zwei String-Gadgets und drei Boolean-Gadgets. Nach diesem Abstecher zur Übersicht unseres Projektes kehren wir zurück zu den vielen Gadgets für das Selektieren eines Namens.

Mit der ersten definierten Gadget-Struktur werden wir nicht weit kommen. Wir brauchen einiges mehr, doch jede Struktur einzeln zu definieren macht ja viel zu viel Arbeit. Erinnern wir uns deswegen an das Beispiel der IntuiText-Strukturen. Dort haben wir auch eine Struktur mehrfach reproduziert. Das läßt sich hier genauso handhaben:

Vor der Hauptfunktion definieren wir diese Gadget-Struktur und noch ein Feld mit noch nicht ausgefüllten Strukturen.

```
struct Gadget Files[15];
```

Dazu brauchen wir dann im Programm eine Funktion, die die allgemeine Struktur dort hineinkopiert und gleichzeitig die Y-Position korrigiert.

```
Make_Files(Files)

struct Gadget Struktur[];

   {
   int i;

   for (i=0; i<15; i++)
      {
      Struktur[i]            = FileGadget;
      Struktur[i].TopEdge    = 30 + i * 8;
      Struktur[i].GadgetID   = i + 1;
      Struktur[i].NextGadget = &Struktur[i+1];
      }
   Struktur[14].NextGadget = NULL;
   }
```

Funktion 3.5: Make-File-Gadgets

Dokumentation

Die Funktion trägt in das Strukturenfeld jedesmal die allgemeine Struktur ein und korrigiert den Y-Wert (TopEdge). Außerdem linkt sie die gesamte Liste der 15 Gadgets zusammen, damit sie nachher durch einen einfachen Aufruf (AddGList()) in das Window gebracht werden kann. Weiterhin bekommt jedes Gadget eine ID beginnend bei 1. Dadurch läßt sich später bei der Auswahl feststellen, welches der User-Gadgets angeklickt wurde. Ein Refresh ist in diesem Fall nicht nötig, da sowieso keine Grafik verwendet wird.

Somit haben wir das erste Problem gelöst.

Fügen Sie jetzt das oben definierte Proportional-Gadget mit in das Programm ein. Verändern Sie dazu die Variable *LeftEdge* auf den Wert 300. Außerdem muß die ID dieses Gadgets einen Wert größer als 15 erhalten, damit es von den anderen zu unterscheiden ist. Ich schlage 20 vor, damit etwas Platz bleibt für Ergänzungen.

Jetzt fehlen uns noch die beiden String-Gadgets. Da es nur zwei sind, können wir sie normal über die Strukturen definieren. Hierbei tritt der wunderbare Zustand ein, daß wir nur einen

Undo-Puffer benötigen, da immer nur ein Gadget zur Zeit aktiv sein kann.

```
unsigned char OrdnerBuffer[512] = "df1:";
unsigned char DateiBuffer[31]   = "";
unsigned char UndoBuffer  [512];
struct StringInfo OrdnerInfo =
   {
   &OrdnerBuffer[0],        /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   0,                       /* Buffer Position     */
   511,                     /* MaxChars            */
   0,                       /* Display Positoin    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL                     /* AltKeyMap           */
   };

struct StringInfo DateiInfo =
   {
   &DateiBuffer[0],         /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   0,                       /* Buffer Position     */
   31,                      /* MaxChars            */
   0,                       /* Display Position    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL                     /* AltKeyMap           */
   };

SHORT StringPairs[] =
   {
   0, 0, 244, 0
   };

struct Border StringBorder =
   {
   0, 10, 1, 0, JAM1, 2, StringPairs, NULL
   };

struct IntuiText OrdnerText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Ordner:" ,NULL
   };
```

```
struct IntuiText DateiText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Datei :" ,NULL
   };

struct Gadget Ordner =
   {
   NULL,                    /* NextGadget          */
   80, 160,                 /* LeftEdge, TopEdge   */
   245, 10,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   STRGADGET,               /* Gadget Type         */
   (APTR)&StringBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &OrdnerText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   (APTR)&OrdnerInfo,       /* SpecialInfo         */
   21,                      /* GadgetID            */
   NULL                     /* UserData            */
   };

struct Gadget Datei =
   {
   &Ordner,                 /* NextGadget          */
   80, 160,                 /* LeftEdge, TopEdge   */
   245, 10,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   STRGADGET,               /* Gadget Type         */
   (APTR)&StringBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &DateiText,              /* GadgetText          */
   NULL,                    /* MutualExclude       */
   (APTR)&DateiInfo,        /* SpecialInfo         */
   22,                      /* GadgetID            */
   NULL                     /* UserData            */
   };
```

Struktur 3.14: String-Gadgets: Datei & Ordner

Als letzte Gadgets für unsere File-Select-Box fehlen nur noch die Boolean-Gadgets. Hier ist es ganz einfach: Mit nur einer Border-Struktur, drei IntuiText-Strukturen und drei Gadget-Strukturen ist alles erledigt:

```
SHORT SelectPairs[] =
   {
   0, 0, 60, 0, 60, 20, 0, 20, 0, 0
   };
```

```
struct Border SelectBorder =
   {
   -2, -2, 1, 0, JAM1, 5, SelectPairs, NULL
   };

struct IntuiText OKText =
   {
   1, 0, JAM2, 1, 5, NULL, (UBYTE *)"  OK   " ,NULL
   };

struct IntuiText AbbruchText =
   {
   1, 0, JAM2, 1, 5, NULL, (UBYTE *)"Abbruch" ,NULL
   };

struct IntuiText ZurueckText =
   {
   1, 0, JAM2, 1, 5, NULL, (UBYTE *)"Zurück!" ,NULL
   };

struct Gadget OK =
   {
   NULL,                    /* NextGadget          */
   10, 40,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &OKText,                 /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   23,                      /* GadgetID            */
   NULL                     /* UserData            */
   };

struct Gadget Abbruch =
   {
   &OK,                     /* NextGadget          */
   10, 65,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &AbbruchText,            /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   24,                      /* GadgetID            */
   NULL                     /* UserData            */
   };
```

```
struct Gadget Zurueck =
   {
   &Abbruch,                  /* NextGadget          */
   10, 90,                    /* LeftEdge, TopEdge   */
   60, 20,                    /* Width, Height       */
   GADGHCOMP,                 /* Flags               */
   RELVERIFY,                 /* Activation          */
   BOOLGADGET,                /* Gadget Type         */
   (APTR)&SelectBorder,       /* GadgetRender        */
   NULL,                      /* Select Render       */
   &ZurueckText,              /* GadgetText          */
   NULL,                      /* MutualExclude       */
   NULL,                      /* SpecialInfo         */
   25,                        /* GadgetID            */
   NULL                       /* UserData            */
   };
```

Struktur 3.15: Boolean-Gadgets: OK, Abbruch & Zurück

Wir haben nun drei gelinkte Gadget-Listen: die File-Gadgets ohne Grafik, die beiden String-Gadgets und die Boolean-Gadgets. Außerdem haben wir noch ein Proportional-Gadget. Bauen Sie jetzt bitte alle Gadget-Definitionen in das Standardprogramm ein, und denken Sie an die Window-Ausmaße! Dann ergänzen wir im Hauptprogramm nach dem Funktionsaufruf Open_All():

```
Make_Files(Files);

FilePos = AddGList(FirstWindow, Files, -1L, 15L, NULL);
StrgPos = AddGList(FirstWindow, Datei, -1L, 2L, NULL);
SlctPos = AddGList(FirstWindow, Zurueck, -1L, 3L, NULL);
PropPos = AddGadget(FirstWindow, Schieber, -1L);

RefreshGList(Datei, FirstWindow, NULL, 2L);
RefreshGList(Zurück, FirstWindow, NULL, 3L);
RefreshGList(Schieber, FirstWindow, NULL, 1L);
```

Programmteil 3.6: Gadgets einbinden

Die Gadget-Abfrage wollen wir jetzt noch nicht mit einer Auswertung beauftragen. Hier setzen wir einfach eine Ausgabe der GadgetID ein, falls es sich um ein selbstdefiniertes handelt. Die nötigen Programmteile finden Sie dazu im Beispiel "Toggle-Gadget zum Auschalten".

Es fehlt lediglich noch die Zeile:

```
printf("Gadget-Nummer: %d\n", GadgetNr);
```

3.4.2.2 Das neue Sizing-Gadget

Nun kommen wir endlich zum neuen Sizing-Gadget. Hier noch einmal eine kurze Funktionsbeschreibung: Das Gadget wird in den Window-Rand eingebaut. Wird es dort angeklickt und hat das Window nicht die maximale Größe, die es beim Screen annehmen kann, wird es vergrößert. Hat es die maximale Größe im Screen, wird es bei erneutem Anklicken auf sein Minimum gebracht. Achten Sie dafür darauf, daß die *MinWidth*- und *MinHeight*-Werte initialisiert sind.

```
struct Gadget Sizing =
   {
   NULL,                        /* NextGadget          */
   -10, 15,                     /* LeftEdge, TopEdge   */
   8, 10,                       /* Width, Height       */
   GADGHCOMP |                  /* Flags               */
   GRELRIGHT,
   RELVERIFY |                  /* Activation          */
   RIGHTBORDER,
   BOOLGADGET,                  /* Gadget Type         */
   (APTR)&SizingImage,          /* GadgetRender        */
   NULL,                        /* Select Render       */
   NULL,                        /* GadgetText          */
   NULL,                        /* MutualExclude       */
   NULL,                        /* SpecialInfo         */
   0,                           /* GadgetID            */
   NULL                         /* UserData            */
   );
```

Struktur 3.16: Sizing-Gadgets

Als Image-Struktur übernehmen Sie bitte die, die wir am Ende des letzten Kapitels dafür erarbeitet haben. Die Abfrageroutine sollte dann folgende Arbeiten vornehmen:

```
Sizing()
{
   SHORT ScreenWidth, ScreenHeight;

   ScreenWidth = Window->WScreen->Width;
```

```
    ScreenHeitgh= Window->WScreen->Height;

    if (Window->LeftEdge == NULL & Window->TopEdge == NULL)
       if (Window->Width == ScreenWidth &
                                  Window->Height == ScreenHeight)
          SizeWindow(Window, -1*(Window->Width - Window->MinWidth),
                             -1*(Window->Height - Window->MinHeight);

    MoveWindow(Window, -1*Window->LeftEdge, -1*Window->TopEdge);
    SizeWindow(Window, ScreenWidth - Window->Width, ScreenHeight -
                                                  Window->Height);
}
```

Funktion 3.6: Size-Window

3.5 Requester: Viele Informationen auf einmal

Requester stellen eine Weiterentwicklung der Gadgets dar. Gadgets sind in erster Linie dafür gedacht, während des Programmablaufs Informationen und Anweisungen zu sammeln.

Es kommt manchmal vor, daß das Programm ganz dringend eine Entscheidung oder Information benötigt. In diesem Fall kann das Programm nicht so lange warten, bis irgendwann einmal die erwartete Information geliefert wird. Hierfür muß der Benutzer "erpreßt" werden, die Entscheidung zu fällen, damit das Programm weiterarbeiten kann. Unter "erpressen" hat man zu verstehen, daß es so lange nicht mehr erlaubt ist, in der Arbeit fortzufahren, bis die Nachfrage beantwortet ist.

In diesem Fall tritt ein Requester in Erscheinung. Das ist eine Ansammlung von Gadgets (mindestens eins), die bei der Beantwortung der Frage helfen. Natürlich können zu den Gadgets noch Texte treten, die das Problem oder die Frage näher erläutern. Es ist sogar möglich, ganze Grafiken in den Requester einzubauen.

Das einfachste Beispiel sind die System-Requester. Sie erscheinen, z.B. sobald irgendeine Diskette eingelegt werden muß. Bis Sie entweder das Einlegen verweigert haben oder Sie die Diskette tatsächlich einlegen, ist das Weiterarbeiten auf der Workbench blockiert. Ein großer Vorteil der Requester ist, daß nicht

nur durch Anklicken eines Requester-Gadgets weitergearbeitet werden kann, sondern auch durch Einlegen einer Diskette. Es sind also mehrere Eingabewege offen.

Das File-Select-Box-Window, das wir im vorhergehenden Kapitel zusammengebaut haben, wollen wir an dieser Stelle in einen Requester umbauen. Dieser dient als Nachfrage für die Anweisung, daß ein File geladen oder gespeichert werden soll. Wir benötigen aber ebenfalls die Information, wie das File heißt und in welchem Verzeichnis es liegt.

Zuvor werden wir uns langsam mit der Materie beschäftigen und uns dazu die einfachen, von Intuition größtenteils unterstützten Requester ansehen. Damit kann man schon einiges machen, und die Programmierung ist wesentlich einfacher.

Ich wünsche viel Spaß bei der Requester-Programmierung!

3.5.1 Der Automatik-Requester

Wie schon erwähnt, sind Requester eine Verknüpfung mehrerer Gadgets. Allerdings haben Sie bestimmt auch schon die Erfahrung gemacht, daß die Gadget-Programmierung zwar einfach, aber dafür doch ziemlich schreibaufwendig ist. Wir kommen selten mit nur einer Struktur aus, denn ohne Grafik läuft hier nicht viel.

Intuition bietet als einfache Lösung für dieses Problem den Automatik-Requester an. Das ist ein Requester, der immer aus zwei Gadgets und einem erläuternden Text besteht. Die einfachen Anfragen lassen sich damit leicht lösen und erfordern keine aufwendige Programmierung. Der besondere Vorteil des Automatik-Requesters ist, daß es dafür nicht einer eigenen Struktur bedarf. Wir übergeben der Funktion einfach die Zeiger auf die IntuiText-Strukturen und bekommen einen Wert zurück, wenn der Requester beantwortet wurde.

Für die Arbeit mit unserem Automatik-Requester benötigen wir erst einmal einen Zeiger auf ein Fenster. Dieser muß aber nicht unbedingt initialisiert sein.

Als nächstes können drei IntuiText-Strukturen angegeben werden. Die erste steht für einen allgemeinen Text, der am besten zur Erläuterung des Problems verwendet wird. Meistens steht hier eine Frage oder Anweisung. Die anderen beiden Texte sind für zwei Gadgets vorgesehen. Sie kennen das sicherlich von den System-Requestern. In der linken Ecke steht der "PositiveText", der ein Zustimmen oder ein Bestätigen kennzeichnen sollte. Der dritte Text stellt genau das Gegenteil dar und wird in der rechten Ecke unseres Requesters ausgegeben. Nehmen Sie hier einen Text, der verneint, abbricht oder ablehnt.

Jetzt fehlen uns noch zwei Flags. Das erste steht für den auslösenden Faktor des zustimmenden Gadgets. Sie können dieses Flag auf Null setzen, denn das Anklicken des Gadgets ist immer auslösender Bestandteil. Das zweite Flag erfüllt genau die gleiche Aufgabe wie das erste, allerdings in bezug auf das ablehnende Gadget. Die letzten beiden Werte kommen Ihnen sicherlich bekannt vor. Da Intuition nicht weiß, welche Ausmaße der Text und die Gadgets haben werden, können Sie an dieser Stelle die Größe des Requesters festlegen. Über die Größe der Umrandung unserer Gadget-Texte brauchen Sie sich überhaupt keine Gedanken zu machen, Intuition berechnet anhand des Textes selbständig die Ausmaße.

Hier haben Sie das allgemeine Format unserer neuen Funktion mit allen Parametern in einer Kurzbeschreibung:

```
Response = AutoRequest(Window, BodyText, PositiveText,
   D0          -348      A0       A1          A2

                       NegativeText, PositiveFlags, NegativeFlags,
                          A3            D0              D1

                       Width, Height);
                         D2     D3
```

Parameter	
Window	Zeiger auf eine Window-Struktur.
BodyText	IntuiText-Struktur für den allgemeinen Text.
PositiveText	IntuiText des Ja-Feldes.
NegativeText	IntuiText des Nein-Feldes.
PositiveFlags	IDCMP-Flag für das Ja-Feld.
NegativeFlags	IDCMP-Flag für das Nein-Feld.
Width	Breite des Auto-Requester-Windows.
Height	Höhe des Auto-Requester-Windows.

Das Beispiel für den Requester benötigt nicht einmal das Standardprogramm. Wir initialisieren einfach die IntuiText-Strukturen und übergeben alle Werte der Funktion. Von dieser erhalten wir nach Beantwortung durch den Benutzer den Wahrheitswert zurück. Nach dessen Auswertung können wir im DOS-Fenster einen entsprechenden Text ausgeben. Das folgende Programm erledigt diese Arbeit für uns:

```
/**********************************
 *                                *
 * Programm: AutoRequester        *
 * ============================== *
 *                                *
 * Autor:  Datum:      Kommentar: *
 * ------  ----------  ---------- *
 * Wgb     27.12.1987             *
 *                                *
 *                                *
 **********************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *Window = NULL;


struct IntuiText Body =
   {
   0, 1,
   JAM2,
   10, 10,
   NULL,
   (UBYTE *)"Wollen Sie Requester Programmieren?",
   NULL
```

```
   };

struct IntuiText JA =
   {
   2, 3,
   JAM2,
   5, 3,
   NULL,
   (UBYTE *)" Ich will !!!",
   NULL
   };

struct IntuiText NEIN =
   {
   2, 3,
   JAM2,
   5, 3,
   NULL,
   (UBYTE *)" Bloß nicht !!!",
   NULL
   };


main()
   {
   BOOL Antwort;

   Open_All();

   Antwort = AutoRequest(Window, &Body, &JA,
                                 &NEIN, 0L, 0L, 320L, 60L);

   if (Antwort == TRUE)
      printf("Wer's glaubt, wird selig!\n");
   else
      printf("Die Warheit ist immer am besten.\n");

   Close_All();
   }


/**********************************
 *                                *
 * Funktion: Library öffnen       *
 * ============================== *
 *                                *
 * Autor:  Datum:      Kommentar: *
 * ------  ----------  ---------- *
 * Wgb     16.10.1987             *
 *                                *
 *                                *
 **********************************/
```

```
Open_All()
   {
   void          *OpenLibrary();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Funktion: Alles Geöffnete schließen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  Intuition     *
 *                                   *
 *                                   *
 *************************************/

Close_All()
   {
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

Programm 3.6: Auto-Requester

Dokumentation

Das Besondere an diesem Programm ist, daß wir einen System-Request auch ausgeben können, wenn wir kein Bezugs-Window haben. Dies ist z.B. wichtig bei der Aufforderung, Speicher frei zu machen, wenn es nicht möglich ist, das Ausgabe-Window zu öffnen. Dafür setzen wir den Window-Pointer auf Null. Denken Sie daran, daß dies zwar bei jeder Requester-Funktion möglich ist, aber dem Zweck wenig dienlich ist. Wir wollen schließlich die Eingabe umleiten.

Ein besonderer Vorteil ist es, daß AutoRequester darüber hinaus nicht nur über die Maus beantwortet werden können. Auch das Einlegen einer Diskette kann die Antwort für eine der beiden Entscheidungen sein. Der Requester läßt sich auch über die Ta-

statur beantworten. Ein besonderer Komfort! Für das "Retry"-Feld kann auch die rechte Amiga-Taste mit V gedrückt werden, das "Cancel"-Feld können Sie durch Amiga-B simulieren. Einfacher geht es nun wirklich nicht!

Der einzige minimale Nachteil der Auto-Requester liegt im Window. Wir können keine Eigenschaft des Requester-Windows bestimmen und sind deshalb auf die Überschrift, die Gadgets und auch die Ausmaße angewiesen. Darüber kann man angesichts der einfachen Programmierung hinwegsehen.

3.5.2 Der System-Requester

Die Bedienung und Programmierung des Auto-Requesters ist sehr einfach. Einziger Nachteil: Es besteht keine Möglichkeit des Eingriffs in den Aufbau.

Deshalb hat man es ermöglicht, einen Teil der AutoRequest-Routine zu nutzen, um dann alles Restliche selber zu verändern. Diese Routine heißt BuildSysRequest() und ist eine Unterroutine von AutoRequest(). Die folgenden Voraussetzungen müssen wir erfüllen, damit wir mit BuildSysRequest() arbeiten können:

Zuerst muß ein Window geöffnet werden, das sich nach der Größe der Texte richten sollte. Dann übergeben wir wie üblich den Text für eine allgemeine Beschreibung und die Texte für die beiden Gadgets. Außerdem benötigen wir noch das IDCMP-Flag für das Requester-Window und als letztes die Größe des Requesters.

Mit diesen Parametern läßt die Routine nun einen Requester entstehen. Dieser wird im Stil dem AutoRequest gleichen: der Body-Text im oberen Teil des Requesters, in der unteren rechten und linken Ecke je ein Gadget mit den beiden anderen Texten. Als Kommunikationskanal bekommen wir den Zeiger auf das Requester-Window zurück. Entweder ist dies der Zeiger, den wir übergeben haben, oder aber wir haben Null übergeben, und die Routine hat selbständig auf der Workbench ein Fenster geöffnet.

Die Abfrageroutine für den neuen Requester müssen wir allerdings selbst schreiben. Dabei können alle IDCMP-Möglichkeiten genutzt werden. Es ist dabei unerheblich, ob Tastatureingabe oder Mausabfrage gewünscht wird. Alle Eingaben sind erlaubt.

Wenn die Routine Probleme beim Öffnen des Requesters hat, wird dieser Requester wie auch ein AutoRequest in eine recoverable Alert umgewandelt. Dann liefert BuildSysRequest() nicht den Zeiger auf ein Window zurück, sondern einen Wahrheitswert, mit dem wir erfahren, welche der beiden Antworten gewählt wurde.

Vor weiteren Beschreibungen sollen Sie erst das Format dieser Funktion kennenlernen:

```
ReqWindow = BuildSysRequest(Window, BodyText, PositiveText,
    D0          -360          A0       A1          A2

                          NegativeText, IDCMPFlags, Width, Height);
                               A3          D0        D2      D3

struct Window          *ReqWindow;
struct Window          *Window;
struct IntuiText       *BodyText;
struct IntuiText       *PositiveText;
struct IntuiText       *NegativeText;
ULONG  IDCMPFlags;
SHORT  Width, Height;
```

Für die Abfrageroutine, die Sie eigenhändig schreiben müssen, brauchen Sie noch einige Informationen über die Gadgets. Bei jedem Gadget sind die folgenden Gadget-Flags gesetzt:

```
BOOLGADGET, RELVERIFY, REQGADGET  und  TOGGLESELECT
```

Die Gadgets sollten in Ihrem Interesse mit dem gleichen Aussehen versehen werden, wie es allgemein üblich ist. Dafür stehen die AUTO-Flags im <intuition/intuition.h>-Include-File zur Verfügung:

Auto-Name	Wert	Beschreibung
AUTODRAWMODE	JAM2	Hintergrund- und Vordergrundfarbe
AUTOFRONTPEN	0L	Zeichenfarbe 1
AUTOBACKPEN	1L	Zeichenfarbe 2
AUTOLEFTEDGE	6L	Positionen des Intui-Textes
AUTOTOPEDGE	3L	
AUTOITEXTFONT	NULL	Zeichensatz für Texte: Workbench-Zs.
AUTONEXTTEXT	NULL	Zeiger auf weitere Texte

Tabelle 3.13: AUTO-Konventionen

Alle Flags können in den IntuiText-Strukturen verwendet werden, damit eine einheitliche Gestaltung gewährleistet ist. Sie sollten das beachten, denn der Benutzer würde einfach überfordert werden, wenn er sich ständig andere Konventionen merken müßte.

Mit dieser Beschreibung sollte es Ihnen möglich sein, einen System-Requester zu öffnen. Allerdings fehlt noch die Reaktion beim Schließen des Requesters. Diese wird gebraucht, um alle Strukturen und jeden Speicher, der für die Arbeit gebraucht wurde, wieder freizugeben. Diese Aufgabe übernimmt die Funktion FreeSysRequest(). Ihr übergeben Sie einfach den Zeiger, den die BuildSysRequest()-Routine zurückgeliefert hatte. Wurde nur ein Wahrheitswert ausgegeben, weil kein Speicher für einen Requester vorhanden war, darf diese Funktion nicht aufgerufen werden.

3.5.3 Wir bauen uns den Requester selbst

Nicht immer kommt man mit den Möglichkeiten des Auto-Requesters weiter. In vielen Fällen bietet er einfach zu wenig mit seinem Ja/Nein-Rückgabewert. Dann tritt der CustomRequester in Aktion!

Für einen CustomRequester braucht das System wieder so viele Angaben, daß es sich lohnt, dafür eine Struktur einzurichten. Diese Struktur enthält alle Informationen über Art, Umfang und Aussehen des Requesters. Allerdings wird der Vorteil der Flexibilität durch die Handhabung wieder ausgeglichen. Wir müssen eine Abfrage programmieren, denn hier kann uns das System bei den vielen verschiedenen Bedürfnissen nicht weiterhelfen.

Sehen Sie einen CustomRequester wie ein Window, in dem wir viele Gadgets zu einem Thema untergebracht haben. Mindestens ein Gadget teilt dem Programm durch Auslösen irgendwann mit, daß die Eingabe beendet wurde. Danach können wir die gewonnenen Daten auswerten. Ein Request ist natürlich nur eine Fläche in einem Window, meistens wird aber extra ein neues geöffnet, das alleine für den Requester benutzt wird. So hat der Benutzer optimale Bedienungsmöglichkeiten.

3.5.3.1 Die Requester-Struktur

Für jeden Requester brauchen wir eine Requester-Struktur. Sie enthält ähnliche Informationen wie die Window-Struktur:

```
struct Requester
{
0x00  00  struct Requester *OlderRequest;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  SHORT RelLeft;
0x0E  14  SHORT RelTop;
0x10  16  struct Gadget *ReqGadget;
0x14  20  struct Border *ReqBorder;
0x18  24  struct IntuiText *ReqText;
0x1C  28  USHORT Flags;
0x1E  30  UBYTE BackFill;
0x20  32  struct Layer *ReqLayer;
0x24  36  UBYTE ReqPad1[32];
0x44  68  struct BitMap *ImageBMap;
0x48  72  struct Window *RWindow;
0x4C  76  UBYTE ReqPad2[36];
0x70  112
};
```

Die Variable *OlderRequester* brauchen wir nicht zu initialisieren, denn sie wird von Intuition nur zu Verwaltungszwecken genutzt.

LeftEdge, *TopEdge*, *Width* und *Height* geben die Position und Größe des Requesters im Window an.

Mit *RelLeft* und *RelTop* läßt sich eine ganz nützliche Funktion unterstützen. Setzen Sie dazu im Flags-Wert das Flag POINTREL! Dann wird der Requester nicht mehr an der unter *LeftEdge* und *TopEdge* angegebenen Position ausgegeben, sondern relativ zur Maus-Cursor-Position. Sie können damit erreichen, den Requester so zu positionieren, daß ein bestimmtes Gadget gleich unter dem Maus-Cursor liegt.

Die nächsten drei Pointer sind in die Struktur übernommen, damit Sie Gadgets, Border und Texte verwenden können. Ohne die Gadgets könnte natürlich der ganze Requester nicht bestehen. Border wurde implementiert, damit Sie dem Requester einen "schönen" Rand geben können, und mit IntuiText ist es möglich, noch einen erläuternden Text auszugeben.

Der wichtigste Wert der Struktur, Flags, ist wieder ein Flag. Wie Sie oben unter *RelLeft*, *RelTop* erfahren haben, können Sie hier das Flag POINTREL setzen, damit der Requester relativ zur Maus-Cursor-Position ausgegeben wird. Es existiert aber noch ein zweites Flag. Mit PREDRAWN signalisieren Sie, daß alle Border-, IntuiText- und Image-Strukturen nicht beachtet werden sollen. Die ganze Grafik wird über die Variable *ImageBMap* geholt. Intuition sucht nach einer initialisierten und mit Grafik gefüllten BitMap. Achten Sie darauf, daß diese neue Grafik auch mit den Klickbereichen der Gadgets übereinstimmt, denn diese haben nun keine eigene Grafik mehr!

In diesem Wert setzt Intuition auch eigene Flags. Es sind dies REQOFFWINDOW, was bedeutet, daß sich der Requester außerhalb des Bezugs-Windows befindet, und REQACTIVE, wenn der Requester gerade aktiv ist. Das Flag SYSREQUEST wird gesetzt, wenn es sich um einen System-Requester handelt (siehe dazu AutoRequest und SystemRequest).

Als nächste Variable steht *BackFill* in der Struktur. Hier findet man die Nummer des Zeichenstiftes, mit dem der Hintergrund des Requesters gefüllt werden soll.

ReqLayer wird von Intuition gefüllt und enthält dann einen Zeiger auf den Layer, der den Requester beinhaltet.

ImageBMap ist ein Zeiger, der auf eine BitMap zeigen kann.

RWindow ist eine Variable, die von Intuition verwaltet wird und auf die Window-Struktur zeigt, in dem der Requester untergebracht worden ist.

3.5.3.2 Eine Requester-Struktur einrichten

Da es nicht so einfach ist, für einen Requester genügend Gadgets zusammenzutragen, wollen wir uns mit einem Beispiel helfen, das Sie auf jeden Fall in Ihre eigenen Programme einbauen können. (Dann lohnt sich auch die Arbeit!) Wir werden die File-Select-Box zu diesem Zweck in einen Requester umrüsten. Damit wird die Handhabung im Programm wesentlich einfacher und strukturierter. Außerdem brauchen wir sowieso eine File-Select-Box für den Editor, unser Großprojekt.

```
struct Requester FileSelectBox;

InitRequest(&FileSelectBox);

FileSelectBox.LeftEdge = 2;
FileSelectBox.TopEdge  = 10;
FileSelectBox.Width    = 360;
FileSelectBox.Height   = 200;
FileSelectBox.ReqGadget= &Zurueck;
```

Programmteil 3.7: Requester-Definition: File-Select-Box

Strukturbeschreibung

Die Ausmaße des Requesters sind denen des Windows angepaßt, das vorher für die File-Select-Box verwendet wurde. Durch die Funktion InitRequest() werden zuerst alle Parameter auf NULL gesetzt. Dann ergänzen wir unsere Werte. Der Gadget-Wert zeigt

auf das erste Gadget und führt sich weiter am NextGadget-Pointer jeder Struktur. Dafür müssen Sie die vier schon geschaffenen Gadget-Blöcke auch noch linken. Weil in jedem Gadget eines Requesters auch noch das REQGADGET-Flag gesetzt werden muß, habe ich zur Verdeutlichung alle gelinkten Gadgets hier noch einmal aufgeführt:

```
unsigned char OrdnerBuffer[512] = "df1:";
unsigned char DateiBuffer[31]   = "";
unsigned char UndoBuffer  [512];

USHORT KnobGrafik[] =
   {
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999
   };

struct Image Knob =
   {
   0, 0, 16, 16, 1, &KnobGrafik[0], 1, 0, NULL
   };

struct PropInfo SchieberProp =
{
   FREEVERT,
   0x8000,   0x8000,
   0x0800,   0x1000,
   0,        0,
   0,        0,
   0,        0
};

struct StringInfo OrdnerInfo =
   {
   &OrdnerBuffer[0], &UndoBuffer[0],
   5, 511, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
   };

struct StringInfo DateiInfo =
   {
   &DateiBuffer[0], &UndoBuffer[0],
   0, 31, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
   };

SHORT StringPairs[] =
   {
   0, 0, 248, 0
   };
```

```
SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border StringBorder =
   {
   0, 9, 1, 0, JAM1, 2, StringPairs, NULL
   };

struct Border SelectBorder =
   {
   -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL
   };

struct IntuiText OrdnerText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Ordner:" ,NULL
         };

struct IntuiText DateiText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Datei :" ,NULL
   };

struct IntuiText OKText =
   {
   1, 0, JAM2, 22, 5, NULL, (UBYTE *)"OK" ,NULL
   };

struct IntuiText AbbruchText =
   {
   1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Abbruch" ,NULL
   };

struct IntuiText ZurueckText =
   {
   1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Zurück!" ,NULL
   };

struct Gadget Files[15];
struct Gadget FileGadget =
   {
   NULL,
   80, 0, 240, 8,
   GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
   NULL, NULL, NULL,
   NULL, NULL, 0, NULL
   };

struct Gadget Schieber =
   {
   &Files[0],
```

```
    322, 30, 24, 120,
    GADGHNONE | GADGIMAGE, RELVERIFY, PROPGADGET | REQGADGET,
    (APTR)&Knob, NULL, NULL,
    NULL, (APTR)&SchieberProp, 1, NULL
    };

struct Gadget Ordner =
    {
    &Schieber,
    80, 165, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &OrdnerText,
    NULL, (APTR)&OrdnerInfo, 21, NULL
    };

struct Gadget Datei =
    {
    &Ordner,
    80, 180, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &DateiText,
    NULL, (APTR)&DateiInfo, 22, NULL
    };

struct Gadget OK =
    {
    &Datei,
    10, 40, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET, BOOLGADGET | REQGADGET,
    (APTR)&SelectBorder, NULL, &OKText,
    NULL, NULL, 23, NULL
    };

struct Gadget Abbruch =
    {
    &OK,
    10, 65, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET, BOOLGADGET | REQGADGET,
    (APTR)&SelectBorder, NULL, &AbbruchText,
    NULL, NULL, 24, NULL
    };

struct Gadget Zurueck =
    {
    &Abbruch,
    10, 90, 60, 20,
    GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
    (APTR)&SelectBorder, NULL, &ZurueckText,
    NULL, NULL, 25, NULL
    };
```

Struktur 3.16: Alle Requester-Gadgets

3.5.3.3 Die Requester-Funktionen

Ausgehend von den oben abgedruckten Gadgets und der Requester-Struktur möchte ich ein Programm mit Ihnen entwickeln, das es ermöglicht, jedes eigene Programm dahingehend zu ergänzen, daß Sie auch die File-Select-Box benutzen können.

Wir brauchen dazu unser Standardprogramm 1. Definieren Sie die NewWindow-Struktur wie abgedruckt und global auch die Requester-Struktur. Vorher müssen noch die Gadgets mit ihrem "Anhang" eingefügt werden. Jetzt kann das Programm beim Ablaufen darauf warten, einen Anstoß zu bekommen, daß es den Requester aktivieren soll. Wir aktivieren ihn sofort. Sie können später eine Bedingung dafür einsetzen, wie z.B. die Auswahl eines Menüpunktes.

```
struct NewWindow FirstNewWindow =
   {
   160, 30,                  /* LeftEdge, TopEdge   */
   370, 220,                 /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   CLOSEWINDOW |             /* IDCMP Flags         */
   GADGETUP |
   REQCLEAR,
   WINDOWDEPTH |             /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                     /* First Gadget        */
   NULL,                     /* CheckMark           */
   (UBYTE *)"Requester Test",
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   100, 50,                  /* Min Width, Height   */
   640, 256,                 /* Max Width, Height   */
   WBENCHSCREEN              /* Type                */
   );
```

Struktur 3.17: NewWindow-Struktur für den Requester

Zum Aufrufen des Requesters benutzen wir die einfache Intuition-Funktion Request(). Sie verlangt nur zwei Parameter. Hier ist das allgemeine Format und die Zeile für unser Programm:

```
Success = Request(Requester, Window);
   D0       -240       A0         A1


Request(&FileSelectBox, FirstWindow);
```

Nach diesem Aufruf ist die Datenleitung für die Eingabe des Windows blockiert. Wir empfangen darüber nur noch Signale vom Requester (wenn überhaupt). Das beste ist, man schreibt sich für die Behandlung der Requester-Eingabe eine Unterroutine. So gestalten wir das Programm übersichtlicher. Rufen wir die Routine gleich nach der Aktivierung so auf:

```
FileSelectBox_Request();
```

In der Routine selbst fragen wir die Datenleitung wie schon bekannt ab. Hinzu kommt, daß wir auf das Flag REQCLEAR achten müssen. Es kennzeichnet, daß der Benutzer die Eingabe beendet hat und wieder in den normalen Programmverlauf zurückkehren möchte. Das Signal wird von allen Gadgets ausgelöst, die unter *Activation* das Flag ENDGADGET gesetzt haben. Dies sind in unserem Fall die Boolean-Gadgets "ABBRUCH" und "OK".

```
/****************************************
 *                                      *
 * Funktion: Request-Abfrage            *
 * ==================================== *
 *                                      *
 * Autor:  Datum:      Kommentar:       *
 * ------  ----------  ----------       *
 * Wgb     28.12.1987                   *
 *                                      *
 *                                      *
 ****************************************/

FileSelectBox_Request()
   {
   BOOL Warten = TRUE;
   ULONG MessageClass;

   struct IntuiMessage *message;
   struct Gadget       *GadgetPtr;
   USHORT GadgetID;

   while (Warten)
      {
      if ((message = (struct IntuiMessage *)
```

```
        GetMsg(FirstWindow->UserPort)) == NULL)
        {
        Wait(1L << FirstWindow->UserPort->mp_SigBit);
        printf("SIGNAL!\n");
        continue;
        }
    MessageClass = message->Class;
    GadgetPtr = (struct Gadget *) message->IAddress;
    GadgetID  = GadgetPtr->GadgetID;
    ReplyMsg(message);

    if (MessageClass == REQCLEAR)
        {
        Warten = FALSE;
        continue;
        }

    if (MessageClass == GADGETUP)
    switch (GadgetID)
        {
        case SCHIEBER : printf("Schieber\n");
                        break;

        case ORDNER   : printf("Ordner\n");
                        break;

        case DATEI    : printf("Datei\n");
                        break;

        case ZURUECK  : printf("Zurück\n");
                        break;

        }

    if (GadgetID <= 16 && 1 <= GadgetID)
        {
           /* Tabellenabfrage */
        printf("Gadget-Nr. %d\n", GadgetID);
        }

    }
}
```

Funktion 3.6: File-Select-Box Requester-Behandlung

3.5.4 Requester: wann es der Benutzer will

Selbst über den Service haben sich die Entwickler von Intuition Gedanken gemacht. Die Service-Eigenschaften werden durch die Möglichkeit eines Double-Menu-Requesters wesentlich gestei-

gert. Mit diesem Requester-Typ kann der Benutzer einen Requester aufrufen, wann immer er seine Funktionen nutzen will. Der Aufruf geschieht durch einen doppelten Druck auf die Menütaste der Maus.

Jedesmal, wenn der Programmbenutzer zweimal auf die rechte Maustaste drückt - das muß in einem Zeitintervall von einem Doppelklick liegen - wird ein DMRequest ausgelöst. Diesen Requester können Sie mit SetDMRequest() für ein Window aktivieren. Wenn der Requester angefordert wird, erhält das Programm eine Nachricht über den IDCMP, daß der erste Requester aktiviert wurde. Es kann dann zu einer speziellen Abfrageschleife verzweigt werden. Hier erfolgt die Behandlung wie bei jedem anderen Requester. Wenn das Programm nicht mehr die Möglichkeit eines DMRequest anbieten will, kann es mit ClearDMRequest() alles rückgängig machen.

Ein Programm gilt als besonders bedienungsfreundlich, wenn es einen DMRequest anbietet. So könnte z.B. die File-Select-Box als DMRequester noch wesentlich vielseitiger ausgebaut werden. Dann müßten Sie nicht nur erlauben, einen File-Namen zu selektieren, der einer Funktion übergeben wird, sondern auch noch die Abfrage anbieten, ob gespeichert, geladen oder gelöscht werden soll. Dann kann jede Diskettenoperation ganz einfach über einen doppelten Druck auf die rechte Maustaste ausgelöst werden. Einfacher geht es wirklich nicht!

Für den Programmierer ändert sich die Requester-Behandlung überhaupt nicht. Er kann den Requester auf die oben beschriebene Methode verarbeiten und auswerten. Der wesentliche Unterschied liegt nur im Aufruf. Während konventionelle Requester durch eine Funktion vom Programm aufgerufen werden, ist dieser Requester nur definiert und kann irgendwann auftreten. Deshalb ist es wichtig, die IDCMP-Flags regelmäßig zu überprüfen. Diese können das Auftreten des ersten Requesters abtesten, denn bei den anderen vielleicht vorhandenen Requester-Routinen wird ClearDMRequest() vorher angesprungen.

Ich wünsche noch viel Spaß bei der Requester-Programmierung!

3.6 Alerts, wenn es ernst wird

Alerts sind enge Verwandte der Requester-Familie. Sie stehen an der Spitze der Dringlichkeit, aber leider am Ende der Programmierfreundlichkeit. Beide, Requester und Alert, helfen, eine Entscheidung vom Benutzer zu verlangen. Gehen wir zuerst daran, die Unterschiede herauszufinden.

- Requester sind weiterentwickelte Windows, die ausschließlich zur Informationssammlung dienen. Alerts sind Bildschirmbereiche, die oft letzte Informationen vor einem Absturz bringen.
- Requester sind in ihrer Größe, Position und Gestaltung frei wählbar. Alerts verdrängen sogar immer alle Screens oder unterbinden deren Darstellung. Außerdem können Alerts nur oberhalb der Screens und auf voller Bildschirmbreite dargestellt werden.
- Requester bieten vielfältige Eingabemöglichkeiten, unterstützt durch alle Gadget-Typen. Alerts lassen nur zwei verschiedene Eingaben über die beiden Maustasten zu.
- Requester blockieren nur den Task, vom dem sie aufgerufen wurden. Alerts legen das gesamte Multitasking-Betriebssystem lahm.

Sie wissen jetzt in etwa, welchen Bedeutungsunterschied die beiden Kommunikationsmittel haben. Wir sollten Alerts in unseren Programmen immer nur dann verwenden, wenn es einer sehr wichtigen Klärung bedarf.

Auch Intuition bringt neben den System-Requestern Alerts. Intuition verwaltet nicht nur die bekannten Guru-Meditations, sondern wandelt auch einen AutoRequest in einen Alert um, wenn nicht genügend Speicher zur Verfügung steht, weil Alerts durch die spartanische Ausnutzung der grafischen Möglichkeiten wesentlich weniger Speicher verbrauchen.

Achtung! Bedenken Sie immer, welchen Schock eine Warnmeldung beim Benutzer auslöst und ob das gerechtfertigt ist!

3.6.1 Verwendungszwecke

Ein Alert sollte nur bei ganz wichtigen Situationen angewendet werden. Würde man bei fast jeder Gelegenheit auf einen Alert zurückgreifen, wäre der Schock nicht mehr da!

Das erste wichtige Anwendungsgebiet für Alerts liegt in dem "Nicht-Erreichen" bestimmter Ziele. Da kann einmal der Versuch, eine Library zu öffnen, fehlgeschlagen sein, oder ein Screen konnte nicht geöffnet werden. Wir haben dafür während der Testphase in unseren Funktionen Open_All() und Close_All() einen Text über printf() ausgegeben. Aber nicht immer hat der Benutzer das DOS-Window geöffnet und darüber das Programm gestartet. Wenn dieser "Nachrichtenkanal" nicht offen ist, teilen wir die äußerst wichtige Nachricht über einen Alert mit.

Dieser kann in jedem Fall benutzt werden, wenn z.B. irgendwelche Voraussetzungen nicht gegeben sind: Eine Steckkarte muß für den Betrieb eines Programms vorhanden sein. Oder die Arbeit ist in der vorliegenden Speicherausführung nicht ausführbar.

Alle genannten Gründe könnten einen Alert rechtfertigen, doch überlegen Sie immer, ob nicht ein einfacher AutoRequester genügt. Immerhin legt der Alert den Ablauf des gesamten Systems lahm und wartet darauf, daß Sie reagieren. Dieser Alert muß in seiner Wichtigkeit also alle anderen Prozesse übertreffen!

3.6.2 Aufbau und Einstellungen für einen Alert

Die Funktion DisplayAlert() erwartet drei Daten, anhand derer der Alert aufgebaut wird.

Zuerst übergeben wir die Alert-Nummer, aus der Intuition abliest, um welchen Alert-Typ es sich handelt. Man unterscheidet grundsätzlich zwei Typen, die am höchstwertigen Byte der LONG-Variable zu erkennen sind. Ein RECOVERY_ALERT - die Funktion kehrt nach der Darstellung wieder zurück zum

Programm - wird über 0 gekennzeichnet, einen DEADEND_ ALERT - nach der Darstellung erfolgt ein Reset - erkennt man am Byte 8.

Weiterhin übergeben wir der Funktion die Höhe unseres Alerts in Bildschirmzeilen. Um diese werden später alle anderen Screens nach unten verschoben.

Das wichtigste Datum ist der Pointer auf einen Alert-String, der aus einem String besteht, der noch um einige Informationen ergänzt wurde:

Der Alert-Text setzt sich zusammen aus vielen Alert-Strings, die über eine Endkennung verbunden sind. Am Anfang enthält jeder Alert-String eine Positionsangabe in Pixel. Darauf folgt der Text, der das Format eines normalen Strings hat. Abschließend finden wir ein Byte, das, wenn es ungleich NULL ist, einen weiteren String "einläutet". Ansonsten ist der Alert-Text damit abgeschlossen.

Leider wird die doch etwas komplizierte Verwaltung eines Alerts wenig von Intuition unterstützt. Man hat leider die Struktur im Include-File nicht eingebunden, so daß ich sie hier nachreichen möchte.

Zuerst richten wir eine Struktur für die Alert-Strings ein:

```
struct AlertMessage
{
0x00  00  SHORT LeftEdge;
0x02  02  BYTE TopEdge;
0x03  03  char AlertText[50];
0x53  83  BYTE Flag;
0x54  84
};
```

Das einzige Problem, das bei der Definition der Struktur auftritt, ist die Länge des Textes. Wir müssen dafür einen Standardwert nehmen. Sie können ihn aber nach Belieben ändern. Achten Sie nur darauf, daß der Alert in TOPAZ_SIXTY ausgegeben wird und damit maximal nur 60 Zeichen in der Zeile erlaubt sind.

Aufgrund der AlertMessage-Struktur können wir jetzt unseren eigenen Alert zusammenstellen. Dafür richten wir uns ein Feld aus dieser Struktur ein:

```
#define NOEND 0xFF
#define END   0x00

struct AlertMessage UserAlert[] =
{
   50, 24, "  Dies ist der erste Alert meines Lebens!!!  ", NOEND,
   50, 34, "  ----------------------------------------  ", NOEND,
   50, 44, "                                            ", NOEND,
   50, 54, " <Left Button>  CANCEL  <Right Button>  RETRY", END
};
```

Struktur 3.18: AlertMessage Test

Denken Sie daran, daß jeder String innerhalb des Feldes mit der Kennung ergänzt werden muß. Um eine möglichst einfache Handhabung zu gewährleisten, habe ich dafür zwei Namen definiert. Mit NOEND kennzeichnen Sie, daß ein weiterer Text folgen wird, und mit END das Ende.

Nachdem unsere erste Alert-Struktur definiert ist, können wir sie mit der DisplayAlert()-Funktion aufrufen. Diese hat folgendes Format:

```
Response = DisplayAlert(AlertNumber, String, Height);
   D0           -90         D0         A0      A1
```

In unserem Test-Programm setzten wir sie mit den bekannten Werten ein:

```
Antwort = DisplayAlert(RECOVERY_ALERT, &UserAlert, 50L);
```

Das ganze Programm sieht dann hoffentlich so aus:

```
/*************************************
 *                                   *
 * Programm: Alert austesten         *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     29.12.1987  erster Alert  *
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;

struct AlertMessage
{
   SHORT LeftEdge;
   BYTE TopEdge;
   char AlertText[50];
   BYTE Flag;
};

#define NOEND 0xFF
#define END   0x00

struct AlertMessage UserAlert[] =
{
   50, 24, "  Dies ist der erste Alert meines Lebens!!!  ", NOEND,
   50, 34, "  ----------------------------------------  ", NOEND,
   50, 44, "                                            ", NOEND,
   50, 54, " <Left Button>  CANCEL  <Right Button>  RETRY", END
};

main()
   {
   BOOL Antwort;

   Open_All();

   Antwort = DisplayAlert(RECOVERY_ALERT, &UserAlert, 50L);

   if (Antwort == TRUE)
      printf("Linke Maustaste wurde gedrückt!\n");
   else
      printf("Rechte Maustaste wurde gedrückt!\n");

   Close_All();
   }
```

```
/****************************************
 *                                      *
 * Funktion: Alles Nötige öffnen        *
 * ==================================== *
 *                                      *
 * Autor:  Datum:      Kommentar:       *
 * ------  ----------  ----------       *
 * Wgb     16.10.1987  nur Intuition    *
 *                                      *
 ****************************************/

Open_All()
   {
   void          *OpenLibrary();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      Close_All();
      exit(FALSE);
      }
   }


/****************************************
 *                                      *
 * Funktion: Alles Geöffnete schließen  *
 * ==================================== *
 *                                      *
 * Autor:  Datum:      Kommentar:       *
 * ------  ----------  ----------       *
 * Wgb     16.10.1987  nur Intuition    *
 *                                      *
 ****************************************/

Close_All()
   {
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

Programm 3.7: Alert

Programmbeschreibung

Das Programm bereitet am Anfang alle nötigen Strukturen für den Funktionsaufruf vor. Dazu gehört es auch, die beiden Include-Files einzubinden, denn ohne deren Definitionen ist die Arbeit nicht möglich, wenn auch nur auf die Variablentypen und Alert-Types zurückgegriffen wird.

Ist der Alert-String mit allen Koordinaten und Kennungen definiert, kann das Hauptprogramm die Funktion Open_All() aufrufen, damit der Zugriff zu Intuition gesichert ist. Danach wird DisplayAlert() gestartet und der Rückgabewert verarbeitet. Zum Schluß kann durch Close_All() alles wieder geschlossen werden.

Hinweis: Um die Wirkung eines DEADEND_ALERT einmal auszuprobieren, sollten Sie ruhig den Alert-Type ändern. Bedenken Sie nur, daß danach die Kontrolle nicht mehr an das Programm übergeben wird, sondern an die Reset-Routine.

3.6.3 Alerts für Programmprojekte

Einen Alert in ein Programm einzubauen bedeutet für den Programmierer Arbeit, die er sich ebensogut sparen könnte. Man versucht lediglich, die Bedienung eindeutiger und interessanter zu gestalten.

Es empfiehlt sich deshalb, erst nach Abschluß aller Arbeiten an einem Programm noch zusätzlich Alerts einzubauen. Diese ersetzen dann meist Fehlermeldungen, die in der Testphase immer in das DOS-Window ausgegeben wurden. Deren Programmierung ist zwar sehr einfach, doch entbehrt sie jeder Professionalität!

Für das nachträgliche Einbinden der Alerts empfehle ich folgende Systematik: Ist ihr Programm fertiggestellt und stehen alle Punkte im Programmtext fest, an denen eine einfache Abbruch-Meldung durch einen Alert ersetzt werden soll, müssen wir zuerst die Texte entwickeln, die später in bestimmten Situationen ausgegeben werden sollen.

Dafür empfehle ich ein allgemeines Aussehen (Design) aller Alerts und vielleicht einen "Multi-Alert", in dem nur eine Spezifikation eingesetzt wird. Aber das nur am Rande. Alle Alert-Texte werden dann gesondert mit der Strukturdefinition in ein File geschrieben. Dieses File wird als letztes "included":

```
#include <exec/devices.h>
#include <graphics/gfxbase.h>

...

#includes "Disk:Verzeichnis/Alert"
```

In dem Alert-File finden wir z.B. für die Textverarbeitung die folgenden Fehlermeldungen:

```
/*****************************************
 *                                       *
 * include : Alert-Struktur mit Definition *
 * ===================================== *
 *                                       *
 * Autor:  Datum:      Kommentar:        *
 * ------  ----------  ----------        *
 * Wgb     29.12.1987  für Textverarbeitung*
 *                                       *
 *****************************************/


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


struct AlertMessage
{
   SHORT LeftEdge;
   BYTE TopEdge;
   char AlertText[50];
   BYTE Flag;
};

#define NOEND 0xFF
#define END   0x00


struct AlertMessage OutMem[] =
{
  50, 24, "  Es stand nicht genügend Speicherplatz zur   ", NOEND,
  50, 34, "  Verfügung! Bitte machen Sie Speicher frei!  ", NOEND,
  50, 44, "  Starten Sie danach das Programm neu!        ", NOEND,
  50, 54, "                                              ", NOEND,
  50, 64, "          Maustaste für Bestätigung           ", END
};

#define OutMemSize 80L


struct AlertMessage NoLibrary[] =
{
```

```
  50, 24, "  Die graphics.library konnte nicht geöffnet  ", NOEND,
  50, 34, "  werden!    Das Programm bricht deshalb ab!  ", NOEND,
  50, 44, "  Sorgen Sie dafür, daß diese Library auf     ", NOEND,
  50, 54, "  der Workbench zu finden ist!                ", NOEND,
  50, 64, "                                              ", NOEND,
  50, 74, "            Maustaste für Bestätigung         ", END
};

#define NoLibraySize 90L


struct AlertMessage Ende[] =
{
  50, 24, "   Lebenswichtige Funktionen können nicht     ", NOEND,
  50, 34, "   erreicht werden!                           ", NOEND,
  50, 44, "  Crash!  Crash! ..... uff                    ", NOEND,
  50, 54, "                                              ", NOEND,
  50, 64, "          Maustaste für Bestätigung           ", END
};

#define EndeSize 80L
```

Programm 3.8: Alert-Texte

Für den Aufruf von DisplayAlert() habe ich zusätzlich die benötigte Höhe jedes Alerts in einem Define abgelegt. Im Programmtext werden dann folgende Zeilen

```
if (Window == NULL)
   {
   printf("Leider kein Window zu öffnen!\n");
   Close_All();
   exit(FALSE);
   }
```

durch diese ersetzt:

```
if (Window == NULL)
   {
   DisplayAlert(RECOVERY_ALERT, &NoWindow, NoWindowSize);
   Close_All();
   exit(FALSE);
   }
```

3.6.4 Auswertung der Guru-Meditation

Auch wenn Sie noch keine Alerts selber programmiert haben, so haben Sie sicherlich schon Bekanntschaft mit den Gurus gemacht. Doch im allgemeinen nimmt man die Meldung meist nur am Rande wahr, da ja sowieso nichts dagegen unternommen werden kann. Damit aber wird der Sinn einer Guru-Meditation verdreht! Der Amiga könnte ja auch sang- und klanglos abstürzen, ohne sich noch einmal zu melden. Die Gurus wurden aber extra eingeführt, damit der Anwender noch die Nachricht erhält, warum das Betriebssystem einen Neustart auslösen mußte.

Das ist besonders wichtig, wenn man selber programmiert. Denn nur so weiß man, warum das eigene Programm in der Testphase fehlerhaft lief und wo die Fehlerquelle zu suchen ist. Das einzige Problem der Guru-Meditation ist aber, daß wir immer nur den gleichen Text erhalten und eine Nummer, mit der keiner etwas anfangen kann. Das soll sich jetzt ändern. Aus den folgenden Tabellen werden Sie alle Informationen schöpfen können, die Ihnen die Decodierung jedes Gurus ermöglichen.

Die Guru-Nummer setzt sich aus mehreren Segmenten zusammen. Wir können sie deshalb in ein allgemeines Format bringen:

```
Guru Meditation : TTSSFFGGGG.AAAAAAAA
```

Die Buchstaben haben folgende Bedeutung:

TT	Alert_Type
SS	Systemklasse
FF	Fehlerklasse
GGGG	Genauere Beschreibung.
AAAAAAAA	Adresse des fehlerauslösenden Tasks.

Für TT gibt es nur zwei Möglichkeiten. Sie kennen diese Werte unter der Bezeichnung Alert_Type. Entweder es handelt sich um einen DEADEND oder einen RECOVERY_ALERT. Nur der letzte Typ kehrt wieder zurück!

Alert_Types

00000000	RECOVERY_ALERT
80000000	DEADEND_ALERT

Tabelle 3.14: Alert-Types

Hier noch eine Information zur Schreibweise. Bei allen Werten schreibe ich den ganzen Term auf. Bei Kombinationen werden diese mit "oder" verknüpft!

Als nächstes steht die Systemklasse zur Diskussion. Hier finden wir die Bezeichnung des "Subsystems", in dem der Fehler aufgetreten ist.

SUBSYSTEM_CODE

00000000	68000er

LIBRARYS

01000000	exec.library
02000000	graphics.library
03000000	layers.library
04000000	intuition.library
05000000	math.library
06000000	clist.library
07000000	dos.library
08000000	ram.library
09000000	icon.library
0A000000	expansion.library

DEVICES

10000000	audio.device
11000000	console.device
12000000	gameport.device
13000000	keyboard.device
14000000	trackdisk.device
15000000	timer.device

RESOURCES

20000000	CIA
21000000	Disk
22000000	Misc

MISC

30000000	Bootstrap
31000000	Workbench
32000000	DiskCopy

Tabellen 3.15: Subsystem-Codes

Beachten Sie die Fehlermeldung am Anfang der Subsystem-Liste! Hier meldet sich der Prozessor selbst zu Wort! Wenn dies der Fall ist, so handelt es sich um einen Prozessor-Trap. Die restlichen Stellen der Guru-Meditation erhalten dadurch eine andere Bedeutung:

TRAP_CODES

00000002	Daten- oder Adreßbus-Fehler beim Takten.
00000003	Adressierungsfehler (ungerade Adresse).
00000004	Illegal Instruktion.
00000005	Division durch Null.
00000006	CHR Instruktion.
00000007	TRAPV Instruktion.
00000008	Privileg-Verletzung.
00000009	Einzelschrittmodus.
0000000A	Line A Emulator (OpCode 1010).
0000000B	Line F Emulator (OpCode 1111).

Tabelle 3.16: Trap-Codes

War es aber ein Fehler nach der Subsystem-Liste, dann können wir ihn weiter spezifizieren. Über "Fehlerklasse" erfahren wir genaueres zum Grund des Fehlers:

ERROR_CLASS

00010000	Nicht genügend Speicher.
00020000	Library konnte nicht aufgebaut werden.
00030000	Library konnte nicht geöffnet werden.
00040000	Device konnte nicht geöffnet werden.
00050000	Keine Reaktion der Hardware.
00060000	I/O-Fehler.
00070000	I/O nicht vorhanden.

Tabelle 3.17: Error-Classes

Über die letzten vier Stellen der Fehlernummer bekommen wir noch detailliertere Informationen. Diese sind allerdings auf die Subsysteme spezifiziert.

exec.library

01000000	Prüfsummenfehler bei Ausnahmen der CPU.
81000002	Prüfsummenfehler der Startadresse.
81000003	Prüfsummenfehler einer Library.
81000004	Nicht genug Speicher für Library.
81000005	Fehlerhafter Speicherlisteneintrag.
81000006	Nicht genug Speicher für Interrupt.
81000007	Zeiger-Fehler.
81000008	Fehlerhafte Semaphore.
81000009	Speicherbereich wurde zum zweiten Mal freigegeben.
8100000A	Zeiger-Fehler bei Ausnahme.

graphics.library

82010001	Nicht genug Speicher für die Copper-Liste.
82010002	Nicht genug Speicher für die Copper-Instruktions-Liste.
82010003	Die Copper-Liste ist vollständig belegt.
82010004	Aufteilungs-Fehler in der Copper-Liste.
82010005	Nicht genug Speicher für den Copper-Listen-Kopf.
82010006	Nicht genug Speicher für "long frame".
82010007	Nicht genug Speicher für "short frame".
82010008	Nicht genug Speicher für die Fill-Routine.
82010009	Nicht genug Speicher für die Text-Routine.
8201000A	Nicht genug Speicher für die BlitterBitMap.
8201000B	Falscher Speicherbereich.
82010030	Fehler während des Einrichtens eines ViewPorts.
82011234	GfxNoLCM (kein Zwischenspeicher vorhanden).

layers.library

03000001	Nicht genug Speicher für Layers.

intuition.library

84000000	Gadget-Typ ist unbekannt.
04000001	Typenfehler beim AN_Gadget.
84010002	Nicht genug Speicher für Erstellung eines Ports.
04010003	Nicht genug Speicher für ein Menü.
04010004	Nicht genug Speicher für ein Sub-Menü.
84010005	Nicht genug Speicher für die Menü-Leiste.
84000006	Falsche Position der Menü-Leiste.
84010007	Nicht genug Speicher für OpenScreen().
84010008	Nicht genug Speicher für Erstellung eines RastPorts.

84000009	Unbekannter oder fehlerhafter SCREEN_TYPE.
8401000A	Nicht genug Speicher für Gadget.
8401000B	Nicht genug Speicher für Window.
8400000C	Statusregister hat falschen Status beim Öffnen d. Library.
8400000D	Falsche Message über den IDCMP.
8400000E	Nicht genug Speicher für Message-Stack.
8400000F	Nicht genug Speicher für Console.Device.

dos.library

07000001	Nicht genug Speicher beim StartUp.
07000002	Der Task wurde nicht beendet.
07000003	Qpkt-Fehler ist aufgetreten.
07000004	Datenpaket wurde nicht erwartet.
07000005	Freier Zeiger ist nicht erreichbar.
07000006	Fehlerhafte Daten eines Disk-Blocks.
07000007	Die BitMap ist zerstört.
07000008	Key wurde schon freigegeben.
07000009	Die Prüfsumme ist fehlerhaft.
0700000A	Disk Error.
0700000B	Key außerhalb des erlaubten Bereichs.
0700000C	Falsches Überschreiben.

ram.library

08000001	Fehlerhafter Eintrag in der Verwaltungsliste.

expansion.library

0A000001	Fehler bei Erweiterung der Hardware.

trackdisk.device

14000001	Fehler während des Suchens.
14000002	Fehler beim Timer-Impuls: Verzögerung.

timer.device

15000001	Fehler beim Zugriff auf das Device.
15000002	Fehler bei Zeitkoordinierung: Netzschwankungen.

disk.resource

21000001	Eingelegte Diskette wurde nicht erkannt.
21000002	Unterbrechung, da kein Laufwerk angeschlossen ist.

bootstrap

30000001	Fehler beim Auswerten der Boot-Daten.

Tabelle 3.18: Fehlermöglichkeiten

3.7 Abfrage des IDCMP

Ein Programm kann erst richtig "leben", wenn es mit seinem Benutzer kommuniziert. Wir wollen kleine Demonstrationsprogramme schreiben und brauchen deshalb immer wieder Entscheidungen und Eingaben während des Programmablaufs. Hier stellt sich zunächst die Frage:

"Wie bekomme ich die Informationen vom Benutzer?"

Als Antwort bietet das Amiga-System mehrere Ein- und Ausgabegeräte und Datenleitungen. Dies sind im einzelnen das Audio.Device, Timer.Device, Trackdisk.Device, Console.Device, Input.Device, Keyboard.Device, Gameport.Device, Narrator.Device, Serial.Device, Parallel.Device, Printer.Device und das Clipboard.Device. Nicht alle von ihnen lassen Ein- und Ausgabe gleichzeitig zu.

So können wir über das Audio.Device, Narrator.Device und Printer.Device nur Daten senden. Umgekehrt erlauben es das Timer.Device, Keyboard.Device und das Gameport.Device sinnvollerweise nur, Daten zu empfangen. Mit den anderen Devices können wir in beiden Richtungen Daten austauschen. Erst das ist eine richtige Kommunikation.

Allerdings haben alle diese Ein- und Ausgabegeräte einen Nachteil. Wollen Sie nämlich mehrere Daten verschiedenen Typs senden und empfangen, so brauchen Sie mehr als ein Device.

Manche Devices bieten auf bestimmten Gebieten Vorteile, auf anderen aber starke Nachteile. Die Folge davon ist es, daß im Programm etliche Devices geöffnet werden, aber der Programmierer am Ende einfach die Übersicht verliert.

Aus diesem Grund haben sich die Entwickler des Amiga Gedanken gemacht, und herausgekommen ist der IDCMP! Er ist eine Synthese aus Input.Device, Gameport.Device, Timer.Device, Trackdisk.Device und Console.Device mit Verknüpfung der Intuition-Einrichtungen.

Trackdisk.Device und Console.Device mit Verknüpfung der Intuition-Einrichtungen.

Der IDCMP ist gedacht zur Datenverarbeitung, Filterung und Bereitstellung. Über das ihm zugeordnete Window können Sie einstellen, welche Daten Sie erhalten wollen und welche er in ein passendes Format umsetzen soll. Oder möchten Sie die Daten lieber unverändert lassen? Auch das erlaubt der IDCMP. Die Ausgabe wurde bei der Entwicklung nicht beachtet. Sie läuft über die Grafik-Unterstützung von Intuition. Möchten Sie noch mehr Flexibilität, dann weichen Sie aus auf die "graphics.library". Es hat sich aber herausgestellt, daß Intuition im Normalfall vollkommen ausreichend ist. Alle Eingaben lassen sich über IDCMP empfangen, und Ausgaben sind mit den Grafik-Strukturen einfach und komfortabel möglich.

Lassen Sie sich entführen in das Reich des IDCMP, und bewundern Sie die Möglichkeiten und die Vielfalt der Eingaben.

3.7.1 Vorbereitung und Nachrichtenempfang

Der IDCMP steht dem Programmierer nicht selbstverständlich zur Verfügung. Intuition muß erkennen, daß eine Notwendigkeit für den Datenempfang besteht.

Als Grundvoraussetzung für den IDCMP benötigen wir ein Window. Ohne Window ist es nicht möglich, Daten über Intuition zu empfangen. Erinnern Sie sich jetzt bitte an die NewWindow-Struktur. Unter dem Eintrag *IDCMPFlags* besteht die Verbindung zu unserem "Intuition Direct Communication Message Ports". Wenn wir dort mit einem Flag kennzeichnen, daß wir an Informationen interessiert sind, richtet Intuition für dieses Window einen Port ein. Um das Gebiet abzustecken, für das wir den Port benutzen können, ist es zuerst wichtig zu wissen, welche Flags gesetzt werden können. Diese unterteilen sich in sechs Gruppen.

In der ersten Gruppe finden wir die wichtigsten Flags. Sie beziehen sich auf das Window des Message-Ports:

SIZEVERIFY

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der Benutzer versucht, die Größe des Windows zu verändern. Das kann wichtig sein, wenn das Programm gerade zeichnet und diese Arbeit beendet sein muß, bevor die Größe verändert wird. Dann kann das Window nicht eher in der Größe geändert werden, bevor auf die Nachricht geantwortet worden ist.

NEWSIZE

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn die Größe des Windows verändert wurde. Die Werte können aus der Window-Struktur ausgelesen werden.

REFRESHWINDOW

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der Inhalt des Windows neu gezeichnet werden muß. Es ist nur bei den Window-Typen SIMPLE_REFRESH und SMART_REFRESH sinnvoll. SMART_REFRESH-Windows benötigen das Refresh-Flag auch nur dann, wenn sie mit einem Sizing-Gadget versehen sind.

ACTIVATEWINDOW

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn das Window aktiviert wurde.

INACTIVATEWINDOW

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn ein anderes Window aktiviert wurde.

Die zweite Gruppe bearbeitet das Problem der Gadget-Handhabung. Diese Flags haben Sie schon bei unseren Gadget-Abfragen kennengelernt:

GADGETDOWN

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn ein Gadget angeklickt wurde, das vom Typ GADGIMMEDIATE ist.

GADGETUP

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn ein Gadget angeklickt wurde, das vom Typ RELVERIFY ist.

CLOSEWINDOW

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn vom Benutzer das Close-Gadget betätigt wird. Intuition schließt das Fenster nicht!

In der dritten Gruppe finden wir die erweiternden Flags zu den Gadgets wieder. Hier steht alles, was ein Requester auslösen kann:

REQSET

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der erste Requester im Window geöffnet wird.

REQCLEAR

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der letzte Requester im Window geschlossen wird.

REQVERIFY

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn Intuition versucht, den ersten Requester zu öffnen. Intuition wartet dann so lange, bis auf die Nachricht geantwortet wird. So können Sie alle Ausgaben fertigstellen, bevor der Requester erscheint. Werden weitere Requester im Window geöffnet, so gibt Intuition keine Nachricht. Es wird davon ausgegangen, daß während der Requester-Bearbeitung die Ausgabe gesichert ist.

Die vierte Gruppe behandelt die Menü-Auswertung. Wenn Ihnen die Programmierung der Menüs noch nicht bekannt ist, so lesen Sie im nächsten Kapitel mehr dazu.

MENUPICK

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der Benutzer den Menü-Button gedrückt und wieder losgelassen hat. Es ist nicht gesichert, daß ein Menüpunkt ausgewählt wurde!

MENUVERIFY

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn versucht wird, die Menüs zu aktivieren. So lange, wie nicht über Reply() geantwortet wird, können die Menüs nicht dargestellt werden.

Für die Mausabfrage sind die Flags der fünften Gruppe vorgesehen:

MOUSEBUTTONS

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn ein Maus-Button gedrückt oder losgelassen wurde. Beachten Sie, daß keine Nachrichten gesendet werden, wenn der linke Button über einem Gadget gedrückt oder wenn der rechte Button zur Handhabung der Menüs verwendet wird.

MOUSEMOVE

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der Benutzer die Maus bewegt. Beachten Sie, daß dadurch sehr viele Nachrichten entstehen können, die auch noch alle verarbeitet werden müssen! Zusätzlich muß auch das REPORTMOUSE-Flag gesetzt sein oder, bei einem Gadget, das FOLLOWMOUSE-Flag.

DELTAMOVE

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn der Benutzer die Maus bewegt. Im Unterschied zum obigen Flag erhalten wir aber nicht die absoluten Positionswerte, sondern die Differenz zur vorhergehenden Position.

In der letzten, der sechsten, Gruppe unserer IDCMP-Flags finden wir die übrigen, die sich nicht einordnen ließen. Das heißt aber nicht, daß diese Flags weniger brauchbar sind. Sie erfüllen

sogar Aufgaben, die manches Programm erst in die Klasse "professionell" erheben.

INTUITICKS

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn eine zehntel Sekunde vergangen ist. (Dies ist die Einbindung des Timer.Device.) Man erhält nur weitere INTUITICK-Nachrichten, wenn auf die letzte geantwortet wurde. So verhindert IDCMP eine "Nachrichtenschwemme"!

NEWPREFS

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn die Preferences-Werte geändert wurden. Das ist z.B. wichtig, wenn sich das Programm auf die Farbeinstellungen oder den Drucker verläßt. Jedes Window, in dem dieses Flag gesetzt wurde, erhält eine Nachricht. Allerdings kann ein Programm, das die Preferences-Daten verändert, bestimmen, ob es dies den anderen mitteilen will.

DISKINSERTED

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn eine Diskette in ein Laufwerk eingelegt wurde. Jedes Window, in dem dieses Flag gesetzt wurde, erhält eine Nachricht. (Dies ist die Einbindung des Trackdisk.Device.)

DISREMOVED

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn eine Diskette aus dem Laufwerk genommen wurde. Jedes Window, in dem dieses Flag gesetzt wurde, erhält diese Nachricht.

RAWKEY

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn eine Taste der Tastatur gedrückt wurde. Der Tastatur-Code ist aber "unbehandelt", d.h. daß Sie keinen nach der Tasturtabelle umgewandelten Code empfangen. (Dies ist die Einbindung des Input.Device.)

VANILLAKEY

Ist dieses Flag gesetzt, so erhält das Programm immer dann eine Nachricht, wenn eine Taste gedrückt wurde. Der Tastur-Code ist nach der Tastaturtabelle verarbeitet und umgewandelt. (Dies ist die Einbindung des Console.Device.)

Abschließend finden Sie hier alle Flags mit ihren Hex-Werten:

Flag-Name	Hex-Wert	Bemerkung
SIZEVERIFY	0x00000001L	Window-Flags
NEWSIZE	0x00000002L	
REFRESHWINDOW	0x00000004L	
ACTIVEWINDOW	0x00040000L	
INACTIVEWINDOW	0x00080000L	
GADGETDOWN	0x00000020L	Gadget-Flags
GADGETUP	0x00000040L	
CLOSEWINDOW	0x00000200L	
REQSET	0x00000080L	Requester-Flags
REQCLEAR	0x00001000L	
REQVERIFY	0x00000800L	
MENUPICK	0x00000100L	Menü-Flags
MENUVERIFY	0x00002000L	
MOUSEBUTTONS	0x00000008L	Maus-Flags
MOUSEMOVE	0x00000010L	
DELTAMOVE	0x00100000L	
INTUITICKS	0x00400000L	Timer.Device
NEWPREFS	0x00004000L	Preferences-Änderung
DISKINSERTED	0x00008000L	Trackdisk.Device
DISKREMOVED	0x00010000L	
RAWKEY	0x00000400L	Input.Device
VANILLAKEY	0x00200000L	Console.Device
WBENCHMESSAGE	0x00020000L	Workbench-Nachricht
LONELYMESSAGE	0x80000000L	Keine Intuition-IDCMP-Nachricht

Tabelle 3.19: Alle IDCMP-Flags

Haben Sie sich entschieden, zu welchen Gebieten Sie Nachrichten und Informationen empfangen möchten, so gibt es zwei Wege, über die Sie es dem System mitteilen können. Der erste und auch der einfachste: Sie setzen gleich in der NewWindow-Struktur die entsprechenden Flags ein. Intuition richtet dann sofort den IDCMP ein, und Sie brauchen sich nur noch um die Abfrage zu kümmern. Der zweite Weg: Sie wollen Nachrichten empfangen und haben schon ein Window geöffnet, ohne aber einen Port einzurichten. In diesem Fall nutzen Sie die Funktion ModifyIDCMP(). Übergeben Sie ihr einfach den Window-Pointer und alle zu setzenden IDCMP-Flags. Der Befehl hat folgendes Format:

```
ModifyIDCMP(Window, IDCMPFlags);
   -150      A0        D0
```

Danach wird entweder ein neuer Port eingerichtet oder ein bestehender geschlossen. Die dritte Möglichkeit wandelt nur die vorhandenen Flags in die neuen Flag-Werte um.

Zum zweiten Weg zählen auch Funktionen wie z.B. ReportMouse(), die den IDCMP beeinflussen. Auch hier geschieht das gleiche, aber nur in bezug auf ein einziges Flag.

3.7.1.1 Die IntuiMessage-Struktur

Nachdem wir ausreichend geklärt haben, zu welchen Situationen uns der IDCMP Nachrichten liefern kann, ist es jetzt an der Zeit, sich die Nachrichtenübermittlung genauer anzusehen.

In der Window-Struktur finden wir an Byte 90 einen Zeiger auf die IntuiMessage-Struktur mit dem Namen *MessageKey*. Außerdem finden wir schon bei Byte 82 einen weiteren Zeiger auf einen MessagePort mit dem Namen *UserPort* und gleich darauf den MessagePort *WindowPort*. Zusammen mit diesem drei MessagePorts lassen sie die Eingaben verarbeiten.

Sehen wir uns zuerst die IntuiMessage-Struktur an:

```
struct IntuiMessage
{
0x00  00  struct Message ExecMessage;
      0x00  00  struct Node mn_Node;
            0x00  00  struct Node *ln_Succ;
            0x04  04  struct Node *ln_Pred;
            0x08  08  UBYTE ln_Type;
            0x09  09  BYTE ln_Pri;
            0x0A  10  char *ln_Name;
            0x0E  14
      0x0E  14  struct MsgPort *mn_ReplyPort;
      0x12  18  UWORD mn_Length;
      0x14  20
0x14  20  ULONG Class;
0x18  24  USHORT Code;
0x1A  26  USHORT Qualifier;
0x1C  28  APTR IAddress;
0x20  32  SHORT MouseX;
0x22  34  SHORT MouseY;
0x24  36  ULONG Seconds;
0x28  40  ULONG Micros;
0x2C  44  struct Window *IDCMPWindow;
0x30  48  struct IntuiMessage *SpecialLink;
0x34  52
      };
```

Strukturbeschreibung

Die Struktur enthält am Kopf eine eingebundene Message-Struktur. Diese wird für den Datentransport der Messages von Exec gebraucht. Interessant wird es erst bei den anderen Werten, die zusätzlich zu der auslösenden Information noch wichtige Daten beinhalten können. Das ist vom Flag abhängig.

ExecMessage wird hauptsächlich von Exec dazu gebracht, die Message in das System einzubinden.

Class ist eine ULONG-Variable, die die Art der gesendeten Daten kennzeichnet. Die Bits sind mit den möglichen der Window-IDCMP-Variablen identisch und können zur Identifizierung der Nachricht gebraucht werden.

Code enthält Daten, die vom auslösenden Flag abhängen. Der Wert hat immer andere Bedeutungen.

Qualifier ist eine Kopie der ie_Qualifier-Variablen, die vom Input.Device übertragen wird.

MouseX, MouseY enthalten die Mauskoordinaten in bezug auf das Window zum Zeitpunkt der Nachrichtenübermittlung.

Seconds, Micros ist eine Kopie der System-Uhrzeit des Amiga zum Zeitpunkt der Nachrichtenübermittlung.

IAddress ist ein Zeiger auf die Struktur, in der man die Nachricht findet. Das kann z.B. eine Gadget-Struktur sein, von der der Impuls ausgelöst wurde.

IDCMPWindow ist ein Zeiger auf das Window, von dem die Nachricht stammt.

SpecialLink ist eine Variable, die vom System benutzt wird. Wie Sie sehen, enthält die IntuiMessage-Struktur alle Daten, die für die Weiterverarbeitung wichtig sein können. Wir finden hier hauptsächlich auf Intuition bezogene Daten, deren Auswertung aber stark vom erhaltenen Impuls abhängig ist.

3.7.1.2 Die MsgPort-Strukturen

Der Auslöser, über den wir den Empfang der Nachricht signalisiert bekommen, liegt aber nicht in der IntuiMessage-Struktur. Dafür haben wir die MsgPort-Struktur mit dem Namen User-Port. Über das Setzen eines Bits in der Variablen mp_SigBit können wir auf eine Nachricht warten. Die Struktur enthält außerdem noch die folgenden Elemente:

```
struct MsgPort
{
0x00  00  struct Node mp_Node;
    0x00  00  struct Node *ln_Succ;
    0x04  04  struct Node *ln_Pred;
    0x08  08  UBYTE ln_Type;
    0x09  09  BYTE ln_Pri;
    0x0A  10  char *ln_Name;
    0x0E  14
0x0E  14  UBYTE mp_Flags;
0x0F  15  UBYTE mp_SigBit;
0x10  16  struct Task *mp_SigTask;
0x14  20  struct List mp_MsgList;
0x22  34
};
```

Strukturbeschreibung
In die Struktur ist am Kopf eine Node-Struktur eingebunden, die für die Erkennung des Ports hilfreich ist.

mp_Flags kennzeichnet die Art der Message:

PA_SIGNAL	wird gesetzt, wenn eine Nachricht ankommt.
PA_SOFTINT	löst einen Software-Interrupt aus.
PA_IGNORE	kennzeichnet Nachrichten, die nicht beachtet werden sollen.

mp_SigBit ist die uns betreffende Variable. Das gesetzte Bit stellt die Ankunft eines Task-Signals für uns dar.

mp_SigTask ist ein Pointer auf den Task, der das Signal ausgelöst hat.

mp_MsgList ist der Kopf einer Liste aller Messages, die zu diesem Port gesendet wurden.

Sicher werden Sie etwas verwirrt sein, wenn Sie dies alles gelesen haben. "Wir wollten doch nur ganz einfach den IDCMP abfragen!", werden Sie sagen.

Sicherlich ist dieser MsgPort nicht ganz einfach, da er voll ins interne Signalsystem des Amiga eingreift. Aber für den IDCMP müssen Sie eigentlich nur die Variable *mp_SigBit* verstehen.

Mit ihr werden wir später darauf warten, daß eine Nachricht für uns ankommt. Es ist nicht einmal wichtig, was dort genau steht.

3.7.1.3 Das Warten auf eine Nachricht

Gehen wir davon aus, daß Intuition für uns MessagePort und IntuiMessage-Struktur eingerichtet hat. Unser Programm soll jetzt so lange verweilen, bis es eine Nachricht empfängt, die es auswerten kann.

Dafür ist es am besten, wenn wir erst einmal nachsehen, ob eine Nachricht auf dem UserPort anliegt. Mit der Exec-Funktion GetMsg() können wir das:

```
Message = GetMsg(Window->UserPort);
```

Als Ergebnis erhalten wir entweder einen Zeiger auf die Message-Struktur - in diesem Fall ist das eine IntuiMessage-Struktur - oder einen Null-Pointer, der uns unverbindlich mitteilt, daß keine Nachricht vorhanden ist. Nun könnte man ständig diese Abfrage wiederholen, bis endlich eine Nachricht anliegt. Das würde so aussehen:

```
FOREVER
   {
   if (message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort))
      {
         /* Auswertung */
      }
   }
```

Programmteil 3.8: IDCMP-Abfrageschleife (Grundaufbau)

Wir hätten dann zwar eine wunderbare Nachrichten-Abfrage, doch besonders die anderen Tasks unseres Multitasking-Computers würden sich freuen (Ironie!). Denn obwohl überhaupt nichts zu tun ist, arbeitet unser Programm ständig daran, immer wieder Nachrichten zu lesen, die ja vielleicht gar nicht vorhanden sind. Besser wäre es, wenn der Task in den "Ruhestand" versetzt wird, bis eine Nachricht gesendet wurde. Auch dafür gibt es einen Exec-Befehl. Mit Wait() erteilen Sie dem System den Auftrag, daß es den Task so lange nicht mehr abarbeiten soll, bis für ihn eine Nachricht abgelegt wird. In unserem Fall sähe das so aus:

```
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
      /* Auswertung */
   }
```

Programmteil 3.9: IDCMP-Abfrageschleife (verbesserte Version)

Zuerst wird ganz gewöhnlich der UserPort auf eine Nachricht hin untersucht. Liegt keine Nachricht vor, lassen wir das System so lange warten, bis es eine empfängt. Dann soll es am Anfang der Schleife die Programmausführung fortsetzen. Liegt aber eine Nachricht an, so wird sofort mit der Auswertung begonnen. Das ist auch dann der Fall, wenn das System aus dem Wait-Status zurückkehrt.

Dann beginnt die Schleife von vorne, wo wir die erhaltene Nachricht einlesen. Diese ist in jedem Fall da und die Auswertung kann beginnen.

3.7.2 Wir werten die Nachrichten aus

Nachdem Sie nun ausführlich über die Möglichkeiten des Nachrichtenempfangs über den UserPort informiert sind, möchte ich damit beginnen, Ihnen die Auswertung zu erklären.

Die ist vollkommen davon abhängig, welche Art von Nachrichten empfangen werden. Wir benötigen getrennte Auswertungen für Gadgets, Menüs, Requester, Keyboard und Maus. Durch diese strikte Unterteilung erhöht sich zwar der Programmieraufwand etwas, wir haben dafür aber eine sehr hohe Flexibilität.

Außerdem unterstüzt die Sprache C unser Vorhaben vorbildlich, so daß es überhaupt keine Mühe macht, die Abfragen zu programmieren. Wenn man erst weiß, wie es geht...

Wenn wir mit Sicherheit eine Nachricht empfangen haben, geht es an die Vorauswertung. Wir müssen zuerst einmal feststellen, in welche Gruppe die Nachricht fällt, d.h. welchen Flag-Wert die *Class*-Variable der IntuiMessage-Struktur enthält.

Für diese Überprüfung holen wir standardmäßig einige Werte aus der Struktur:

```
MessageClass = Message->Class;
MessageCode  = Message->Code;
```

Wir können dann anhand von *MessageClass* feststellen, welches IDCMP-Flag die Nachricht ausgelöst hat. In diesem Sinne müssen wir dann die Untersuchungen fortsetzen. Es ist aber auch sehr wichtig, nicht zu vergessen, auf die Nachricht zu antworten, damit das Nachrichten-System weiterlaufen kann. Denn ohne Antwort können keine weiteren Nachrichten empfangen werden!

```
ReplyMsg(Message);
```

Machen wir uns jetzt daran, für die einzelnen Flag-Gruppen Abfragen zu konstruieren, mit denen wir die Datenflut bewältigen können.

3.7.2.1 Die altbekannten Gadgets

Wenn wir erstmal den Zeiger auf die IntuiMessage-Struktur haben, können wir mit der Auswertung beginnen. Zuerst sehen wir uns an, wie man diese für Gadgets programmiert. Sie haben dazu zwar schon im Verlauf der Intuition-Kapitels einiges lesen können, doch hier erfahren Sie, wie man von Grund auf eine Gadget-Abfrage entwickelt und seinen eigenen Bedürfnissen anpaßt.

Zuerst testen wir, ob es sich um die Gadget-IDCMP-Flags handelt:

```
if (MessageClass & (GADGETDOWN | GADGETUP))
   {
      /* Gadget-Abfrage */
   }
```

Dann besorgen wir uns den Zeiger auf die Gadget-Struktur:

```
struct Gadget *GadgetPtr;
      GadgetPtr = (struct Gadget *) Message->IAddress;
```

Für die Unterscheidung mehrerer Gadgets brauchen wir jetzt noch die *GadgetID*:

```
USHORT GadgetID;
      GadgetID  = GadgetPtr->GadgetID;
```

(Ich empfehle fortlaufende Nummern, die am Anfang des Programms durch DEFINES ersetzt werden, um die Gadgets namentlich leichter erkennen zu können.)

Nach *GadgetID* unterschieden, können zum Schluß die einzelnen Aktionen ausgelöst werden. Am übersichtlichsten gestalten Sie Ihren Programmtext, wenn Sie vom Programmtext längere Reaktionen in einer Funktion ablegen:

```
switch(GadgetID)
   {
   case TEXTGADGET : Text_Auswerten();
                     break;

   case SCHIEBER   : Neue_Werte();
                     break;

   case ....

   }
```

Alle weiteren Informationen kann sich das Programm über den Pointer auf die Gadget-Struktur holen. In der IntuiMessage-Struktur wird bei Gadgets nichts weiter abgelegt, es sei denn, Sie interessieren sich für die Mauskoordinaten. Diese sind natürlich immer dort zu finden. Die Abfrage ist ganz einfach:

```
MausX = (SHORT) Message->MouseX;
MausY = (SHORT) Message->MouseY;
```

Denken Sie aber daran, daß die Koordinaten-Angaben relativ zum Window zu interpretieren sind.

3.7.2.2 Das Window macht Meldung!

Anders als bei den Gadgets läuft die Abfrage der Window-Flags. Sie sind nicht allgemein und müssen auch nicht weiter erforscht werden. Window-Flags geben sofort über einen bestimmten Zustand Auskunft!

Dafür bieten sich zwei Abfragemöglichkeiten an: Die erste fügt sich in das System ein. Zuerst selektieren wir alle Window-Flags

heraus und untersuchen anschließend weiter. Das läßt sich auf einfache Weise realisieren:

```
if (MessageClass & (NEWSIZE ¦ REFRESHWINDOW ¦ ACTIVATEWINDOW ¦
                                              INACTIVATEWINDOW))
   {
   WindowPtr = (struct Window *) Message->IAddress;
   switch(MessageClass)
      {
      case NEWSIZE          : Breite = WindowPrt->Width;
                              Hoehe  = WindowPrt->Height;
                              break;

      case REFRESHWINDOW    :
                              break;

      case ACTIVATEWINDOW   :
                              break;

      case INACTIVATEWINDOW: ActivateWindow(WindowPtr);
                              break;

   }
```

Programmteil 3.10: Window-Nachrichten-Abfrage

Bitte beachten Sie, daß wir natürlich nicht das Flag SIZEVERIFY auf diese Art abfragen können, da vor der genaueren Untersuchung schon mit ReplyMsg() auf die Nachricht geantwortet wird. Hier reicht aber eine einfache Abfrage des Flags vor jeder Antwort auf die Nachricht. Sie können dann einfach die ReplyMsg()-Funktion überspringen.

Ein witziger Trick wurde in der letzten Überprüfung verwendet. Das Window unseres Programms wird durch diese Zeilen immer wieder aktiviert. Sobald der Benutzer irgendein anderes Window anklickt, wird sofort wieder unseres aktiviert.

Auch die Workbench benutzt diese Methode! Wenn Sie einen File-Namen ändern, dann wird ein Fenster mit einem String-Gadget geöffnet. Das Fenster hat die Größe des Gadgets und ist ohne jede System-Grafik. Klicken Sie jetzt irgendwo in die Workbench, wird das Rename-Window sofort wieder aktiviert.

3.7.2.3 Requester werfen den Ablaufplan über den Haufen

Das ist in der normalen Programmierung nicht der Fall. Requester können nur vom Programm ausgelöst werden und somit nicht den Ablaufplan gefährden. Es sei denn, Sie bieten dem Benutzer den Komfort, daß er über den DMRequest jederzeit eine bestimmte Funktion aufrufen kann. Dann ist der Zeitpunkt der Unterbrechung natürlich nicht festgelegt. Dafür sind hauptsächlich die Requester-Flags gedacht.

Setzen Sie im IDCMP des Windows REQSET und REQCLEAR, und Sie erhalten immer Nachricht, wenn der erste Requester aktiviert wird - unser DMRequest - und wenn der letzte Requester verschwindet - auch unser DMRequest.

Da das End-Flag sowieso über die Requester-Routine bearbeitet wird, brauchen wir uns im Hauptprogramm nur um REQSET kümmern. Dazu ist nur eine Zeile notwendig:

```
if (MessageClass & REQSET) DMRequest_Auswertung();
```

Über REQVERIFY können Sie, genau wie bei SIZEVERIFY, bestimmen, ob der Prozeß sofort ausgelöst werden soll oder vorher noch wichtige Arbeiten zu erledigen sind. Wenn das Flag gesetzt ist, erhält das Programm schon diese Nachricht, bevor Intuition den Requester öffnet. Dieser kann nämlich erst bearbeitet werden, nachdem auf die Nachricht mit ReplyMsg() geantwortet wurde.

3.7.2.4 Menü-Flags mit Auswertung

Menüs sind neben den Gadgets die wichtigste Eingabemöglichkeit für den Anwender, die Intuition bietet. Auch sie erfordern eine Auswertung.

Ist vom Benutzer die Menütaste gedrückt und wieder losgelassen worden, erhält das Programm über den IDCMP eine Nachricht, wenn das MENUPICK-Flag gesetzt wurde. Wir können dann im Feld Code der IntuiMessage-Struktur nach der Nummer des Menüpunktes suchen. Nun werden die Menüpunkte für den Be-

diener offensichtlich nicht mit Nummern verwaltet. Aber intern wurde ein System entwickelt, um die Identifizierung möglichst einfach zu halten. Allerdings liegt in dem System auch der Grund, warum wir nur eine Stufe Untermenüs programmieren können, obwohl doch eine unendliche Verschachtelung denkbar wäre.

Sehen wir uns nun die Kodierung der ausgewählten Menüs an! Die beiden Bytes sind wie folgt "verschlüsselt":

```
UUUUU MMM|MMM NNNNN
```

Wir haben zwei Bytes in drei Bit-Gruppen. Die erste NNNNN beschreibt die Menü-Nummer. Wir müssen diese nur von den anderen befreien, und schon sind wir im Besitz der Zahl. Diese Arbeit erledigt die Definition MENUNUM(Code). Sie ist im *<intuition/intuition.h>-Include-File* zusammen mit den folgenden zu finden und klammert einfach die entsprechenden Bits aus.

Die zweite Gruppe, MMMMMM, stellt die Nummer des Menüpunktes dar. Sie wurde doppelt so groß gewählt, damit auch wirklich genügend Menüs (31) und genügend Menüpunkte (63) benutzt werden können. Möchten Sie auch die Nummer des Menüpunktes feststellen, dann verwenden Sie dazu die Definition ITEMNUM(Code).

In der dritten Gruppe, UUUUU, finden Sie die Nummer des Untermenüpunktes, wenn überhaupt einer vorhanden ist. Auch hier sind wieder 31 Werte möglich. Diesen Wert erreichen Sie über die Definition SUBITEM(Code).

Durch die Definitionen wird uns die Menüabfrage wirklich einfach gemacht. In unserer Abfrageschleife für alle IDCMP-Nachrichten müssen wir zuerst feststellen, ob überhaupt eine Menü-Nachricht vorliegt. Dann können wir zu einer Funktion verzweigen, die weitere Auswertungen vornimmt.

```
if (MessageClass & MENUPICK) Menu_Auswertung(Code);
```

In der Funktion werden schließlich die Werte berechnet und nacheinander ausgewertet:

```
Menu_Auswertung(Menunummern)
USHORT Menunummer;
{
   USHORT Menu, MenuItem, SubItem;

   Menu     = MENUNUM(Menunummern);
   MenuItem = ITEMNUM(Menunummern);
   SubItem  = SUBITEM(Menunummern);

   switch(Menu)
   {
      case FILEMENU : FileAuswertung(MenuItem, SubItem);
                      break;

      case WORKMENU : WorkAuswertung(MenuItem);
                      break;
   }
}


FileAuswertung(MenuItem, SubItem)
USHORT MenuItem, SubItem;
{
   switch(MenuItem)
   {
      case LADEN    : Laden();
                      break;

      case SPEICHERN: Speichern();
                      break;

      case LOESCHEN : switch(SubItem)
                      {
                         case FILE : DelFile();
                                     break;

                         case TEXT : DelText();
                                     break;
                      }
                      break;
   }


WorkAuswertung(MenuItem)
USHORT MenuItem;
{
   switch(MenuItem)
   {
      case MARKIEREN: Mark();
```

```
                    break;

    case EDITIEREN: Edit();
                    break;

    case SUCHEN   : Suchen();
                    break;
  }
```

Funktion 3.7: Menü-Abfrage

Ich habe anstatt der Nummern für die Menüs, Menüpunkte und Untermenüs definierte Ausdrücke gewählt. Dadurch wird der Programmtext leichter verständlich. Hier sind die DEFINES für die obigen Routinen:

```
/* Menünamen */

#define FILEMENU
#define WORKMENU

/* 1. Menü  FILEMENU */

#define LADEN
#define SPEICHERN
#define LOESCHEN

/* Untermenü LOESCHEN */

#define FILE
#define TEXT

/* 2. Menü WORKMENU */

#define MARKIEREN
#define EDITIEREN
#define SUCHEN
```

3.7.2.5 Mäuse laufen gerne. Aber wohin?

Nicht nur zum Auslösen von Gadgets und Markieren von Menüpunkten ist die Maus geschaffen. Der Programmierer kann alle Tätigkeiten eigenständig abfragen. Für die direkte Mausabfrage wird auch der IDCMP genutzt. Wir kennzeichnen hier durch Setzen der Flags, ob wir über das Drücken oder Loslassen der Maus-Buttons informiert werden möchten oder lieber über die

Bewegung der Maus. Alles das kann über den IDCMP empfangen werden.

In einem Zeichenprogramm kann es durchaus wichtig sein zu wissen, wann der Mausknopf gedrückt wird, damit darauf z.B. an der Stelle ein Punkt gesetzt wird. Allerdings gibt es zwei Maus-Buttons, und bei beiden kann sowohl das Niederdrücken als auch das Loslassen abgefragt werden. Dementsprechend gibt es vier Ergebnisse:

SELECTDOWN, SELECTUP
Die "select"-Flags stehen für die linke Maustaste.

MENUDOWN, MENUUP
Die "menu"-Flags stehen für die rechte Maustaste.

Beachten Sie aber, daß die linke Maustaste nur Signale sendet, wenn sie nicht für die Arbeit mit Gadgets gebraucht wird. Deren Handhabung hat eine höhere Priorität! Gleiches gilt für die rechte Maustaste. Wird sie für die Menüauswahl benötigt, dann erfährt das Programm überhaupt nichts davon. Nur wenn extra das Flag RMBTRAP gesetzt wurde, ist die rechte Maustaste registrierbar. Darauf sollten Sie achten!

Die Abfrage sieht im Programm sehr einfach aus. Nehmen wir an, daß das Programm beide Tasten benötigt und keine Menüs verwaltet. Wir setzen also im Window das IDCMP-Flag MOUSEBUTTONS und RMBTRAP. Immer, wenn wir eine Nachricht erhalten, finden wir im Code-Feld der IntuiMessage-Struktur die Art des Maus-Buttons:

```
MausX = (SHORT) Message->MouseX;
MausY = (SHORT) Message->MouseY;

if (MessageClass & MOUSEBUTTONS) MausAbfrage(Code, MausX, MausY);
```

Diese Zeile verzweigt in ein Unterprogramm, in dem weitere Auswertungen stattfinden:

```
MausAbfrage(Flag, MausX, MausY)
USHORT Flag;
SHORT MausX, MausY;
{
   switch(Flag)
   {
      case SELECTDOWN : Begin_Line(MausX, MausY);
                        break;

      case SELECTUP   : End_Line(MausX, MausY);
                        break;

      case MENUDOWN   : Begin_Erase(MausX, MausY);
                        break;

      case MENUUP     : End_Erase(MausX, MausY);
                        break;
   }
}
```

Funktion 3.8: Maus-Abfrage

Die Beispiel-Funktion bekommt als Übergabewerte den Status der Maus, zusammen mit den Koordinaten. Je nachdem, welcher Maus-Status gerade herrscht, wird entweder die Linien-Funktion des Programms aufgerufen und gestartet oder beendet. Oder aber die Flächen-Lösch-Funktion tritt in Aktion, und wir übergeben den ersten Eckpunkt oder den zweiten.

Der nächste äußerst wichtige Punkt der Mausabfrage ist die Positionsübermittlung. In dem ersten Beispiel haben wir diese schon teilweise benutzt, denn zu jeder IntuiMessage wird ja auch die Mausposition relativ zum Fenster geliefert. Wollen wir aber z.B. eine ständige Koordinatenanzeige programmieren, so können wir nicht darauf warten, daß der Benutzer immer eine Taste drückt, wenn er die Koordinaten sehen möchte. Für ihn ist es entscheidend, immer den aktuellen Stand zu haben. Auch dafür bietet Intuition einen Nachrichten-Typ. Setzen Sie einfach das IDCMP-Flag REPORTMOUSE, und Sie empfangen jede kleine Bewegung!

Hinweis: An dieser Stelle möchte ich Sie schon warnen! Auch das Bewegen der Maus um nur einen Pixel löst schon eine Nachricht aus. Das kommt sehr oft vor! Bedenken Sie auch, daß für das genaue Positionieren meist

sehr viele Schritte notwendig sind, bei denen Ihr Programm einen "Infarkt" erleiden kann.

Auch bei den Mausbewegungen empfehle ich eine normale Abfrage:

```
MausX = (SHORT) Message->MouseX;
MausY = (SHORT) Message->MouseY;

if (MessageClass & MOUSEMOVE) MausBewegung(MausX, MausY);
```

Ein Unterprogramm kann dann die Auswertung vornehmen. Mit der oben beschriebenen Methode erhalten Sie bei jeder Mausbewegung die absoluten Werte relativ zum Fenster des IDCMP. In manchen Anwendungen ist es aber einfacher, wenn man nur den Unterschied zu letzten Position erfährt.

Für diese Umstellung müssen Sie nur das DELTAMOVE-Flag setzen.

3.7.2.6 Erkennunng der Tastatureingaben

Wir erkennen Tastatureingaben nur, wenn eins der beiden Flags VANILLAKEY oder RAWKEY gesetzt ist.

Damit stellen wir den Modus ein, unter dem wir die Tastatur-Codes erhalten wollen. Es ist nur ein Modus pro Window möglich.

RAWKEY unterscheidet sich von VANILLAKEY dadurch, daß wir bei RAWKEY "unübersetzte" Codes erhalten. Das heißt, wir bekommen für jede Taste eine Nummer als Nachricht.

Ist VANILLAKEY aktiv, werden die Tastaturnummern nach der Tastaturtabelle übersetzt. Wir erhalten dann die Zeichen der Tastaturtabelle.

Somit richtet sich das Programm nach der Tastaturbelegung des jeweiligen Computers. In Deutschland verwenden wir eine

deutsche Tastaturbelegung, in England eine englische, in Frankreich eine französische usw...

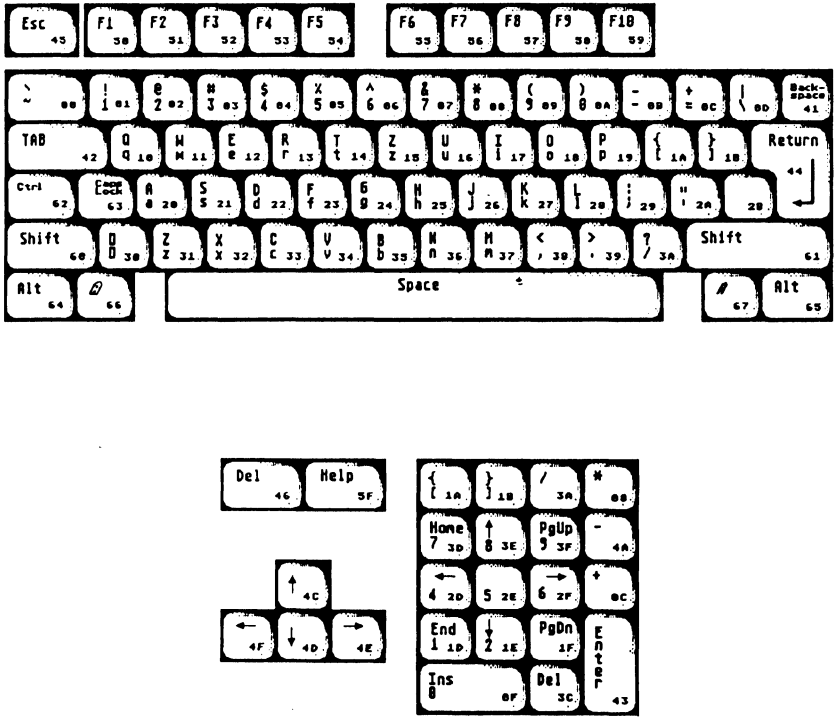


Abbildung 3.12: RAW-Tastatur-Codes I

Sie sehen anhand der Abbildung die Tastatur-Codes für jede einzelne Taste. (Die Tastatur wurde aus Platzgründen in der Abbildung aufgeteilt.) Zu beachten ist, daß selbst der Zehnerblock eine eigene Numerierung besitzt. Wollen Sie eine Abfrage programmieren, dann müssen Sie im *Class*-Feld nach dem Flag RAWKEY suchen und im Feld *Code* nach der Tastennummer. Zusätzlich finden Sie im Feld *Qualifier* einen Statusbericht der Sondertasten wie Shift, Alt, Ctrl usw.

Sie können mit dieser Tabelle gedrückte Tasten identifizieren:

QUALIFIER_TYPES	Hex-Wert
IEQUALIFIER_LSHIFT	0x0001L
IEQUALIFIER_RSHIFT	0x0002L
IEQUALIFIER_CAPSLOCK	0x0004L
IEQUALIFIER_CONTROL	0x0008L
IEQUALIFIER_LALT	0x0010L
IEQUALIFIER_RALT	0x0020L
IEQUALIFIER_LCOMMAND	0x0040L
IEQUALIFIER_RCOMMAND	0x0080L
IEQUALIFIER_NUMERICPAD	0x0100L
IEQUALIFIER_REPEAT	0x0200L
IEQUALIFIER_INTERRUPT	0x0400L
IEQUALIFIER_MULTIBROADCAST	0x0800L
IEQUALIFIER_MIDBUTTON	0x1000L
IEQUALIFIER_RBUTTON	0x2000L
IEQUALIFIER_LEFTBUTTON	0x4000L
IEQUALIFIER_RELATIVEMOUSE	0x8000L

Tabelle 3.20: Qualifier-Typen

Das Input-Device läßt selbst eine Unterscheidung in den Tastengruppen zu. So können Sie die linke, rechte und auch die Caps-Shift-Taste erkennen. Weiterhin erfahren wir etwas darüber, ob die Information vom Numericpad kam und ob sich die Taste schon im Repeat-Status befindet. Sie können hierüber sogar die Maustasten abfragen.

Wollen Sie doch lieber mit Codes arbeiten, die schon nach der Tastaturtabelle verarbeitet sind, dann erhalten Sie bei *Class* eine Nachricht des Types VANILLAKEY. Unter *Code* finden wir den übersetzten Zeichen-Code. Diese Möglichkeit ist die einfachste. Egal, welche Methode Ihnen in einem Programm besser gefällt, der Aufruf ähnelt den bereits bekannten:

```
    if (MessageClass & RAWKEY) RAW_Auswertung(Code, Qualifier);
    if (MessageClass & VANILLAKEY) VANILLA_Auswertung((char)Code);
```

Entscheidende Unterschiede werden erst in der Auswertung des empfangenen Codes sichtbar. Während VANILLAKEY ganz einfach mit dem Zeichen arbeiten kann, muß die RAWKEY-Auswertung erst einmal eine eigene Tastaturtabelle bemühen.

```
VANILLA_Auswertung(Zeichen)
char Zeichen;
{
   Print(Zeichen);
   InputTextBuffer(Zeichen);
   ...
}
```

Funktion 3.9: Vanilla-Auswertung

```
RAW_Auswertung(Code, Qualifier);
USHORT Code, Qualifier;
{
   /* 1. Umwandlung nach Tabelle */

   Zeichen = TastTab[Qualifier][Code];

   /* oder 2. Umwandlung mit Einzelunterscheidung */

   switch(Qualifier)
   {
      case IEQUALIFIER_LSHIFT :
         switch(Code)
            {
               /* Auswertung linke Shift-Taste */
            }
         break;

      case IEQUALIFIER_RSHIFT :
         switch(Code)
            {
               /* Auswertung rechte Shift-Taste */
            }
         break;

      switch(Code)
         {
               /* Auswertung ohne Shift-Taste */
         }
   }
}
```

Funktion 3.10: RawKey-Auswertung

3.7.2.7 Die Diskettenstation

Ich möchte an dieser Stelle noch einmal auf die File-Select-Box zurückkommen. Diese soll ja das Inhaltsverzeichnis der eingelegten Diskette anzeigen.

Was passiert nun, wenn wir die Diskette aus dem Laufwerk nehmen und sie wechseln?

Nichts! Immer noch die gleiche Anzeige mit den gleichen Files, aber eigentlich sollte doch das Verzeichnis der neuen Diskette angezeigt werden.

Dieses Problem läßt sich einfach über die beiden Disk-Flags lösen. Mit dem ersten, DISKREMOVED, erfahren wir immer, wenn eine Diskette entfernt wird. Dann kann gegebenenfalls die File-Tabelle gelöscht werden.

Mit DISKINSERTED können wir den Lesevorgang starten, der das neue Verzeichnis analysiert.

```
if (MessageClass & DISKREMOVED)
   {
      /* Tabelle löschen */
   }

if (MessageClass & DISKINSERTED)
   {
      /* Tabelle neu einlesen */
   }
```

3.8 Menüs, die Auswahl der Konsumgesellschaft

Wie ist das bei Ihnen? Sie haben ein Programm geschrieben, das mit vielen, vielen Funktionen aufwartet. Zu jeder Kleinigkeit haben Sie ein Unterprogramm geschrieben.

Der Benutzer wird überflutet mit Auswahlmöglichkeiten. Nur ...

... wissen Sie noch nicht, wie Sie ihm alle Funktionen darbieten sollen. Nehmen wir Tastaturbelegungen, dann heißt es eine

Schablone anzufertigen, denn auswendig lernen dauert lange, und man ist ja so vergeßlich.

Warum nicht Gadgets? Wir können ja nicht den ganzen Bildschirm mit Gadgets vollpacken. Immerhin haben wir auch noch ein Ein- und Ausgabe-Window. Wohin dann mit den Funktionen? (Hätten Sie bloß nicht so viele programmiert!)

Dem Amiga macht das gar nichts aus! Er hat für jede Menge an Auswahlpunkten das richtige Medium! Gadgets eignen sich gut für einige kleine Aufrufe, und wenn gleich in "Zehnersalven" gearbeitet wird, nehmen wir die Menüs.

Mit den von Intuition angebotenen Menüs können wir alle Funktionen unseres Programms in Gruppen unterteilen. Da gibt es dann meist eine Gruppe für "Datei", "Bearbeiten". In jeder dieser Gruppen finden wir Funktionen zu einem Thema. Manche Funktionen sind noch weiter untergliedert, und wir können dann auswählen.

Die kleinsten, aber auch ersten Menüs, die Sie als Amiga-Besitzer kennengelernt haben, sind die Workbench-Menüs. Wir finden drei Hauptgruppen vor: "Workbench", "Disk" und "Special". Jede dieser Hauptgruppen enthält Unterpunkte, die bei Auswahl eine Aktion auslösen. So etwas wollen wir jetzt auch programmieren! Wir wollen ein oder mehrere Menüs schreiben, die mehrere Menüpunkte haben. Manche der Menüpunkte sollen auch noch Untermenüs besitzen, in denen man weitere Punkte findet. Dieses Menü soll alle Möglichkeiten ausnutzen, die programmiertechnisch möglich sind. Lassen Sie sich überraschen, denn es ist weit mehr erlaubt, als Sie von der Workbench kennen.

3.8.1 Der allgemeine Aufbau eines Menüs

Es ist wichtig zu wissen, welchen Aufbau die Menüs haben, damit wir unsere gestalterischen Grenzen kennenlernen. Nehmen wir als erstes Beispiel, weil es jedermann zugänglich ist, die Workbench-Menüs.

Wenn die rechte Maustaste gedrückt ist, erkennt man, daß die Titelleiste des Screens für eine neue Leiste weichen muß, in der wir drei Namen finden. Dies sind die Namen der Funktionsgruppen der drei Menüs. Die Leiste selbst paßt sich dem Font des jeweiligen Screens an. Den Überschriftentext können wir als Programmierer pixelgenau in X-Richtung einstellen.

Betrachten wir nun das erste Menü: Das Workbench-Menü hat sechs Funktionen, die alle nicht anwählbar sind. Alle Texte werden dafür in der sog. Geisterschrift dargestellt (engl. ghosted). Erst wenn wir z.B. ein Icon anklicken, ist entsprechend der Möglichkeiten eine Auswahl zugelassen. Gleiches gilt für die beiden anderen Menüs.

Als zweites Beispiel möchte ich die Menüleiste eines Zeichenprogramms benutzen. Dieses Menü benutzt auch Grafikelemente. Wir können deshalb als ersten Punkt festhalten, daß Menüs nicht unbedingt aus Textzeilen bestehen müssen. Wir können überall Grafiken verwenden. Die zweite Besonderheit ist, daß wir zusätzlich nicht daran gebunden sind, in einer "Zeile" einen Text oder eine Grafik zu verwenden. Es ist uns freigestellt, wie viele Texte oder Grafiken pro Zeile in einem Menü stehen. Aber achten Sie als Programmierer darauf, daß sich die einzelnen Punkte nicht überlappen!

Drittens möchte ich die freie Wahl vorstellen, mit der Sie über die Größe des Menükastens bestimmen. Es liegt ganz an Ihnen, ob wir ein Menü haben, das vielleicht nur zwei Zeichen breit ist, oder eines, das den ganzen Screen ausfüllt. Die Größe wird automatisch den Menüpunkten angepaßt.

3.8.2 Wir basteln ein Menü

Sie haben jetzt einen groben Überblick durch die beiden Beispiele gewonnen. Es sollte unser erstes Ziel sein, eine eigene Menüleiste aufzubauen. Um möglichst freie Gestaltungsmöglichkeiten zu haben, möchte ich dazu einen neuen Screen mit eigenen Parametern öffnen. Außerdem benötigen wir ein Window,

ohne das keine Menüleiste existieren kann, denn sie wird an das Window angeheftet.

Das folgende Listing entspricht dem Standardprogramm für Screens und enthält zusätzlich noch eine Zeichensatzeinstellung:

```
/*************************************
 *                                   *
 * Programm: Demonstrations Menu-Strip *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     13. 1.1988  Menu-Strip    *
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct TextAttr ScreenFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct NewScreen FirstNewScreen =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 256,                /* Width, Height       */
   3,                       /* Depth               */
   0, 1,                    /* DetailPen, BlockPen */
   HIRES,                   /* ViewModes           */
   CUSTOMSCREEN,            /* Type                */
   &ScreenFont,             /* Font                */
   (UBYTE *)"Screen Test",
   NULL,                    /* Gadgets             */
   NULL                     /* CustomBitMap        */
   };
```

```
struct NewWindow FirstNewWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 100,                /* Width, Height       */
   0, 1,                    /* DetailPen, BlockPen */
   CLOSEWINDOW,             /* IDCMP Flags         */
   WINDOWDEPTH |            /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   (UBYTE *)"Test Menü-Strip",
   NULL,                    /* Screen (kommt noch) */
   NULL,                    /* BitMap              */
   100, 50,                 /* Min Width, Height   */
   640, 256,                /* Max Width, Height   */
   CUSTOMSCREEN             /* Type                */
   };


main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;

      ReplyMsg(message);
      switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
         }
      }
   }
```

```
/*********************************************
 *                                           *
 * Funktion: Library, Screen und Window öffnen *
 * ========================================= *
 *                                           *
 * Autor:  Datum:      Kommentar:            *
 * ------  ----------  ----------            *
 * Wgb     16.10.1987  funktioniert!         *
 *                                           *
 *                                           *
 *********************************************/

Open_All()
   {
   void          *OpenLibrary();
   struct Window *OpenWindow();
   struct Screen *OpenScreen();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstScreen = (struct Screen *)
       OpenScreen(&FirstNewScreen)))
      {
      printf("Screen hat keine Zeit!\n");
      Close_All();
      exit(FALSE);
      }

   FirstNewWindow.Screen = FirstScreen;

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will nicht aufgehen!\n");
      Close_All();
      exit(FALSE);
      }
   }
```

```
/*************************************
 *                                   *
 * Funktion: Alles Geöffnete schließen *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  Window, Screen *
 *                     und Intuition *
 *                                   *
 *************************************/

Close_All()
   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (FirstScreen)     CloseScreen(FirstScreen);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

Programm 3.9: Menu-Strip-Demo

3.8.2.1 Die Menüleiste als Grundstock

Mit Hilfe dieses Testprogramms werden wir nun die Menüleiste aufbauen. Der englische Fachbegriff für Menüleiste ist übrigens Menu-Strip. Dafür benötigen wir zuerst wieder eine Struktur für jedes Menü. Diese enthält neben dem Namen des Menüs noch ein Flag, Koordinatenangaben und zwei Link-Pointer.

```
struct Menu
{
0x00  00  struct Menu *NextMenu;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  USHORT Flags;
0x0E  14  BYTE *MenuName;
0x12  18  struct MenuItem *FirstItem;
0x16  22  SHORT JazzX;
0x18  24  SHORT JazzY;
0x1A  26  SHORT BeatX;
0x1C  28  SHORT BeatY;
0x1E
};
```

Strukturbeschreibung

**NextMenu*

Zuerst enthält die Menu-Struktur, wie so viele Intuition-Strukturen, einen Zeiger auf eine Folgestruktur. Er wird für die Verkettung mehrerer Menüs benötigt.

LeftEdge, Width

Die Werte beschreiben die Position und die Breite der Select-Box. Das ist der Bereich, in den der Mauspfeil gefahren werden muß, bis das Menü aktiviert wird.

TopEdge, Height

Leider werden diese beiden Werte nicht genutzt. Sie wurden lediglich implementiert, um auch für Erweiterungen zur Verfügung zu stehen. So soll es später möglich sein, auch Grafiken in die Menu-Strip einzubinden, was momentan noch nicht erlaubt ist.

Flags

Ein Flag-Wert, über den wir einstellen, ob das gesamte Menü anwählbar sein soll oder nicht. Dann werden nämlich alle Menüpunkte "ghosted" ausgegeben. Um die Auswahl zuzulassen, ist das Flag MENUENABLED zu setzen. Das Flag MIDRAWN wird von Intuition gesetzt, wenn das Menü gerade vom Benutzer ausgewählt wurde.

**MenuName*

Als nächstes finden wir einen Pointer auf den Namen des Menüs. Hier ist nur ein String erlaubt, der im Default-Zeichensatz des Screens ausgegeben wird.

**FirstItem*

Hier finden wir die Verkettung mit einer weiteren Struktur. Sie enthält alle Informationen für die einzelnen Menüpunkte.

Alle weiteren Variablen werden von Intuition benutzt und müssen nicht initialisiert werden.

Für den Anfang bauen wir drei Menüs in unser Testprogramm ein. Das erste soll alle Disketten-Funktionen unterstützen und heißt "Datei". Sie können es fast in jedem Programm verwenden, denn die Arbeit mit der Diskette ist für gute Programme notwendig. Das zweite Menü schlägt schon einen spezielleren Weg ein. Es ist für eine Textverarbeitung gedacht. Sie sollen in ihm Einstellungen für das Textaussehen machen können. Es wird "Design" heißen. Im dritten Menü werden wir uns den Grafik-Programmen widmen. Hier können Sie sich den Zeichenstift aussuchen oder die Zeichenfarbe wählen. Es heißt "Grafik".

Ich habe alle drei Strukturen initialisiert vorbereitet. Sie brauchen sie nur noch abzutippen und in das Testprogramm einzubauen.

```
struct Menu Grafik =
{
   NULL,
   220, 0,
   70, 10,
   MENUENABLED,
   (UBYTE *) "Grafik",
   NULL
};

struct Menu Design =
{
   &Grafik,
   100, 0,
   70, 10,
   MENUENABLED,
   (UBYTE *) "Design",
   NULL
};

struct Menu Datei =
{
   &Design,
   10, 0,
   60, 10,
   MENUENABLED,
   (UBYTE *) "Datei",
   NULL
};
```

Struktur 3.19: Menu-Strukturen zum Demo-Programm

Fügen Sie nun zwei neue Befehle in unsere _All()-Funktionen ein. Zuerst müssen wir diese Menü-Leiste in das Window einbinden. Dafür gibt es die Intuition-Funktion SetMenuStrip(). Sie hat folgendes Aussehen:

```
Success = SetMenuStrip(Window, Menu);
   D0          -264        A0     A1
```

Wir übergeben das Window und die Adresse der ersten Menu-Struktur. Schon ist die Menü-Leiste aktiv. Sie können mit ihr arbeiten, wann immer das Window aktiv ist. Für unsere Open_All()-Funktion bedeutet das, daß Sie diese Zeile nach der OpenWindow()-Funktion ergänzen müssen:

```
SetMenuStrip(FirstWindow, &Datei);
```

Ich habe hier mit Absicht den Rückgabewert *Success* nicht beachtet, da er sowieso immer TRUE lautet. Wir ersparen uns dadurch eine Variablendefinition.

In der Close_All()-Funktion hieße die Ergänzung dann:

```
if (FirstWindow)    ClearMenuStrip(FirstWindow);
```

Das allgemeine Format von ClearMenuStrip() ist:

```
ClearMenuStrip(Window);
      -54        A0
```

Denken Sie daran, daß die Menü-Leiste entfernt werden muß, bevor das Window geschlossen wird, da es sonst zu einem Systemabsturz kommt. Außerdem ist die Menüleiste vor jeder Änderung zuerst vom Window zu entfernen. Nach der Änderung kann sie dann wieder mit SetMenuStrip() angehängt werden.

Programmbeschreibung

Nach dem Start des Programms sehen Sie einen Screen mit einem Window. Solange das Window inaktiv ist, können Sie über die rechte Maustaste, die Menütaste, keine Menüleiste erreichen. Erst nachdem das Fenster aktiviert wurde, ist auch die Menüleiste sichtbar. Sie finden dort drei Menüs mit den definierten Titeln vor, die aber noch überhaupt keine Punkte beinhalten. Das

soll sich gleich ändern! Zuvor sei noch erwähnt, daß Sie das Programm über das Close-Gadget des Windows beenden.

3.8.2.2 Die ersten Menüpunkte und die MenuItem-Struktur

Als zweites Ziel wollen wir uns den Entwurf der ersten Menüpunkte setzen. Wie greifen dabei zuerst auf Texte zurück, weil deren Handhabung besonders einfach ist. Wir benötigen lediglich für jeden Text eine IntuiText-Struktur. Das dürfte aber nicht weiter schwierig sein. Hier sind die von mir geplanten Menüpunkt-Texte:

```
struct TextAttr Font =
   {
   (STRPTR)"topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct IntuiText LadenText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Laden", NULL
   };

struct IntuiText SpeichernText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Speichern", NULL
   };

struct IntuiText LoeschenText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Löschen", NULL
   };

struct IntuiText ProgrammendeText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Programmende", NULL
   };
```

Struktur 3.20: Menü-Texte

Wir haben jetzt zwar die Texte, benötigen aber noch für jeden Menüpunkt eine Struktur. Das ist die MenuItem-Struktur, die alle wichtigen Daten für jeden Menüpunkt aufbewahrt.

```
struct MenuItem
{
0x00  00  struct MenuItem *NextItem;
0x04  04  SHORT LeftEdge
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  USHORT Flags;
0x0E  14  LONG MutualExclude;
0x12  18  APTR ItemFill;
0x16  22  APTR SelectFill;
0x1A  26  BYTE Command;
0x1B  27  struct MenuItem *SubItem;
0x1F  31  USHORT NextSelect;
0x21  33
};
```

Strukturbeschreibung

**NextItem*

Ein Zeiger auf den nächsten Menüpunkt. Wir brauchen ihn, um mehr als einen Menüpunkt pro Menü zu verwalten. Alle Menüpunkte eines Menüs werden durch diese Kette verknüpft

LeftEdge, TopEdge

Die Position des Aktivierungsbereiches dieses Menüpunktes, der ab jetzt Select-Box genannt werden soll. Die Koordinaten sind relativ zu *LeftEdge* und *TopEdge* der Menüstruktur!

Width, Height

Die Ausdehnung der Select-Box des Menüpunktes.

Flags

Wieder ein Flag-Wert, mit dem wir die Eigenschaften des Menüpunktes einstellen können.

Normalerweise vermutet Intuition unter *ItemFill* und *SelectFill* jeweils einen Pointer auf eine Image-Struktur. Durch Setzen des Flags ITEMTEXT stellen Sie eine IntuiText-Struktur ein.

Auf welche Art der Menüpunkt dargestellt werden soll, der augenblicklich mit dem Mauspfeil angewählt wird, kann mit den HIGHFLAGS-Flags ausgesucht werden. Setzen Sie HIGHCOMP,

wird die Select-Box invertiert. Setzen Sie HIGHBOX, wird um die Select-Box ein Kasten gezeichnet. Über HIGHIMAGE können Sie einstellen, daß ein neuer Text oder eine neue Grafik gezeigt wird. Das letzte Flag, HIGHNONE, weist Intuition an, nichts zu tun.

Genau wie auch bei der Menü-Überschrift können wir bei jedem Menüpunkt angeben, ob er angewählt werden soll oder nicht. Dazu verwenden wir das Flag ITEMENABLED. Setzen Sie ITEMENABLED, wenn der Menüpunkt angewählt werden darf.

Zu jedem Menüpunkt kann eine Kommando-Taste definiert werden. Falls eine existiert, ist das Flag COMMSEQ zu setzen.

Als weiteren Komfort erlaubt es Intuition, bestimmte Menüpunkte abzuhaken. Dies kennzeichnet das Flag CHECKIT. Der Menüpunkt ist dann nicht mehr für einen Funktionsaufruf gedacht, sondern stellt einen aktiven oder inaktiven Zustand dar.

Soll ein Menüpunkt, der abgehakt werden kann, auch direkt abgehakt sein, wenn das Menü initialisiert wird, dann müssen Sie zusätzlich noch das CHECKED-Flag setzen. Sie können dieses dann später auch abfragen.

Intuition hat in der Variablen *Flags* zwei weitere Flags für sich reserviert. So wird ISDRAWN gesetzt, wenn Untermenüpunkte des Menüpunktes ausgegeben sind. Das Flag HIGHITEM wird gesetzt, wenn der Menüpunkt mit dem Maus-Cursor aktiviert wird.

MutualExclude

Über die Bits dieser Variablen können wir einstellen, ob andere Menüpunkte bei Auswahl dieses Menüpunktes deaktiviert werden sollen.

ItemFill

Dieser Pointer zeigt entweder auf eine Image-Struktur oder auf eine IntuiText-Struktur. Sie kennzeichnen dies mit dem Flag ITEMTEXT.

SelectFill

Der zweite Pointer enthält auch wieder einen Zeiger auf eine der beiden Strukturen. Allerdings wird diese nur dargestellt, wenn der Menüpunkt aktiviert wird und auch das Highlighting-Flag HIGHIMAGE gesetzt ist.

Command

Falls unter *Flags* eine Kommando-Taste definiert wurde, ist hier der ASCII-Code des Zeichens zu finden, das den Menüpunkt aktivieren soll.

**SubItem*

Wie Sie vielleicht schon wissen, ist es möglich, die Menüs noch auf einer weiteren Ebene zu verschachteln. Dieser Zeiger zeigt auf den ersten Untermenüpunkt dieses Menüpunktes.

NextSelect

Eine Variable, die erst nach Auswahl des Menüpunktes von Intuition gefüllt sein kann. Dann ist nämlich noch ein weiterer Menüpunkt ausgewählt worden, und Sie finden hier seine Nummer!

Beginnen wir nach dieser anstrengenden Strukturanalyse mit der Programmierung unserer ersten Menüpunkte. Oben haben Sie hoffentlich schon die IntuiText-Strukturen gelesen und auch abgetippt. Ich habe vier dazu passende MenuItem-Strukturen:

```
struct MenuItem Programmende =
{
   NULL,
   1, 40,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&ProgrammendeText,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Loeschen =
{
```

```
    &Programmende,
    1, 26,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&LoeschenText,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Speichern =
{
    &Loeschen,
    1, 14,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&SpeichernText,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Laden =
{
    &Speichern,
    1, 2,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&LadenText,
    NULL,
    0,
    NULL,
    0
};
```

Struktur 3.21: MenuItem-Strukturen zum 1. Menü

3.8.2.3 Kleine Extras in der Menügestaltung

Nun soll es Leute geben, die mit den obigen Strukturen, Definitionen, Texten und Einstellungen zufrieden sind. Wir aber nicht! Abgesehen davon, daß noch zwei Menüs ohne Menüpunkte existieren und das erste Menü auch noch Untermenüs bekommen soll, ist dieses Menü viel zu fade entworfen. Es fehlt einmal das

Extravagante und das Außergewöhnliche, was ein gutes Programm auszeichnet, und es fehlt der Bedienerkomfort! Gehen wir Schritt für Schritt, unter Beachtung und vollständiger Ausschöpfung aller von Intuition angebotenen Extras, vor.

Zuerst ist der Bedienungskomfort an der Reihe. Er steht leider wirklich fast an letzter Stelle. Versetzen Sie sich in folgende Situation: Sie sind langjähriger Anwender des Programms und kennen seine Möglichkeiten, müssen aber immer noch die Maus in die Hand nehmen, die Menüs entlangfahren, nur um eine Funktion anzuwählen. Ständig der Wechsel zwischen Tastatur, Maus und ...

Viel leichter wäre es, wenn man die Menü-Funktionen auch von der Tastatur aus aufrufen könnte. Das ist auch zugelassen, und warum sollen wir es dann nicht nutzen?

Dafür ist ganz einfach das Flag COMMSEQ im Feld *Flags* der MenuItem-Struktur zu setzen und der Parameter *Command* mit dem ASCII-Wert der Taste zu "füttern". Dann wird im Menü auf der rechten Seite zusätzlich noch die Amiga-Taste mit unserem Zeichen dargestellt. Erweitern Sie deshalb die Select-Box des Menüpunktes um mindestens die Breite von vier Zeichen (40 Pixel). Ein Zeichen als Zwischenraum, zwei Zeichen für das Amiga-Symbol und ein Zeichen für die Taste. Die Kommando-Sequenz wird immer mit der rechten Amiga-Taste aufgerufen.

Erweitern Sie nun alle Select-Box-Breiten um vierzig Pixel, und ergänzen Sie das Flag COMMSEQ in den Flag-Feldern. Vergessen Sie nicht, überall einen Buchstaben einzusetzen. Ich würde folgende empfehlen: L für Laden, S für Speichern, D für Löschen und E für Programmende. Nun sehen die MenuItem-Strukturen so aus:

```
struct MenuItem Programmende =
{
   NULL,
   1, 40,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ProgrammendeText,
```

```
    NULL,
    0x45,
    NULL,
    0
};

struct MenuItem Loeschen =
{
    &Programmende,
    1, 26,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&LoeschenText,
    NULL,
    0x44,
    NULL,
    0
};

struct MenuItem Speichern =
{
    &Loeschen,
    1, 14,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&SpeichernText,
    NULL,
    0x53,
    NULL,
    0
};

struct MenuItem Laden =
{
    &Speichern,
    1, 2,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&LadenText,
    NULL,
    0x4C,
    NULL,
    0
};
```

Struktur 3.22: MenuItem-Strukturen zum 1. Menü, 2. Version

Hinweis: Beachten Sie, daß Intuition Groß- und Kleinbuchstaben unterscheidet, Sie können so jede Taste zweimal benutzen.

Die magere Ausstattung der Menüpunkte ist nicht zufriedenstellend. Der Benutzer erkennt zwar durch die Invertierung, welchen Menüpunkt er gerade ausgewählt hat, aber wir können noch mehr unterbringen. Wir wäre es z.B. mit einem Textwechsel bei der Löschen-Funktion. Hier könnte doch so etwas wie "Wirklich?" erscheinen, damit der Anwender aufgeschreckt wird. Immerhin wird auch das über die Highlighting-Flags wunderbar unterstützt. Auf geht's!

Dazu brauchen wir zunächst eine neue IntuiText-Struktur. Am besten verändern wir damit gleichzeitig die Farbe:

```
struct IntuiText LoeschenText =
   {
   5, 1, JAM2, 1, 1, &Font, (UBYTE *)"Löschen  ", NULL
   };

struct IntuiText LoeschenTextII =
   {
   5, 1, JAM2, 1, 1, &Font, (UBYTE *)"WIRKLICH?", NULL
   };
```

Struktur 3.23: IntuiTexte zu "Löschen"

Sie können natürlich auch die Position oder den Zeichensatz ändern. Hier sind stehen Ihnen alle Möglichkeiten offen. Probieren Sie ruhig einige Varianten aus!

Vorher dürfen wir aber nicht vergessen, die MenuItem-Struktur dahingehend zu verändern. Zum einen muß das Flag HIGHCOMP durch HIGHIMAGE ersetzt werden, und danach muß noch der Pointer *SelectFill* mit dem Zeiger auf unseren Text versehen werden:

```
struct MenuItem Loeschen =
{
   &Programmende,
   1, 26,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHIMAGE | COMMSEQ,
```

```
    0,
    (APTR)&LoeschenText,
    (APTR)&LoeschenTextII,
    0x44,
    NULL,
    0
};
```

Struktur 3.24: MenuItem-Struktur "Löschen"; verbessert

Ich habe die Kommando-Sequenz beibehalten, weil das immer am praktischsten ist! Testen Sie doch jetzt mal die Menüauswahl! Zur grafischen Gestaltung Ihrer Menüs möchte ich Ihnen raten, auch auf die anderen beiden Highlighting-Modi zurückzugreifen. Warum sollte Ihr Menü nicht alle Punkte mit einem Kasten umrahmen? Auch das Flag HIGHNONE läßt sich verwenden. Gibt es z.B. in einem Menü viele Funktionen, so wäre es gut, wenn diese auch noch gegliedert wären. Ich habe dazu einfach die Select-Box der Programmende-Funktion etwas tiefer gerückt. So entsteht schon eine thematische Trennung. Reicht Ihnen das aber nicht, so können Sie einfach einen Grafik- oder Text-Menüpunkt mit einem Strich verwenden und diesen nicht abfragen. Für den Benutzer darf dieser Punkt natürlich nicht anwählbar sein, und deshalb setzen Sie das HIGHNONE-Flag! Vermeiden Sie außerdem von Anfang an eine Fehlbedienung! Denn es ist unsinnig, daß die Speicher-Funktion schon aktiv, d.h. anwählbar ist, wenn noch nichts zum Abspeichern geschaffen wurde. Daraufhin sollten Sie alle Menüs und Menüpunkte überprüfen, damit ersparen Sie Programm und Benutzer viel Arbeit, die durch Requester entsteht (ITEMENABLE-Flag weglassen).

```
struct MenuItem Speichern =
{
    &Loeschen,
    1, 14,
    170, 10,
    ITEMTEXT | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&SpeichernText,
    NULL,
    0x53,
    NULL,
    0
};
```

Struktur 3.25: MenuItem-Struktur "Speichern"; verbessert

3.8.2.4 Komplexere Verschachtelungen: Untermenüs

Die Gestaltung ist noch nicht ausgereizt, aber unser Weg soll uns tiefer in die Menü-Leiste führen. Dort treffen wir auf die Untermenüs! Diese Menüs erscheinen, wenn wir in einem Menü einen Menüpunkt ausgewählt haben. Es ist durchaus möglich, daß dieser nur ein Oberbegriff war und noch eine weitere Auswahl erforderlich ist.

Wie kommen wir nun zu unserem Untermenü? Dafür sieht man sich am besten noch einmal die MenuItem-Struktur an. In ihr finden wir einen Pointer auf einen *SubItem*, und genau das ist der Eingang zum Untermenü. Fügt man hier den Zeiger auf eine neue MenuItem-Kette ein, ist das der Anfang zum neuen Untermenü.

Mit dieser Methode ließe sich z.B. das Datei-Menü ergänzen. Ist das Datei-Menü für eine Textverarbeitung gedacht, dann muß zuvor angegeben werden, welchen Text-Typ das Programm laden soll, und dafür benutzen wir das Untermenü. Es stehen drei Typen zur Auswahl: Text (das Textformat der Textverarbeitung), ASCII (Austauschformat zwischen allen Computern) und IFF (genormtes Format vom Electronic Arts).

```
struct IntuiText TextText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"Text", NULL
   };

struct IntuiText ASCIIText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"ASCII", NULL
   };

struct IntuiText IFFText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"IFF", NULL
   };

struct MenuItem IFF =
{
   NULL,
   140, 22,
   100, 10,
```

```
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&IFFText,
   NULL,
   0x49,
   NULL,
   0
};

struct MenuItem ASCII =
{
   &IFF,
   140, 12,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ASCIIText,
   NULL,
   0x41,
   NULL,
   0
};

struct MenuItem Text =
{
   &ASCII,
   140, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&TextText,
   NULL,
   0x54,
   NULL,
   0
};

struct MenuItem Laden =
{
   &Speichern,
   1, 2,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&LadenText,
   NULL,
   0,
   &Text,
   0
};
```

Struktur 3.26: Untermenü Laden

Das gleiche Untermenü muß nun auch noch bei der Speicher-Funktion ergänzt werden.

Sie müssen nur die MenuItem-Struktur mit dem Pointer auf die Untermenüs ergänzen und gleichzeitg die Kommando-Sequenz entfernen:

```
struct MenuItem Speichern =
{
   &Loeschen,
   1, 14,
   170, 10,
   ITEMTEXT | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&SpeichernText,
   NULL,
   0,
   &SpeicherText,
   0
};
```

Struktur 3.27: MenuItem-Struktur "Speichern" verbessert

Leider müsssen auch die SubItems neu definiert werden, weil die alten Kommando-Sequenzen nicht doppelt verwendet werden können:

```
struct MenuItem SpeicherIFF =
{
   NULL,
   140, 22,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&IFFText,
   NULL,
   0x46,
   NULL,
   0
};

struct MenuItem SpeicherASCII =
{
   &SpeicherIFF,
```

```
    140, 12,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&ASCIIText,
    NULL,
    0x43,
    NULL,
    0
};

struct MenuItem SpeicherText =
{
    &SpeicherASCII,
    140, 2,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&TextText,
    NULL,
    0x51,
    NULL,
    0
};
```

Struktur 3.28: Untermenü "Speichern"

Die IntuiText-Strukturen können wir ohne Bedenken ein zweites Mal verwenden, da sie immer nur einmal zur Zeit benutzt wird!

Für den Menüpunkt "Löschen" hatte ich mir vorgestellt, daß wir dort die Auswahl zwischen einer Datei auf der Diskette und dem Text (Bild) im Speicher in einem Untermenü bringen. Aber ich bin sicher, daß Sie das alleine schaffen, ohne daß es abgedruckt werden muß.

3.8.2.5 Eine Anleitung zum Design-Menü

Sie sind jetzt endlich an der Reihe und sollen ein einfaches Menü selbständig programmieren. Die einzige Hilfe von mir wird die Beschreibung des Menüs sein. Sie können dann nach Herzenslust die Strukturen definieren und verketten. Also machen Sie sich an die Arbeit!

Das Menü soll aus sechs Schrifttypen bestehen. Diese können natürlich alle auch über einen Command-Key aufgerufen werden. Folgende Schriftarten gehören zum Standard: "Normal", "Kursiv", "Fett", "Invers", "80 Zeichen" und "60 Zeichen". Jeder Menüpunkt muß als IntuiText-Struktur mit eigener TextAttr-Struktur definiert werden. In der TextAttr-Struktur geben Sie dann immer die jeweilige Schriftart an. Somit hat der Benutzer später eine leichte Auswahl und kann sich sofort ein Bild vom Textaussehen machen.

Wenn das Menü aktiviert wird, soll der Anwender auch sofort erkennen, welche Schrifttypen momentan angewählt sind. Dafür bietet sich das Checkmark an. Setzen Sie dazu das CHECKIT-Flag im Flags-Wert. Für das Checkmark selbst, den Haken, müssen im IntuiText noch Lücken vor den Text gestellt werden. Zwei Leerzeichen dürften ausreichen. Dies empfiehlt sich auch, falls das MENUTOGGLE-Flag verwendet wird, weil damit zuerst eine Schriftart eingeschaltet und bei erneutem Anwählen wieder ausgeschaltet wird.

Als nächstes müssen wir uns noch um den MutualExclude kümmern. Damit können wir einstellen, daß bestimmte Menüpunkte, die auch abgehakt werden können, nicht mehr abgehakt sind, wenn ein bestimmter anderer abgehakt wurde. Das hört sich vielleicht etwas kompliziert an, ist aber sehr einfach und nützlich. Nehmen wir ein konkretes Beispiel: Wenn der Benutzer die Schriftart "Normal" anwählt, beabsichtigt er, daß alle anderen Schrifttypen wieder deaktiviert werden. Wir erreichen das durch MutualExclude. Gleiches gilt auch für die beiden Schriftbreiten. Nur eine kann benutzt werden, die andere muß jeweils ausgeschlossen werden.

Die MutualExclude-Methode arbeitet folgendermaßen: Wir haben einen LONG-Wert zur Verfügung, von dem jedes Bit einen Menüpunkt repräsentiert. Alle Menüpunkte, die wieder ausgeschaltet werden sollen, wenn unser bestimmter Menüpunkt angewählt wird, sind dann mit einem gesetzten Bit zu markieren. Dabei entspricht das Bit 0 dem ersten Menüpunkt, das Bit 1 dem zweiten Menüpunkt usw.

3.8.2.6 Grafik-Menü mit Grafiken

Nachdem im Design-Menü die Schriftart gewählt werden kann, werden Sie im "Grafik"-Menü die Auswahl mehrerer Grafiken haben und gleichzeitig eine Farbauswahl. Dieses Menü würde gut in ein Grafikprogramm passen.

Das Menü setzt sich aus zwei Menüpunkten zusammen. Der erste, Pinsel, hat ein Untermenü mit zwei weiteren Punkten: einen Pinsel mit feiner Spitze und einen mit dicker. (Für Ergänzungen bin ich Ihnen jederzeit dankbar.) Der zweite Menüpunkt, Palette, weist 8 Untermenüpunkte auf, die zum Einstellen der Zeichenfarbe dienen. Er ist sowohl für ein Grafikprogramm als auch für eine Textverarbeitung zu gebrauchen.

Hier die MenuItem-Strukturen für das Pinsel-Menü:

```
struct MenuItem PinselII =
{
   NULL,
   90, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PinselIIImage,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem PinselI =
{
   &PinselII,
   102, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PinselIImage,
   NULL,
   0,
   NULL,
   0
};
```

Struktur 3.29: Untermenü "Pinsel"

Und die MenuItem-Struktur für das Hauptmenü:

```
struct IntuiText PinselText =
   {
   0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Pinsel", NULL
   };

struct MenuItem Pinsel =
{
   &Farben,
   2, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PinselText,
   NULL,
   0,
   &PinselI,
   0
};
```

Struktur 3.30: MenuItem "Pinsel"

Fast hätte ich die beiden Image-Strukturen mit den Werten vergessen:

```
USHORT PinselIDaten[] =
   {
   0x00, 0x00,
   0x01, 0xF0,
   0x00, 0x00,
   0x00, 0x00,
   };

USHORT PinselIIDaten[] =
   {
   0x00, 0x00,
   0x01, 0xF0,
   0x01, 0xF0,
   0x00, 0x00,
   };

struct Image PinselIImage =
   {
   1, 1,
   8, 8,
   1,
   &PinselIDaten[0],
   2, 0,
   NULL
```

```
   };

struct Image PinselIIImage =
   {
   1, 1,
   8, 8,
   2,
   &PinselIIDaten[0],
   2, 0,
   NULL
   };
```

Struktur 3.31: Daten/Struktur Pinsel-Image

Nun folgen die Daten zum Farbauswahl-Untermenü. Wir definieren dazu eine ganz normale MenuItem-Struktur für den Menüpunkt. Im Untermenü wird jede Farbe durch eine IntuiText-Struktur vertreten. Aber nun erst zum Hauptmenü:

```
struct IntuiText FarbenText =
   {
   0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Farben", NULL
   };

struct MenuItem Farben =
{
   NULL,
   2, 14,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&FarbenText,
   NULL,
   0,
   &Farbe0,
   0
};
```

Struktur 3.32: MenuItem "Farben"

Für die Farben nehmen wir wie oben eine IntuiText-Struktur, die nur Leerzeichen enthält. Dafür wird die Hintergrundfarbe gewechselt, so daß Farbbalken entstehen. Ich zeige hier nur die ersten drei, alle anderen sind leicht zu konstruieren:

```
struct IntuiText Col0Text =
   {
   0, 0, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
```

```
   };

struct IntuiText Col1Text =
   {
   0, 1, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
   };

struct IntuiText Col2Text =
   {
   0, 2, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
   };

...
```

Struktur 3.33: IntuiTexte Menü "Farben"

Nach dem gleichen Prinzip erfolgt die Definition der MenuItem-Strukturen für das Untermenü:

```
...

struct MenuItem Farbe2 =
{
   &Farbe3,
   90, 26,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col2Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Farbe1 =
{
   &Farbe2,
   90, 14,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col1Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Farbe0 =
{
   &Farbe1,
```

```
    90, 2,
    50, 10,
    ITEMTEXT | ITEMENABLED | HIGHBOX,
    0,
    (APTR)&ColOText,
    NULL,
    0,
    NULL,
    0
};
```

Struktur 3.34: MenuItem-Strukturen Menü "Farben"

3.8.3 Die Abfrage der gesamten Menüleiste

Die Menüleiste dieses Kapitels stellt einen Universaltyp dar, da sie aus mehr als einem Menü, einem Menüpunkt und mehr als einem Untermenüpunkt besteht. Außerdem finden wir Menüpunkte, die Aktionen auslösen, genauso wie solche, die für eine bestimmte Eigenschaft stehen. Aus diesem Grund eignet sich die Menü-Leiste besonders für eine allgemeine Abfrage. Sie können die Unterroutine in jedem eigenen Programm verwenden, weil die Menü-Abfrage besonders flexibel ist.

Wie für jede Abfrage des IDCMP, müssen zuerst die richtigen Vorausssetzungen geschaffen werden. Hierfür setzen Sie bitte das MENUPICK-Flag in der Variable IDCMP des Windows.

Als nächstes testen wir in der bekannten Abfrageschleife auf eine Menünachricht:

```
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
   MessageClass = message->Class;
   Code = message->Code;

   ReplyMsg(message);
   switch (MessageClass)
         {
```

```
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;

            case MENUPICK    : Menu_Auswertung(Code);
                               break;
            }
         }
      }
```

Programmteil 3.11: Abfrageschleife Menütest

Haben wir eine Nachricht vom Typ MENUPICK erhalten, so wird in die Funktion Menu_Auswertung verzweigt, wo wir einen Punkt nach dem anderen prüfen können.

```
Menu_Auswertung(Menunummern)
USHORT Menunummer;
{
   USHORT Menu, MenuItem, SubItemm, NextMenu;
   struct MenuItem *MenuPunkt;

   Menu     = MENUNUM(Menunummern);
   MenuItem = ITEMNUM(Menunummern);
   SubItem  = SUBITEM(Menunummern);

   MenuPunkt = ItemAddress(&Datei, Menunummern);

   switch(Menu)
   {
      case 0 : /* Datei - Menü */

         break;

      case 1 : /* Design - Menü */

         break;

      case 2 : /* Grafik - Menü */

         break;
   }

   NextMenu = MenuPunkt->NextSelect;
   if (NextMenu) Menu_Auswertung(NextMenu);

}
```

Funktion 3.11: Menü-Auswertung

Hier haben wir zuerst die Unterscheidung nach den jeweiligen Menüs. Außerdem berechnet das Programm anhand der Funktion ItemAddress() die Adresse der MenuItem-Struktur. Somit können wir weitere Daten auslesen. Dies wird auch am Ende der neuen Funktion genutzt, wenn nämlich der Wert *NextSelect* auf eine neue Menünummer überprüft wird. Somit unterstützt diese Routine auch das Auswählen von mehreren Menüpunkten!

Gehen wir die Abfragen der Reihe nach durch, zuerst das Datei-Menü. Wir unterscheiden vier Menüpunkte:

```
switch(MenuItem)
   {
   case 0 : /* Laden */

      break;

   case 1 : /* Speichern */

      break;

   case 2 : /* Löschen */
      Delete();
      break;

   case 3 : /* Programmende */
      Ende = FALSE;
      break;
   }
```

Programmteil 3.12: Austesten der Menüs

Die letzten beiden Punkte lassen es zu, sofort eine Aktion auszulösen. Nur bei den ersten muß weiter geprüft werden:

```
case 0 : /* Laden */

   switch(SubItem)
      {
      case 0 : /* Text */
         Load(TEXT);
         break;
      case 1 : /* ASCII */
         Load(ASCII);
         break;
      case 2 : /* IFF */
         Load(IFF);
         break;
```

```
      }
   break;

case 1 : /* Speichern */

   switch(SubItem)
      {
      case 0 : /* Text */
         Save(TEXT);
         break;
      case 1 : /* ASCII */
         Save(ASCII);
         break;
      case 2 : /* IFF */
         Save(IFF);
         break;
      }
   break;
```

Programmteil 3.13: Austesten der Untermenüs "Laden" und "Speichern"

Alle Überprüfungen eines Untermenüs rufen die gleiche Funktion auf. Allerdings werden dieser jedesmal andere Parameter übergeben. In diesem Fall sind es selbst definierte Flags. So sagt TEXT dem Unterprogramm Load() oder Save(), daß es die eigenen Kommandosequenzen erwarten oder schreiben soll. ASCII unterbindet jede Formatangabe, und IFF schreibt die allgemein definierten Formatangaben vor.

Beim zweiten Menü müssen wir einen anderen Weg der Abfrage beschreiten. Wir haben hier keine Untermenüs, sondern nur ein Hauptmenü, in dem mehrere Eigenschaften stehen, von denen fast alle miteinander kombinierbar sind. Nur die Schriftbreiten vertragen sich nicht. Nimmt man an, daß das Programm eine Variable für den Schrifttyp und eine für die Zeichenbreite verwaltet, so könnte die folgende Abfrage sinnvoll sein:

```
switch(MenuItem)
   {
   case 0 : /* Normal */
      TextStyle = NULL;
      break;

   case 1 : /* Kursiv */
      TextStyle ^= KURSIV;
      break;
```

```
case 2 : /* Fett */
   TextStyle ^= FETT;
   break;

case 3 : /* Invers */
   TextStyle ^= INVERS;
   break;

case 4 : /* 80 Zeichen */
   TextFont = 80L;
   break;

case 5 : /* 60 Zeichen */
   TextFont = 60L;
   break;
}
```

Programmteil 3.14: Austesten des Menüs "Design"

Im dritten Menü kann wieder die einfache Abfrage benutzt werden. Wir unterscheiden hier zwei Klassen. Die erste beschäftigt sich mit den "Pinseln". Man findet in der Variablen *PinselNr* später dann die Nummer des ausgewählten Pinsels vor. Das gleiche in der Farbauswahl: In der Variablen *FarbNr* steht nach der Menüauswahl für das Programm die Farbnummer.

```
switch(MenuItem)
   {
   case 0 : /* Pinsel */
      PinselNr = SubItem + 1;
      break;

   case 1 : /* Farbe */
      FarbNr = SubItem;
      break;
   }
```

Programmteil 3.15: Austesten Untermenüs "Grafik"

3.8.4 Die Arbeit mit Source-Code-Utilities

Was versteht man eigentlich unter einem Source-Code-Utility?

Wir bezeichnen damit Programme, die dem Programmierer bei seiner Arbeit am Programmtext helfen. Das sind meist Pro-

gramme, die aufwendige Arbeiten erledigen, die sehr schwer theoretisch zu bearbeiten sind, weil viel von der Grafik abhängt. Sie erfordern aber keine schwierigen Dateneingaben, sondern arten mehr in Schreibarbeit aus. Diese Schreibarbeit nimmt einem das Source-Code-Utility ab. Es besorgt alle Definitionen, die eindeutig sind, und erlaubt es dem Entwickler, diese nachträglich zu verändern.

Gerade bei der Programmierung von Intuition erweisen sich solche Utilities als sehr hilfreich. Sie werden gemerkt haben, daß bereits für ein neues Window einige Parameter notwendig waren. Noch komplexer wurde es bei den Gadgets, die immer mit einer, zwei oder drei Strukturen erweitert wurden. Bei den Requestern merkte man, was für Fingerarbeit geleistet werden mußte.

Nun sind wir aber an einem Punkt angelangt, an dem es nicht mehr tragbar ist, alle Strukturen selber zu schreiben. Da Menüs nur in ein Programm eingebaut werden, wenn es viele Funktionen gibt, die darüber angeboten werden können, ergibt es sich von selbst, daß wir pro Menüpunkt mindestens zwei Strukturen brauchen. Es ist nicht selten, daß ein Programm über 50 Menüpunkte verwaltet. BECKERtext von DATA BECKER hat z.B. derzeit 5 Menüs mit insgesamt 63 Menüpunkten und 63 Untermenüpunkten. Sie müßten dafür 126 MenuItem-Strukturen und genausoviele IntuiText-Strukturen definieren, hätten dann aber nur die rohen Menüs. Diese Arbeit kann uns ein Programm abnehmen.

3.8.4.1 Die Bedienung von PowerWindows

PowerWindows ist ein Programm, das verspricht, dem Programmierer die Arbeit abzunehmen. Es ist aber nicht nur für die Programmierung von Menü-Strips vorgesehen. Es unterstützt ebenfalls die Handhabung von Windows und Gadgets. Es werden demnach die drei wichtigsten Grundelemente Intuitions verarbeitet. Aus den Gadgets kann der Programmierer dann z.B. selbst Requester bauen.

Kommen wir nun zu der Bedienung dieses Programms. Nach dem Laden erscheint ein großes Window, zu dem wir nur die Menüleiste finden. Hier können wir nur wählen zwischen dem Laden eines alten Windows, dem Erstellen eines neuen Windows oder dem Verlassen des Programms. Natürlich wählt man ein neues Window.

Das Window erscheint sofort und eine neue Menüleiste tut sich dem Entwickler auf: Wir haben jetzt fünf Menüs, von denen drei die Intuition-Elemente betreffen. Die anderen werden für File-Arbeit und allgemeine Einstellungen benutzt.

In der File-Arbeit ist es möglich, einen Assembler-Code, einen C-Code oder einen programminternen Code zu erzeugen. Der letzte ist dafür da, ein Projekt noch weiter behandeln zu können, da es in den anderen beiden Formen nicht mehr für PowerWindows zu verarbeiten ist.

Das Preferences-Menü erlaubt einige Einstellungen zum Quelltext. So kann dieser mit Kommentaren (Standard) versehen werden, und auch der Tabulator im Text ist einstellbar. Weiterhin können Sie das Programm beauftragen, darauf zu achten, daß sich z.B. keine Gadgets überlappen.

3.8.4.2 Eine Menüleiste erstellen "lassen"

Machen wir uns jetzt daran, die ersehnte Menüleiste aufzubauen. Dafür sehen wir ins Menü "Menü" und entdecken nur einen Menüpunkt, der anwählbar ist. Wir dürfen unserem Window ein ganzes Menü anfügen. Das wählen wir aus, und ein Requester erscheint. In diesen kann der Name des neuen Menüs eingegeben und mit OK bestätigt werden. Diese Bestätigung stört in der späteren Arbeit noch unangenehm, denn der Return-Tasten-Druck im String-Gadget sollte nach Amiga-Konventionen eigentlich genügen.

Klickt man jetzt sein kleines Window an und sieht in die Menüleiste, findet man wirklich sein erstes Menü. Sogar ein Menüpunkt wurde schon eingesetzt, er ist allerdings ausgeschal-

tet. Jetzt wird ein weiteres Problem sichtbar, das sich leider nicht beheben läßt. Zum Ausführen eines Befehls von PowerWindows muß immer wieder das PowerWindow-Fenster angeklickt werden und zum Betrachten der Menüleiste das eigene Fenster.

Geplant ist, die Menüleiste, die wir zu Anfang dieses Kapitels zusammen programmiert haben, auch mit PowerWindows zu erstellen. Damit haben wir einen guten Vergleich. Ich schaffe deshalb erst einmal drei Menüs mit den Titeln "Datei", "Design" und "Grafik".

Dafür stehen noch mehr Funktionen als nur "Append a menu" zur Verfügung. Hier soll aber keine Programmbeschreibung abgedruckt werden, sondern nur eine kurze Kommentierung der Arbeitsschritte.

Sind die drei Menüs fertig, kommen wir zu den Menüpunkten. Wählen Sie dafür die Funktion "Work on MenuItems for ..." an. In einem Untermenü können Sie das Menü bestimmen, dessen Menüpunkte Sie editieren wollen. Nehmen wir zuerst "Datei".

Eine andere Menüleiste hat sich uns zur Verfügung gestellt. Sie dient ausschließlich der Bearbeitung der MenuItems. Fügen wir einen nach dem anderen an: "Laden", "Speichern", "Löschen" und "Programmende". Sehen Sie sich einmal das Menü an. Es fällt auf, daß die Positionierung automatisch gemacht wird und leider auch nicht beeinflußt werden kann.

Kommen wir jetzt zu den Untermenüs. Zuerst das "Laden"-Menü. Gehen Sie über das Menü in "Work on subitems for ..." und stellen Sie "Laden" ein, wieder erscheint eine neue Menüleiste, die aber der vorangegangenen sehr ähnlich ist. Hier fügen Sie bitte die Punkte "Text", "ASCII" und "IFF" ein. Machen Sie danach das gleiche für das "Speichern"-Menü.

Beim Betrachten des bisher erstellten Menüs fällt etwas auf. Wir können die Position des Untermenüs nicht bestimmen! Außerdem wird das Untermenü nicht gerade großzügig verwaltet. Alle

diese Mängel können nachher am Quelltext behoben werden, doch eigentlich wollte man sich Arbeit sparen.

Wählen Sie jetzt den Menüpunkt "Options for" im SubItem- oder Item-Menü an. Dort können Sie alle Flags und Ergänzungen einstellen. Die Handhabung eines Kommando-Keys wird so einfach, und die Farbeinstellungen sind auch vorbildlich gelöst.

Das erste Menü ist somit abgeschlossen. Sie können natürlich noch Kommando-Sequenzen und anderes einfügen, wir gehen aber vorher schon an das zweite Menü. Dort sind sechs Menüpunkte einzurichten, die Ihnen ja bekannt sind. Denken Sie daran, für jeden eine Taste und ein Checkmark vorzusehen. Allerdings bekommen wir Probleme, wenn wir außerdem Kursivschrift oder die anderen Typen darstellen wollen. Hier muß die Änderung zu einem späteren Zeitpunkt durchgeführt werden. Auch MutualExclude wird nicht vom Programm eingestellt.

Im letzten Menü gibt es Probleme mit den Grafiken. Diese werden von PowerWindows nicht unterstüzt. Allerdings kann man sich dabei mit Texten helfen, die später ersetzt werden. Es ist verständlich, denn eine Grafikverwaltung würde erheblich mehr Aufwand bedeuten. Die Farbauswahl dürfte eigentlich keine Schwierigkeiten machen.

Damit ist die Arbeit abgeschlossen. In dieser Beschreibung finden Sie zwar nicht alle Schritte, aber die Möglichkeiten sind Ihnen vertraut. Sie werden im folgenden Quelltext auch noch genau die Einstellungen ablesen können.

3.8.4.3 Der entstandene Quelltext

PowerWindows erlaubt das Erzeugen von Quelltexten für Assembler- und C-Programme. Uns interessiert natürlich nur das letztere. Nach dem Abspeichern im Format von PowerWindows habe ich das oben beschriebene Menü mit Kommentaren und einem Tabulator von 3 Zeichen als C-Quelltext schreiben lassen. Sehen Sie, was daraus geworden ist:

```
struct IntuiText IText1 = {
   3,7,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem8 = {
   NULL,   /* next SubItem structure */
   34,62, /* XY of Item hitbox relative to
             TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText1,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText2 = {
   3,6,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem7 = {
   &SubItem8,   /* next SubItem structure */
   34,52,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText2,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText3 = {
   3,5,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem6 = {
   &SubItem7,   /* next SubItem structure */
```

```
   34,42,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText3,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText4 = {
   3,4,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem5 = {
   &SubItem6,   /* next SubItem structure */
   34,32,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText4,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText5 = {
   3,3,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem4 = {
   &SubItem5,   /* next SubItem structure */
   34,22,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText5,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};
```

```
struct IntuiText IText6 = {
   3,2,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem3 = {
   &SubItem4,   /* next SubItem structure */
   34,12,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText6,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText7 = {
   3,1,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem2 = {
   &SubItem3,   /* next SubItem structure */
   34,2,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText7,  /* Item render  (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText8 = {
   3,0,JAM2,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "      ",   /* pointer to text */
   NULL   /* next IntuiText structure */
};
```

```
struct MenuItem SubItem1 = {
   &SubItem2,   /* next SubItem structure */
   34,-8,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   68,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX+CHECKED,  /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText8,   /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText9 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Farbe",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem2 = {
   NULL,   /* next MenuItem structure */
   0,11,   /* XY of Item hitbox relative to
                                 TopLeft of parent hitbox */
   49,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText9,   /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   &SubItem1,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText10 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "!!!",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem10 = {
   NULL,   /* next SubItem structure */
   59,0,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   25,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText10,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
```

```
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText11 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "???",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem9 = {
   &SubItem10,   /* next SubItem structure */
   34,0,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   25,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText11,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText12 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Pinsel",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem1 = {
   &MenuItem2,   /* next MenuItem structure */
   0,0,   /* XY of Item hitbox relative
             to TopLeft of parent hitbox */
   49,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText12,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   &SubItem9,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct Menu Menu3 = {
   NULL,   /* next Menu structure */
   135,0,   /* XY origin of Menu hit box
               relative to screen TopLeft */
   66,0,   /* Menu hit box width and height */
   MENUENABLED,   /* Menu flags */
```

```
   "Grafik",   /* text of Menu name */
   &MenuItem1   /* MenuItem linked list pointer */
};

struct IntuiText IText13 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "60 Zeichen",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem8 = {
   NULL,   /* next MenuItem structure */
   0,55,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP, /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText13,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "6",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText14 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "80 Zeichen",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem7 = {
   &MenuItem8,   /* next MenuItem structure */
   0,44,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP+CHECKED,
             /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText14,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "8",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText15 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Invers",   /* pointer to text */
```

```
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem6 = {
   &MenuItem7,   /* next MenuItem structure */
   0,33,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
            /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText15,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "V",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText16 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Fett",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem5 = {
   &MenuItem6,   /* next MenuItem structure */
   0,22,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
            /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText16,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "T",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText17 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Kursiv",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem4 = {
   &MenuItem5,   /* next MenuItem structure */
   0,11,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
```

```
   CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
            /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText17,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "K",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText18 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   20,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Normal",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem3 = {
   &MenuItem4,   /* next MenuItem structure */
   0,0,   /* XY of Item hitbox relative
             to TopLeft of parent hitbox */
   140,10,   /* hit box width and height */
 CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP+CHECKED,
            /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText18,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "N",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct Menu Menu2 = {
   &Menu3,   /* next Menu structure */
   63,0,   /* XY origin of Menu hit box
              relative to screen TopLeft */
   66,0,   /* Menu hit box width and height */
   MENUENABLED,   /* Menu flags */
   "Design",   /* text of Menu name */
   &MenuItem3   /* MenuItem linked list pointer */
};

struct IntuiText IText19 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Programmende",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem12 = {
   NULL,   /* next MenuItem structure */
```

```
   0,33,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   137,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText19,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "E",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText20 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Löschen",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem11 = {
   &MenuItem12,   /* next MenuItem structure */
   0,22,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   137,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText20,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "D",   /* alternate command-key */
   NULL,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText21 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "IFF",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem13 = {
   NULL,   /* next SubItem structure */
   122,12,   /* XY of Item hitbox relative
                to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText21,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "F",   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
```

```
};

struct IntuiText IText22 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "ASCII",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem12 = {
   &SubItem13,   /* next SubItem structure */
   122,2,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText22,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "C",   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText23 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Text",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem11 = {
   &SubItem12,   /* next SubItem structure */
   122,-8,   /* XY of Item hitbox relative
                to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText23,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "X",   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText24 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Speichern",   /* pointer to text */
   NULL   /* next IntuiText structure */
};
```

```
struct MenuItem MenuItem10 = {
   &MenuItem11,   /* next MenuItem structure */
   0,11,   /* XY of Item hitbox relative
              to TopLeft of parent hitbox */
   137,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText24,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   &SubItem11,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText25 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "IFF",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem16 = {
   NULL,   /* next SubItem structure */
   122,12,   /* XY of Item hitbox relative
                to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText25,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "I",   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText26 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "ASCII",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem15 = {
   &SubItem16,   /* next SubItem structure */
   122,2,   /* XY of Item hitbox relative
               to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText26,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "A",   /* alternate command-key */
```

```
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText27 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Text",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem SubItem14 = {
   &SubItem15,   /* next SubItem structure */
   122,-8,   /* XY of Item hitbox relative
                to TopLeft of parent hitbox */
   81,10,   /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText27,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   "T",   /* alternate command-key */
   NULL,   /* no SubItem list for SubItems */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct IntuiText IText28 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,   /* XY origin relative to container TopLeft */
   NULL,   /* font pointer or NULL for defaults */
   "Laden",   /* pointer to text */
   NULL   /* next IntuiText structure */
};

struct MenuItem MenuItem9 = {
   &MenuItem10,   /* next MenuItem structure */
   0,0,   /* XY of Item hitbox relative
             to TopLeft of parent hitbox */
   137,10,   /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,   /* Item flags */
   0,   /* each bit mutually-excludes a same-level Item */
   (APTR)&IText28,  /* Item render (IntuiText or Image or NULL) */
   NULL,   /* Select render */
   NULL,   /* alternate command-key */
   &SubItem14,   /* SubItem list */
   0xFFFF   /* filled in by Intuition for drag selections */
};

struct Menu Menu1 = {
   &Menu2,   /* next Menu structure */
   0,0,   /* XY origin of Menu hit box
             relative to screen TopLeft */
   57,0,   /* Menu hit box width and height */
   MENUENABLED,   /* Menu flags */
```

```
   "Datei",   /* text of Menu name */
   &MenuItem9   /* MenuItem linked list pointer */
};

#define MenuList Menu1

struct NewWindow NewWindowStructure = {
   275,85,   /* window XY origin relative to TopLeft of screen */
   150,50,   /* window width and height */
   0,1,   /* detail and block pens */
   NULL,   /* IDCMP flags */
   NULL,   /* other window flags */
   NULL,   /* first gadget in gadget list */
   NULL,   /* custom CHECKMARK imagery */
   "Your new window",   /* window title */
   NULL,   /* custom screen */
   NULL,   /* custom bitmap */
   5,5,   /* minimum width and height */
   640,200,   /* maximum width and height */
   WBENCHSCREEN   /* destination screen type */
};

/* end of PowerWindows source generation */
```

Struktur 3.35: PowerWindows Menu-Strip

3.8.4.4 So verarbeitet man den Quelltext weiter

Zugegeben, der Text ist sehr lang, aber PowerWindows hat das daraus gemacht. Leider läßt es sich nicht einstellen, daß z.B. für jede Struktur eine Zeile mit allen Daten generiert wird.

Wir wollen uns als letztes ansehen, wie man den Text weiterbearbeiten kann. Zusätzlich zur Menu-Strip hat uns PowerWindows noch eine NewWindow-Struktur erstellt. Wir hätten dieses Window natürlich auch einstellen können, sogar noch komfortabler als die Menüs. Wir bleiben jedoch bei den Menüs.

Es wird immer eine IntuiText-Struktur zusammen mit einer MenuItem-Struktur definiert. Diese sind aus unerklärlichen Gründen durchnumeriert. Möchte man jetzt durch das Programm welche ergänzen, müssen die Nummern geändert werden. Leider geben diese Nummern nicht viel her an Informationen. Schöner wäre es gewesen, wenn die Namen der Menüpunkte dort stehen

würden. Schließlich kann man mit diesen mehr verbinden, und das Einsetzen neuer Menüpunkte fällt leichter.

Gut anrechnen kann man dem Programm die Kommentare, obwohl sie in Englisch sind. Auch bei späteren Änderungen weiß man sofort, wofür der Parameter stand. Das ist besonders wichtig bei Strukturen, die viele Null-Parameter haben. Dann vergißt man doch leicht, welche Bedeutung damit verknüpft war. Man muß zugeben, daß die Koordinatenauswahl ziemlich eigenwillig ist. Da wird auf "Pixel und Pfennig" gerechnet. Sie müssen später für mehr Platz systematisch alle Strukturen verändern. Viel Spaß!

Beim ersten Untermenü des dritten Menüs (Grafik) müssen noch die beiden Grafiken eingesetzt werden. Entfernen Sie dafür die IntuiText-Strukturen, und setzen Sie die Image-Strukturen ein. Nicht vergessen, das Flag ITEMTEXT zu entfernen!

Hinweis: Als abschließende Bemerkung möchte ich mir erlauben, Ihnen zu raten, möglichst wenig Änderungen auf einmal durchzuführen. Es hat sich nämlich in der Praxis gezeigt, daß hier und dort immer ein Fehler auftritt und dieser dann später sehr schwer zu finden ist. Die einfachste Methode ist, den ganzen Programmtext auszudrucken und dann die geänderten Stellen mit Textmarkern zu markieren. Ist die Operation gelungen, können Sie neue Änderungen vornehmen. Aber immer zwischendurch compilieren, damit Sie das Ergebnis vor Augen haben.

3.9 Kontakt über das Console.Device

Beim Arbeiten mit dem IDCMP fällt auf, daß nur Daten empfangen werden können. Man kann zwar selektieren, welche Daten gesendet werden sollen, ist jedoch immer auf das Empfangen beschränkt.

Der IDCMP bietet für die Arbeit zwei Tastaturabfragen an. Die RAWKEY-Abfrage sendet immer den Tastatur-Code, ohne ihn auf irgendeine Art zu verändern. Die VANILLAKEY-Abfrage

übersetzt den Tastatur-Code nach der Tastaturtabelle. Somit kann sich das Programm jeder internationalen Tastaturbelegung anpassen.

Es bleibt aber immer noch das Problem, daß keine Daten gesendet werden können. Um auch dies zu ermöglichen, müssen wir auf das Console.Device zurückgreifen. Das Console.Device ist ein eigenständiges Gerät, mit dem wir Daten austauschen können. Es bietet sowohl den Datenempfang auf RAWKEY- als auch auf VANILLAKEY-Art des IDCMP. Außerdem bearbeitet Console.Device die Ausgabe. Es können Texte ausgegeben werden, die unter Beachtung der Window-Maße bearbeitet werden. Damit ist z.B. gewährleistet, daß kein Wort am Zeilenende abgeschnitten wird.

Weiterhin arbeitet das Console.Device mit vielen Formatierungshilfen, es scrollt unsere Textausgabe, bearbeitet den Text grafisch und erstellt Nachrichten über herrschende Bedingungen.

3.9.1 Aufbau der Kommunikationsleitungen

Wie für jedes Device, so müssen wir auch für das Console.Device erst einen Read- und einen Write-Port aufbauen, über die wir dann mit IOStandardMessages kommunizieren können. Zum Erstellen der Ports und der StdIOs benötigen wir zwei Funktionen, die uns Exec zur Verfügung stellt. CreatePort() und CreateStdIO() rufen wir in der Open_All()-Routine auf.

Zuerst der Port zum Schreiben von Daten:

```
ConsoleWritePort = CreatePort("wgb-con-dev.write", 0L);
if (!ConsoleWritePort)
   {
   printf("Beim neuen WritePort hat etwas nicht geklappt!\n");
   Close_All();
   exit(FALSE);
   }

ConsoleWriteMsg  = CreateStdIO(ConsoleWritePort);
if (!ConsoleWriteMsg)
   {
   printf("CreateStdIO für Write wollte nicht so recht!\n");
```

```
   Close_All();
   exit(FALSE);
   }
```

Programmteil 3.15: Daten schreiben

Das gleiche zum Lesen von Daten:

```
ConsoleReadPort = CreatePort("wgb-con-dev.read", 0L);
if (!ConsoleReadPort)
   {
   printf("Beim neuen ReadPort hat etwas nicht geklappt!\n");
   Close_All();
   exit(FALSE);
   }

ConsoleReadMsg  = CreateStdIO(ConsoleReadPort);
if (!ConsoleReadMsg)
   {
   printf("CreateStdIO für Read wollte nicht so recht!\n");
   Close_All();
   exit(FALSE);
   }
```

Programmteil 3.16: Daten lesen

Nachdem die beiden Datenleitungen stehen, kann das Device geöffnet werden. Zuvor müssen aber zwei Felder der IOStdRequest-Struktur initialisiert werden. Dies ist unüblich für das Öffnen eines Device, wird aber beim Console.Device benötigt!

```
ConsoleWriteMsg->io_Data   = (APTR)FirstWindow;
ConsoleWriteMsg->io_Length = sizeof(*FirstWindow);

conDevErr = OpenDevice ("Console.device", 0L,
                                ConsoleWriteMsg, 0L);
if (conDevErr)
   {
   printf("OpenDevice machte Schwierigkeiten!\n");
   Close_All();
   exit(FALSE);
   };

ConsoleReadMsg->io_Device  = ConsoleWriteMsg->io_Device;
ConsoleReadMsg->io_Unit    = ConsoleWriteMsg->io_Unit;
```

Programmteil 3.17: Device öffnen

Die ConsoleWriteMsg-Struktur wird damit auf ein Window fixiert, über das jetzt der Datenaustausch gehen soll. Nach dem Öffnen wird auch die ConsoleReadMsg-Struktur mit dem *Device* und der *Unit* versehen. Jetzt kann die Ein- und die Ausgabe gestartet werden.

Alle Tastatureingaben, die zu dem spezifizierten Window laufen, werden zum Console.Device geleitet und dort verarbeitet. Erst dann bekommt das Programm eine Nachricht vom Console.Device, die die aufbereiteten Tastatur-Codes enthält.

Will man irgendwelche Nachrichten an das Console.Device schicken, dann verwendet man die ConsoleWriteMsg-Struktur. Über sie richten wir Befehle und Anweisungen an das Device, wovon es einige gibt.

3.9.2 Wir empfangen die ersten Daten

Von dem Aufbau zur Datenleitung bis zum Empfangen von Daten ist es kein weiter Weg. Jetzt können wir schon die ersten Daten, d.h. Tastatureingaben, empfangen. Aber wie läuft das ab?

Dazu müssen wir an das Console.Device die Nachricht senden, daß wir über eine bestimmte Anzahl von Tastatureingaben informiert werden möchten. Diese Anweisung wird mit einer StandardRequest-Struktur losgeschickt. Das Programm wartet aber nicht, bis es eine Antwort erhält, denn in der Zeit könen ja andere Dinge erledigt werden.

Das einfachste wird es sein, wenn man für den Auftrag, Tasten zu lesen, eine Funktion schreibt. Diese Funktion wird in der Fachliteratur mit QueueRead() bezeichnet. Weil dies hier ein Fachbuch ist, werden wir es ebenso tun.

```
/*************************************
 *                                   *
 * Funktionen: Eingabe über ConsoleDev *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     24.10.1987  Verbesserte L&D *
 *                                   *
 * StdReq   Zeiger auf IOStdReq      *
 * buffer   Adresse des Textpuffers  *
 * length   Länge des Textpuffers    *
 *                                   *
 *************************************/


QueueRead(StdReq, buffer, Length)

struct IOStdReq *StdReq;
char            *buffer;
ULONG            length;

   {
   StdReq->io_Command = CMD_READ;
   StdReq->io_Data    = (APTR)buffer;
   StdReq->io_Length  = length;
   SendIO(StdReq);
   }
```

Funktion 3.11: Daten vom Console.Device lesen

Der Funktion übergeben Sie den Zeiger auf die IOStdReq-Struktur, die wir zum Schreiben eingerichtet haben. Außerdem übergeben Sie den Zeiger auf einen Puffer, in dem die empfangenen Daten später abgelegt werden, und zum Schluß noch die Länge des Puffers bzw. die Anzahl der Zeichen, die empfangen werden sollen.

```
QueueRead(consoleReadMsg, &Buffer[0], 1L);
```

Jetzt kann sich unsere Abfrageschleife "auf die Lauer legen" und darauf warten, daß über GetMsg(ConsoleReadPort) eine Erfolgsmeldung kommt. Dann tritt eine neue Abfrage in Aktion:

```
if (GetMsg(consoleReadPort))
   Write(consoleWriteMsg, &Buffer[0], 1L);
```

Es wäre besser, wenn man vor der Ausgabe das Zeichen noch auf Steuer-Codes testen würde, aber wir gehen zuerst den einfachen Weg und geben das Zeichen gleich wieder über das Console.Device aus.

3.9.3 Wie läuft nun die Ausgabe?

Auch dafür nimmt man eine Funktion. Diese arbeitet grundsätzlich wie die Lese-Funktion: Wieder wird ein IOStdReq ausgefüllt, die Option Schreiben wird angewählt, der Textpuffer übertragen und die Länge eingesetzt. Dann wird die Nachricht losgeschickt. Es wird aber so lange gewartet, bis alles abgelaufen ist. So können keine Überlappungen entstehen.

```
/*************************************
 *                                   *
 * Funktionen: Ausgabe über ConsoleDev *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     23.10.1987  Grundlage L&D *
 *                                   *
 * StdReq   Message Port der Nachricht *
 * Zeichen  Anzahl der Zeichen       *
 *                                   *
 *************************************/


ConWrite(StdReq, Text, Laenge)

struct IOStdReq *StdReq;
char            Text[];
int             Laenge;

   {
   StdReq->io_Command = CMD_WRITE;
   StdReq->io_Data    = (APTR)Text;
   StdReq->io_Length  = Laenge;
   DoIO(StdReq);
   }
```

Funktion 3.12: Daten über das Console.Device schreiben

Mit dem obigen Aufruf dieser Funktion wird das Zeichen am Pufferbeginn ausgegeben. Wir können auch einen String ausgeben, der ja bekanntlich durch ein Null-Byte gekennzeichnet ist.

Hierfür verwendet man die Längenangabe -1L. Ließe man dieses Programm laufen, Intuition-Library öffnet ein Window, Abfrageschleife auf Close-Gadget, Console.Device mit Ein- und Ausgabe, dann erhielte man schon einen Full-Window-Editor. Probieren Sie es aus!

3.9.4 Die Steuersequenzen des Console.Device

Im Gegensatz zu dem IntuiText-Strukturen können beim Console.Device auch Steuersequenzen übermittelt werden, die Einstellungen verändern oder Aktionen auslösen. Diese Steuersequenzen bestehen aus einfachen Zeichen-Codes oder komplexeren Zeichenketten. Sie brauchen nur mit der Schreibanweisung losgeschickt zu werden, und bei manchen bekommt man sogar noch eine Nachricht zurück.

Sehen Sie in der ersten Tabelle die Steuerzeichen, die vollkommen alleine stehen und sehr einfache Aktionen auslösen:

Bezeichnung	Zeichensequenz	Kommentar
Backspace	0x08	Entferne Zeichen links vom Cursor. Wie Backspace-Taste.
Linefeed	0x0A	Bewege Cursor eine Zeile tiefer unter Berücksichtigung der Mode-Funktion. Nächste Zeile, aber nicht erste Spalte.
Vertical Tab	0x0B	Bewege Cursor eine Zeile hoch.
Form Feed	0x0C	Lösche Fensterinhalt.
Carriage Return	0x0D	Bewege Cursor in erste Textspalte. Nur in erste Spalte, nicht Zeile tiefer.
Shift In	0x0E	Schalte Shift Out aus. (Textmodus)
Shift Out	0x0F	Shifte alle Zeichen, d.h. setze Bit 7 aller Character-Codes. (Grafikmodus)

Bezeichnung	Zeichensequenz	Kommentar
ESC	0x1B	Escape. Einleitung zu Escape-Sequenzen (wird mehr bei Druckern benutzt).
CSI	0x9B	control sequence introducer. Einleitung zu Control-Sequences (bei Console.Device, siehe unten).

Tabelle 3.20: Einfache Steuerzeichen

Wichtig sind die letzten beiden Zeichen. CSI leitet eine Steuersequenz ein und ist deshalb sehr wichtig, ESC wird für ähnliche Aufgaben benutzt.

Bezeichnung	Zeichensequenz	Kommentar
Reset	ESC 0x63	In den Urzustand bringen.
Insert [n] Characters	CSI [n] 0x40	Füge [n] Zeichen ein. Wenn kein [n] angegeben, dann n = 1.
Cursor Up [n] Lines	CSI [n] 0x41	Bewege Cursor [n] Zeilen nach oben. Wenn kein [n] angegeben, dann n = 1.
Cursor Down [n] Lines	CSI [n] 0x42	Bewege Cursor [n] Zeilen nach unten. Wenn kein [n] angegeben, dann n = 1.
Cursor Forward [n] Characters	CSI [n] 0x43	Bewege Cursor [n] Zeichen n. rechts. Wenn kein [n] angegeben, dann n = 1.
Cursor Backward [n] Characters	CSI [n] 0x44	Bewege Cursor [n] Zeichen n. links. Wenn kein [n] angegeben, dann n = 1.
Cursor Next Line [n]	CSI [n] 0x45	Bewege den Cursor in Zeile [n]. Wenn kein [n] angegeben, dann n = 1.
Cursor Preceding Line [n]	CSI [n] 0x46	Bewege Cursor in verherg. [n] Zeile. Wenn kein [n] angegeben, dann n = 1.
Move Cursor To Row [n] (Column [m])	CSI [n] (0x3B [m]) 0x48	Cursor in Zeile [n] (Spalte [m]).
Erase To End Of Display	CSI 0x4A	Lösche Fenster bis zum Ende.
Erase To End Of Line	CSI 0x4B	Lösche bis zum Ende der Zeile.
Insert Line	CSI 0x4C	Füge eine Zeile ein.
Delete Line	CSI 0x4D	Lösche eine Zeile.
Delete Character [n]	CSI [n] 0x50	Lösche Zeichen rechts vom Cursor.

Bezeichnung	Zeichensequenz	Kommentar
Scroll Up [n] Lines	CSI [n] 0x53	Schiebe Text um [n] Zeilen hoch.
Scroll Down [n] Lines	CSI [n] 0x54	Schiebe Text um [n] Zeilen runter.
Set Mode	CSI 0x32 0x30 0x68	Mache Linefeed zu Return-Linefeed.
Reset Mode	CSI 0x32 0x30 0x6C	Setze Linefeed nur als Linefeed.
Device Status Report	CSI 0x36 0x6E	Frage nach einem Statusbericht.

Tabelle 3.21: Kombinierte Steuerzeichen

Es seien noch einige Anmerkungen zu den kombinierten Steuerzeichen gestattet. Diese unterscheiden sich von den einfachen, da sie aus mehreren Zeichen zusammengesetzt werden. Aber die zweite Gruppe läßt sich noch weiter unterscheiden. Wir finden nämlich überall einen Zahlenwert. Dieser wird in ASCII-Ziffern übergeben und nicht als ASCII-Code eingesetzt (Achtung!). Das ist untypisch für die konventionelle Datenübertragung.

In allen Fällen, in denen eine Zahl eingesetzt werden kann, ist es auch möglich, diese wegzulassen. Dann nimmt das Console.Device einen Standardwert, der auch in der Tabelle angegeben ist. Manche der Kommando-Sequenzen fordern das Console.Device auf, eine Nachricht zurückzugeben. Diese Nachricht enthält dann angeforderte Informationen, wie z.B die Cursor-Position.

Weil alle Steuersequenzen an das Console.Device aus mehreren Zeichen bestehen, ist es ratsam, sich dafür ein Feld einzurichten, auf das man zurückgreift, wenn man eine Sequenz braucht. Bei der Abfrage gilt zuerst eine grundsätzliche Unterscheidung zwischen normalen Zeichen wie Buchstaben oder Zahlen und Sonderzeichen. Die Sonderzeichen sind in weitere Gruppen zu unterteilen. Hier benötigt man zusätzliche Abfragen der Cursor-Tasten, der Funktionstasten und der mit CSI eingeleiteten Steuersequenzen, die man zurückbekommt. Hier ist zuerst ein Beispiel für wichtige Sonderzeichen und eine Kommandotabelle:

```
#define SPACE                   0x20
#define BACKSPACE               0x08
#define LINEFEED                0x0A
#define RETURN                  0x0D
#define CSI                     0x9B
#define DEVICE_STATUS_REPORT    0x36, 0x6E
#define WINDOW_STATUS_REQUEST   0x30, 0x20, 0x71

#define CURSOR_UP               0x41
#define CURSOR_DOWN             0x42
#define CURSOR_Shift_UP         0x54
#define CURSOR_Shift_DOWN       0x53


UBYTE Sonderzeichen[] =
   {
   SPACE,
   BACKSPACE,
   LINEFEED,
   RETURN,
   CSI, DEVICE_STATUS_REPORT,
   CSI, WINDOW_STATUS_REQUEST
   };
```

Programmteil 3.18: Steuerzeichendefinition

3.9.5 Auswertung aller Nachrichten vom Console.Device

Die Kommunikation ist ausgereift: Wir können über zwei Ports Daten senden und empfangen. Wir wissen, welche "normalen" Zeichen empfangen werden können und welche Steuersequenzen es gibt. Damit sind wir gut gerüstet für einen Datenempfang. Nur gilt das auch für unser Programm?

Das Empfangen einfacher Tastenanschläge stellt kein Problem für ein Programm dar. Kompliziert wird die ganze Sache, wenn eine Nachricht aus mehreren Zeichen besteht. Dann liegt die Gefahr in der Luft, daß man die Nachricht falsch interpretiert oder gar verstümmelt auswertet. Diesem Fehler wollen wir gleich von Anfang an vorbeugen und eine hieb- und stichfeste Auswertung schreiben. Damit diese Auswertung auch einen Sinn hat, stellt sie den Grundstock für einen Editor dar. Dieser Editor wird soweit erweitert, daß er das alte NewCLI-Window ablösen kann. Weil wir uns aber zuerst nur mit der Abfrage beschäftigen wollen, setzen Sie bitte voraus, daß wir einen genügend großen

Textpuffer für das ganze Window haben. Zuerst benötigen wir eine Abfrageschleife, die aus unserem Datenpuffer Daten holt:

```
if (GetMsg(consoleReadPort))
   {
   switch (Buffer[0])
      {
         /* Untersuchung */
      }
   QueueRead(consoleReadMsg, &Buffer[mode], 1L);
   }
```

Programmteil 3.19: Abfrage Console.Device

Wie immer hat die Abfrageschleife einen ganz primitiven Grundaufbau: nachsehen, ob Daten vorhanden sind, erstes Zeichen untersuchen, neues Zeichen holen. Angenommen, es handelt sich um einen Buchstaben oder ein normales Zeichen, das nicht gesondert behandelt werden muß, dann kann es folgende Zeilen bearbeiten:

```
default        : ConWriteChr(consoleWriteMsg, &Buffer[0]);
                 Coords = FALSE;
                 WindowBuffer[y][x-1] = Buffer[0];
                 if (WindowBuffer[y][79] < x)
                     WindowBuffer[y][79] = x;
                 break;
```

Programmteil 3.20: Allgemeine Tastenauswertung

Die kleine Bearbeitungsroutine gibt zuerst das Zeichen aus. Dann wird das Koordinaten-Flag auf FALSE gesetzt, damit das Programm weiß, daß die aktuellen Koordinaten nun nicht mehr bekannt sind. Als nächstes wird das Zeichen in den Window-Puffer übertragen. Dieser speichert alle Zeichen, die im Window zu sehen sind. Als nächstes wird geprüft, ob das neue Zeichen die Zeile verlängert. Wenn ja, wird die neue Zeilenlänge übertragen.

Diese Methode wurde aus folgendem Grund gewählt: Ganz zu Anfang wird der Window-Puffer gelöscht. Er enthält dann nur Leerzeichen (0x20). Würde in einer Zeile Return getippt werden, dann soll diese ja an das DOS übergeben werden, damit der CLI-Befehl ausgeführt werden kann. Werden aber auch alle Leerzeichen an das DOS übergeben, dann kann es nicht erken-

nen, wo der Befehl abgeschlossen wurde. Dadurch ist es nicht möglich, einen Befehl zu senden, um z.B. sein Format zu erfahren, denn dafür darf nur das Befehlswort ohne jedes weitere Zeichen übermittelt werden.

Apropos Bestätigung einer Zeile. Hier haben wir das erste Zeichen, das gesondert betrachtet werden muß. Denn dann soll ja die Zeile, in der sich der Cursor befindet, an das CLI übermittelt werden:

```
case RETURN     : ConWriteChr(consoleWriteMsg, &Buffer[0]);
                  ConWriteChr(consoleWriteMsg, &Sonderzeichen[2]);
                  Coords = FALSE;
                  if (WindowBuffer[y][79] != 0)
                     {
                     WindowBuffer[y][WindowBuffer[y][79]] = NULL;
                     Execute(&WindowBuffer[y][0], 0, 0);
                     WindowBuffer[y][WindowBuffer[y][78]] = 0x20;
                     }
                   break;
```

Programmteil 3.21: Return-Auswertung

Diese zweite Routine zur Zeichenauswertung bearbeitet zuerst die Grafik. Das heißt, es wird zuerst der Cursor an den Zeilenanfang und in die nächste Zeile gebracht. Wieder wird das Koordinaten-Flag gesetzt. Danach prüft eine Abfrage, ob in der Zeile überhaupt Zeichen eingegeben wurden. Wenn ja, wird diese Zeile mit Null abgeschlossen, um an die Execute()-Funktion des DOS übergeben werden zu können. Nachher wird der Abschluß wieder mit einem Space "übertüncht".

Das letzte Zeichen, das gesondert betrachtet werden muß, ist das der Backspace-Taste. Weder die grafische Ausgabe wird vom Console.Device erledigt noch die Korrektur des Puffers.

```
case BACKSPACE : ConWriteChr(consoleWriteMsg, &Buffer[0]);
                 ConWrite(consoleWriteMsg, &Sonderzeichen[0], 2);
                 WindowBuffer[y][x-1] = SPACE;
                 Coords = FALSE;
                 break;
```

Programmteil 3.22: BACKSPACE-Auswertung

Naja, eine Beschreibung brauchen Sie hoffentlich nicht dafür!

Das letzte und zugleich wichtigste Problem, die Verwaltung der Steuersequenzen, wird verschoben. Wir helfen uns mit der folgenden "Auswertung":

```
        case CSI        : mode += 1;
                          break;
```

Programmteil 3.23: CSI-Erkennung

Diese Abfrageschleife setzt einfach einen Zähler, der später abgefragt wird, um eins höher. Aus dem Zähler läßt sich erkennen, wie viele Zeichen noch zu bearbeiten sind. Es kann deshalb auftreten, daß mehrere Steuersequenzen im Puffer bereitliegen, weil auch die Funktions- und Cursor-Tasten mit dem CSI gehandhabt werden. Aber sehen wir uns gleich die Abfrage dazu an:

```
/*********************************
 *                               *
 *  Behandlung der Steuer-       *
 *  sequenzen                    *
 *                               *
 *********************************/

else if (mode)
   {
       /* Abfrage */

   mode = 0;
   QueueRead(consoleReadMsg, &Buffer[mode], 1L);
   }
```

Programmteil 3.24: Steuerzeichenbehandlung

Das Else bezieht sich auf den If-Befehl, der den Console-Puffer auf ein Zeichen vom Device testet. Solange nämlich Nachrichten vorliegen, werden diese erst in den Puffer geholt, denn das Console.Device speichert nur die letzten 32 Zeichen. Stehen mehr Zeichen in der Warteschlange, dann werden diese einfach ignoriert. Deshalb wird die Variable *mode* verwendet, um immer neu eintreffende Zeichen in die nächste Stelle unseres Puffers einzutragen.

Betrachten wir jetzt die erste Auswertung einer Steuersequenz. Es handelt sich dabei um die Nachricht der Cursor-Position:

```
/*********************************
 *                               *
 *  Control Sequence :           *
 *                               *
 *  CURSOR POSITION REPORT       *
 *                               *
 *********************************/

if (Buffer[mode-1] == 82)
   {
   Coords = TRUE;
   x = 0; y = 0; i = 1;

   while (Buffer[i] < 0x3A & 0x2F < Buffer[i])
      {
      y *= 10;
      y += Buffer[i] - 0x30;
      i ++;
      }
   i ++;
   while (Buffer[i] < 0x3A & 0x2F < Buffer[i])
      {
      x *= 10;
      x += Buffer[i] - 0x30;
      i ++;
      }
   }
```

Programmteil 3.25: Cursor-Position auswerten

Der Programmteil geht ein Zeichen nach dem anderen durch und berechnet daraus die X- und Y-Koordinate. Der DEVICE STATUS REPORT hat das Format: *CSI Reihe; Zeile R*. Wir erkennen ihn also an dem abschließenden R. Dann wird der Puffer Zeichen für Zeichen ausgewertet. Die Berechnung des ersten Wertes schließt ab, wenn das Programm auf ein Semikolon trifft. Danach kann der X-Wert berechnet werden. Wichtig ist auch, daß das Koordinaten-Flag wieder gelöscht wird. Denn später folgt ein Test, der bei gesetztem Flag die Nachricht ans Console.Device schickt, daß es doch bitte die Koordinaten auf die Datenleitung legen soll. Dann hätten wir eine Endloskommunikation, die aus den Koordinaten des Cursors bestehen würde.

Die nächsten Überprüfungen gehen davon aus, daß immer noch eine Control Sequence vorliegt. Allerdings nicht der DEVICE STATUS REPORT. Zuerst wird überprüft, ob vielleicht die Cursor-Tasten betätigt wurden. Das Console.Device erlaubt deren Benutztung auch mit Shift. Dann wird nicht die Position des Cursors geändert, sondern der ganze Fensterinhalt gescrollt. Dieses Scrollen muß natürlich auch im Textpuffer geschehen. Aber betrachten Sie vorher die Abfrage:

```
else
   {
   switch (Buffer[mode-1])
      {

      /**********************************
       *                                *
       *  Control Sequence :            *
       *                                *
       *  SHIFTED CURSOR MOVE           *
       *                                *
       **********************************/

      case CURSOR_Shift_UP    : Move_Down();
                                break;
      case CURSOR_Shift_DOWN  : Move_Up();
                                break;

      /**********************************
       *                                *
       *  Control Sequence :            *
       *                                *
       *  NORMAL CURSOR MOVE            *
       *                                *
       **********************************/

      case CURSOR_UP          : if (y == 1) Move_Down();
                                break;
      case CURSOR_DOWN        : if (y == Ymax) Move_Up();
                                break;

      }
```

Programmteil 3.26: Cursor-Tasten auswerten

Die letzten beiden Vergleiche gehen auf normale Cursor-Tastenbewegungen ein. Denn wenn der Cursor den oberen oder unteren Rand erreicht hat, wir auch hier gescrollt. Für das Scrollen des Window-Puffers wurden eigens zwei neue Funktionen defi-

niert. Sie stehen in Zusammenhang mit den Textpuffer-Funktionen. Hier sind sie gleich alle:

```
/*************************************
 *                                   *
 * Funktionen: Textpufferbearbeitung *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb      2.11.1987                *
 *                                   *
 * y  Nummer der Zeile               *
 * x  Nummer der Spalte              *
 *                                   *
 *************************************/


Clear_Buffer()
   {
   int y;
   for(y=0; y<80; y ++)
      Clear_Line(y);
   }


Clear_Line(y)
int y;
   {
   int x;
   for(x=0; x<79; x ++)
      WindowBuffer[y][x] = 0x20;
   WindowBuffer[y][78] = NULL;  /* Zeilenendekennung */
   WindowBuffer[y][79] = NULL;  /* Anzahl eingegebener Zeichen */
   }


Move_Up()
   {
   int x, y;
   for(y=1; y<80; y ++)
      for(x=0; x<80; x ++)
         WindowBuffer[y-1][x] = WindowBuffer[y][x];
   Clear_Line(79);
   }


Move_Down()
   {
   int x, y;
   for(y=79; y>0; y --)
      for(x=0; x<80; x ++)
         WindowBuffer[y][x] = WindowBuffer[y-1][x];
```

```
Clear_Line(0);
}
```

Funktion 3.13: Pufferbehandlung

Clear_Buffer()
Wird zu Anfang des Programms aufgerufen und initialisiert den Puffer wie schon besprochen. Die Anzahl der eingegebenen Zeichen pro Zeile wird auf Null gesetzt, genauso wie auch das Zeilenende durch eine Null gekennzeichnet wird.

Clear_Line()
Die eigentliche Routine, die von Clear_Buffer() aufgerufen wird. Dadurch ist es möglich, Zeilen selektiert zu löschen, was ja mit Steuersequenzen unterstützt wird.

Es ist nicht möglich, die Abfrage und Behandlung aller möglichen Steuersequenzen in diesem Kapitel zu beschreiben. Deshalb folgt als letztes einen allgemeiner Abschnitt, der alle unbehandelten Routinen zur Ausgabe weitergibt. Damit unterschlagen wir nichts, und das Console.Device ist zufrieden.

```
/*********************************
 *                               *
 *  Control Sequence :           *
 *                               *
 *  Handhabung jeder Sequenz     *
 *                               *
 *********************************/

  ConWrite(consoleWriteMsg, &Buffer[0], mode);
  Coords = FALSE;
  }
```

Programmteil 3.27: Durchschleifen jeder Sequenz

Wer Lust hat, kann aber gerne zu allen weiteren Steuersequenzen Abfragen schreiben. Es ist z.B. noch nicht beschrieben worden, wie man eine einfache Funktionstastenbehandlung realisiert. Das ist aber ganz einfach. Vorausgesetzt, Sie nehmen unveränderbare Belegungen, dann richten Sie einfach ein Pointer-Feld mit den gewünschten Texten ein. Hier ein Beispiel für den CLI-Editor:

```
UBYTE *Funktionstasten[] =
   {
   "dir"+RETURN,
   "cd df1:"+RETURN,
   "cd sys:"+RETURN,
   "list"+RETURN,
   "makedir ram:c"+RETURN+"copy sys:c ram:c"+RETURN,
   "execute make "
   };
```

Programmteil 3.28: Funktionstasten-Definition

Die Abfrage für die Funktionstasten muß diese nur über "~" identifizieren und die Nummer berechnen. Dann kann der Pointer an ConWrite(); übergeben werden:

```
ConWrite(consoleWriteMsg, Funktionstasten[Nummer], -1L);
```

3.9.6 Zum Schluß der CLI-Editor

Für die Arbeit zwischen Intuition, dem Window des Editors, und dem Console.Device wird empfohlen, das Console.Device auf eine etwas andere Art zu öffnen. Ich möchte auch diese hier kurz beschreiben:

Abwohl das Console.Device in den IDCMP eingebaut wurde, sind doch dadurch nicht alle Möglichkeiten ausgeschöpft. Für eine totale Offenheit in der Bedienung müssen wir zuerst eine IOStdReq-Struktur definieren, in die wir den Zeiger auf unser Window im Wert *io_Data* "einpflanzen". Danach kann das Device wie gewohnt geöffnet werden. Jetzt ist es mit unserem Intuition-Window verbunden, und die Kommunikation kann über die verwendete Struktur abgewickelt werden. Sie können sie sowohl zum Schreiben als auch zum Lesen verwenden. Die Lese- und Schreibfunktionen, die am Anfang des Kapitels dafür entwickelt wurden, gelten nach wie vor.

Setzen Sie jetzt alle Abfrageelemente, alle Routinen zum Öffnen und Schließen und die Textpufferbehandlung zusammen. Vergessen Sie nicht, die nötigen Variablen und Felder zu definieren! Fertig! Ein neuer CLI-Editor!

Hinweis: In diesen Editor können Sie auch das vorher im Buch beschriebene Gadget einbauen, das die Größe des Editorfensters ändert.

3.10 Einrichten selbstdefinierter Tastaturtabellen

Immer wird davon gesprochen, daß das Console.Device die RAWKEY-Codes in Zeichen umwandelt. Nun fragt sich aber der aufmerksame Programmierer, nach welchen Anweisungen, Befehlen oder Definitionen diese Umwandlung geschieht.

Für die Transferierung der RAWKEY-Codes zu richtigen Zeichen wird eine aufwendige Tabelle verwendet. Das ist die sog. Tastaturtabelle. In ihr ist genauestens verzeichnet, was bei welchem Tastendruck wie umgewandelt werden soll.

Diese Tastaturtabelle ist noch weiter unterteilt. Sie enthält Abschnitte für die Shift-, Alt- oder Ctrl-Taste. Außerdem wird die CapsLock-Taste noch gesondert behandelt. Das macht es z.B. möglich, daß bei einer deutschen Tastaturbelegung nur die Buchstaben geshiftet werden und nicht die Sonderzeichen oder die Zahlen.

Durch die Tastaturtabellen ist es uns auch möglich, auf eine Taste einen ganzen Text zu legen. Dieser kann aus normalen Zeichen bestehen oder auch mit lauter Steuer-Codes belegt werden.

Sie sehen, welche Möglichkeiten das Console.Device zur Verfügung stellt.

3.10.1 Aufbau der Tastaturtabellen

Wir wissen jetzt, daß die Übersetzung der RAWKEY-Codes über Tastaturtabellen abgewickelt wird. Aber wie sind diese Tabellen aufgebaut, und welche Informationen enthalten sie genau?

Das ist die Frage, die uns zuerst beschäftigen soll.

Wenn man von der Tastaturtabelle spricht, so meint man damit alle kleineren Tabellen, aus denen sich diese zusammensetzt.

Für den Zusammenhalt aller Untertabellen ist die KeyMap-Struktur zuständig. Sie enthält nur Zeiger auf jede der kleineren Tabellen:

```
struct KeyMap
{
0x00  00  UBYTE *km_LoKeyMapTypes;
0x04  04  ULONG *km_LoKeyMap;
0x08  08  UBYTE *km_LoCapsable;
0x0C  12  UBYTE *km_LoRepeatable;
0x10  16  UBYTE *km_HiKeyMapTypes;
0x14  20  ULONG *km_HiKeyMap;
0x18  24  UBYTE *km_HiCapsable;
0x1C  28  UBYTE *km_HiRepeatable;
0x20  32
};
```

Zuerst erkennt man, daß die KeyMap-Struktur in zwei große Teile zerfällt: einen Lo- und einen Hi-Abschnitt.

Der erste ist für die Zeichen mit dem RAWKEY-Code vom 0x00 bis 0x3F zuständig.

Der zweite Abschnitt enthält Informationen zu den Codes von 0x40 bis 0x67. Diese Teilung wurde vorgenommen, damit die Tastatur noch für weiteren Tasten frei ist, die dann in die Hi-Tabellen eingetragen werden können.

Sehen Sie dazu noch einmal das RAWKEY-Schaubild an, dem Sie den Code jeder Taste entnehmen können. Gehen wir aber nun jede Tabelle einzeln durch:

km_LoKeyMapTypes & km_HiKeyMapTypes

Diese beiden Tabellen enthalten für jede Taste ein Byte. In diesem Byte sind Flags untergebracht, die etwas über später folgende Informationen aussagen. Es ist leider nicht möglich, für jede Taste alle Kombinationen zu belegen. Das heißt, Sie können nicht zu jeder Taste einen normalen, einen Shift-, einen Alt- und einen Ctrl-Wert definieren. Weil auch die Kombinationen von zwei Qualifiers, so werden die Shift-, die Alt- und die

Ctrl-Taste genannt, berücksichtig wurden, können insgesamt nur vier Zeichen auf eine Taste gelegt werden. Allerdings können Sie bestimmen, welche Qualifier welche Zeichen auslösen. Oder Sie entscheiden sich für einen ganzen String auf der Taste, dann müssen Sie aber in Kauf nehmen, daß nur zwei Belegungen für die Taste zulässig sind.

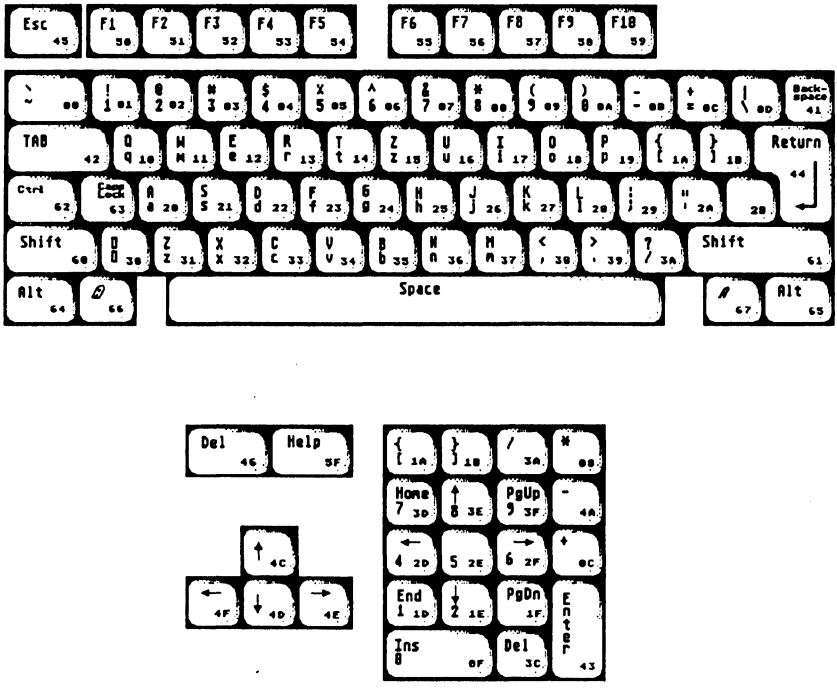


Abbildung 3.13: Die RAW-Tastatur-Codes II

km_LoKeyMap & km_HiKeyMap

Hier haben wir jetzt die beiden Tabellen, die in ihren ULONG-Werten die Zeichen enthalten, die beim Tastendruck erscheinen sollen. Da der ULONG-Wert aus vier Bytes besteht, finden wir in jedem Byte den Code des Zeichens, das bei einer der vier

Kombinationen ausgegeben werden soll. Es sei denn, es handelt sich um einen String. Dann stellt der ULONG-Wert einen Pointer auf den String dar.

km_LoCapsable & km_HiCapsable

Diese beiden Tabellen enthalten keine Zeichen-Codes, die ausgegeben werden sollen. In den Tabellen finden wir nur gesetzte und ungesetzte Bits für jeden RAWKEY-Code. So zeigt ein gesetztes Bit an, daß diese Taste bei gedrückter CapsLock-Taste wie bei Shift behandelt werden soll. Ist das Bit nicht gesetzt, wird die Taste normal ausgegeben.

km_LoRepeatable & km_HiRepeatable

Die nächsten beiden Tabellen arbeiten genauso wie die Capsable-Tabellen. Auch hier haben wir wieder für jeden RAWKEY-Code ein Bit. Ist dieses gesetzt, dann wird die Taste nach dem Wert aus Preferences wiederholt. Wurde das Bit gelöscht, dann wird die Taste nicht wiederholt.

3.10.2 Die Arbeit mit den Tabellen

Ist es gewünscht, Einfluß auf die Tastaturtabelle zu nehmen, so ist es nötig, erst einmal den momentanen Stand zu erfahren. Diese Arbeit erledigt nicht eine neue Library, sondern es besteht ein Weg über das Console.Device. Immerhin kann man nur mit Tastaturtabellen arbeiten, wenn das Console.Device aktiv ist.

Es existieren Kommandos, die an das Console.Device gesendet einen Report über den aktuellen Zustand geben. Ebenso wird es unterstützt, die Zeiger auf eine neue Tastaturtabelle zu übertragen. Leider werden die beiden Vorgänge nicht über Library-interne Funktionen erledigt. Die Aufträge werden ganz einfach mit einem IOStdReq übermittelt. Deshalb empfiehlt es sich, dafür eigene Funktionen zu schreiben.

Das Kommando zum Holen der aktuellen KeyMap-Belegung heißt CD_ASKKEYMAP. Wir brauchen dazu die Größe und Adresse unserer KeyMap-Struktur, in die die Daten gelegt werden sollen. Danach kann der Request aufgerufen werden:

```
AskKeyMap(ConWriteRequest, OwnKeyMap)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;

{
   BYTE Error;
   ConWriteRequest->io_Command = CD_ASKKEYMAP;
   ConWriteRequest->io_Lenght  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;

   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

Funktion 3.14: AskKeyMap()

Wir finden nach erfolgreichem Arbeiten der selbstdefinierten Funktion in der Struktur OwnKeyMap die Zeiger auf die aktiven Tastaturtabellen. Diese können analysiert und beeinflußt werden. Mit dieser Methode kann man z.B. ganz einfach einzelne Zeichen der Tastatur ändern.

Hat man aber eine sehr spezielle Belegung, die nicht mit wenigen Änderungen aktivierbar ist, oder will man eine Tabelle laden, dann eignet sich diese Methode nicht. Dafür geht man anders vor: Zuerst wird der Speicher mit den Tastaturwerten belegt. Dies kann entweder im Programmtext oder durch Laden abgewickelt werden. Danach erstellt man die neue KeyMap-Struktur. Diese ist aber das Kernstück, aus dem das Console.Device alle Informationen nimmt. Also müssen wir die neue KeyMap an das Console.Device übertragen. Dafür gibt es ein Kommando, und dieses läßt sich am einfachsten in einer eigens dafür entwickelten Funktion aufrufen:

```
SetKeyMap(ConWriteRequest, OwnKeyMap)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;

{
```

```
   BYTE Error;
   ConWriteRequest->io_Command = CD_SETKEYMAP;
   ConWriteRequest->io_Lenght  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;

   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

Funktion 3.15: SetKeyMap

Im Prinzip ist das die gleiche Funktion wie die vorhergehende. Einziger Unterschied ist das Kommando. Wollen Sie beide Funktionen durch eine ersetzen, so ist das nicht unmöglich! Wir erweitern nur den Funktionsaufruf um einen Parameter:

Beispiel:

```
DoKeyMap(WriteRequest, KeyMapPointer, CD_SETKEYMAP)

DoKeyMap(ConWriteRequest, OwnKeyMap, Mode)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;
UWORD            Mode;

{
   BYTE Error;
   ConWriteRequest->io_Command = Mode;
   ConWriteRequest->io_Lenght  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;

   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

Funktion 3.16: DoKeyMap()

Mit dieser letzten Funktion ist es möglich, zwei weitere Kommandos anzusprechen: CD_ASKDEFAULTKEYMAP und CD_SETDEFAULTKEYMAP. Beide arbeiten genauso wie die beiden bekannten. Allerdings beeinflussen wir mit ihnen die Default-KeyMap, die jedem Programm beim Öffnen des Console.Device zur Verfügung steht.

In dieser KeyMap werden alle Zeichen in die Gruppen Lo und Hi unterteilt. Die Lo-Gruppe wird folgendermaßen behandelt:

- Alle Tasten, die alleine gedrückt werden, erzeugen ihren ASCII-Code.
- Alle Tasten, die mit Shift gedrückt werden, erzeugen ihren geshifteten ASCII-Code.
- Alle Tasten, die mit Alt gedrückt werden, erzeugen ihren ASCII-Code, bei dem das höchste Bit (0x80) gesetzt ist.
- Alle Tasten, die mit Alt und Shift gedrückt werden, erzeugen ihren geshifteten ASCII-Code, bei dem das höchste Bit (0x80) gesetzt ist.
- Alle Tasten, die mit Ctrl gedrückt werden, erzeugen ihren ASCII-Code, bei dem Bit 5 und 6 gelöscht werden.

Die Hi-Gruppe der Default-KeyMap wird anders behandelt:

Da in der High-KeyMap fast nur Steuertasten enthalten sind, besteht diese Tabelle nicht aus Einzel-Code. Wir treffen hier fast nur Strings an. Das ist auch der Grund, warum die Steuertasten höchstens zweifach belegt werden können.

Taste	Übersetzter Wert
BACKSPACE	0x08
ENTER	0x0D
DEL	0x7F

Tabelle 3.22: Tasten ohne jeden Qualifier

Taste	Wert ohne -	Wert mit -	Qualifier
SPACE	0x20	0xA0	Alt
RETURN	0x0D	0x0A	Ctrl
ESC	0x1B	0x9B	Alt
-	0x2D	0xFF	Alt

Tabelle 3.23: Tasten mit einem Qualifier

Taste	Wert	Wert mit Shift
TAB	0x09	0x9B "Z"
UP	0x9B "A"	0x9B "T"
DOWN	0x9B "B"	0x9B "S"
FORWARD	0x9B "C"	0x9B " @"
BACKWARD	0x9B "D"	0x9B " A"
F1	0x9B "0~ "	0x9B "10~ "
F2	0x9B "1~ "	0x9B "11~ "
F3	0x9B "2~ "	0x9B "12~ "
F4	0x9B "3~ "	0x9B "13~ "
F5	0x9B "4~ "	0x9B "14~ "
F6	0x9B "5~ "	0x9B "15~ "
F7	0x9B "6~ "	0x9B "16~ "
F8	0x9B "7~ "	0x9B "17~ "
F9	0x9B "8~ "	0x9B "18~ "
F10	0x9B "9~ "	0x9B "19~ "
HELP	0x9B "?~ "	

Tabelle 3.24: Tasten der High-KeyMap

3.10.3 Eine Tastaturtabelle selber zusammensetzen

Ihnen ist jetzt bekannt, aus welchen Einzeltabellen sich die Tastaturtabelle zusammensetzt. Auch die Abfrage der bestehenden und das Setzen einer neuen Tabelle ist erklärt worden. Wie erstellt man nun seine eigene Tabelle und auf was muß man achten? Unter dieser Fragestellung werden wir dieses Kapitel beschließen.

KeyMapTypes

Als erstes benötigen wir die beiden Types-Tabellen. Diese enthalten, wie oben schon besprochen, zu jeder Taste Flags, aus denen sich schließen läßt, ob es ein String oder vier Einzelzeichen sind und mit welchen Qualifiern die Zeichen zu erreichen sein werden. Folgende Flags sind möglich:

Name	Wert	Beschreibung
KC_NOQUAL	0L	Ohne jeden Qualifier.
KC_VANILLA	7L	Standardwert mit Ctrl.
KCF_SHIFT	0x01L	Gedrückte Shift-Taste.
KCF_ALT	0x02L	Gedrückte Alt-Taste.
KCF_CONTROL	0x04L	Gedrückte Ctrl-Taste.
KCF_DOWNUP	0x08L	Reagiert nur, wenn Taste niedergedrückt und losgelassen wurde.
KCF_DEAD	0x20L	Taste reagiert nicht.
KCF_STRING	0x40L	Taste gibt einen String aus.

Tabelle 3.25: KeyMapTypes

Wir können bei Auswahl der Types nun verschiedene Belegungen erreichen. Dabei haben die vier Bytes die folgenden Bedeutungen:

Qualifier-Kombination	1. Byte	2. Byte	3. Byte	4. Byte
KC_DEAD	-	-	-	-
KC_NOQUAL	-	-	-	ohne
KC_SHIFT	-	-	Shift	ohne
KC_ALT	-	-	Alt	ohne
KC_CONTROL	-	-	Ctrl	ohne
KC_ALT + KC_SHIFT	Shift+Alt	Alt Shift	ohne	
KC_CONTROL + KC_ALT	Ctrl+Alt	Ctrl Alt	ohne	
KC_CONTROL + KC_SHIFT	Ctrl+Shift	Ctrl Shift	ohne	
KC_VANILLA	Shift+Alt	Alt Shift	ohne	
	Außerdem ist noch Ctrl möglich.			
KC_STRING	Pointer auf String			

Tabelle 3.26: KeyMapTypes-Kombinationen

Hiermit müßte es Ihnen schon möglich sein, für beliebige Belegungen eine Types-Tabelle zu erstellen. Das am häufigsten benutzte Qualifier-Flag ist KC_VANILLA, weil die üblichen Tasten Shift und Alt integriert sind. Aber zusätzlich wird auch noch Ctrl unterstützt, indem Bit 5 und 6 des NOQUAL-Wertes gelöscht werden. Durch geschickte Wahl kann man auch damit sehr gut zurecht kommen.

KeyMap

Eine Lo- oder Hi-KeyMap läßt sich jetzt sehr einfach zusammenbauen. Allerdings empfiehlt es sich hier für C-Programmierer, in den Assembler-Modus des Quelltextes umzuschalten, da so das Problem viel einfacher zu lösen ist. Fertigen Sie dann einfach eine Tabelle an, die in Vierer-Gruppen unterteilt ist. Jede Gruppe entspricht einer Taste und ist nach der oben stehenden Tabelle geordnet. Jedes Byte enthält ein Zeichen, das je nach Tastenkombination ausgegeben wird.

Handelt es sich aber um einen String, so steht an dieser Stelle nicht eine Gruppe aus vier Bytes, sondern ein LONG-Wert, der den Zeiger auf unseren String darstellt. Leider wird nicht gleich auf einen String gezeigt, das wäre ja viel zu einfach. Der Pointer zeigt zuerst auf eine weitere Tabelle, die eigens für diese Taste die Werte enthält. Und zwar ist das zuerst die Länge des Strings, der bei normalem Tastendruck erscheinen soll, dann ein Offset vom Anfang der String-Tabelle zum eigentlichen String selbst. Als weiteren Wert finden wir die Länge des Strings, der bei einem Tastendruck mit Shift ausgegeben werden soll. Auch dazu steht der Offset vom Beginn der Tabelle.

Capsable

Wie von der Schreibmaschine bekannt, hat auch die Amiga-Tastatur eine CapsLock-Taste, die wie ein Dauer-Shift funktioniert. Da die Tastatur aber einen Tastaturprozessor hat, wird nicht einfach nur ein Dauer-Shift simuliert. Wir haben es mit einer neuen Art der Shift-Taste zu tun. Für die CapsLock-Taste kann bestimmt werden, auf welche Tasten sie anwendbar ist und auf welche nicht. Diese Methode bietet den Vorteil, daß z.B. die Satzzeichen nicht geshiftet werden und dort ganz normal weiter-

geschrieben werden kann, ohne vorher den CapsLock-Modus aufheben zu müssen.

Für alle Tasten haben wir 8 Bytes mit 64 Bits zur Verfügung. Jedes dieser Bits repräsentiert eine Taste:

```
Bit 0 Byte 0 RAWCODE 0x00, Bit 1 Byte 0 RAWCODE 0x01 ...
Bit 0 Byte 1 RAWCODE 0x08 ... Bit 7 Byte 7 RAWCODE 0x4F.
```

Setzen Sie alle die Bits, für deren Tasten Sie eine Shift-Behandlung wünschen.

Repeatable

Die gleiche Methode, die der Capsable-Tabelle zu Grunde liegt, wurde auch bei der Repeatable-Tabelle angewendet. Nur hier indiziert jedes gesetzte Bit, daß die Taste nach der Wiederholungszeit und der Wiederholfrequenz der Preference-Werte wiederholt werden soll. So können manche Tasten von der Wiederholfunktion ausgeschlossen werden. Das ist sinnvoll, wenn eine Taste nicht mit Wiederholung gebraucht wird oder eine Wiederholung großen Schaden anrichten könnte. Meistens wird die Return-Taste von der Wiederholung ausgeschlossen.

3.11 Speicherverwaltung

Irgendwann kommt es bei Anwenderprogrammen immer vor, daß ein gewisser Speicher gebraucht wird. Die einfachste Speicherform sind die Variablen. Hier übersieht man schon bei der Programmgestaltung, welche benötigt werden. Die etwas komplexere Form davon bilden die Variablenfelder. Aber auch dabei ist man immer darüber informiert, welche Ausmaße diese haben werden.

Doch schon hier setzt das Problem an. Es könnte für den Programmablauf z.B. ein Variablenfeld doppelt genauer Zahlen benötigt werden. Dieses soll die Ausmaße 1000*1000 für die Farbwerte einer hochauflösenden Grafik haben. Unser Compiler wird dann im fertigen Programm 1.000.000mal eine doppelt genaue Variable reservieren. Sie können sich vorstellen, welche Ausmaße das annimmt!

An einem kleinen Beispiel möchten ich Ihnen jetzt zeigen, wie Sie Speicherbereiche ganz individuell vom System holen und nach Gebrauch wieder zurückgeben. Drei Methoden der Speicherorganisation werden daran demonstriert. Das Beispiel selbst ist natürlich aus dem Themenkreis Intuition gewählt. Wir werden versuchen, uns vier Textpuffer für die entsprechenden String-Gadgets zuweisen zu lassen.

3.11.1 Die Speicherorganisation des Amiga

Bevor ein Programm erstellt werden kann, sollte man sich mit der Speicherorganisation vertraut machen. Der Amiga hat eine zweigeteilte dynamische Speicherverwaltung.

In dieser Tabelle finden Sie die Speicherbelegung:

Adresse	Belegung	Kommentar
0x00000000	Chip-RAM	Spiegelung des Kickstart-ROM
0x08000000		Dreimalige Spiegelung des Chip-RAM
0x20000000	8 MByte Fast-RAM	
0xA0000000	CIAs	
0xC0000000	512 KByte-Erweiterung	(nur 500er & 2000er)
0xC8000000	Unbenutzt	
0xDC000000	Echtzeituhr	(nur 500er & 2000er)
0xDF000000	Custom-Chips	
0xE0000000	Unbenutzt	
0xE8000000	Expansion-Slots	
0xF0000000	ROM-Module	
0xF8000000	Kickstart-ROM	Spiegelung
0xFC000000	Kickstart-ROM	

Tabelle 3.27: Speicherbelegung des Amiga

Der ganze Speicher ist in zwei Bereiche unterteilt. Die ersten 512 KByte sind Chip-RAM, d.h. sie können auch von den Custom-Chips angesprochen werden. Der zweite Bereich besteht aus Fast-RAM oder ROM und kann nur vom 68000er verwaltet werden.

Die Dynamik liegt darin, daß die 8 MByte Fast-RAM bei der Grundversion noch nicht existieren und dynamisch ergänzt werden können.

Wenn wir nun unserem Programm vom System einen Speicherbereich zuteilen lassen wollen, dann müssen wir angeben, ob es Chip- oder Fast-RAM sein soll. Geben wir nichts an, so wird zuerst versucht, Fast-RAM zu bekommen, und erst wenn dies nicht gelingt, wird Chip-RAM belegt. Dies liegt an der "Kostbarkeit" des Chip-RAM.

Hinweis: Hierin liegt auch das Problem begraben, daß alte Programme bei Speicher-Erweiterungen abstürzen. Diese geben nämlich nicht genau an, welchen Speicherbereich sie brauchen, denn ohne Erweiterung liegt ja nur Chip-RAM vor. Dieses kann auch von den Custom-Chips angesprochen werden. Bei einer Speicher-Erweiterung wird aber Fast-RAM zugeteilt. Bei einem Zugriff der Chips auf diesen Bereich stürzt das System ab!

3.11.2 Erste Gehversuche mit AllocMem()

Für die ersten Versuche, Speicher zu belegen, greifen wir auf eine Exec-Funktion zurück. Sie heißt AllocMem() und belegt einen Speicherbereich von angegebener Größe und Kriterium. Man übergibt also zwei Parameter und erhält als Ergebnis die Adresse des zugeteilten Speichers. Wurde kein Speicher in der Größe und den Eigenschaften gefunden, erhält man einen Null-Pointer = Fehler zurück.

```
MemoryBlock = AllocMem(ByteSize, Requirements);
    D0          -198       D0        D1
```

Für die Angabe der Eigenschaften des Speicherbereichs stehen uns die oben angesprochenen Flags MEMF_FAST und MEMF_CHIP zur Auswahl. Hinzu kommt noch MEMF_PUBLIC, das verbietet, den Speicherbereich zu verschieben. Diese Option ist aber noch gar nicht vorgesehen. Weiterhin können Sie mit MEMF_CLEAR gleich Anweisung geben,

daß der Speicherbereich gelöscht, d.h. mit Nullen gefüllt werden soll. Der kleinste Speicherbereich kann übrigens die Größe von 8 Bytes nicht unterschreiten!

Hier ein Programm, das versucht, vier Textpuffer zu reservieren:

```
/*************************************
 *                                   *
 * Speicherverwaltung : AllocateMemory *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur für Test- *
 *                     zwecke        *
 *                                   *
 *************************************/


#include <exec/types.h>
#include <exec/memory.h>

#define MemoryType MEMF_CHIP | MEMF_CLEAR


UBYTE *UndoBuffer, *FileBuffer, *DiskBuffer,
                   *SuffBuffer, *AllocMem();


main()
   {
   UndoBuffer = AllocMem(512L, MemoryType);
   if (!UndoBuffer)
      {
      printf("Leider Probleme beim Undo-Puffer!\n");
      FreeMemory();
      exit(FALSE);
      }

   FileBuffer = AllocMem(30L, MemoryType);
   if (!FileBuffer)
      {
      printf("Leider Probleme beim FileBuffer!\n");
      FreeMemory();
      exit(FALSE);
      }

   DiskBuffer = AllocMem(512L, MemoryType);
   if (!DiskBuffer)
      {
      printf("Leider Probleme beim DiskBuffer!\n");
      FreeMemory();
```

```
      exit(FALSE);
      }

   SuffBuffer = AllocMem(10L, MemoryType);
   if (!SuffBuffer)
      {
      printf("Leider Probleme beim SuffBuffer!\n");
      FreeMemory();
      exit(FALSE);
      }


   printf("Speicher reserviert!\n");

   FreeMemory();
   exit(TRUE);
   }


FreeMemory()
   {
   if (UndoBuffer)  FreeMem(UndoBuffer, 512L);
   if (FileBuffer)  FreeMem(FileBuffer, 30L);
   if (DiskBuffer)  FreeMem(DiskBuffer, 512L);
   if (SuffBuffer)  FreeMem(SuffBuffer, 10L);

   printf("Speicher wieder freigegeben!\n");
   }
```

Programm 3.10: Speicherbelegung AllocMem()

Programmbeschreibung

Wichtig für unsere Arbeit ist das Include-File <exec/memory.h>. Es enthält die Speichertypendefinition. Wir fassen auch gleich zwei zusammen und definieren so eine Bezeichnung. Dann benötigen wir die vier Pointer auf die Speicherbereiche. Auch die Funktion liefert solch einen Wert und wird deshalb mitdefiniert.

Im Hauptprogramm wird dann nach und nach versucht, jeden einzelnen Bereich zu erreichen. Klappt dies bei einem nicht, so wird zur Freigaberoutine verzweigt, und das Programm bricht ab. Diese Routine arbeitet ähnlich der bekannten Close_All()-Funktion. Jeder Pointer wird geprüft, ob er auf einen Speicherblock zeigt. Nur wenn er das tut, wird der Speicher wieder freigegeben. So vermeiden wir den Fehler, der auftritt, wenn ein

Speicher freigegeben wird, der gar nicht belegt war (Guru Meditation).

Für das Freigeben nutzen wir eine weitere Exec-Funktion:

```
FreeMem(Memoryblock, ByteSize);
  -210       A1          D0
```

Der Nachteil dieser Methode ist, daß wir bei der Freigabe jeden Speicherbereich einzeln freigeben müssen und auch die Größe bekannt sein muß. Das ist unnötiger Aufwand für den Programmierer. Außerdem erhöht das die Programmzeilenzahl, denn es muß jedesmal wieder getestet werden.

3.11.3 Verbesserung mit AllocEntry()

Natürlich ist das Problem bekannt, daß oft mehrere Speicherbereiche vielleicht unterschiedlicher Eigenschaften gebraucht werden. Die AllocMem()-Funktion ist dafür zwar verwendbar, doch nicht optimal. Hilfe bietet, wie so häufig auf dem Amiga, eine Struktur. Diese Struktur enthält alle Speicherbereiche mit ihren Eigenschaften. Übergeben wir dann den Zeiger auf AllocEntry(), so wird versucht, alle Speicherbereiche zu erreichen.

Die Grundstruktur mit dem Namen MemList:

```
struct MemList
{
0x00  00  struct Node ml_Node;
0x0E  14  UWORD ml_NumEntries;
0x10  16  struct MemEntry ml_ME[1];
0x18  24
};
```

Die altbekannte Node am Anfang wird wie immer zum Verknüpfen gebraucht. Mit *ml_NumEntries* geben wir an, wie viele Einträge die Speicherliste haben wird. Die zweite eingebundene Struktur MemEntry gibt nähere Informationen zum ersten Speicherblock.

```
struct MemEntry
{
union
   {
   ULONG meu_Reqs;
   APTR meu_Addr;
   }
0x00  me_Un;
0x04  04  ULONG me_Length;
0x08  08
};

Definition:
me_un me_Un
me_Reqs me_Un.meu_Reqs
me_Addr me_Un.meu_Addr
```

Der Wert *me_Un* enthält zu Anfang die Definition der Eigenschaften dieses Speicherblocks. Nach Zuweisung findet man hier die Startadresse. In *me_Length* ist die Länge des Speicherbereichs festgehalten.

Aus diesen beiden Strukturen, MemList und MemEntry, müssen wir uns noch eine eigene Struktur zusammensetzen, bevor wir AllocEntry() benutzen können. Dies sieht für unsere Aufgabe so aus:

```
struct Speicherbedarf
   {
   struct MemList Kopf;
   struct MemEntry UndoBuffer;
   struct MemEntry FileBuffer;
   struct MemEntry DiskBuffer;
   struct MemEntry SuffBuffer;
   } Speicher;
```

Struktur 3.35: Speicherbedarf

Diese selbstgeschusterte Struktur wird einfach mit den gewünschten Werten gefüllt. Dann kann AllocEntry() aufgerufen werden. Es wird versucht, den Speicher zu bekommen, und in den einzelnen MemEntry-Strukturen finden wir die Daten.

```
MemList = AllocEntry(Entry);
   D0       -222      A0

          FreeEntry(Entry);
            -228     A0
```

Zum Abschluß hier das komplette Listing:

```
/*************************************
 *                                   *
 * Speicherverwaltung : MemoryEntries *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur für Test- *
 *                     zwecke        *
 *                                   *
 *************************************/


#include <exec/types.h>
#include <exec/memory.h>

#define MemoryType MEMF_FAST | MEMF_CLEAR


struct MemList *SpeicherPtr, *AllocEntry();
APTR UndoData, FileData, DiskData, SuffData;

struct Speicherbedarf
   {
   struct MemList Kopf;
   struct MemEntry UndoBuffer;
   struct MemEntry FileBuffer;
   struct MemEntry DiskBuffer;
   struct MemEntry SuffBuffer;
   } Speicher;


main()
   {
   Speicher.Kopf.ml_NumEntries   = 5;

   Speicher.Kopf.ml_me[0].me_Length = 0;
   Speicher.UndoBuffer.me_Reqs   = MemoryType;
   Speicher.UndoBuffer.me_Length = 512L;
   Speicher.FileBuffer.me_Reqs   = MemoryType;
   Speicher.FileBuffer.me_Length = 30L;
   Speicher.DiskBuffer.me_Reqs   = MemoryType;
   Speicher.DiskBuffer.me_Length = 512L;
   Speicher.SuffBuffer.me_Reqs   = MemoryType;
   Speicher.SuffBuffer.me_Length = 10L;

   SpeicherPtr = (struct MemList *)AllocEntry(&Speicher);

   if ((ULONG)SpeicherPtr & (1<<31))
      {
      printf("Die Liste konnte nicht aufgestellt werden!\n");
```

```
    exit(FALSE);
    }
else
    {
    UndoData = SpeicherPtr->ml_me[1].me_Addr;
    FileData = SpeicherPtr->ml_me[2].me_Addr;
    DiskData = SpeicherPtr->ml_me[3].me_Addr;
    SuffData = SpeicherPtr->ml_me[4].me_Addr;
    }

printf("Speicher gesichert!\n");

FreeEntry(SpeicherPtr);
printf("Speicher wird wieder freigegeben!\n");
exit(TRUE);
}
```

Programm 3.11: Speicherbelegung AllocEntry()

3.11.4 Die beste Lösung

Die beiden vorgestellten Funktionen sind aus der Exec-Library. Wir befinden uns hier aber im Intuition-Kapitel, und deswegen finden Sie hier auch eine von Intuition unterstützte Speicherverwaltung. Diese ist mehr auf die Bedürfnisse eines Intuition-Programmierers zurechtgeschnitten und wird sich für unsere Problemstellung auch als die beste erweisen.

Die beiden Funktionen heißen AllocRemember() und FreeRemember(). Wie der Name schon sagt, hilft uns Intuition beim Merken von reservierten Bereichen. Wir übergeben der Funktion einfach die Größe des gewünschten Speicherbereichs, den Speichertyp und den Zeiger auf eine Remember-Struktur. Zurück bekommen wir entweder eine Fehlermeldung, die durch NULL gekennzeichnet wird, oder die Adresse unseres Speicherbereichs. Die Remember-Struktur wird von Intuition zur Verwaltung benutzt und erinnert an die bisher belegten Speicherbereiche. Nun kommt aber der Clou: Wenn wir die Speicherbereiche freigeben wollen, dann müssen wir FreeRemember() nur den Zeiger auf die Remember-Struktur übergeben, und alle bisher reservierten Blöcke werden wieder freigegeben.

Zuerst die beiden Funktionen mit ihrem allgemeinen Format:

```
MemBlock = AllocRemember(RememberKey, Size, Flags);
    D0          -396          A0        D0     D1

           FreeRemember(RememberKey, ReallyForget);
              -408          A0            D0
```

Die Remember-Struktur sieht wie folgt aus:

```
struct Remember
{
0x00  00  struct Remember *NextRemember;
0x04  04  ULONG RememberSize;
0x08  08  UBYTE *Memory;
0x0C  12
};
```

Strukturbeschreibung

Die Struktur ist eigentlich sehr simpel aufgebaut (aber wirkungsvoll). Zu Anfang steht ein Zeiger auf eine weitere Remember-Struktur. So werden mehrere Speicherblöcke in einer Liste verbunden. Über *RememberSize* erfährt man die Größe des Speicherbereichs, *Memory stellt die Adresse dar.

Der Trick liegt nun darin, daß man sich zuerst nur einen Zeiger auf eine noch nicht vorhandene Remember-Struktur besorgt, der sog. *RememberKey*, und diesen auf NULL setzt. Übergeben wir diesen Zeiger an AllocRemember(), dann wird sofort die erste Remember-Struktur angelegt. Bei jedem weiteren Aufruf mit dem gleichen *RememberKey* wird dann die Remember-Liste verlängert.

Die Arbeit für FreeRemember() ist zum Schluß ganz einfach. Die Funktion geht nach und nach die Liste durch und gibt jeden Bereich frei. Auch der Speicher, den die Liste selbst einnimmt, kann auf Wunsch des Programmierers freigegeben werden.

Sie sehen: Der Vorteil liegt eindeutig in der einfachen Freigabe alle unterschiedlichen Speicherbereiche und in der Organisation, die von Intuition vorgenommen wird. Hier ist das Programm:

```
/*************************************
 *                                   *
 * Speicherverwaltung : Remember     *
 * ================================= *
 *                                   *
 * Autor:  Datum:      Kommentar:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  nur für Test- *
 *                     zwecke        *
 *                                   *
 *************************************/


#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>

#define MemoryType MEMF_CHIP | MEMF_CLEAR


struct IntuitionBase *IntuitionBase;
struct Remember      *RememberPtr = NULL;

VOID *OpenLibrary();

UBYTE *UndoBuffer, *FileBuffer, *DiskBuffer,
                   *SuffBuffer, *AllocRemember();


main()
   {
   if (!(IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", 0L)))
      {
      printf("Keine Intuition Library gefunden!\n");
      exit(FALSE);
      }

   UndoBuffer = AllocRemember(&RememberPtr, 512L, MemoryType);
   FileBuffer = AllocRemember(&RememberPtr,  30L, MemoryType);
   DiskBuffer = AllocRemember(&RememberPtr, 512L, MemoryType);
   SuffBuffer = AllocRemember(&RememberPtr,  10L, MemoryType);

   printf("Speicher reserviert!\n");

   FreeRemember(RememberPtr, TRUE);
   printf("Speicher wieder freigegeben!\n");
   CloseLibrary(IntuitionBase);
   }
```

Programm 3.12: Speicherbelegung AllocRemember()

4. Betriebssystemprogrammierung

Im folgenden wollen wir Ihnen anhand eines längeren Programms das Zusammenspiel der verschiedenen Komponenten des Betriebssystems zeigen. Als Programm haben wir einen Texteditor gewählt, da dieser auf relativ viele Funktionen des Betriebssystems zugreift: Man denke nur an die Speicherverwaltung, die Ausgabe des Textes im Fenster und an Eingaben über die Tastatur. Außerdem können Sie diesen Editor zum Erstellen Ihrer C-Programme benutzen, sobald wir ihn fertiggestellt haben. Damit für Abwechslung gesorgt ist, werden wir Ihnen zwischendurch ein paar Tips und Tricks zeigen, z.B. wie man mit dem Debugger auf Fehlersuche geht oder wie man Assembler-Programme einbindet.

Zusätzlich zu den normalen Fähigkeiten, die jeder Editor besitzt, soll unser Editor noch ein paar kleine Extras für die Erstellung von Programmtexten enthalten. Damit sind nicht nur C-Programme gemeint, vielmehr sollte der Editor flexibel sein und sich an verschiedene Programmiersprachen anpassen lassen.

Geschrieben wird der Editor mit dem Aztec-C-Compiler. Besitzer anderer Compiler brauchen deswegen nicht zu verzweifeln. Das Programm ist so gehalten, daß es mit jedem Compiler übersetzt werden können sollte. Wenn wir auf speziellere Funktionen des Compilers oder seiner Hilfsprogramme (z.B.: Make) eingehen, so werden wir diese soweit beschreiben, daß die Besitzer anderer Compiler diese sinngemäß verwenden können. Dies ist besser, als jedesmal im Text einen Abschnitt für Aztec-C-Besitzer und einen für Lattice-Besitzer (und einen für ???) zu schreiben. In diesem Sinne: Viel Spaß!

4.1 Planung des Editors

Wenn man ein neues Programm anfängt, sollte man sich zuerst einmal hinsetzen und aufschreiben, was einem zu diesem Programm alles einfällt. Dies hilft, Denkfehler vorzeitig zu erken-

nen und große Änderungen im Programm zu vermeiden. Dabei sollte man nicht nur die Funktionen aufschreiben, sondern auch die Ideen, die man zur Realisation von Einzelproblemen hat. Diese kann man später bei der Programmierung verwenden. Beginnen wollen wir dieses Brainstorming damit, daß wir die grundlegenden Forderungen aufstellen, die unser Editor erfüllen soll: schnell, flexibel, programmierbar und komfortabel.

Beginnen wir mit der ersten Forderung: Schnell soll er sein, unser Editor. Zum Problem der Geschwindigkeit gehört die Art und Weise, wie der Editor den Text im Speicher ablegt. Je besser dieses organisiert ist, um so schneller kann der Text manipuliert werden. Wir könnten beispielsweise einen Editorspeicher mit frei wählbarer Größe anlegen und den Text einfach als zusammenhängenden Block darin ablegen. Dies hätte den Vorteil, daß wir nur wenig Verwaltungsaufwand treiben müßten und somit sparsam mit dem Speicher umgingen. Auch ginge das Laden und Abspeichern des Textes in kürzester Zeit vonstatten, da wir den gesamten Text in einem Rutsch abspeichern bzw. laden könnten.

Schwierig wird es dagegen beim Einfügen von Zeichen in den Text. Wenn wir für jedes Zeichen den gesamten nachfolgenden Text im Speicher um ein Zeichen nach hinten schieben wollten, so würde der Editor bereits bei kurzen Texten unerträglich langsam, da stets ein relativ großer Speicherbereich verschoben werden müßte. Günstiger ist es, wenn man die Zeile, in der Eingaben vorgenommen werden, in einen Puffer kopiert und dort die Zeichen einfügt. Da dieser Puffer nur eine geringe Größe hat, geht das Verschieben entsprechend schneller vonstatten. Der Puffer wird erst dann in den Text zurückkopiert, wenn der Benutzer die Zeile mit dem Cursor wieder verläßt.

Dieses Verfahren eignet sich recht gut, wenn Sie Speicherplatz sparen und nicht zu große Texte bearbeiten wollen. Aber auch hier ist irgendwo eine Grenze: Je nachdem, wie schnell der compilierte Code arbeitet, der den Text verschiebt, können recht lange Wartezeiten entstehen; besonders dann, wenn der Text größer als 30 KByte wird. Noch ein Nachteil fällt vor allem beim Scrollen auf: Um von einer Zeile in die nächste zu kom-

men, müssen wir stets die gesamte Zeile nach dem Zeilenende durchsuchen, und das dauert etwas länger, als wenn wir den Anfang der nächsten Zeile einfach berechnen könnten.

Eine andere Möglichkeit, den Text im Speicher abzulegen, besteht in der Verwendung eines Arrays. Dazu legen wir die maximale Anzahl von Zeichen pro Zeile und die maximale Zeilenanzahl fest und definieren ein Array mit den entsprechenden Dimensionen: zeilenfeld[MaxZeile][MaxSpalte]. Dadurch haben wir einen sehr schnellen Zugriff auf jede einzelne Zeile und jedes einzelnen Zeichen innerhalb einer Zeile. Auch das Einfügen von Zeichen innerhalb einer Zeile könnte direkt im Text vorgenommen werden und wäre schnell genug, da wir nur die Zeichen dieser einen Zeile verschieben müßten. Allerdings hat diese Methode auch Nachteile:

- Wenn wir neue Zeilen einfügen wollen, so müssen wir wieder den gesamten Text verschieben.
- Die Größe des Arrays ist festgelegt. Wenn wir einen größeren Text bearbeiten wollen, so müssen wir zuerst die Größe des Arrays ändern, wobei dessen Inhalt gelöscht wird.
- Nicht alle Zeilen nutzen die maximale Zeilenlänge wirklich aus. Die Praxis hat gezeigt, daß stets nur wenige Zeilen wirklich lang sind, während die meisten etwa ein Drittel bis die Hälfte der möglichen Zeilenbreite ausnutzen.

Aufgrund dieser Nachteile werden wir eine andere Methode wählen und die Zeilen als verkettete Liste ablegen. Dann ist die maximale Anzahl von Zeilen nur durch den vorhandenen Speicher begrenzt, und wir kommen schnell von einer Zeile in die nächste, indem wir uns den Zeiger auf die nächste Zeile holen. Die Zeilen haben keine feste Länge, so daß kurze Zeilen auch nur wenig Speicher beanspruchen. Dies hat zwar zur Folge, daß die Speicherverwaltung verkompliziert wird, wie Sie später noch sehen werden, aber dafür haben wir einen guten Kompromiß zwischen Geschwindigkeit und geringem Platzbedarf.

Nachdem wir wissen, wie unsere Zeilen aussehen, müssen wir uns überlegen, wo wir den Speicher für jede einzelne Zeile her-

nehmen. Prinzipiell könnten wir den Speicher jedesmal vom Betriebssystem anfordern. Dies würde allerdings dazu führen, daß der Speicher zerstückelt würde, da das Betriebssystem nicht darauf ausgelegt ist, eine große Anzahl kleiner Speicherbereiche zu vergeben. Besser ist es, wenn wir jeweils einen Block fester Größe (z.B. 5 KByte) vom Betriebssystem anfordern und diesen nach Bedarf selbst zerteilen. Die entsprechenden Funktionen schreiben wir so, daß wir uns später, wenn wir im Programm eine neue Zeile anfordern wollen, keine Gedanken mehr über das "wie" zu machen brauchen. Lassen wir es bei diesem groben Modell bewenden. Ins Detail gehen wir erst, wenn es ernst wird, wir also die Speicherverwaltung implementieren wollen.

Jetzt ein paar Ideen, die uns zum Thema Komfort eingefallen sind: Man sollte die Änderungen, die man in einer Zeile gemacht hat, durch eine Undo-Funktion wieder rückgängig machen können. Da wir die Zeilen zum Editieren ja ohnehin in einen Puffer kopieren müssen, läßt sich die Undo-Funktion einfach dadurch realisieren, daß man die Pufferzeile löscht und wieder den alten Inhalt, der sich ja noch im Speicher befindet, sichtbar macht.

Etwas, was uns beim Arbeiten mit Editoren und Textverarbeitungen immer sehr gestört hat, auch wenn es nur eine Kleinigkeit war, ist, daß sich diese nur selten merken konnten, ob der Text verändert wurde. Beim Verlassen des Editors bekam man auf jeden Fall die Frage "Sind Sie sicher, daß Sie den Text schon abgespeichert haben?" auf dem Monitor angezeigt. Besser ist dies bei Ed gelöst, der nur dann diese Frage stellt, wenn man den Text wirklich verändert hat. Um aber festzustellen, ob der Text verändert wurde, muß bei jeder Veränderung ein Flag gesetzt werden. Dies ist auch der Grund, warum wir dieses Thema hier schon zur Sprache bringen. Vergessen wir nämlich später auch nur an einer einzigen Stelle das Setzen dieses Flags, so funktioniert die ganze Sache nicht, und wir müssen uns auf die mühevolle Suche nach dem Fehler machen.

Ebenfalls von Anfang an im Auge zu behalten ist, daß wir später mehrere Fenster gleichzeitig mit dem Editor bearbeiten können wollen. Dies ist vor allem dann von großem Nutzen, wenn man Textstücke zwischen verschiedenen Texten austauschen will.

Haben Sie nur ein Fenster, so ergibt dies ein lästiges Laden und Abspeichern. Aber wir können nicht so einfach mehrere Fenster öffnen. Jedes Fenster braucht seinen eigenen Editorspeicher, seinen eigenen Cursor und evtl. noch weitere Dinge. Dies bedeutet, das wir alle wichtigen Informationen über den Text wie Anzahl der Zeilen, Cursor-Position etc. nicht global abspeichern dürfen, sondern in einer Struktur, von der es dann für jedes Editorfenster genau eine gibt. Ferner gibt es dann noch einen Zeiger auf die aktuelle Struktur, deren Fenster gerade aktiv ist. Dies alles bedeutet zwar am Anfang mehr Arbeit für uns, aber die Implementation mehrerer Fenster wird dafür hinterher um so einfacher.

Eine interessante Fähigkeit des Amiga ist die Möglichkeit, jede Taste mit einer beliebigen Zeichenkette zu versehen. So können Sie z.B. einstellen, daß der Amiga jedesmal, wenn Sie Alt + "Ä" drücken, ein "ae" sendet oder "\344" (Oktal-Code von "ä"). Oder Sie können sich aussuchen, ob die Return-Taste ein Linefeed (LF), ein Return (CR) oder ein CRLF sendet. Oder Sie können Sonderzeichen auf beliebige Tasten legen, an die Sie sonst nicht herankommen. Die Möglichkeiten, die Ihnen hierdurch gegeben werden, lassen sich wahrscheinlich erst im praktischen Einsatz überblicken.

Auch sollte unser Editor mit echten Tabulatoren arbeiten können und nicht einfach nur Blanks einfügen. Wenn Sie Ihre Programme schön strukturieren (einrücken) wollen, so müssen Sie beim Arbeiten mit Ed viele Blanks einfügen, die auch viel Speicher kosten. Wenn Sie einen Aztec-C-Compiler besitzen, so können Sie natürlich dessen Editor Z benutzen, der echte Tabulatoren einfügt. Allerdings ist die Bedienung von Z nichts für sanfte Gemüter, denn es bedarf schon einer gehörigen Eingewöhnungszeit, bis man damit umgehen kann. Am besten wäre es natürlich, wenn man die Tabulatoren beliebig setzen könnte, also nicht nur jede dritte (vierte, fünfte...) Spalte, sondern einen in Spalte 3, einen in Spalte 10 und einen in Spalte 20. Dies ist zwar aufwendiger zu programmieren, macht den Editor aber flexibler.

Auto-Indent hängt ebenfalls, wenn auch nicht direkt, mit Tabulatoren zusammen. Darunter versteht man, daß der Editor,

wenn Sie am Ende einer Zeile Return drücken, nicht in die erste Spalte der nächsten Zeile geht, sondern in die Spalte, in der auch in der darüberliegenden Zeile der erste Buchstabe steht (bzw. das erste Zeichen, das weder Blank noch Tabulator ist). Wenn Sie Ihre Programme ordentlich strukturieren, so sparen Sie sich mit Auto-Indent jede neue Zeile auch neu einrücken zu müssen.

Der Editor sollte auch Sonderzeichen anzeigen und nicht einfach wie Ed "file contains binary" melden und abbrechen. Mit Sonderzeichen sind insbesondere die Zeichen mit den ASCII-Codes unterhalb von Space gemeint. Da diese Zeichen nicht im Zeichensatz des Amiga vorhanden sind, könnte man diese in einer anderen Farbe ausgeben, wobei man auf den ASCII-Code des Zeichens einen konstanten Wert (z.B. 64) addiert, so daß man wieder ein ausgebbares Zeichen erhält. So würde das Zeichen mit dem ASCII-Code Eins beispielsweise als rotes A im Text erscheinen (Code(A) = 65 = 1 + 64). Ferner könnte man bei der Ausgabe die reservierten C-Befehlswörter im Fettdruck ausgeben lassen. Dies würde die Programme noch übersichtlicher machen.

Zum Thema Komfort ist uns schließlich noch das Folding eingefallen. Der Vorteil von Folding ist, daß man einzelne Programmteile, die eine bestimmte Funktion zu erfüllen haben, einfach wegfalten (unsichtbar machen) kann. Dann sieht man nur noch einen Kommentar, aus dem die Funktion des entsprechenden Programmteils hervorgeht. Um an den Text zu gelangen, der in der Falte steht, kann man entweder in die Falte eintreten, so daß man dann nur noch die Falte sieht, oder diese auffalten. Die Programme werden dadurch übersichtlicher und besser lesbar.

Wenden wir uns nun der Programmierbarkeit unseres Editors zu. Alle Funktionen des Editors sollten sich über einfache Kommandos aufrufen lassen. Aus diesen Kommandos lassen sich Kommandofolgen bilden, indem man die einzelnen Kommandos durch ein Semikolon trennt. Ferner sollten sich Kommandofolgen klammern lassen und einzelne Kommandos oder geklammerte Kommandofolgen mehrfach ausführen lassen, indem man eine Zahl direkt davor schreibt. Kommt Ihnen bis jetzt alles

recht bekannt vor, hmm? Richtig, soviel kann Ed auch, doch wir wollen noch ein paar Schritte weitergehen.

Zuerst erlauben wir die Verwendung von Variablen, und zwar von Variablen, die Zahlen (Cursor-X/Y-Position o.ä.), Strings oder Kommandofolgen enthalten. Schreiben wir eine Zahl-Variable vor ein Kommando, so wird dieses sooft ausgeführt, wie der Inhalt der Variablen angibt. Außerdem erlauben wir erweiterte Kontrollstrukturen, also For-To-Do-Schleifen, While-Do-Schleifen, und If-Then-Else. Selbstverständlich können wir die Funktionstasten beliebig mit solchen Kommandofolgen belegen. Weitere Ideen werden uns bei den einzelnen Befehlen noch einfallen, z.B. die Möglichkeit, beim Suchen und Ersetzen einen Bereich (von Zeile bis Zeile) anzugeben, in dem gesucht werden soll.

Ebenfalls nicht schlecht wäre es, wenn wir CLI-Befehle direkt vom Editor aus aufrufen könnten. Dann bräuchten wir den Editor nämlich gar nicht mehr zu verlassen, um z.B. auf einen Tastendruck unseren Text abzuspeichern und den Compiler aufzurufen.

Nachdem wir nun bereits eine ganze Menge an Ideen zusammengetragen haben, die wir im Zuge dieses Buches gar nicht alle realisieren können, müssen wir uns noch Gedanken über die Datenstrukturen machen, auf denen unser Editor aufbaut. Die Datenstrukturen gehören zu den wichtigsten Dingen, über die man sich vor der Programmierung Gedanken machen muß. So ist es z.B. ein besonderes Merkmal des Amiga, daß alle Datenstrukturen, auf die das Betriebssystem zugreift, ähnlich aufgebaut sind; man denke nur an die Listen. Dies erleichtert die Manipulation dieser Objekte. Die erste Datenstruktur unseres Editors ist die Zeile:

```
struct Zeile
{
   struct Zeile *succ;        Zeiger auf nächste Zeile
   struct Zeile *pred;        Zeiger auf vorige Zeile
   UBYTE flags;               Diverse Flags
   UBYTE len;                 Länge der Zeile
   /* UBYTE zeile[] */
}
```

Mit den ersten beiden Zeigern wird später die verkettete Liste aufgebaut. Zwei Zeiger brauchen wir deswegen, weil wir ja nicht nur von einer Zeile in die nächste, sondern auch in die vorige kommen wollen. Außerdem stellt uns der Amiga Funktionen für doppelt verkettete Listen zur Verfügung, die wir auch für unsere Zeilen verwenden können. Ist (flags & 128) ungleich null, so befindet sich die Zeile gerade im Puffer, weil der Benutzer diese zu ändern gedenkt. Dieses Flag werden wir bei der Ausgabe brauchen, da wir den Puffer ausgeben müssen, wenn wir auf eine Zeile stoßen, bei der dieses Flag gesetzt ist. Weitere Flags werden wir nach Bedarf definieren.

Die Bedeutung von len dürfte dagegen klar sein. Es gibt die gesamte Länge der Zeile an, also inklusive CR, LF oder CRLF, die ja das Zeilenende signalisieren. Auf diese Art und Weise können wir auch Zeilen ohne ein Zeilenende zulassen, zum Beispiel dann, wenn eine Zeile länger werden soll, als der Bildschirm breit ist. Anstatt diese Zeile nach rechts aus dem Bildschirm heraus fortzusetzen, könnten wir sie am rechten Bildschirmrand trennen und daraus zwei Zeilen machen, wovon die erste nicht durch ein CR, LF oder CRLF beendet wird. Auf diese Art und Weise werden übrigens bei Textverarbeitungen Absätze behandelt.

Die Zeile selbst folgt direkt hinter len. Da ihre Länge nicht festgelegt ist, können wir sie auch nicht in der Struktur aufführen. Die Länge der Struktur einschließlich Zeile ergibt sich zu:

```
Gesamtlänge = (sizeof(struct Zeile) + Zeile.len + 1) & -2.
```

Die Gesamtlänge muß geradzahlig sein, da beim Amiga Zeiger (oder generell: Wörter und Langwörter) an geraden Adressen im Speicher liegen müssen. Dies setzt die CPU MC68000 voraus, andernfalls erhalten Sie eine Guru-Meditation mit der Fehlernummer 00000003.

Die nächste Struktur gehört zu den Speicherblöcken, in denen die Zeilen liegen. Die Speicherblöcke sind ebenfalls über eine verkettete Liste miteinander verbunden. Ferner benötigen wir einen Zeiger auf eine Liste aller freien Speicherstücke, die in

diesem Block liegen, die Länge des Blocks und die Anzahl der unbenutzten Bytes. Das Ganze sieht also wie folgt aus:

```
struct Speicherblock
{
   struct Speicherblock *succ;      Nächster Speicherblock
   struct Speicherblock *pred;      Voriger Speicherblock
   ULONG laenge;                 Maximale Anzahl der freien Bytes
   ULONG frei;                   Momentane Anzahl der freien Bytes
   struct FList freiliste;       Liste der unbelegten Stücke
}
```

Die ersten beiden Variablen dienen wieder dem Einfügen des Speicherblocks in eine doppelt verkettete Liste. Die Größe des Blocks ohne die Speicherblock-Struktur gibt laenge an. Dies entspricht der Anzahl der freien Bytes, die in einem unbenutzen Block vorhanden sind, abzüglich der Länge einer Speicherstueck-Struktur, die nachfolgend beschrieben wird. Frei dagegen gibt an, wie viele Bytes in diesem Block tatsächlich noch frei sind. Jetzt tritt das Problem auf, wo diese freien Bytes liegen. Klar ist, daß diese nicht unbedingt hintereinander im Speicher liegen, sondern an beliebigen Stellen. Wenn nämlich eine Zeile gelöscht wird, so entsteht ein freies Stück Speicher, das nicht zwingend an einem anderem Stück freien Speicher liegen muß. Da jede Zeile mindestens 10 Bytes lang ist (sizeof(struct Zeile)!), haben wir 10 Bytes zum Aufbauen einer Liste aller freien Speicherstücke eines Blocks zur Verfügung. Jedes Element dieser Liste hat folgenden Aufbau:

```
struct Speicherstueck
{
   struct Speicherstueck *succ;
   struct Speicherstueck *pred;
   UWORD len;
}
```

Die ersten beiden Elemente wurden mit Rücksicht auf die Listen des Betriebssystems gewählt. Dabei handelt es sich auch um doppelt verkettete Listen, für deren Verwaltung eine Reihe von Funktionen in der Exec-Library bestehen. Dadurch, daß wir dieselbe Verkettung benutzen, können wir diese Funktionen auch für unsere Listen verwenden und müssen keine eigenen schreiben. Das Element len enthält diesmal die Länge des gesamten Speicherstücks, also einschließlich der Speicherstueck-

Struktur. Was noch fehlt, ist die FList-Struktur, die denselben Aufbau hat wie der Listenkopf einer Liste des Betriebssystems:

```
struct FList
{
   struct Speicherstueck *head;
   struct Speicherstueck *tail;
   struct Speicherstueck *tailpred;
}
```

Die Struktur sollte Ihnen von den Exec-Lists bekannt sein, so daß wir diese hier nicht noch einmal erklären wollen (siehe Anhang: Exec-Library). Es sei nur soviel gesagt, daß head auf das erste und tailpred auf das letzte Element der Liste zeigen, während tail immer Null ist. Diese Vereinbarung vereinfacht das Einfügen und Löschen in Listen.

Ziemlicher Aufwand für die Verwaltung der Zeilen, meinen Sie? Tja, da haben Sie nicht ganz unrecht. Wir könnten doch den Speicher für jede Zeile einfach vom Betriebssystem anfordern? Im Prinzip schon, aber ... Bedenken Sie, daß ein Text aus 1000 und mehr Zeilen bestehen kann. Jedesmal, wenn Sie eine Zeile ändern, müssen Sie, wenn sich deren Länge ändert, den alten Speicher freigeben und neuen allokieren. Wenn Sie den Text einige Zeit editiert haben, sieht Ihr Speicher aus wie ein Schweizer Käse: ein Loch neben dem anderen.

Nun meinen Sie, daß es uns nicht anders ergehen würde, wenn wir das Problem selbst in die Hand nehmen? Das stimmt nicht ganz! Wenn wir eine neue Zeile anfordern, so werden wir zuerst nach einem Speicherstück suchen, das genau die gesuchte Länge hat. Finden wir ein solches nicht, so nehmen wir den Speicher von dem größten Speicherstück, das wir haben. Dadurch stellen wir sicher, daß wir kleine Speicherstücke nicht unnötig weiter verkleinern. Stellen Sie sich vor, wir brauchen 14 Bytes, und das nächstgrößere Speicherstück faßt 16 Bytes. Würden wir dieses aufteilen, so erhielten wir zwei Speicherstücke, wovon eines 14 und das andere 2 Bytes groß wäre. Diese zwei Bytes würden allerdings nicht mehr dazu ausreichen, das Speicherstück wieder in die Liste aller freien Speicherstücke einzuhängen, da wir dazu mindestens 10 Bytes für die Speicherstueck-Struktur benötigten.

Erst wenn wir nicht mehr genug Platz zur Verfügung haben, allokieren wir einen neuen Speicherblock und fügen ihn in die Blockliste ein.

Was uns zur vorerst letzten Struktur bringt. Die Struktur, ohne die nichts läuft, ist die Editor-Struktur, von der für jedes Editorfenster eine existiert und die alle wichtigen Daten aufnimmt:

```
struct Editor
{
   struct Editor *succ;
   struct Editor *pred;
   struct Window *window;          Zeiger auf das Editorfenster
   struct BList block;             Liste der Speicherblöcke
   struct ZList zeilen;            Liste der Zeilen
   UBYTE puffer[MAXBREITE];        Puffer zum Editieren
   UBYTE tabstring[MAXBREITE];     Tabulatoren
   UWORD anz_zeilen;               Anzahl der Zeilen
   struct Zeile *aktuell;          Zeiger auf aktuelle Zeile
   struct Zeile *top;              Zeiger auf oberste Zeile,
                                   die im Bildschirm sichtbar ist.
   UWORD toppos;                   Nummer der obersten Zeile
   UWORD xpos,ypos;                Cursor-Position
   UWORD changed:1;                Ist 1, falls Text verändert
   UWORD insert:1;                 Ist 1, falls Einfügemodus
}
```

Dazu gehören noch zwei weitere Strukturen, die aber einfach nur Listenköpfe darstellen:

```
struct BList
{
   struct Speicherblock *head;
   struct Speicherblock *tail;
   struct Speicherblock *tailpred;
}

struct ZList
{
   struct Zeile *head;
   struct Zeile *tail;
   struct Zeile *tailpred;
}
```

Die einzelnen Elemente der Editor-Struktur dürften Ihnen eigentlich keine Probleme bereiten, und falls doch, so werden diese spätestens beim Programmieren gelöst. Allerdings sei schon jetzt gesagt, daß die Editor-Struktur noch um einige Elemente

erweitert werden wird, da sich bei der Programmierung zeigen wird, daß noch die eine oder die andere Variable nützlich für den Editor wäre, an die wir in diesem frühen Stadium noch nicht gedacht haben.

4.2 Ein Programmgerüst

Beginnen wollen wir die Programmierung mit einem Gerüst, in das wir dann unseren Editor einbauen werden. Das folgende Programm öffnet nur die Librarys, über die auf dem Amiga auf die Funktionen des Betriebssystems zugegriffen wird, und gibt eine Meldung aus, wenn dies erfolgreich geschehen ist. Beachten Sie bitte, daß Sie nur dann Funktionen einer Library aufrufen können, wenn Sie diese zuvor mit OpenLibrary geöffnet haben. Auch gehört es zum guten Ton, am Programmende alle geöffneten Librarys wieder zu schließen.

```
<Editor.c>

/************
 *
 * Includes:
 *
 ************/

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

/***********
 *
 * Defines:
 *
 ***********/

#define REV 33L

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

struct Library *OpenLibrary();
void CloseLibrary();
```

```
/*********************
 *
 * Globale Variablen:
 *
 *********************/

struct Library *IntuitionBase = NULL, *GfxBase = NULL;
struct Library *DosBase = NULL;

/******************
 *
 * Hauptprogramm:
 *
 ******************/

main()
{
   if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
      goto Ende;

   if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
      goto Ende;

   if ( !(DosBase = OpenLibrary("dos.library",REV)))
      goto Ende;

printf("Alles geöffnet\n");

Ende:
   if (DosBase) CloseLibrary(DosBase);
   if (GfxBase) CloseLibrary(GfxBase);
   if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```

Die #include-Befehle binden die Definitionen der Datenstrukturen ein, auf die wir uns beziehen, wenn wir auf Betriebssystemroutinen zugreifen. Wenn wir diese Befehle wegließen, so würde der C-Compiler mehrere Fehlermeldungen wegen unbekannter Strukturen ausgeben. Auch sollten Sie alle Funktionen, die Sie innerhalb eines Programms verwenden, zuvor deklarieren, damit der C-Compiler überprüfen kann, ob Sie den Wert einer Funktion auch an eine Variable vom richtigen Typ übergeben.

Beim Compilieren wird Ihnen die lange Zeit auffallen, die der Compiler braucht, um die Include-Dateien einzulesen. Weil dies anscheinend auch die Entwickler des Aztec-C-Compilers gestört hat, haben Sie eine Möglichkeit eingebaut, solche Include-Da-

teien zu präcompilieren. Diese präcompilierten Includes lassen sich erheblich schneller einbinden als die normalen Includes. Zu diesem Zweck schreiben Sie eine Datei, die nur die #include-Anweisungen Ihres Quelltextes enthält. In unserem Beispiel sähe die Datei also wie folgt aus:

```
<Editor.prelist>

    #include <exec/types.h>
    #include <intuition/intuition.h>
```

Diese Datei wird nun compiliert, wobei Sie allerdings dem Compiler mitteilen müssen, daß er eine Datei mit den präcompilierten Includes erzeugen soll. Dies geschieht beim Aztec mit der Option +H:

```
cc +Hpre/editor.pre pre/editor.prelist
```

Dabei sind wir davon ausgegangen, daß sich die Datei mit den Include-Anweisungen unter dem Namen "editor.prelist" im pre-Verzeichnis befindet. Erzeugt wird im selben Verzeichnis die Datei "editor.pre", die Sie mit der Option +I beim Compilieren des eigentlichen Programms einbinden können:

```
cc +Ipre/Editor.pre src/Editor.c
```

Das Verzeichnis src (Source = Quelltext) soll dabei die Datei Editor.c beinhalten. Sie werden feststellen, daß der Compiler nun wesentlich schneller arbeitet. Eine weitere Möglichkeit zur Vereinfachung der Übersetzung besteht in der Verwendung von Make. Make kommt immer dann zum Einsatz, wenn ein Programm aus mehreren Modulen besteht, die erst beim Linken zusammengebunden werden. Bei größeren Programmen ist es sinnvoll, diese in kleinere Teilprogramme - Module - zu zerlegen, die getrennt übersetzt und dann erst vom Linker zusammengebunden werden. Dadurch bleibt das Programm übersichtlich, und bei Änderungen muß jeweils nur das geänderte Modul neu übersetzt werden und nicht das gesamte Programm.

Make übernimmt das Compilieren und das Linken der Module, so daß Sie sich nicht mehr darum kümmern müssen. Sie sagen Make, aus welchen Modulen Ihr Programm besteht und wie

diese zu compilieren und zu linken sind. Wenn Sie Make aufrufen, so überprüft es anhand des Datums und der Uhrzeit, ob Sie Teile Ihres Programms geändert haben, übersetzt ggf. diese Module neu und linkt alle Module zum fertigen Programm zusammen.

Damit Make dies tun kann, teilen Sie ihm in der Make-Datei (Make-File) mit, welche Dateien Ihres Programms von welchen anderen Dateien abhängen. So hängt z.B. die Objektdatei eines Moduls von dessen Quelltext ab, und das fertige Programm hängt von den Objektdateien aller Module ab. Aufgrund dieser Abhängigkeiten und aufgrund des Datums und der Uhrzeit, zu der diese Dateien erzeugt worden sind, bestimmt Make, ob welche geändert wurden und welche Schritte auszuführen sind, um das Programm neu zu erstellen. Es ist also wichtig, daß die Uhrzeit und das Datum in Ihrem Amiga stets richtig eingestellt sind.

Um dies etwas anschaulicher zu machen, wollen wir nun Make verwenden, um unseren Editor übersetzen zu lassen. Dazu schreiben wir zunächst einmal auf, aus welchen Dateien unser Editor bisher besteht:

```
Editor
src/Editor.o
src/Editor.c
pre/Editor.pre
pre/Editor.prelist
```

Eigentlich gehören die Include-Dateien auch noch dazu, aber da wir diese nicht ändern werden und da diese in pre/Editor.pre schon enthalten sind, können wir auf sie verzichten. Wie hängen nun diese fünf Dateien voneinander ab? Das fertige Programm Editor hängt sicherlich von dem Objektmodul src/Editor.o ab, aus dem es gelinkt wird. Das dabei auch die Bibliotheken eine Rolle spielen, ist hier nicht von Belang, da wir diese nicht ändern werden. Die Abhängigkeiten brauchen wir nur für die Dateien aufzustellen, die wir im Laufe der Programmentwicklung zu ändern gedenken.

Das Objektmodul src/Editor.o hängt seinerseits vom Quelltext src/Editor.c und von den präcompilierten Includes pre/Editor.

pre ab; diese wiederum von pre/Editor.prelist, die die Namen der Includes enthält, die präcompiliert werden sollen. Wenn wir für Make die Abhängigkeiten aufschreiben wollen, so schreiben wir den Dateinamen in die erste Spalte einer Zeile, gefolgt von einem Doppelpunkt. Dahinter schreiben wir die Dateien, von denen diese Datei abhängt. Für unseren Editor sähe das Ganze wie folgt aus:

```
Editor:          src/Editor.o
src/Editor.o:    src/Editor.c pre/Editor.pre
pre/Editor.pre: pre/Editor.prelist
```

Die Dateien src/Editor.c und pre/Editor.prelist hängen von keinen anderen Dateien ab. Damit Make weiß, wie alle diese Dateien zu compilieren bzw. zu linken sind, müssen wir noch einige Anweisungen in die Make-Datei schreiben. Dabei schreiben wir diese jeweils direkt unter die Zeile, die die Abhängigkeit beschreibt. Die Befehle dürfen übrigens nicht in der ersten Spalte beginnen, da dies den Abhängigkeiten vorbehalten ist. Sie sollten also mindestens ein Blank davor setzen. Sie können hinter jede Abhängigkeit beliebig viele Anweisungen setzen. Make führt diese solange aus, bis es wieder an eine Abhängigkeit oder an das Ende der Make-Datei gelangt.

Stellen wir nun die Anweisungen zusammen, die unseren Editor erzeugen. Um aus dem Objektmodul das fertige Programm zu machen, muß folgender Befehl ausgeführt werden:

```
ln src/Editor.o -lc -o Editor
```

Zum Übersetzen des Quelltextes:

```
cc +Ipre/Editor.pre src/Editor.c
```

Zum Erzeugen der präcompilierten Includes wollen wir zwei Anweisungen verwenden. Wenn Sie diese so erzeugen, wie wir es oben beschrieben haben, so erzeugt der C-Compiler eine Objektdatei, obwohl dies gar nicht nötig wäre, da uns ja nur die precompilierten Includes interessieren. Auch wird der Assembler mit aufgerufen, was zusätzlich Zeit kostet. Wir werden daher das Aufrufen des Assemblers mit der Option -A unterbinden und

die Assembler-Datei, die der Compiler trotzdem erzeugt, auf die RAM-Disk schreiben lassen und anschließend löschen:

```
cc -A +Hpre/Editor.pre pre/Editor.prelist -O ram:Ed.asm
delete ram:Ed.asm
```

Unsere Make-Datei zum Erzeugen des Editors sieht damit wie folgt aus:

```
Editor: src/Editor.o
   ln src/Editor.o -lc -o Editor

src/Editor.o: src/Editor.c pre/Editor.pre
   cc +IpreEditor.pre src/Editor.c

pre/Editor.pre: pre/Editor.prelist
   cc -A +Hpre/Editor.pre pre/Editor.prelist -O ram:Ed.asm
   delete ram:Ed.asm
```

Make erzeugt nun die Datei, deren Namen sich zuerst in der Make-Datei findet. In unserem Fall also Editor. Wollen Sie eine andere Datei erzeugen lassen, z.B. src/Editor.o, so geben Sie deren Namen bereits beim Aufruf vom CLI aus an:

```
make src/Editor.o
```

Nun wollen wir es damit aber nicht bewenden lassen, sondern Make noch ein bißchen weiter ausnutzen. Da unser Programm später aus mehreren Modulen besteht, können wir die Make-Datei jetzt schon mal darauf vorbereiten. Als erstes wollen wir die Objektmodule, aus denen unser Editor besteht, in einer Variablen definieren:

```
OBJ=src/Editor.o
```

Die Variable muß dabei direkt am Anfang der Zeile stehen, gefolgt von einem Gleichheitszeichen. Dahinter folgt die Auflistung der Objektmodule. Nun fragen Sie sich sicherlich, ob dies noch einen anderen Zweck hat als die Make-Datei übersichtlicher zu gestalten. Es hat! Da wir später mit an Sicherheit grenzender Wahrscheinlichkeit den Debugger benutzen müssen, um unser Programm von etwaigen Fehlern zu befreien, ist es wünschenswert, das Programm mit der Option -w zu linken. Diese Option sorgt dafür, daß wir, wenn wir den Debugger benutzen,

auf die Namen von Funktionen und globalen Variablen zugreifen können, was das Debuggen für uns extrem vereinfacht. Nun haben wir in unserer Make-Datei zwei Link-Anweisungen, in denen die Namen sämtlicher Objektmodule auftauchen. Eine Link-Anweisung erzeugt ganz normal den Editor, während die andere zusätzlich eine Symboltabelle einbindet, also mit Option -w linkt. Verwenden wir dagegen die Variable OBJ, so brauchen wir die Objektmodule nur einmal zu definieren und geben bei den Link-Anweisungen nur die Variable an:

```
OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor
```

Um die Variable benutzen zu können, muß diese geklammert und vor die Klammer ein $ geschrieben werden. Stellen Sie sich nun vor, unser Programm bestände aus zwanzig Modulen (20!), und Sie müßten diese für beide Link-Befehle aufschreiben. So definieren Sie die Module einmal und können sicher sein, daß bei beiden Link-Anweisungen das gleiche Programm erzeugt wird und daß Sie kein Modul bei einer der Anweisungen vergessen haben.

Wenn Sie den Editor mit der Option -w linken lassen wollen, so rufen Sie Make so auf:

```
make Debug
```

Make stellt dann fest, daß die Datei Debug gar nicht existiert und leitet daher alle Schritte ein, die zu deren Erzeugung laut Make-Datei notwendig sind. Daß dabei gar keine Datei namens Debug erzeugt wird, stört Make nicht.

Aber kommen wir auf die zwanzig Module zurück, aus denen unser Programm bestehen könnte. Wenn Sie für jedes Objektmodul angeben wollten, wie dies aus dem entsprechenden Quelltext zu erzeugen ist, so würden Sie auch hier viel Platz vergeuden und sich unnötig Arbeit machen. Um dies zu verhindern, kennt Make Regeln. Eine Regel gibt an, wie aus einer Datei mit einer

bestimmten Endung (z.B.: .c) eine Datei mit demselben Namen, aber einer anderen Endung (z.B.: .o) zu erzeugen ist:

```
.c.o:
   /* Rufe Compiler auf */
```

Anders als bei den Abhängigkeiten wird hier zuerst die Endung aufgeschrieben, von der die zweite Endung abhängt. Die Reihenfolge ist also vertauscht! Auch ist die Schreibweise anders als bei den Abhängigkeiten: Zuerst wird die Endung, die übriges mit einem Punkt beginnen muß, von der die zweite Endung abhängt, an den Anfang der Zeile geschrieben. Direkt dahinter folgt die zweite Endung, die wiederum mit einem Punkt beginnen muß. Abgeschlossen wird die Zeile mit einem Doppelpunkt. In die folgenden Zeilen kommen die Anweisungen, wie Sie es bereits von den Abhängigkeiten her gewohnt sind.

Da Sie bei den Regeln keine direkten Dateinamen angeben können, stellt Ihnen Make zwei Variablen zur Verfügung, die beide den Dateinamen enthalten, auf den diese Regel angewendet werden soll. Dabei entspricht $@ dem vollen Dateinamen und $* nur dem Namen der Datei ohne Endung und ohne Pfadname. Der Dateiname hat übrigens stets dieselbe Endung wie die zweite Endung der Regel. Nehmen wir als Beispiel src/Editor.o. Dann wäre $@ = src/Editor.o und $* = Editor. Stellen wir nun eine Regel auf, wie aus einer prelist-Datei eine pre-Datei zu erzeugen ist:

```
.prelist.pre:
   cc -a -o ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm
```

Der einzige Unterschied zur oben definierten Abhängigkeit ist, daß die Datei, die auf der RAM-Disk erzeugt wird, den Namen Editor.preasm bekommt, wenn die Datei pre/Editor.pre erzeugt werden soll. Nun können wir die Anweisungen, die oben auf unsere Abhängikeit gefolgt sind, entfernen, da diese von der Regel bestimmt werden.

Die nächste Regel, die wir definieren wollen, gibt an, wie aus einem Quelltext eine Objektdatei zu erzeugen ist. Hier sollten wir beachten, daß unser Programm, wenn es auch aus mehreren

Modulen besteht, nur ein Hauptmodul enthält, in dem die Funktion main() definiert ist. Nun erlaubt es der C-Compiler durch Setzen des Flags +B bei allen anderen Modulen etwas Speicher zu sparen. Während das Hauptmodul nämlich mit einem Sprung in eine Initialisierungsroutine startet, ist dies bei den anderen Modulen nicht notwendig, da diese Routine nur einmal vor Beginn von main aufgerufen werden muß. Wir werden die Regel so definieren, daß sie alle Module außer dem Hauptmodul erzeugen kann:

```
.c.o:
   cc +B +Ipre/Editor.pre -O $@ src/$*.c
```

Nun stellt unser Hauptmodul aber die Ausnahme von dieser Regel dar. Hier ist es jedoch ausreichend, hinter die Abhängigkeitsdefinition für das Hauptmodul src/Editor.o eine entsprechende Anweisung zu schreiben, so wie wir es oben bereits getan haben. Denn nur dann, wenn Make zu einer Abhängigkeit keine Anweisungen vorfindet, benutzt es eine entsprechende Regel. Unsere Make-Datei ist nun fertig und sie hat folgendes Aussehen:

<makefile>

```
.c.o:
   cc +B +Ipre/Editor.pre -O $@ src/$*.c

.prelist.pre:
   cc -A -O ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm

OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor

pre/Editor.pre: pre/Editor.prelist
src/Editor.o: src/Editor.c pre/Editor.pre
   cc +Ipre/Editor.pre src/Editor.c
```

Nehmen wir jetzt noch eine weitere Datei für den Editor in Angriff, die später häufig verwendet werden wird, nämlich die Datei, die die Datenstrukturen des Editors definiert:

```
<src/Editor.h>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>

/*******************
 *
 * Datenstrukturen:
 *
 *******************/

struct Zeile
{
   struct Zeile *succ;
   struct Zeile *pred;
   UBYTE flags;
   UBYTE len;
};

#define ZLF_USED 128
```

Diese Datei enthält momentan nur die Definition für Zeilen, was sich aber noch ändern wird. Die Make-Datei müssen wir auch dementsprechend anpassen:

```
<makefile>

.c.o:
   cc +B +Ipre/Editor.pre -O $@ src/$*.c

.prelist.pre:
   cc -A -O ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm

OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor

pre/Editor.pre: pre/Editor.prelist src/Editor.h
src/Editor.o: src/Editor.c src/Editor.h pre/Editor.pre
   cc +Ipre/Editor.pre src/Editor.c
```

Editor.h wird also momentan von Editor.pre und Editor.o benutzt. Um besser zu verstehen, was beim Aufruf von Make passiert, geben Sie doch mal Editor.c, Editor.h, Editor.prelist und das "makefile" ein. Nun müssen Sie noch Make auf Ihre Diskette kopieren und dann "make Editor" aufrufen. Zuerst stellt Make dann fest, daß Editor von src/Editor.o abhängt, welches noch nicht existiert. Daraufhin merkt sich Make, daß es src/Editor.o erzeugen muß und stellt fest, daß dieses von src/Editor.c, src/Editor.h und pre/Editor.pre abhängt. Während die beiden ersten Dateien bereits existieren, ist dies bei der dritten Datei nicht der Fall. Also merkt sich Make wieder, daß es pre/Editor.pre erzeugen soll, und schaut nach, wovon dieses abhängt. pre/Editor.pre hängt von pre/Editor.prelist und von src/Editor.h ab. Da beide existieren, kann Make aufgrund der Regel ".prelist.pre" die Datei pre/Editor.pre erzeugen, indem es den folgenden Befehl ans CLI schickt:

```
cc -A -O ram:Editor.preasm +Hpre/Editor.pre pre/Editor.prelist
```

und danach die überflüssige Datei ram:Editor.preasm löscht:

```
delete ram:Editor.preasm
```

Nachdem nun pre/Editor.pre existiert, kann Make src/Editor.o erzeugen, und es führt daher folgenden Befehl aus:

```
cc +Ipre/Editor.pre src/Editor.c
```

Zwar gibt es auch eine Regel, um aus einer *.c-Datei eine Objektdatei zu machen, da wir aber für src/Editor.o extra eine Anweisung angegeben haben, wird die Regel ignoriert. Zum Schluß wird dann noch der Editor selbst erzeugt, nachdem die Objektdatei ja existiert:

```
ln src/Editor.o -lc -o Editor
```

Damit haben wir nun nicht nur die allererste Version unseres Programms fertig, sondern auch eine stabile Grundlage, auf der aufbauend wir uns der weiteren Programmierung des Editors widmen können.

4.3 Schritt für Schritt

Im folgenden werden wir beschreiben, wie wir Schritt für Schritt den Editor aufbauen, bis wir am Ende soweit sind, diesen benutzen zu können.

4.3.1 Öffnen eines Fensters

Zuerst soll unser Editor ein Fenster öffnen. Nun könnten wir dazu eines der Beispielprogramme aus Kapitel 2 kopieren und entsprechend unseren Wünschen anpassen. Da wir später jedoch mehrere Fenster öffnen können wollen, müssen wir ohnehin unsere eigenen Routinen schreiben, die ein Fenster öffnen und dieses in eine zuvor beschaffte Editor-Struktur einbinden. Des weiteren brauchen wir eine Funktion, die ein Fenster wieder schließt und die dazugehörige Editor-Struktur freigibt.

Doch zunächst zum Öffnen eines Fensters. Als erstes soll diese Funktion versuchen, sich Speicher für die Editor-Struktur zu beschaffen. Ist dies gelungen, so wird ein Fenster geöffnet und der Zeiger auf dieses Fenster in die Struktur eingetragen. Dann muß die Struktur initialisiert werden, was insbesondere für die Speicherblock- und Zeilen-Listen gilt. Die Funktion zum Schließen des Fensters schließt analog zuerst das Fenster und gibt dann den Speicher wieder frei, den die Editor-Struktur besetzt hatte.

Im Prinzip wäre das schon alles, aber... wenn wir es wirklich so machen würden, so hätten wir später, wenn wir tatsächlich mehrere Fenster einbauen, einige Änderungen vorzunehmen. Vergegenwärtigen wir uns daher nochmals, wie das mit den Eingaben über Fenster funktioniert. Jedes Fenster besitzt einen MessagePort, an den das Betriebssystem seine Nachrichten schickt. Jedes Fenster! Da auch aus der IntuiMessage-Struktur hervorgeht, an welches Fenster eine Nachricht geschickt worden ist, würde für unsere Zwecke ein MessagePort ausreichen. Somit sparen wir kostbaren Speicher, den wir noch anderweitig "vergeuden" wollen.

Eigener MessagePort

Wir müssen also Intuition klarmachen, daß wir für alle unsere Fenster einen einzigen MessagePort haben wollen. Zu diesem Zweck löschen wir die IDCMP-Flags in der NewWindow-Struktur, bevor wir OpenWindow aufrufen. Da das Fenster scheinbar keine Eingaben erhalten soll, kriegt es von Intuition auch keinen MessagePort. Ist ein Fenster geöffnet worden, so tragen wir die Adresse unseres MessagePorts, den wir zuvor von CreatePort haben initialisieren lassen, in das Feld UserPort der Window-Struktur ein. Danach rufen wir ModifyIDCMP auf und definieren die Eingaben, die wir erhalten wollen. Da Intuition bereits einen MessagePort für das Fenster vorfindet, beschafft es keinen neuen. Voilà! Nur beim Schließen des Fensters müssen wir aufpassen und das Feld UserPort der Window-Struktur auf Null setzen, bevor wir CloseWindow aufrufen, da sonst auch unser MessagePort geschlossen werden würde.

Damit die nächste Stufe unseres Programms nicht allzu sinnlos wird, lassen wir uns die gedrückten Tasten anzeigen. Hierfür bietet das Betriebssystem zwei Möglichkeiten an: VANILLAKEY und RAWKEY. Während Sie bei VANILLAKEY direkt die ASCII-Codes der entsprechenden Tasten erhalten, erhalten Sie bei RAWKEY nur sogenannte Scan-Codes, die angeben, welche Taste gedrückt wurde, und nicht, welchem Buchstaben dies entspricht. In unserem Editor werden wir mit RAWKEY arbeiten, da dies die einzige Möglichkeit ist zu erfahren, ob die Funktions- oder Cursor-Tasten gedrückt wurden. Diese werden von VANILLAKEY "verschluckt", da diese Tasten keine ASCII-Codes senden.

Doch zuerst komplettieren wir die Strukturen, die wir für unseren Editor brauchen und die wir in Editor.h eingetragen haben:

```
<src/Editor.h>

/************
 *
 * Includes:
 *
 ************/
```

```
#include <exec/types.h>
#include <intuition/intuition.h>

/***********
 *
 * Defines:
 *
 ***********/

#define MAXBREITE 80

/*******************
 *
 * Datenstrukturen:
 *
 *******************/

struct Zeile
{
   struct Zeile *succ;
   struct Zeile *pred;
   UBYTE flags;
   UBYTE len;
};

#define ZLF_USED 128

struct ZList
{
   struct Zeile *head;
   struct Zeile *tail;
   struct Zeile *tailpred;
};

struct Speicherstueck
{
   struct Speicherstueck *succ;
   struct Speicherstueck *pred;
   UWORD len;
};

struct FList
{
   struct Speicherstueck *head;
   struct Speicherstueck *tail;
   struct Speicherstueck *tailpred;
};

struct Speicherblock
{
   struct Speicherblock *succ;
   struct Speicherblock *pred;
   ULONG laenge;
   ULONG frei;
```

```
   struct FList freiliste;
};

struct BList
{
   struct Speicherblock *head;
   struct Speicherblock *tail;
   struct Speicherblock *tailpred;
};

struct Editor
{
   struct Editor *succ;
   struct Editor *pred;
   struct Window *window;
   struct BList block;
   struct ZList zeilen;
   UBYTE puffer[MAXBREITE];
   UBYTE tabstring[MAXBREITE];
   UWORD anz_zeilen;
   struct Zeile *aktuell;
   struct Zeile *top;
   UWORD toppos;
   UWORD xpos,ypos;
   UWORD changed:1;
   UWORD insert:1;
};

struct EList
{
   struct Editor *head;
   struct Editor *tail;
   struct Editor *tailpred;
};
```

Die Strukturen sind ja bereits in Kapitel 4.1 beschrieben worden, bis auf die letzte. Die EList-Struktur dient zum Verwalten aller Editorfenster, so wie die BList-Struktur zum Verwalten aller Speicherblöcke und die ZList-Struktur zum Verwalten aller Zeilen dient. Als nächstes folgt der Quelltext des Editors, der ebenfalls an Umfang gewonnen hat:

```
<src/Editor.c>

/************
 *
 * Includes:
 *
 ************/
```

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define REV 33L

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

struct Library *OpenLibrary();
struct Window *OpenWindow();
void CloseLibrary(),NewList(),AddTail(),CloseWindow(),free();
void DeletePort(),ModifyIDCMP(),ReplyMsg();
struct Editor *malloc(),*RemHead();
struct MsgPort *CreatePort();
struct IntuiMessage *GetMsg();
ULONG Wait();

/*********************
 *
 * Globale Variablen:
 *
 *********************/

struct Library *IntuitionBase = NULL,
               *GfxBase = NULL, *DosBase = NULL;

struct EList editorList;

struct NewWindow newEdWindow =
{
   100,50,440,156,
   AUTOFRONTPEN,AUTOBACKPEN,
   REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW,
   WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH |
                  WINDOWCLOSE | SIZEBBOTTOM
   | SIMPLE_REFRESH | ACTIVATE,
   NULL,NULL,
   (UBYTE *)"Editor",
   NULL,NULL,
   100,50,640,256,
   WBENCHSCREEN
};
```

```
struct MsgPort *edUserPort = NULL;

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

/***********************************
 *
 * OpenEditor()
 *
 * Öffnet Editor-Fenster und
 * initialisiert Editor-Struktur
 *
 * Gibt Zeiger auf Editor-Struktur
 * zurück, oder NULL falls Fehler.
 *
 ***********************************/

struct Editor *OpenEditor()
{
   register struct Editor *ed = NULL;
   register struct Window *wd;
   register ULONG flags;

   /* IDCMPFlags retten, in Struktur auf NULL setzen!
    => eigenen UserPort verwenden! */
   flags = newEdWindow.IDCMPFlags;
   newEdWindow.IDCMPFlags = NULL;

   /* Speicher für Editor-Struktur beschaffen: */
   if (ed = malloc(sizeof(struct Editor)))

      /* Fenster öffnen: */
      if (wd = OpenWindow(&newEdWindow))
      {
         ed->window = wd;

         /* UserPort einrichten: */
         wd->UserPort = edUserPort;
         ModifyIDCMP(wd,flags);

         /* Parameter initialisieren: */
         NewList(&(ed->block));
         NewList(&(ed->zeilen));
         ed->anz_zeilen = 0;
         ed->aktuell    = NULL;
         ed->top        = NULL;
         ed->toppos     = 0;
         ed->xpos       = 1;
         ed->ypos       = 1;
         ed->changed    = 0;
         ed->insert     = 1;
```

```
      }
      else
      {
         free(ed);
         ed = NULL;
      }

   newEdWindow.IDCMPFlags = flags;
   return (ed);
}

/***************************
 *
 * CloseEditor(ed)
 *
 * Schließt Editor-Fenster wieder.
 *
 * ed ^ Editor-Struktur.
 *
 ***************************/

void CloseEditor(ed)
struct Editor   *ed;
{
   ed->window->UserPort = NULL;
   CloseWindow(ed->window);
   free(ed);
}

/*****************
 *
 * Hauptprogramm:
 *
 *****************/

main()
{
   register struct Editor *ed;
   BOOL running = TRUE;
   register struct IntuiMessage *imsg;
   ULONG signal,class;
   UWORD code,qualifier;
   APTR iaddress;
   register UWORD n = 1;

   /* Librarys öffnen: */
   if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
      goto Ende;
   if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
      goto Ende;
   if ( !(DosBase = OpenLibrary("dos.library",REV)))
      goto Ende;
```

```
   /* UserPort öffnen */
   if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

   /* Editor-Liste initialisieren: */
   NewList(&editorList);

   /* Erstes Editorfenster öffnen: */
   if (ed = OpenEditor())
      AddTail(&editorList,ed);
   else
      goto Ende;

   do
   {
      signal = Wait(1L << edUserPort->mp_SigBit);

      while (imsg = GetMsg(edUserPort))
      {
         class     = imsg->Class;
         code      = imsg->Code;
         qualifier = imsg->Qualifier;
         iaddress  = imsg->IAddress;

         ReplyMsg(imsg);

         /* Event bearbeiten: */
         switch (class)
         {
            case RAWKEY:
               printf(
               "%3d RawKey-Event: Code = $%4x,
                                  Qualifier = $%4x.\n",
               n++,code,qualifier);
               break;

            case CLOSEWINDOW:
               running = FALSE;
               break;

            default:
               printf("Nicht bearbeitbarer Event: %lx\n",class);

         } /* of case */
      } /* of while (GetMsg()) */
   } while (running);

Ende:
   /* Alle Editorfenster wieder schließen: */
   while (ed = RemHead(&editorList))
      CloseEditor(ed);

   /* UserPort schließen: */
   if (edUserPort) DeletePort(edUserPort);
```

```
    /* Librarys schließen: */
    if (DosBase) CloseLibrary(DosBase);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```

Die Funktion OpenEditor funktioniert genauso, wie wir es festgelegt haben, ebenso CloseEditor. Im Hauptprogramm öffnen wir zuerst wieder die Librarys und dann einen eigenen MessagePort (edUserPort), der bei OpenEditor in den UserPort des Fensters eingetragen wird. Danach wird die Editorliste (editorList) initialisiert und ein Editorfenster geöffnet und in die Editorliste eingebunden (AddTail). Anschließend folgt die Hauptschleife, in der auf Eingaben gewartet wird. Hier ist es von Vorteil, daß wir nur einen MessagePort haben, denn sonst müßten wir die SignalFlags aller Ports holen und miteinander ver-"odern", damit auch wirklich auf Messages aus allen Fenstern gewartet wird.

In der Hauptschleife (do {...} while (running)) wird zunächst darauf gewartet, daß das Betriebssystem ein Ereignis signalisiert (Wait). Anschließend werden solange Messages abgeholt (GetMsg), bis keine mehr vorhanden sind; es kann nämlich sein, daß der Editor nur ein Signal, aber mehrere Messages erhalten hat. In dem switch-case-Befehl wird dann nach dem Message-Typ unterschieden und dementsprechend gehandelt. In unserem Fall wird bei RAWKEY nur der Code der gedrückten Taste ausgegeben, ein Zähler (n) heraufgezählt, bei CLOSEWINDOW running auf FALSE gesetzt und damit das Programm beendet. Zum Schluß werden zuerst alle Editorfenster wieder geschlossen, dann der MessagePort und zu allerletzt die Librarys.

Ein paar Bemerkungen zum Programm:

- Um Programme zu schreiben, die wirklich auf allen Compilern laufen, sollten Sie die Variablentypen aus <exec/types.h> verwenden und nicht die normalen C-Standard-Datentypen, da vor allem der Datentyp integer je nach Compiler mal als 16 Bit und mal als 32 Bit implementiert ist.
- Wenn Sie sich casting (Zwangs-Typ-Umwandlung) ersparen wollen, so können Sie Funktionen so deklarieren, wie

Sie wollen. Insbesondere Funktionen, die Zeiger auf Objekte zurückliefern, können so deklariert werden, daß sie Zeiger auf alles mögliche zurückliefern, wie es im Programm mit malloc, RemHead und GetMsg gemacht wurde.

- Sie sollten unbedingt alle Funktionen, die Sie im Programm benutzen, auch deklarieren, und sei es nur als void. Andernfalls kann es allzuleicht passieren, daß Sie eine Funktion übersehen und es deswegen zum Programmabsturz kommt, weil Sie z.B. statt eines 32-Bit-Zeigers nur einen 16-Bit-Wert erhalten haben. Auch gibt es dann keine Probleme, wenn Sie, aus welchen Gründen auch immer, statt mit 16-Bit-Integer plötzlich mit 32-Bit-Integer rechnen müssen oder umgekehrt.

- Der oft verteufelte Befehl goto erfüllt im Zusammenhang mit der Fehlerbehandlung seinen Zweck besser, als eine Aneinanderreihung von If-Befehlen. Auch das nicht unbeliebte:

```
if (fehler)
{
   CloseWindow(xxx);
   CloseLibrary(yyy);
   exit(FALSE);
}
```

  das für jeden Fehler im Hauptprogramm einmal hingeschrieben wird, ist, was den Speicherbedarf betrifft, im Vergleich mit goto um ein Vielfaches uneffizienter, denn schließlich wollen die entsprechenden Befehle vor dem exit auch codiert werden. Die Verwendung von gotos spart hier Speicher und ist sicherlich nicht unübersichtlicher.

- Die Funktionen NewList, AddTail und RemHead sind eigentlich für Exec-Listen gedacht. Wenn Sie sich die Strukturen von Lists und Nodes ansehen (siehe Anhang: Exec-Library), so werden Sie feststellen, daß diese minimal umfangreicher sind als unsere Strukturen. Allerdings funktionieren die genannten Funktionen auch mit unseren Strukturen, da diese nicht auf die zusätzlichen Elemente zugreifen. Anders wäre dies, wenn wir auch die Funktion Enqueue benutzen würden, da diese auch das Prioritätsfeld der Exec-Node benutzt, das unsere Struktur nicht besitzt.

Probieren Sie das Programm doch einmal aus, und drücken Sie ein paar Tasten. Wie erwartet erhalten Sie je eine Message für das Drücken und eine für das Loslassen der Taste. Wenn Sie jetzt aber eine Taste drücken und gedrückt halten, so passiert etwas, was Sie vielleicht nicht unbedingt erwartet haben. Sobald eine gewisse Zeitspanne vergangen ist (der sogenannte KeyRepeatDelay), erhalten Sie plötzlich reihenweise weitere Messages für diese Taste, und zwar so lange, bis Sie die Taste wieder loslassen. Der einzige Unterschied zu der ersten Message für diese Taste ist, daß beim Qualifier das neunte Bit gesetzt ist, was bedeutet, daß diese Taste aufgrund der Tastaturwiederholfunktion gesendet wurde.

Eine Wiederholfunktion der Tastatur hätten Sie wahrscheinlich nur bei VANILLAKEY-Messages erwartet? Für unseren Editor ist dies aber nur von Vorteil, da wir nun die Tastaturwiederholfunktion nicht per Hand programmieren müssen. Stellt sich nur noch die Frage, wie wir die RAWKEYs in normale ASCII-Zeichen umwandeln können. Eine eigene Umwandlungstabelle scheidet mit Sicherheit aus, da diese nicht auf Amigas mit anderen Tastaturbelegungen funktionieren würde. Aber auch hier stellt das Betriebssystem erfreulicherweise eine Funktion zur Verfügung, die uns diese Aufgabe abnimmt, und dabei die eingestellte Tastaturbelegung berücksichtigt.

4.3.2 Von RAWKEY nach ASCII

Die Funktion ist Teil des Console.Devices und hört auf den Namen RawKeyConvert. Allerdings brauchen wir nicht das Console.Device zu öffnen, um an diese Funktion zu kommen, sondern es reicht aus, einen Zeiger auf die ConsoleBase zu holen, da sich die Funktion wie eine Funktion einer Library aufrufen läßt. Den Zeiger auf die ConsoleBase erhalten wir, indem wir zuerst

```
OpenConsole("console.device",-1L,IOStdReq,0L);
```

aufrufen und uns dann den Zeiger aus dem io_Device-Feld der IOStdReq-Struktur holen. Danach können wir RawKeyConvert wie eine normale Funktion aufrufen. Übergeben müssen Sie da-

bei einen Zeiger auf eine InputEvent-Struktur, einen Zeiger auf einen Puffer, dessen Länge und einen Zeiger auf eine KeyMap-Struktur, den wir aber vorerst auf Null setzen, da zur Konvertierung die Standard-Tastaturtabelle verwendet werden soll. Die InputEvent-Struktur müssen wir vor dem Aufruf füllen, und zwar mit dem RAWKEY-Key-Code und dem Qualifier aus der IntuiMessage-Struktur. Um vom Programm statt der Rawkeys nun die ASCII-Zeichen ausgeben zu lassen, sind folgende Änderungen notwendig:

Zuerst definieren wir die maximale Länge einer Tastatureingabe auf 128 Zeichen, was sicher nicht zu wenig ist:

```
#define MAXINPUTLEN 128L
```

Dann deklarieren wir die beiden Funktionen OpenDevice und RawKeyConvert, die neu in unserem Programm sind:

```
ULONG OpenDevice();
SHORT RawKeyConvert();
```

Jetzt kommen noch ein paar globale Variablen dazu, die wir bereits besprochen haben. In der InputEvent-Struktur setzen wir das ie_Class-Feld auf RAWKEY, damit wir dies nicht explizit im Programm tun müssen:

```
struct IOStdReq ioStdReq;

UBYTE inputPuffer[MAXINPUTLEN];
UWORD inputLen = 0;

struct InputEvent inputEvent =
{
   0,
   IECLASS_RAWKEY,0,
   0,0
};
```

Im inputPuffer werden von der Funktion RawKeyConvert die Zeichen abgelegt, mit der die entsprechende Taste belegt ist. Zu Beginn des Hauptprogramms definieren wir eine Variable l, die wir später noch brauchen werden:

```
SHORT l;
```

Im Hauptprogramm holen wir uns den Zeiger auf das Console.Device, nachdem wir die Librarys geöffnet haben:

```
if ( !(OpenDevice("console.device",-1L,&ioStdReq,0L)))
   ConsoleDevice = ioStdReq.io_Device;
else
   goto Ende;
```

Und zu guter Letzt bauen wir in der Hauptschleife die Behandlung von RAWKEY-Messages folgendermaßen um:

```
case RAWKEY:
   if (! (code & IECODE_UP_PREFIX))
   {
      inputEvent.ie_Code  = code;
      inputEvent.ie_Qualifier = qualifier;
      if ((l = RawKeyConvert(&inputEvent,
                  inputPuffer,MAXINPUTLEN,NULL)
            ) > 0)
      {
         inputPuffer[l] = 0;
         printf("%3d> Laenge = %d: ",n++,l);
         for (l = 0; inputPuffer[l]; l++)
            printf("%2x",(UWORD)inputPuffer[l]);
         if (inputPuffer[0] >= 32 && inputPuffer[0] <= 127)
            printf(" %s",inputPuffer);
         printf("\n");
      }
   }
   break;
```

Zuerst testen wir, ob die entsprechende Taste gedrückt oder losgelassen worden ist; im zweiten Fall (IECODE_UP_PREFIX) interessiert sie uns nicht. Dann werden die Werte von code und qualifier in die InputEvent-Struktur übertragen, und anschließend wird RawKeyConvert aufgerufen. Den Rückgabewert speichern wir in l, nicht in inputLen, ab und testen, ob dieser größer als 0 ist. War er kleiner null, so ist ein Fehler aufgetreten (Puffer war zu klein), und war er null, so ergab die Taste keine Zeichen, so daß wir uns in beiden Fällen nicht weiter darum zu kümmern brauchen.

War der Rückgabewert jedoch größer als null, so gibt er die Länge des Strings an, mit dem die Taste belegt ist, und wir setzen hinter den String ein Null-Byte, damit wir diesen mit printf ausgeben können. Normalerweise sind die Tasten nur mit einem

einzigen Zeichen belegt, so daß wir als Länge den Wert Eins erhalten, aber es gibt auch Ausnahmen. Danach geben wir zuerst eine fortlaufende Nummer und dann die Länge des Strings aus. Es folgt der String selbst, und zwar zuerst in Hex-Codes und dann als richtiger ASCII-String, aber dieser nur, wenn das erste Zeichen zwischen einem Space und einem Delete liegt, also ein druckbares Zeichen ist. Wenn wir diese Abfrage nicht treffen, wird unsere Ausgabe im CLI-Fenster in Mitleidenschaft gezogen, da das CLI z.B. bestimmte ASCII-Codes in Cursor-Bewegungen umsetzt.

Alles klar? Also los: Eintippen, compilieren und ausprobieren. Während das Ausprobieren der "normalen" Tasten keine ungewöhnlichen Ergebnisse hervorbringt, erhalten Sie, wenn Sie die Funktionstasten drücken, drei bis vier (bei Shift) Zeichen geliefert. Dies müssen wir bei unserer späteren Tastaturabfrage berücksichtigen. Außerdem ist dies der Grund dafür, daß Sie bei VANILLAKEY keine Informationen über die Funktionstasten erhalten. Schließlich kann die IntuiMessage nur ein einziges Zeichen zurückliefern und nicht drei auf einmal. Wenn Sie alle Sondertasten ausprobieren und die gelieferten Codes mit denen in der Dokumentation (AmigaDOS-Manual) vergleichen, so werden Sie vielleicht feststellen, daß einige damit nicht übereinstimmen; aber vielleicht sind die (wenigen) Fehler in Ihrer Ausgabe bereits behoben.

Den kompletten Quelltext zu diesem Teil des Editors finden Sie im Verzeichnis V0.1 auf der Diskette zum Buch. Wenn Sie Besitzer eines Aztec-C-Compilers sind, so können Sie diesen compilieren, indem Sie in das Verzeichnis V0.1 wechseln und dann "make Editor" eingeben. Zuvor müssen Sie allerdings dafür sorgen, daß sich Make irgendwo befindet, so daß das CLI dies auch wirklich aufrufen kann, z.B. indem Sie es auf die RAM-Disk kopieren:

```
copy make to ram:
path ram: add
```

und dann zum Compilieren folgendes eingeben:

```
cd V0.1
make Editor
```

Der einzige Unterschied zu den hier besprochenen Dingen ist, daß das pre-Verzeichnis woanders liegt, nämlich im Hauptverzeichnis. Da es aber nur einmal für alle Versionen des Editors existiert, sollten Sie darauf achten, daß die Datei :pre/Editor.pre für jede neue Version des Editors auch neu erzeugt wird! Löschen Sie diese also vor dem Aufruf von Make. Wenn Sie den Editor nicht extra übersetzen wollen, so finden Sie das fertige Programm ebenfalls im Verzeichnis V0.1, so daß Sie diesen nur noch zu starten brauchen.

4.3.3 Die Speicherverwaltung

Nächster Punkt der Tagesordnung ist die Speicherverwaltung. Bevor wir eine Datei laden und anzeigen lassen können, was unser nächstes Ziel sein wird, müssen wir die Möglichkeit zur Verfügung stellen, Zeilen abzuspeichern und zu verketten. Dazu brauchen wir eine Funktion, die uns eine neue Zeile-Struktur besorgt, in die wir dann einen String, die eigentliche Zeile also, kopieren. Der Kopf dieser Funktion soll wie folgt aussehen:

```
struct Zeile *neueZeile(len)
UWORD len;
```

Dabei ist len die Länge des Strings, einschließlich Zeilenende. Da wir aber nur Speicherstücke gerader Länge verwenden können, müssen wir in der Funktion die Länge geradzahlig machen. Da dies sicher öfter vorkommt, definieren wir ein entsprechendes Define:

```
#define EVENLEN(x) (((x)+1)&-2)
```

Bevor wir die Funktion implementieren, wollen wir uns überlegen, wie diese funktionieren soll. Wenn sich noch kein Speicherblock in der Blockliste befindet, der Textspeicher also leer ist, wird ein neuer Block mit 10 KByte Größe beschafft und dann eine Zeile geholt. Andernfalls wird direkt versucht, eine Zeile zu beschaffen. Mißlingt dies, so wird eine Garbage Collection durchgeführt, also der Speicher aufgeräumt, bevor ein zweites

Mal versucht wird, eine Zeile zu beschaffen. Mißlingt auch dies, so ist anscheinend in den vorhandenen Speicherblöcken nicht mehr genügend Platz für eine weitere Zeile vorhanden, und es wird ein neuer Speicherblock beschafft, der diesmal aber nur 5 KByte groß ist. Wir arbeiten deshalb mit zwei unterschiedlichen Blockgrößen, damit wir für kleine Texte, die kürzer als 10 KByte sind, nur einen Speicherblock benötigen. Der folgende Ablaufplan verdeutlicht unsere Vorgehensweise:

```
Falls noch kein Speicherblock in Blockliste
   Hole neuen Block (Länge = 10 KByte)
   hole Zeile
   Fertig
sonst
   hole Zeile
   Falls keine gefunden
      Garbage-Collection
      hole Zeile
      Falls keine gefunden
         hole neuen Block (5 KByte)
         hole Zeile
   Fertig
```

Die nächste Funktion beschafft einen neuen Speicherblock:

```
struct Speicherblock *neuerBlock(len)
ULONG len;
```

Konnte ein neuer Speicherblock beschafft werden, so wird dessen Speicherblock-Struktur initialisiert und ein Zeiger auf den Block zurückgegeben, ansonsten wird Null zurückgeliefert. Der Speicherblock muß dann noch in die Speicherblock-Liste eingefügt werden. Dazu brauchen wir jedoch noch einen Zeiger auf die aktuelle Editor-Struktur:

```
struct Editor *aktuellerEditor;
```

Der Speicherblock wird mit AllocMem beschafft und nicht mit der C-Standard-Funktion malloc. Dies ist günstiger, da malloc den beschafften Speicher in eine Liste einhängt, damit es am Programmende allen Speicher, der nicht explizit freigegeben worden ist, doch noch freigeben kann. Diese Liste kostet aber etwas Speicherplatz, und da wir selbstverständlich allen Speicher wieder freigeben, bevor wir das Programm beenden, verwenden wir besser AllocMem.

Aus demselben Grund verwenden wir auch nicht die Funktion AllocRemember, die die Intuition-Library zur Speicherverwaltung zur Verfügung stellt. Der Vorteil dieser Funktion ist, daß sie sich allen Speicher merkt, denn man allokiert hat, und daß man diesen mit einem Befehl wieder freigeben kann. Allerdings muß wieder die Remember-Struktur mit abgespeichert werden, was zusätzlich Platz kostet.

Ansonsten dürfte die Funktion keine Schwierigkeiten bereiten, so daß sich weitere Erklärungen erübrigen. Anders sieht dies da schon bei der nächsten Funktion aus:

```
struct Zeile *holeZeile(len)
UWORD len;
```

Diese Funktion durchsucht alle Speicherblöcke des aktuellen Editors und nach einem Speicherstück, das die Zeile aufnehmen kann. In len wird dabei wieder die tatsächliche Länge der Zeile übergeben, die innerhalb der Funktion geradzahlig gemacht werden muß. Es stellt sich die Frage, welches Speicherstück wir nehmen sollen? Das erste, das groß genug ist, oder dasjenige, das genug Platz bietet und am nächsten dran ist? Da wir jedoch nicht umsonst diesen Aufwand mit den Speicherblöcken getrieben haben, werden wir anders vorgehen, um die Speicherstücke möglichst optimal aufzuteilen:

Zuerst versuchen wir ein Speicherstück zu finden, das exakt die gesuchte Länge hat. Dies ist sicherlich der optimale Fall. Hat dies nicht geklappt, so suchen wir das größte Speicherstück. Dadurch vermeiden wir weitgehendst den Fall, daß wir ein ohnehin schon kleines Speicherstück noch kleiner machen. Wenn jedoch ein Speicherstück so klein ist, daß, nachdem die neue Zeile davon abgeschnitten worden ist, nicht einmal mehr genug Platz für die Speicherstueck-Struktur und mindestens zwei weitere Bytes ist, so war die Suche erfolglos. Denn der Rest muß mindestens so groß sein, daß man darin eine weitere Zeile ablegen kann.

Rufen Sie sich nochmals in Erinnerung, daß jeder Speicherblock aus zwei Dingen besteht: den Zeilen, die darin liegen, und den freien Speicherstücken, aus denen der Platz für neue Zeilen gewonnen wird.

Die letzte Funktion, die von neueZeile benutzt wird, ist:

```
struct Speicherblock *garbageCollection(len)
UWORD len;
```

Diese Funktion reorganisiert den freien Speicher innerhalb der Speicherblöcke so, daß es pro Speicherblock nur noch ein einziges Speicherstück mit maximaler Größe gibt. Bedenken Sie, daß jedes Speicherstück zehn Bytes Verwaltungsinformation benötigt (sizeof(Speicherstueck)). Je mehr Speicherstücke in einem Speicherblock existieren, um so weniger freien Speicher enthält dieser. Die Garbage-Collection wird Speicherblock für Speicherblock durchgeführt, so lange, bis ein Speicherblock mit einem freien Speicherstück der Größe len Bytes gefunden ist oder bis das Ende der Speicherblockliste erreicht ist. War die Suche erfolgreich, so wird ein Zeiger auf den entsprechenden Speicherblock zurückgeliefert, andernfalls Null. Der Zeiger kann eventuell dazu verwendet werden, der Funktion holeZeile das erneute Durchsuchen aller Speicherblöcke zu ersparen.

Alle genannten Funktionen schreiben wir in ein neues Modul und nennen dieses Speicher.c. Das ganze Modul sieht wie folgt aus:

```
<src/Speicher.c>

/*************
 *
 * Includes:
 *
 *************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <exec/memory.h>
#include "src/Editor.h"

/************
 *
 * Defines:
 *
 ************/

#define BIGBLOCK (10240L-sizeof(struct Speicherblock))
#define BLOCKSIZE (5120L-sizeof(struct Speicherblock))
```

```
/**********************
 *
 * Externe Funktionen:
 *
 **********************/

struct Speicherblock *AllocMem();
void NewList(),AddTail(),Remove(),FreeMem();

/**********************
 *
 * Externe Variablen:
 *
 **********************/

extern struct Editor *aktuellerEditor;

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

/*****************************************
 *
 * freeSpeicherblock(blk)
 *
 * Gibt den Speicher des Speicherblocks
 * wieder frei, ohne Ruecksicht auf Zeilen,
 * die noch darin liegen.
 *
 * blk ^ Speicherblock.
 *
 *****************************************/

void freeSpeicherblock(blk)
struct Speicherblock  *blk;
{
   Remove(blk);
   FreeMem(blk,blk->laenge + sizeof(struct Speicherblock));
}

/*********************************************
 *
 * neuerBlock(len):
 *
 * Beschafft Speicher fuer neuen Speicherblock
 * mit len Laenge und liefert Zeiger darauf
 * Zurueck. Falls Fehler => NULL.
 *
 * len = Laenge des Speicherblocks.
 *
 *********************************************/
```

```
struct Speicherblock *neuerBlock(len)
ULONG                            len;
{
   register struct Speicherblock *blk;
   register struct Speicherstueck *spst;

   /* Speicher beschaffen: */
   if (blk = AllocMem(len+sizeof(struct
                      Speicherblock),MEMF_CLEAR))
   {
      /* Speicherblock-Struktur initialisieren: */
      blk->laenge = len;
      blk->frei = len - sizeof(struct Speicherstueck);
      NewList(&(blk->freiliste));
      spst = (struct Speicherstueck *)(blk + 1);
      AddTail(&(blk->freiliste),spst);
      spst->len = (UWORD)blk->frei;
   }

   return (blk);
}

/***********************************************
 *
 * sucheSpeicherstueck(block,len)
 *
 * Sucht Speicherstueck mit len Laenge
 * im Block block, und liefert Zeiger darauf
 * zurueck, oder Null falls kein Platz.
 *
 * block ^ Speicherblock.
 * len = Laenge der Zeile (geradzahlig).
 *
 ***********************************************/

struct Speicherstueck *sucheSpeicherstueck(block,len)
struct Speicherblock                      *block;
UWORD                                            len;
{
   register UWORD minl,l;
   register struct Speicherstueck *st,*fnd = NULL;

   /* Nur Durchsuchen, wenn ueberhaupt noch genug Platz: */
   if ((ULONG)len <= block->frei)
   {
      /* Minimale Laenge, falls gefundene Laenge <> len */
      minl = len + sizeof(struct Speicherstueck) + 2;

      /* Durchsuche Liste: */
      for (st = block->freiliste.head; st->succ; st = st->succ)
         if ((l = st->len) == len)
         {
            /* Optimale Groesse gefunden: */
            fnd = st;
```

```
            break;
         }
         else
            if ((l >= minl) && (l > ((fnd)? fnd->len : 0)))
               /* Suche groessten Block: */
               fnd = st;
   }

   return (fnd);
}

/*****************************************
 *
 * ConvertSpstToZeile(blk,st,len)
 *
 * Wandelt Speicherstueck in Zeile um.
 * War das Stueck laenger als die Zeile,
 * so wird das Stueck verkuerzt, andernfalls
 * wird das Stueck aus der Liste entfernt.
 *
 * blk ^ Speicherblock des Speicherstuecks.
 * st ^ Speicherstueck.
 * len = Laenge der Zeile
 *
 *****************************************/

struct Zeile *ConvertSpstToZeile(blk,st,len)
struct Speicherblock            *blk;
register struct Speicherstueck      *st;
register UWORD                          len;
{
   register UWORD l;
   register struct Zeile *z;

   if (st->len == (l = EVENLEN(len)))
   {
      /* Gefundenes Stueck passt genau: */
      Remove(st);
      z = (struct Zeile *)st;
   }
   else
   {
      /* Stueck aufteilen: */
      z = (struct Zeile *)
            ((UBYTE *)st + sizeof(struct Speicherstueck) + st->len
          - (l += sizeof (struct Zeile)) );
      st->len -= l;
   }
   blk->frei -= l;

   z->flags = 0;
   z->len = len;
```

```
   return (z);
}

/***********************************************
 *
 * holeZeile(len)
 *
 * Versucht Zeile mit len Laenge zu beschaffen,
 * und sucht dabei alle Speicherbloecke durch.
 * Liefert Zeiger auf Zeile zurueck, oder
 * Null, falls kein Platz.
 *
 * len = Laenge der Zeile.
 *
 ***********************************************/

struct Zeile *holeZeile(len)
UWORD                   len;
{
   register UWORD l;
   register struct Speicherblock *blk,*fblk = NULL;
   register struct Speicherstueck *st,*fst = NULL;

   /* Laenge geradzahlig machen: */
   l = EVENLEN(len);

   /* Liste der Bloecke durchlaufen: */
   for (blk = aktuellerEditor->block.head; blk->succ;
                                        blk = blk->succ)
      if (st = sucheSpeicherstueck(blk,l))
         if (st->len == l)
         {
            /* Optimales Speicherstueck gefunden: */
            fblk = blk;
            fst = st;
            break;
         }
         else
            if (st->len > ((fst)? fst->len : 0))
            {
               /* Suche groesstes Speicherstueck: */
               fblk = blk;
               fst = st;
            }

   if (fst)
      return (ConvertSpstToZeile(fblk,fst,len));
   else
      /* Kein Stueck gefunden: */
      return (NULL);
}
```

```
/*******************************************
 *
 * optimiereBlockblock)
 *
 * Durchsucht alle Speicherstuecke des Blocks
 * und macht aus hintereinanderliegenden
 * Speicherstuecken ein einziges.
 *
 * block ^ Speicherblock.
 *
 *******************************************/

void optimiereBlock          (blk)
register struct Speicherblock *blk;
{
   register struct Speicherstueck *st1,*st2,*st1e;

   for (st1 = blk->freiliste.head; st1->succ; st1 = st1->succ)
   {
      /* Bestimme Zeiger direkt hinter st1: */
      st1e = (struct Speicherstueck *)
           (((UBYTE *)st1) + st1->len + sizeof(struct
                                        Speicherstueck));

      for (st2 = blk->freiliste.head; st2->succ; st2 = st2->succ)
         if (st1e == st2)
         {
            /* Speicherstueck st2 liegt direkt hinter st1: */
            st1->len += st2->len + sizeof(struct Speicherstueck);
            blk->frei += sizeof(struct Speicherstueck);
            Remove(st2);
            st1e = (struct Speicherstueck *)
               (((UBYTE *)st1)+st1->len+sizeof(struct
                                        Speicherstueck));

            /* Wieder am Anfang der Liste beginnen: */
            st2 = (struct Speicherstueck *) &(blk-
                                          >freiliste.head);
         }
   }
}

/*******************************************
 *
 * garbageCollection(len)
 *
 * Raeumt solange den Speicher auf,
 * bis mindestens len freie Bytes.
 * Liefert Zeiger auf Speicherblock, in
 * dem noch Platz ist zurueck, oder
 * Null, falls kein Platz.
 *
```

```
 * len = Anzahl Bytes, die untergebracht
 *       werden sollen.
 *
 ****************************************/

struct Speicherblock *garbageCollection(len)
UWORD                                   len;
{
   struct Speicherblock *blk;
   struct Speicherstueck *st,spst;
   register UBYTE *ptr,*end;
   register struct Zeile *z,*fz;
   register UWORD n;

   for (blk = aktuellerEditor->block.head; blk->succ;
                                        k = blk->succ)
      /* Nur Garbage-Collection, wenn mehr
                                  als ein Speicherstueck! */
      if ((st = blk->freiliste.head->succ)? st->succ : FALSE)
      {
         /* Anfang und Ende des Blockspeichers bestimmen: */
         ptr = (UBYTE *)(blk + 1);
         end = ptr + blk->laenge;

         /* Zeilen an den Anfang legen: */
         while (ptr < end)
         {
            /* Suche Zeile, die am naechsten an ptr liegt: */
            fz = NULL;
            for (z = aktuellerEditor->zeilen.head; z->succ;
                                              z = z->succ)
               if ((z >= ptr) && (z < end))
                  if ((z < fz) || (fz == NULL))
                     fz = z;

            if (fz)
               if (fz != ptr)
               {
                  /* Alte Zeiger merken: */
                  z = fz;
                  st = (struct Speicherstueck *)ptr;
                  spst = *st;

                  /* Zeile nach ptr herunterschieben: */
                  for (n = (EVENLEN(fz->len)+sizeof(struct
                                                  Zeile))>>1;
                       n; n--)
                     *((UWORD *)ptr)++ = *((UWORD *)fz)++;

                  /* Zeiger korrigieren: */
                  fz = (struct Zeile *)st;
                  fz->succ->pred = fz;
                  fz->pred->succ = fz;
                  if (aktuellerEditor->aktuell == z)
```

```
                    aktuellerEditor->aktuell = fz;
                 if (aktuellerEditor->top == z)
                    aktuellerEditor->top = fz;

                 st = (struct Speicherstueck *)ptr;
                 st->succ = spst.succ;
                 st->pred = spst.pred;
                 st->len = spst.len;
                 spst.succ->pred = st;
                 spst.pred->succ = st;

                 /* Speicherstuecke zusammenschieben: */
                 optimiereBlock(blk);
              }
              else
                 /* ptr hochsetzen: */
                 ptr += EVENLEN(fz->len) + sizeof(struct Zeile);
           else
              /* Scheinbar keine Zeilen mehr im Block! */
              break;
        } /* of while */

        /* Teste, ob bereits genug Platz: */
        if (sucheSpeicherstueck(blk,len))
           return (blk);
     } /* of if (mindestens 2 Bloecke) */

  return (NULL);
}

/*****************************************
 *
 * neueZeile(len)
 *
 * Beschafft Platz fuer neue Zeile,
 * in die len Zeichen passen.
 * Liefert Zeiger auf neue Zeile zurueck,
 * oder Null, falls Fehler.
 *
 * len = Anzahl der Zeichen.
 *
 *****************************************/

struct Zeile *neueZeile(len)
UWORD                  len;
{
  register struct Zeile *z;
  register struct Speicherblock *blk;
  register struct Speicherstueck *st;
  register UWORD l;

  if (aktuellerEditor->block.head->succ)
     /* Es existieren bereits Bloecke: */
     if (z = holeZeile(len))
```

```
            return (z);
         else
            /* Zeile nicht direkt verfuegbar: Garbage-Collection */
            if (blk = garbageCollection(l = EVENLEN(len)))
               if (st = sucheSpeicherstueck(blk,l))
                  return (ConvertSpstToZeile(blk,st,len));
               else
                  /* Darf eigentlich nicht auftreten !!! */
                  return (NULL);
            else
               /* Garbage-Collection erfolglos
                           -> neuen Block holen */
               if (blk = neuerBlock(BLOCKSIZE))
               {
                  AddTail(&(aktuellerEditor->block),blk);
                  return (holeZeile(len));
               }
               else
                  return (NULL);
      else
         /* Noch kein Block in der Block-Liste: */
         if (blk = neuerBlock(BIGBLOCK))
         {
            AddTail(&(aktuellerEditor->block),blk);
            return (holeZeile(len));
         }
         else
            return (NULL);
}
```

Es sind mehr Funktionen geworden, als wir uns eigentlich vorgenommen hatten. Dies liegt daran, daß sich einige Aufgaben besser in Funktionen unterbringen ließen und daß ein paar Kleinigkeiten von uns in der Planung nicht berücksichtigt worden waren. Daher nun ein paar Worte zu den zusätzlichen Funktionen:

```
void freeSpeicherblock(blk)
struct Speicherblock  *blk;
```

Diese Funktion gibt den Speicher wieder frei, den der Speicherblock blk belegt hatte. Dabei wird keine Rücksicht darauf genommen, ob eventuell noch Zeilen in diesem Block liegen. Diese Funktion wird in CloseEditor dazu benutzt werden, den gesamten Speicher des Editors wieder freizugeben.

```
struct Speicherstueck *sucheSpeicherstueck(block,len)
struct Speicherblock                    *block;
UWORD                                          len;
```

Diese Funktion durchsucht den angegebenen Block nach einem Speicherstueck, das möglichst genau die Länge len hat, ansonsten nach dem größten vorhandenem Speicherstueck. Zurückgeliefert wird ein Zeiger auf das gefundene Speicherstueck oder Null, falls keines mit ausreichender Länge gefunden wurde. Zu beachten ist insbesondere, daß blk->frei die Summe der Längen der einzelnen Speicherstuecke dieses Blocks enthält, so daß sich eine erste grobe Abfrage, ob in dem Block überhaupt noch genug Platz vorhanden ist, entsprechend einfach gestaltet. Konnte die Länge nicht genau gefunden werden, so sind nur die größeren Speicherstuecke von Interesse, die um (sizeof(struct Speicherstueck) + 2) Bytes größer sind als len, da das gefundene Speicherstueck in eine Zeile und ein Speicherstueck aufgeteilt werden muß. Wenn aber kein Platz mehr für die Strukturen ist, so kann er auch nicht aufgeteilt werden!

Für alle, denen die folgende Zeile Schwierigkeiten bereitet, hier ein kleiner Tip:

```
if ((l >= minl) && (l > ((fnd)? fnd->len : 0)))
```

Wenn fnd gleich null ist, also noch kein Speicherstueck gefunden worden ist, kann nicht getestet werden, ob l größer als die Länge des bereits gefundenen Speicherstückes (fnd) ist. In diesem soll auf jeden Fall der zweite Vergleich TRUE sein. Der seltsame Ausdruck ((fnd)? fnd->len : 0) ist null, wenn auch fnd null ist, und sonst fnd->len, also die Länge des Speicherstückes fnd. Da l aber immer positiv ist, ist l > 0 immer TRUE, und somit haben wir den Sonderfall, daß fnd noch null ist, elegant umgangen.

```
struct Zeile *ConvertSpstToZeile(blk,st,len)
struct Speicherblock            *blk;
struct Speicherstueck               *st;
UWORD                                   len;
```

Wenn wir eine neue Zeile beschaffen wollen, so müssen wir zwangsweise ein Speicherstueck in eine Zeile umwandeln oder doch zumindest eine Zeile von einem Speicherstück abschneiden. Diese Arbeit nimmt uns die zuvor genannte Funktion ab. War das Speicherstück genauso lang wie die Zeile, so wird es aus der Liste aller Speicherstücke entfernt, andernfalls wird die Zeile

vom Ende des Speicherstücks abgezwackt und dessen Länge entsprechend verringert. In jedem Fall wird die Anzahl der freien Bytes des Blocks, zu dem das Speicherstück gehört, entsprechend vermindert. Zurückgeliefert wird ein Zeiger auf die Zeile.

```
void optimiereBlock  (blk)
struct Speicherblock *blk;
```

Die einzige Aufgabe der Funktion ist es, direkt hintereinander liegende Speicherstücke zusammenzufügen. Diese Funktion brauchen wir immer dann, wenn wir ein Speicherstück verlängern oder ein neues Speicherstück in die Liste einhängen. Dann nämlich müssen wir sicherstellen, daß keine zwei Speicherstücke direkt hintereinander liegen, da dies uns jedesmal zehn Bytes (sizeof(struct Speicherstueck)) kostet.

Zur Funktion garbageCollection gibt es allerdings noch etwas zu sagen, da wir uns bei der Festlegung der Aufgaben dieser Funktion ziemlich kurz gefaßt haben. Diese Funktion arbeitet in etwa wie folgt:

```
FOR alle Blöcke
  Liegen in diesem Block mindestens zwei Speicherstücke?
    Schiebe alle Zeilen, die in diesem Block liegen,
    an dessen Anfang.
    Fasse alle Speicherstücke zu einem zusammen.
    Ist in diesem Block bereits genug Platz?
      Liefere Zeiger auf Block zurück.

Sonst: Liefere Null zurück.
```

Dabei ist vor allem das Zusammenschieben aller Zeilen an den Blockanfang so eine Sache. Existieren irgendwo im Programm Zeiger auf diese Zeile, so müssen diese umgebogen werden, so wie dies auch mit der Verkettung geschieht. Im Moment könnten dies höchstens die beiden Zeiger aktuell und top aus der Editor-Struktur sein, die wir später für die Ausgabe und das Editieren brauchen. Wenn wir später jedoch weitere Zeiger auf Zeilen in das Programm einfügen, so müssen wir diesen Umstand unbedingt beachten, sonst stimmt nach einer Garbage Collection gar nichts mehr.

4.3.4 Testumgebung

Jetzt müssen wir nur noch die Funktionen austesten, die uns das Modul Speicher zur Verfügung stellt. Allerdings können wir dies nicht so ohne weiteres, da wir nicht überprüfen können, ob unsere Funktionen auch wirklich richtig funktionieren. Darum schreiben wir eine Testumgebung, von der aus wir alle wichtigen Funktionen aufrufen können und die uns den belegten Speicher anzeigt:

```
<src/Test.c>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

struct Zeile *neueZeile();
UBYTE *gets(),getchar();
UWORD strlen();
void strcpy(), AddTail();

/*********************
 *
 * Externe Variablen:
 *
 *********************/

extern struct Editor *aktuellerEditor;
```

```
/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

void print()
{
   register struct Zeile *z;

   for (z = aktuellerEditor->zeilen.head; z->succ; z = z->succ)
      printf("%s\n",z+1);
}

void print_blocks()
{
   register struct Speicherblock *blk;

   printf("Blockliste:\n\n");
   for (blk = aktuellerEditor->block.head; blk->succ;
                                        blk = blk->succ)
      printf("Block %8lx, Laenge = %ld, Frei = %ld\n",
             blk,blk->laenge,blk->frei);

   printf("\n");
}

void input()
{
   UBYTE s[80];
   register struct Zeile *z;
   register UWORD l;

   printf("Neue Zeile: ");
   if (gets(s))
   {
      l = strlen(s) + 1;
      if (z = neueZeile(l))
      {
         strcpy(z+1,s);
         AddTail(&(aktuellerEditor->zeilen),z);
      }
      else
         printf("Fehler bei neueZeile!!!\n");
   }
   else
      printf("Fehler bei gets!!!\n");
}

void Test()
{
   UBYTE c;

   while ((c = getchar()) != 27)
```

```
    {
        while (getchar() != 10) ;  /* CR einlesen! */
        switch (c)
        {
            case 'a':
                input();
                break;
            case 'p':
                print();
                break;
            case 'b':
                print_blocks();
                break;
        }
    }
}
```

Die Funktion print gibt dabei den gesamten Text aus, den wir eingegeben haben. Die Funktion print_blocks gibt die Startadressen, Längen und die Anzahl der freien Bytes aller Speicherblöcke aus, die der Editor momentan belegt hat. input dagegen erwartet die Eingabe einer Zeile, die dann ans Ende der Liste aller Zeilen angefügt wird. Anders, als wir es später im richtigen Editor machen werden, gehört auch ein Null-Byte als Zeilenabschluß mit zu der Zeile, dies aber nur, um die Ausgabe aller Zeilen bei print so einfach wie möglich zu gestalten. Dort erwartet die Funktion printf nämlich einen Zeiger auf einen String als Parameter, der durch ein Null-Byte abgeschlossen ist.

Aufgerufen werden diese Funktionen von der Funktion Test aus, die von Ihnen die Eingabe eines Zeichens erwartet. So können Sie mit a (Append) eine Zeile anfügen, mit p (Print) den bisherigen Text ausgeben und mit b (Blocks) eine Liste aller Speicherblöcke erhalten. Nach dem Drücken des Buchstabens müssen Sie allerdings noch Return betätigen, da das CLI-Fenster, über das diese Eingaben laufen, zeilenorientiert arbeitet und Eingaben erst nach dem Drücken der Return-Taste an das Programm abschickt. Da uns das Return jedoch stört, wird es vor dem Switch-Case abgeschnitten.

Jetzt müssen wir nur noch unser Hauptprogramm so ändern, daß die Funktion Test automatisch aufgerufen wird. Wir ändern das Modul Editor daher wie folgt:

Zuerst definieren wir die externen Funktionen:

```
void Test();
void freeSpeicherblock();
```

Dann ändern wir die Funktion CloseEditor so, daß auch wirklich aller Speicher freigegeben wird:

```
void CloseEditor(ed)
struct Editor  *ed;
{
   register struct Speicherblock *blk;

   /* Zuerst Speicherblöcke wieder freigeben: */
   while (blk = (struct Speicherblock *)RemHead(&(ed->block)))
      freeSpeicherblock(blk);

   /* Dann Fenster schliessen: */
   ed->window->UserPort = NULL;
   CloseWindow(ed->window);
   free(ed);
}
```

Und im Hauptprogramm rufen wir vor der Hauptschleife, also vor dem Wait-Befehl, die Funktion Test auf:

```
Test();
```

Und zu guter Letzt ändern wir noch das Make-File ab, so daß die beiden neuen Module auch mit übersetzt werden. Dazu ergänzen wir die Zeile, in der OBJ definiert wird, wie folgt:

```
OBJ=src/Editor.o src/Speicher.o src/Test.o
```

Am Ende der Datei fügen wir noch folgende zwei Zeilen ein:

```
src/Speicher.o: src/Speicher.c src/Editor.h pre/Editor.pre
src/Test.o: src/Test.c src/Editor.h pre/Editor.pre
```

Die (wirklich) letzte Änderung, die wir machen sollten, besteht darin, die beiden Defines BIGBLOCK und BLOCKSIZE auf kleinere Werte zu setzen, damit wir später nicht 10 KByte Text eingeben müssen, bevor wir einen zweiten Speicherblock erhalten. Wir ändern also 10 KByte in 120 Bytes und 5 KByte in 60 Bytes um. Damit können wir auch gleich ausprobieren, was pas-

siert, wenn wir eine Zeile einfügen wollen, die länger als ein völlig leerer Block ist.

Damit heißt es mal wieder: compilieren und ausprobieren. Beim Einfügen einer Zeile, die größer als der Block ist, fällt auf, daß ein neuer Block beschafft wird, obwohl die Zeile dann auch dort nicht hineinpaßt. Da wir aber später die Blöckgröße wieder auf 5 KBytes heraufsetzen und wir ja wohl kaum Zeilen eingeben werden, die auch nur annähernd an diese Größe herankommen, werden wir mal ein Auge zudrücken. Ebenfalls interessant ist folgender Effekt: wenn z.B. nur noch 22 Bytes in einem Block frei sind, so paßt dort entweder eine Zeile mit 21 oder 22 Zeichen oder eine mit weniger als zehn Zeichen hinein, da dann auch noch Platz für eine Speicherstueck-Struktur geschaffen werden muß!

4.3.5 Zeilen löschen

Allerdings haben wir bisher nur die Funktionen zum Einfügen einer neuen Zeile getestet. Um die Funktionen optimiereBlock und garbageCollection testen zu können, müssen wir eine Funktion schreiben, die eine beliebige Zeile löschen kann. Diese Funktion entfernt zuerst die Zeile aus der Liste, sucht den Block, in dem diese liegt, und wandelt die Zeile anschließend in ein Speicherstück um. Abschließend wird noch der Block optimiert. In C sieht das Ganze so aus:

```
void loescheZeile     (zeile)
register struct Zeile *zeile;
{
   register struct Speicherblock *blk,*fblk = NULL;
   register struct Speicherstueck *st;

   /* Zeile aus Liste entfernen: */
   Remove(zeile);

   /* eventuelle Zeiger auf diese Zeile auf Null setzen: */
   if (aktuellerEditor->aktuell == zeile)
      aktuellerEditor->aktuell = NULL;
   if (aktuellerEditor->top == zeile)
      aktuellerEditor->top = NULL;
```

```
   /* Stelle fest, in welchem Block die Zeile liegt: */
   for (blk = aktuellerEditor->block.head; blk->succ;
                                        blk = blk->succ)
      if ((zeile > blk)&&(zeile < (((UBYTE *)(blk+1))
                                        + blk->laenge)))
      {
         fblk = blk;
         break;
      }

   if (fblk)
   {
      /* Zeile in Speicherstueck umwandeln, in Liste einfuegen: */
      fblk->frei += ( ((struct Speicherstueck *)zeile)->len
                     = EVENLEN(zeile->len) );
      AddTail(&(fblk->freiliste),zeile);

      optimiereBlock(fblk);
      if (fblk->laenge == (fblk->frei + sizeof(struct
                                        Speicherstueck)))
         /* Block enthaelt keine Zeilen mehr: Loeschen! */
         freeSpeicherblock(fblk);
   }
}
```

Wenn die Zeile die letzte in dem Speicherblock war, so wird dieser daraufhin gelöscht. Bevor wir jetzt noch ein paar zusätzliche Funktionen zum Testen schreiben, noch eine Bemerkung zur Funktion ConvertSpstToZeile: Weil es am einfachsten zu realisieren war, haben wir dort, falls die einzufügende Zeile kleiner war als das Speicherstück, dieses so aufgeteilt, daß die Zeile ans Ende kam. Dies bedeutet, daß die Zeilen in einem Block von hinten nach vorne wachsen.

Allerdings haben wir es bei der Garbage Collection genau andersherum gehalten, nämlich so, daß alle Zeilen an den Blockanfang kommen. Solche Unstimmigkeiten bemerkt man immer erst hinterher, aber es ist ja noch nicht zu spät. Aus diesem Grund ändern wir die Funktion ConvertSpstToZeile so, daß die Zeilen von vorne nach hinten wachsen:

```
struct Zeile *ConvertSpstToZeile(blk,st,len)
struct Speicherblock            *blk;
register struct Speicherstueck      *st;
register UWORD                          len;
{
   register UWORD l;
```

```
    register struct Zeile *z;

    z = (struct Zeile *)st;

    if (st->len == (l = EVENLEN(len)))
       /* Gefundenes Stueck passt genau: */
       Remove(st);
    else
    {
       /* Stueck aufteilen: */
       ((UBYTE *)st) += (l += sizeof (struct Zeile));
       ( st->succ = ((struct Speicherstueck *)z)->succ )
                                              ->pred = st;
       ( st->pred = ((struct Speicherstueck *)z)->pred )
                                              ->succ = st;
       st->len = ((struct Speicherstueck *)z)->len - l;
    }
    blk->frei -= l;

    z->flags = 0;
    z->len = len;

    return (z);
}
```

Kleiner Nachteil dieser Problemlösung: Die Zeiger für die Verkettung müssen umgebogen werden. Dies gibt uns dafür aber wieder einmal Gelegenheit, mit den Möglichkeiten von C zu spielen. Der Assembler-Code, den die beiden Zeilen ergeben, die die Zeiger umbiegen, ist übrigens ziemlich optimal. Schauen Sie sich doch mal den Assembler-Code an. Dazu übersetzen Sie das Modul Speicher wie folgt:

```
cc +B +Ipre/Editor.pre src/Speicher.c -a -t -o ram:sp.asm
```

Der erzeugte Assembler-Code befindet sich dann unter dem Namen sp.asm auf der RAM-Disk. Da der Assembler-Code als Kommentare die C-Befehle enthält, ist es nicht allzu schwierig, die Stelle zu finden: Suchen Sie zuerst nach dem Label _ConvertSpstToZeile, dann gehen Sie mit dem Cursor solange nach unten, bis Sie als Kommentar die oben genannten Zeilen sehen. Direkt darunter sollte sich der Assembler-Code dieser Zeilen befinden. Bei uns sah diese Stelle wie folgt aus:

```
;  ( st->succ = ((struct Speicherstueck *)z)->succ )->pred = st;
;
 move.l (a3),a0       ;a3 = z
 move.l a0,(a2)       ;a2 = st;
```

```
 move.l a2,4(a0)
;  ( st->pred = ((struct Speicherstueck *)z)->pred )->succ = st;
;
 move.l 4(a3),a0
 move.l a0,4(a2)
 move.l a2,(a0)
```

Daran läßt sich wirklich nichts mehr von Hand optimieren. Vielleicht schauen Sie sich bei der Gelegenheit auch gleich an, wie der Assembler-Code der anderen Funktionen aussieht. Dabei werden Sie nämlich bemerken, daß man durchaus noch etwas verbessern kann, z.B. bei folgendem Ausschnitt aus der Funktion garbageCollection, wobei ein Stück Speicher verschoben werden soll:

```
;       /* Zeile nach ptr herunterschieben: */
;       for (n = (EVENLEN(fz->len) + sizeof(struct Zeile)) >> 1;
;            n; n--)
 move.l d5,a0          ;d5 = fz
 move.l #0,d0          ;d6 = n
 move.b 9(a0),d0       ;a2 = ptr
 add.w #1,d0
 bclr.l #0,d0
 add.w #10,d0
 move.w d0,d6
 lsr.w #1,d6
 bra .74
.73
;        *((UWORD *)ptr)++ = *((UWORD *)fz)++;
 move.l d5,a0
 add.l #2,d5
 move.l a2,a1
 add.l #2,a2
 move.w (a0),(a1)
.71
 sub.w #1,d6
.74
 tst.w d6
 bne .73
.72
;
```

Hier kann man vor allem in der Schleife ab Label .73 einiges verbessern. So lassen sich die fünf Befehle direkt nach Label .73 durch einen einzigen ersetzen (MOVE.W (An)+,(Am)+), wenn man die Variable fz in ein Adreßregister packt statt in ein Datenregister. Auch die drei Befehle nach Label .71 und .74 lassen sich durch einen DBRA-Befehl zusammenfassen. Wenn sich

diese Funktion im späteren Betrieb als zu langsam herausstellt, dann können wir unter anderem an dieser Stelle mit Verbesserungen ansetzen.

Als nächstes brauchen wir noch Funktionen für unser Test-Modul, mit denen wir Zeilen löschen können. Auch wäre es interessant zu wissen, wie der Speicher der einzelnen Speicherblöcke aufgeteilt ist. Sehen Sie sich dazu folgende Funktionen an:

```
void print_speicher()
{
   struct Speicherblock *blk;
   register UBYTE *ptr,*end;
   register struct Zeile *z;
   register struct Speicherstueck *st;
   struct Zeile *fz;
   struct Speicherstueck *fst;

   printf("Speicheraufbau:\n");
   for (blk = aktuellerEditor->block.head; blk->succ;
                                    blk = blk->succ)
   {
      printf("\nBlock %8lx, Laenge = %ld, Frei = %ld\n",
             blk,blk->laenge,blk->frei);

      ptr = (UBYTE *)(blk + 1);
      end = ptr + blk->laenge;
      while (ptr < end)
      {
         fz = NULL;
         fst = NULL;

         /* Suche Zeile, die am naechsten an ptr liegt: */
         for (z = aktuellerEditor->zeilen.head;
                                 z->succ; z = z->succ)
            if ((z >= ptr) && (z < end))
               if (z < ((fz)? fz : (struct Zeile *)end))
                  fz = z;

         /* Keine Zeile gefunden oder zeile >
                                     ptr -> suche Sp.stueck */
         if ((fz > ptr) || !fz)
         {
           for (st = blk->freiliste.head; st->succ; st = st->succ)
               if ((st >= ptr) && (st < end))
                  if (st < ((fst)? fst :
                                 (struct Speicherstueck *)end))
                     fst = st;

            /* Falls Speicherstueck > ptr -> Fehler!!!!! */
            if ((fst > ptr) || !fst)
```

```
            {
               printf(" Unbenutztes Speicherstueck!\n");
               if (fz)
                  if (fst)
                     if (fst < fz)
                        ptr = (UBYTE *)fst;
                     else
                        ptr = (UBYTE *)fz;
                  else
                     ptr = (UBYTE *)fz;
               else
                  if (fst)
                     ptr = (UBYTE *)fst;
                  else
                     break;
            }
            else
            {
               /* Speicherstueck: */
               printf("Speicherstueck (%d)\n",fst->len);
               ptr += sizeof(struct Speicherstueck) + fst->len;
            }
         }
         else
         {
            /* Zeile */
            printf("Zeile: %s\n",fz+1);
            ptr += sizeof(struct Zeile) + EVENLEN(fz->len);
         }

      }
   }

   printf("\n");
}

void loesche_zeile()
{
   register struct Zeile *z;
   UWORD nr;

   printf("Nummer der Zeile, die geloescht werden soll: [0..n] ");
   scanf("%d",&nr);
   while (getchar() != 10) ;  /* CR einlesen! */

   for (z=aktuellerEditor->zeilen.head; z->succ && nr; z=z->succ,nr--);

     if (z->succ)
     {
        printf("Loesche: %s\n",z+1);
        loescheZeile(z);
     }
}
```

Die Funktion print_speicher sieht ja ziemlich wild aus, aber sie ist nur dafür zuständig, nacheinander jeden Block durchzugehen und die Zeilen und Speicherstücke in der Reihenfolge, in der sie im Block liegen, auszugeben.

Lediglich, wenn die Funktion feststellt, daß der Block ein Stück Speicher enthält, das weder Zeile noch Speicherstueck ist, wird eine Fehlermeldung ausgegeben und versucht, den Fehler dadurch abzufangen, daß der Zeiger ptr auf den Anfang der nächstgelegenen Zeile oder des Speicherstücks gesetzt wird. Der Zeiger ptr zeigt dabei stets auf den Anfang des Speicherbereichs, der noch zu durchsuchen ist, und end auf das Ende desselben.

Viel einfacher sieht da schon die zweite Funktion, loesche_zeile, aus! Diese verlangt nur nach der Eingabe einer Zahl und löscht dann die entsprechende Zeile, wobei wir von Null an aufwärts zählen.

Die While-Schleife mit dem getchar muß leider sein, da sonst ein Zeilenende überbleibt, das dafür sorgt, daß wir in der Funktion Test erst nach erneutem Drücken der Return-Taste Eingaben vornehmen können.

Es mag zwar elegantere Lösungen geben, aber schließlich handelt es sich hier um ein Testprogramm, bei dem es nur auf Funktionalität ankommt, und die Testfunktionen werden im fertigen Programm ohnehin nicht mehr enthalten sein.

Die Switch-Case-Anweisung der Funktion Test wird um folgende Zeilen bereichert, damit wir auch die neuen Testfunktionen ausprobieren können:

```
case 's':
   print_speicher();
   break;
case 'd':
   loesche_zeile();
   break;
```

Und mit dem Ausprobieren können wir sofort anfangen. Übersetzen Sie das Programm, und probieren Sie die einzelnen Funktionen aus: Fügen Sie neue Zeilen an, löschen Sie andere, und

lassen Sie sich zwischendurch immer wieder mal die Speicherbelegung anzeigen. Vielleicht finden Sie ja noch die eine oder andere Stelle, an der sich noch etwas verbessern läßt. Den kompletten Quelltext und den bereits compilierte Editor finden Sie auf der Diskette zum Buch, im Verzeichnis V0.2!

4.3.6 Ausgeben von Text im Fenster

Im folgenden werden wir einige Funktionen implementieren, die es uns ermöglichen, den Text im Editorfenster auszugeben und nicht wie bisher im CLI-Fenster. Wie in (fast) allen Bereichen, so bietet der Amiga bzw. sein Betriebssystem hier verschiedene Möglichkeiten, um zum Ziel zu gelangen. Wenden wir uns zuerst dem Fenster zu. Wenn Sie sich im Intuition-Teil dieses Buches die verschiedenen Möglichkeiten ansehen, die Sie beim Umgang mit Fenstern haben, so haben Sie die Qual der Wahl, welche Sie davon nutzen wollen.

Das Fenster, mit dem wir bisher gearbeitet haben und mit dem wir auch in Zukunft arbeiten werden, ist ein "ganz normales" Fenster. Schauen wir uns dazu nochmals die Daten der NewWindow-Struktur aus unserem Hauptprogramm an:

```
100,50,440,156,
```

Dies ist die Startposition des Fensters. Diese werden wir allerdings ändern (Bildschirm-füllend), sobald der Editor halbwegs funktioniert.

```
AUTOFRONTPEN,AUTOBACKPEN,
```

Diese Zeile sorgt dafür, daß unser Fenster dieselben Farben hat wie ein normales Workbench-Fenster. Als nächstes folgen die IDCMP-Flags:

```
REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW | NEWSIZE,
```

Wir wollen also mitgeteilt bekommen, wenn der Inhalt des Fensters neu ausgegeben werden muß, wenn der Benutzer die Maus- oder Tastaturtasten gedrückt hat, wenn er das Schließfeld des Fensters angeklickt hat und wenn die Größe des Fensters verän-

dert wurde. Der letzte Punkt ist neu, auf ihn kommen wir noch zu sprechen. Als letztes wollen wir uns noch den Windowflags zuwenden:

```
WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE |
SIZEBBOTTOM  | SIMPLE_REFRESH | ACTIVATE,
```

Das Fenster soll sich also vergrößern, verkleinern, bewegen und nach vorne und hinten bringen lassen. Ferner wollen wir nicht auf das Schließfeld verzichten, sonst wäre das CLOSEWINDOW-Flag bei den IDCMP-Flags ja sinnlos. Das Feld zum Vergrößern und Verkleinern soll sich im unteren Rahmen des Fensters befinden, so daß wir mehr Breite zur Verfügung haben. Das letzte Flag sorgt dafür, daß das Fenster beim Öffnen gleichzeitig aktiviert wird, was bei einem Editor sinnvoll ist.

Ein Flag, das wir wegen seiner Komplexität bis jetzt noch nicht besprochen haben, ist das SIMPLE_REFRESH-Flag. Dieses Flag bestimmt, was das Betriebssystem machen soll, wenn der Inhalt des Fensters zerstört wird, wenn also zum Beispiel ein anderes Fenster über unser Editorfenster gelegt wird. Hier gibt es insgesamt drei Alternativen, die im vorigen Kapitel bereits näher beleuchtet wurden, so daß Sie deren Vor- und Nachteile kennen. Sinnvoll für unser Programm ist nur SIMPLE_REFRESH oder SMART_REFRESH. Da unser Fenster aber ohnehin die meiste Zeit ganz vorne liegen und damit nicht verdeckt werden wird, reicht SIMPLE_REFRESH für unsere Zwecke aus, vor allem, da SIMPLE_REFRESH-Fenster weniger Speicher verbrauchen als SMART_REFRESH-Fenster.

Einen Nachteil haben allerdings SIMPLE_REFRESH-Fenster, den wir aber erst in Erfahrung bringen können, wenn wir das Scrolling einbauen. Haben Sie schon mal in einem SIMPLE_REFRESH-Fenster gescrollt, wenn ein anderes Fenster dieses teilweise verdeckt hat? (Vorausgesetzt, das Scrolling geschieht mittels der Graphics-Funktion ScrollRaster, also durch Verschieben des Fensterinhaltes.) Das Ergebnis sieht etwas seltsam aus, da das Betriebssystem keine Informationen darüber hat, was in dem verdeckten Fensterteil liegt; aber wann scrollt man denn schon mal, wenn ein Teil des Fensters verdeckt ist? Wenn Sie dies stört, so können Sie SIMPLE_REFRESH auch durch

SMART_REFRESH ersetzen, was dann allerdings mehr Speicher belegt.

Ein weiteres Problem tritt bei der Ausgabe von Texten in unserem Fenster auf. Es gilt dabei zu beachten, daß wir diese nicht in den Fensterrahmen schreiben. Allerdings gibt es auch zu diesem Zweck ein Flag, nämlich das GIMMEZEROZERO-Flag. Dieses sorgt dafür, daß wir einen eigenen RastPort für die Ausgabe innerhalb des Fensters bekommen, und daß der Fensterrahmen (Border) in einem seperaten RastPort liegt, so daß wir diesen nicht zerstören können. Aber auch dies ist mit erhöhtem Speicherbedarf verbunden, und leider wird die Ausgabe auch noch langsamer, da ja nun zwei RastPorts von Intuition verwaltet werden müssen. Wir werden daher auch hier den weniger einfachen, dafür aber effizienteren Weg gehen, und das sogenannte Clipping selbst vornehmen, also dafür sorgen, daß unser Text nicht in die Fensterumrahmung hineinragt.

Für das eigentliche Ausgeben des Textes gibt es wieder zwei Möglichkeiten bzw. Funktionen. Eine davon stammt aus der Graphics-Library und heißt Text, die zweite kommt von Intuition und nennt sich PrintIText. An die Graphics-Funktion Text übergeben Sie nur einen Zeiger auf den RastPort, den Text und dessen Länge, während PrintIText einen Zeiger auf den RastPort und einen auf eine IntuiText-Struktur, ferner noch X- und Y-Koordinaten braucht. In der IntuiText-Struktur können Sie Schreibmodus, Farbe, Position und den Zeichensatz festlegen, in dem der Text ausgegeben wird. Ferner muß der Text, auf den die IntuiText-Struktur zeigt, durch ein Null-Byte abgeschlossen sein.

Auf den ersten Blick ist die Intuition-Funktion komfortabler zu handhaben, während sich die Graphics-Funktion dagegen eher ärmlich ausnimmt. Allerdings können Sie sich ausrechnen, daß der Komfort durch einen höheren Zeitbedarf erkauft wird. Sie wissen ja, daß man nichts umsonst bekommt! Welche "Sonderfunktionen" brauchen wir denn überhaupt für unseren Editor, was die Ausgabe von Texten angeht? Wir müssen die Ausgabeposition festlegen können (Move aus Graphics-Library). Control-Codes wollten wir in Rot ausgeben, also müssen wir die

Farbe umschalten können (SetAPen aus Graphics-Library). Ferner haben wir uns überlegt, bestimmte Wörter (reservierte C-Wörter) in Fettdruck auszugeben (SetSoftStyle). Sonst brauchen wir eigentlich nichts, so daß die Intuition-Funktion fast schon unterfordert wäre. Wir wählen also zum dritten Mal die primitivere Funktion, aber nicht, weil wir die komfortablen Funktionen des Betriebssystems nicht zu schätzen wüßten, sondern weil diese für unsere Anwendung geeigneter (schneller) sind.

Wie sorgen wir nun dafür, daß unser Text nicht doch den Fensterrahmen überschreibt? Ganz einfach: Wir bestimmen, wie viele Zeichen unser Fenster bzw. der Innenraum unseres Fensters (das Fenster ohne Rahmen also) breit und hoch ist.

Dann brauchen wir bei der Ausgabe nur noch darauf zu achten, daß wir nicht mehr Zeichen ausgeben. Breite und Höhe des Fensters legen wir in der Editor-Struktur ab, da diese Werte bei mehreren Fenstern verschieden sein können.

Da sich diese beiden Werte aber ändern, wenn die Größe des Fensters verändert wird, müssen wir davon in Kenntnis gesetzt werden. Dies geschieht über das NEWSIZE-Flag, das wir neu in unsere IDCMP-Flags aufgenommen hatten. Jedesmal, wenn wir eine entsprechende Nachricht erhalten, berechnen wir die Fensterbreite und -höhe neu. Da wir ein SIMPLE_REFRESH-Fenster haben, bekommen wir direkt danach auch noch eine REFRESHWINDOW-Nachricht und können den Inhalt des Fensters gleich neu ausgeben.

Nun bleiben aber rechts und unten zwei Ränder, die weder zum Fensterrahmen noch zum Text gehören. Diese entstehen dadurch, daß die Breite des Fensters nicht zwingendermaßen durch die Breite der Buchstaben teilbar ist. Gleiches gilt analog für die Höhe. Was machen wir nun mit dem Rand, der in der folgenden Zeichnung hell gepunktet ist, wärend der Fensterrahmen dunkel gehalten ist:

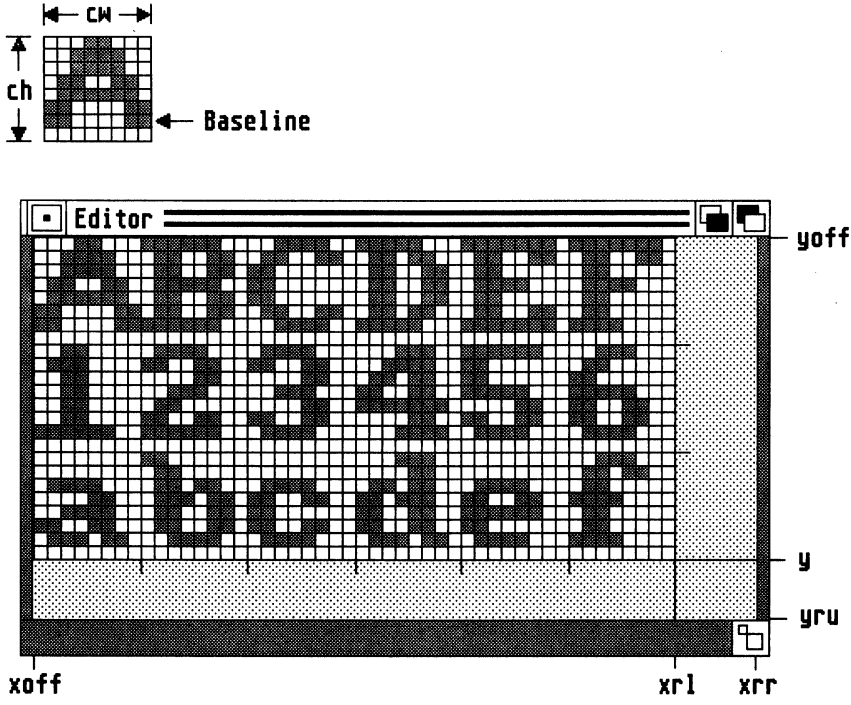


Abbildung 4.1: Window-Beispiel

Die Buchstaben wurden dabei überproportional groß dargestellt, damit auch die Ränder größer werden und damit besser erkennbar sind. Über dem Fenster sehen Sie einen Buchstaben, an dem exemplarisch dessen Breite (cw = CharacterWidth) und Höhe (ch = CharacterHeight) angezeichnet ist. Da wir glücklicherweise keinen proportionalen Zeichensatz verwenden, sind diese Werte für alle Buchstaben gleich. Die Breite des Fensters in Zeichen (wcw = WindowCharacterWidth) erhalten wir, indem wir die tatsächliche Breite des Fensters nehmen (window->Width), die Breite des linken (window->BorderLeft) und rechten Fensterrahmens (window->BorderRight) abziehen und das Ergebnis durch die Breite eines Buchstabens (cw) dividieren. Für die Höhe gehen wir analog vor:

```
wch = (window->Height - window->BorderTop
                      - window->BorderBottom) / ch;
```

Für die Ausgabe brauchen wir ebenfalls die Breite des linken und des oberen Fensterrahmens, da diese beiden die oberste linke Position unseres Fensters beschreiben. Die beiden Werte speichern wir als xoff (X-Offset) und yoff (Y-Offset) in unsere Editor-Struktur ab, ebenso wie cw und ch, die durchaus für verschiedene Fenster unterschiedlich ausfallen können, wenn man mittels Preferences den Zeichensatz ändert und danach ein weiteres Editorfenster öffnet! Auch in unsere Editor-Struktur gehört ein Zeiger auf den RastPort des dazugehörigen Fensters, damit wir nicht immer über die Window-Struktur gehen müssen, um an diesen heranzukommen. Sie wissen ja, daß auf dem Amiga grafische Ausgaben nur über RastPorts laufen.

Unsere Frage, was wir mit den Rändern machen, haben wir aber immer noch nicht beantwortet. Wir brauchen diese zwar nicht, aber störenden "Grafikmüll" sollten diese auch nicht enthalten. Daher löschen wir sicherheitshalber den Inhalt der beiden Ränder jedesmal, wenn wir eine REFRESHWINDOW-Nachricht erhalten. Dazu benötigen wir zusätzliche Informationen, wie zum Beispiel die Position der Ränder. In der obigen Zeichnung sind die außer xoff und yoff zusätzlichen Koordinaten mit xrl (X, Rand, linke Ecke), xrr (X, Rand, rechte Ecke) und yru (Y, Rand, untere Ecke) bezeichnet. Zusätzlich benötigen wir die obere Ecke des Randes, die in der Zeichnung mit Y angegeben ist. Dieses Y brauchen wir aber jetzt noch nicht zu berechnen, da es sich quasi von selbst ergibt: Wir werden nämlich den Text Zeile für Zeile ausgeben und dabei die Y-Koordinate heraufzählen. Wenn entweder das Fenster voll oder kein Text mehr da ist, haben wir die Y-Koordinate der Zeile direkt unter der letzten Textzeile, die exakt der oberen Ecke des Randes entspricht.

Weitere Einzelheiten ergeben sich im Programm. Beim Positionieren des Textes gilt es aber zu beachten, daß der Text immer relativ zu seiner Baseline ausgegeben wird. Wir müssen daher die Y-Position um den Abstand der Baseline von der oberen Kante des Buchstabens erhöhen, bevor wir Move aufrufen. Doch genug der Feinheiten! Bauen wir jetzt die einzelnen Funktionen zusammen, die für die Ausgabe des Textes sorgen:

```
void printAll()
```

Diese Funktion gibt den gesamten Text aus. Dabei wird zunächst die oberste Zeile im Fenster bestimmt, ab dieser werden solange Zeilen ausgegeben, bis das Fenster voll oder das Textende erreicht ist. Dabei wird die Funktion printLine aufgerufen, an die ein Zeiger auf die Zeile und die aktuelle Y-Position übergeben wird. Danach werden die beiden Ränder gelöscht, sofern diese breiter (höher) als null sind. Die nächste Funktion ist damit logischerweise:

```
void printLine(zeile,y)
```

Diese gibt die Zeile linksbündig an der übergebenen Y-Position aus. Doch dies ist gar nicht so einfach, wie es sich zunächst anhört! So arbeitet unser Editor mit "richtigen" Tabulatoren, die bei der Ausgabe nur als Kästchen erscheinen würden. Wenn wir die Ausgabe über das Console.Device laufen lassen würden, so würden Tabulatoren richtig ausgegeben werden. Allerdings hat das Console.Device viel zu viel Overhead, mit dem wir nichts anfangen könnten und der die Ausgabe langsamer macht. Also müssen wir die Tabulatoren selbst umwandeln. Dazu verwenden wir eine weitere Funktion:

```
void convertLineForPrint(text,länge,breite,puffer)
```

Diese erwartet einen Zeiger auf den Text (und nicht mehr auf die Zeile-Struktur) und dessen Länge. Ferner müssen Sie einen Zeiger auf einen Puffer übergeben, in den die konvertierte Zeile geschrieben wird, sowie die Breite einer Zeile, da alles bis zum Zeilenende ggf. mit Blanks aufgefüllt wird. Da das CR bzw. LF am Ende einer Zeile aus optischen Gründen nicht mit ausgegeben werden sollte, wird es von dieser Funktion abgeschnitten. Wird die Zeile nicht durch ein CR, LF oder CRLF beendet, so wird der Rest der Zeile mit Punkten (CHR$(183)) aufgefüllt, was bedeuten soll, daß die Zeile noch weitergeht. Dieses gilt auch für Zeilen, die nicht ganz in das Fenster passen, also zu lang sind. So bedeutet ein Punkt (CHR$(183)) am Ende einer Zeile immer, daß die Zeile noch weitergeht, mehrere Punkte, daß die Zeile nicht durch ein normales Zeilenende abgeschlossen ist.

Bei der Ausgabe der konvertierten Zeile könnten wir es uns nun einfach machen und diese einfach mittels der Graphics-Funktion Text ausgeben. Da aber eine Zeile stets auch aus Blanks besteht, sei es, weil die Zeile nicht so breit wie das Fenster ist, sei es, daß die Zeile aus Formatgründen eingerückt ist, wäre dies uneffizient. Das Ausgeben von Blanks kostet genausoviel Zeit wie das Ausgeben eines Buchstabens. Schneller ist es, wenn wir hintereinander liegende Blanks auf den Bildschirm bringen, indem wir einfach ein Rechteck zeichnen, das dieselbe Größe hat. Diese Aufgabe nimmt uns die folgende Funktion ab:

```
void printAt(puffer,länge,x,y)
```

Außerdem behandelt diese Funktion auch die Ausgabe von Control-Codes, die in Rot ausgegeben werden, also CHR$(1) als rotes "A" ("A" = CHR$(1 | 64)). Übergeben wird ein Zeiger auf den Text, dessen Länge, die normalerweise der Bildschirmbreite in Zeichen entspricht, und die Position, wobei die X-Koordinate in der Regel xoff entspricht. Damit haben wir nun die wesentlichen Funktionen zur Ausgabe des Textes auf dem Bildschirm zusammen. Fehlt nur noch eine Funktion, die die ganzen Werte bestimmt, von denen die Ausgabe abhängt, also wcw, wch, xoff etc.:

```
void initWindowSize(ed)
```

Die Variable ed zeigt dabei auf die Editor-Struktur, deren Werte bestimmt werden sollen. Lassen Sie uns, bevor wir an die Realisierung der Funktionen gehen, nochmals unsere Liste mit den Eigenschaften durchgehen, die unser Editor haben soll. Vielleicht finden wir ja etwas, das wir bereits jetzt berücksichtigen sollten, damit wir später weniger Arbeit damit haben. Folding! Wenn wir Folding in den Editor einbauen, so liegen zwei Zeilen, die im Fenster direkt untereinander stehen, im tatsächlichen Text vielleicht einige Zeilen auseinander; man kann ja beliebig viel Text "wegfalten". Das bedeutet aber, daß es nicht reicht, uns nur einen Zeiger auf die oberste Zeile zu merken (Editor-Struktur: top), wie wir es bisher vorgesehen hatten. Wenn wir dann nämlich auf eine Zeile zugreifen wollen, so müßten wir uns schlimmstenfalls durch ein paar hundert Zeilen durchkämpfen, bis wir die Zeile erreicht haben, die wir suchen.

Schneller geht es, wenn wir uns für jede Zeile, die im Fenster sichtbar ist, und am besten für die nächste Zeile auch noch, einen Zeiger darauf merken. Wenn wir die Position im Fenster kennen, dann kommen wir direkt an den Zeiger und über den Zeiger sofort zur Zeile. Nachteil: Wir müssen stets die Zeiger aktualisieren. Dies kostet aber weniger Zeit als das Durchhangeln durch die Liste. Da die maximale Anzahl der Zeilen, die auf den Bildschirm passen, beschränkt ist, und zwar auf 64 (gleich 512 (maximale Auflösung) durch 8 (Höhe eines Zeichens)), können wir auch das Feld auf diese Größe festlegen. Dabei werden allerdings nur die Zeiger vermerkt, deren Zeilen auch wirklich auf dem Bildschirm zu sehen sind (plus die erste Zeile, die nicht mehr zu sehen ist), und nicht alle 64, es sei denn, das Fenster ist wirklich 63 Zeilen hoch; alle anderen setzen wir sicherheitshalber auf Null.

Als erstes wollen wir die Include-Datei Editor.h entsprechend erweitern:

```
#define MAXHOEHE 64
#define FGPEN 1L
#define BGPEN 0L
#define CTRLPEN 3L
#define CONTROLCODE(c) (((c)&127)<32)

struct Editor
{
   struct Editor *succ;
   struct Editor *pred;
   struct Window *window;
   struct BList block;
   struct ZList zeilen;
   UBYTE  puffer[MAXBREITE];
   UBYTE  tabstring[MAXBREITE];
   UWORD  anz_zeilen;
   struct Zeile *aktuell;
   struct Zeile *zeilenptr[MAXHOEHE];
   UWORD  toppos;
   UWORD  xpos,ypos;
   UWORD  changed:1;
   UWORD  insert:1;
   struct RastPort *rp;
   UWORD  xoff,yoff;
   UWORD  cw,ch;
   UWORD  wcw,wch;
   UWORD  xrl,xrr,yru,bl;
};
```

Außer der maximalen Fensterhöhe werden auch noch die Standard-Vordergrundfarbe (FGPEN = ForegroundPen), die Hintergrundfarbe (BGPEN BackgroundPen) und die Farbe für die Control-Codes (CTRLPEN) definiert. Es wird ebenfalls festgelegt, was überhaupt ein Control-Code ist (CONTROLCODE), nämlich ein Zeichen, dessen Code kleiner als 32 oder größergleich 128 und kleiner als 160 ist. Dies sind die Zeichen, die im Amiga-Zeichensatz nur durch Rechtecke dargestellt werden. Wenn Sie Ihren eigenen Zeichensatz verwenden wollen, in dem Sie diesen Zeichen eigene Darstellungen zugewiesen haben, so können Sie die Definition von CONTROLCODE ja entsprechend ändern.

Geändert wurde die Editor-Struktur: Statt des Zeigers top ist dort nun das Feld zeilenptr vorhanden, wobei zeilenptr[0] dieselbe Funktion hat, die zuvor top zugedacht war. Des weiteren wurden zusätzliche Elemente für die Ausgabe am Ende der Struktur hinzugefügt, die aber schon besprochen wurden. Entsprechend müssen auch zwei Funktionen des Moduls Speicher umgeschrieben werden, die auf top Bezug nehmen. In der Funktion garbageCollection müssen die Zeilen:

```
if (aktuellerEditor->top == z)
   aktuellerEditor->top = fz;
```

durch die folgenden Zeilen ersetzt werden:

```
for (n = 0, zptr = aktuellerEditor->zeilenptr;
     n <= aktuellerEditor->wch; n++, zptr++)
   if (*zptr == z)
   {
      *zptr = fz;
      break;
   }
```

wobei zptr noch als "struct Zeile **zptr" definiert werden muß. Auch in der Funktion loescheZeile ist eine Änderung notwendig. Dort müssen die Zeilen:

```
if (aktuellerEditor->top == zeile)
   aktuellerEditor->top = NULL;
```

durch folgende ersetzt werden:

```
    for (n = 0, zptr = aktuellerEditor->zeilenptr;
         n <= aktuellerEditor->wch; n++, zptr++)
       if (*zptr == zeile)
       {
          *zptr = NULL;
          break;
       }
```

Auch hier müssen zptr und n (UWORD) definiert werden. Kommen wir nun zu den Änderungen im Hauptmodul. Als externe Funktion kommt nicht nur printAll hinzu:

```
    void initWindowSize(),SetDrMd(),SetAPen(),SetBPen(),printAll();
    void BeginRefresh(),EndRefresh();
```

Die beiden folgenden neuen Funktionen bestimmen jeweils die auf eine Zeile folgende bzw. die vorige Zeile. Im Moment sieht dies zwar noch etwas überflüssig aus, aber spätestens, wenn der Editor Folding beherrschen wird, werden Sie diese Funktionen zu schätzen wissen, da sie das "Durchhangeln" durch den Text vereinfachen:

```
/*********************************************
 *
 * nextLine(zeile)
 *
 * Liefert Zeiger auf naechste Zeile zurueck,
 * (und beruecksichtigt dabei Folding!)
 * Null, falls keine naechste Zeile.
 *
 * zeile ^ Zeile-Struktur.
 *
 *********************************************/

struct Zeile *nextLine(zeile)
register struct Zeile *zeile;
{
   register struct Zeile *z;

   if (zeile)
   {
      if (z = zeile->succ)
         if (!(z->succ))
            z = NULL;

      return (z);
   }
   else
      return (NULL);
      }
```

```
/*****************************************
 *
 * prevLine(zeile)
 *
 * Liefert Zeiger auf vorige Zeile zurueck,
 * (und beruecksichtigt dabei Folding!)
 * Null, falls keine vorige Zeile.
 *
 * zeile ^ Zeile-Struktur.
 *
 *****************************************/

struct Zeile *prevLine(zeile)
register struct Zeile *zeile;
{
   register struct Zeile *z;

   if (zeile)
   {
      if (z = zeile->pred)
         if (!(z->pred))
            z = NULL;

      return (z);
   }
   else
      return (NULL);
}
```

Die Funktion OpenEditor wurde wesentlich erweitert:

```
struct Editor *OpenEditor()
{
   register struct Editor *ed = NULL;
   register struct Window *wd;
   register struct RastPort *rp;
   register ULONG flags;
   register UBYTE *ptr;
   register struct Zeile **zptr;
   register UWORD n;

   /* IDCMPFlags retten, in Struktur auf NULL setzen!
      => eigenen UserPort verwenden! */
   flags = newEdWindow.IDCMPFlags;
   newEdWindow.IDCMPFlags = NULL;

   /* Speicher fuer Editor-Struktur beschaffen: */
   if (ed = malloc(sizeof(struct Editor)))
      /* Fenster oeffnen: */
      if (wd = OpenWindow(&newEdWindow))
      {
         ed->window = wd;
```

```
        ed->rp = (rp = wd->RPort);

        /* Schreibmodi setzen: */
        SetDrMd(rp,JAM2);
        SetAPen(rp,FGPEN);
        SetBPen(rp,BGPEN);

        /* UserPort einrichten: */
        wd->UserPort = edUserPort;
        ModifyIDCMP(wd,flags);

        /* Parameter initialisieren: */
        NewList(&(ed->block));
        NewList(&(ed->zeilen));
        ed->anz_zeilen = 0;
        ed->aktuell = NULL;
        ed->toppos = 0;
        ed->xpos = 1;
        ed->ypos = 1;
        ed->changed = 0;
        ed->insert = 1;

        /* zeilenptr-Array initialisieren: */
        for (n = 0, zptr = ed->zeilenptr;
                              n < MAXHOEHE; n++, zptr++)
           *zptr = NULL;

        /* Tabulatoren initialisieren: */
        ptr = ed->tabstring;
        *ptr++ = 1;
        for (n = 1; n < MAXBREITE; n++)
           if (n % 3)
              *ptr++ = 1;
           else
              *ptr++ = 0;

        ed->wch = 0;
        initWindowSize(ed);
     }
     else
     {
        free(ed);
        ed = NULL;
     }

  newEdWindow.IDCMPFlags = flags;
  return (ed);
}
```

Das Initialisieren der Tabulatoren geschieht, damit wir auf jeden Fall ein paar definierte Tabulatoren zur Verfügung haben. In der

Abfrage des Nachrichtentyps im Hauptprogramm fügen wir zwei weitere Flags hinzu:

```
case NEWSIZE:
   initWindowSize(aktuellerEditor);
   print();
   break;

case REFRESHWINDOW:
   BeginRefresh(aktuellerEditor->window);
   printAll();
   EndRefresh(aktuellerEditor->window,TRUE);
   break;
```

Die beiden Funktionen BeginRefresh und EndRefresh sollte man nach einem REFRESHWINDOW-Event auf jeden Fall aufrufen, damit Intuition weiß, daß das Fenster restauriert wurde. Außerdem sorgen diese Funktionen dafür, daß nur die Teile des Fensters neu ausgegeben werden, die es nötig haben, aber das wissen Sie ja bereits aus dem zweiten Kapitel. Die Funktion print stammt aus der Testumgebung und wird noch etwas abgewandelt werden. Bevor wir uns dieser zuwenden, wollen wir das neue Modul "Ausgabe" implementieren:

```
<src/Ausgabe.c>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define CR 13
#define LF 10
#define TAB 9
```

```
/**********************
 *
 * Externe Funktionen:
 *
 **********************/

void SetAPen(),RectFill(),Text(),Move();
struct Zeile *nextLine();

/**********************
 *
 * Externe Variablen:
 *
 **********************/

extern struct Editor *aktuellerEditor;

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

/***********************************
 *
 * initWindowSize(ed)
 *
 * Initialisiert die Breite/Hoehe-
 * Angaben des aktuellen Fensters.
 * Ferner wird das zeilenptr-Feld
 * restauriert.
 *
 * ed ^ Editor-Struktur.
 *
 ***********************************/

void initWindowSize    (ed)
register struct Editor *ed;
{
   register struct Window *w;
   register struct RastPort *rp;
   register struct Zeile *z,**zptr;
   register UWORD n,newh;

   w = ed->window;
   rp = ed->rp;

   ed->xoff = (UWORD)w->BorderLeft;
   ed->yoff = (UWORD)w->BorderTop;

   ed->cw = (rp->TxWidth)? rp->TxWidth : 8;
   ed->ch = (rp->TxHeight)? rp->TxHeight : 8;

   ed->wcw = (w->Width - ed->xoff - w->BorderRight) / ed->cw;
```

```
   newh  = (w->Height- ed->yoff - w->BorderBottom)/ ed->ch;

   ed->xrl = ed->xoff + ed->wcw*ed->cw;
   ed->xrr = w->Width - w->BorderRight - 1;
   ed->yru = w->Height - w->BorderBottom - 1;
   ed->bl = rp->TxBaseline;

   /* Nun ggf zeilenptr-Feld restaurieren: */
   if (newh != ed->wch)
   {
      if (newh < ed->wch)
      {
         /* Ueberfluessige Zeiger auf Null setzen: */
         zptr = ed->zeilenptr + newh + 1;
         for (n = newh; n < ed->wch; n++)
            *zptr++ = NULL;
      }
      else
      {
         zptr = ed->zeilenptr + ed->wch;
         z  = *zptr++;
         for (n = ed->wch; n < newh; n++)
            *zptr++ = (z = nextLine(z));
      }

      ed->wch = newh;
   }
}

/****************************************
 *
 * convertLineForPrint(zeile,len,buf)
 *
 * Konvertiert zeile so, dass diese
 * einfach ausgegeben werden kann.
 * Dabei werden v.a. Tabs in Spaces
 * umgewandelt.
 *
 * zeile ^ Zeile.
 * len = deren Laenge.
 * w = maximale Breite.
 * buf ^ Puffer, in den die konvertierte
 *   Zeile abgelegt wird.
 *
 ****************************************/

void convertLineForPrint(zeile,len,w,buf)
UBYTE                  *zeile;
register WORD                 len;
register UWORD                    w;
register UBYTE                     *buf;
{
   register UBYTE *tab;
   register UWORD l = 1;
```

```
   register UBYTE lastchar;

   /* Feststellen, ob mit CR oder LF beendet: */
   if (len)
   {
      tab = zeile + len - 1;
      if (*tab == CR)
      {
         len--;
         lastchar = ' ';
      }
      else if (*tab == LF)
      {
         len--;
         lastchar = ' ';
         if ((len) && (*--tab == CR))
            len--;
      }
      else
         lastchar = 183;
   }
   else
      lastchar = 183;

   /* Zeile konvertieren: */
   tab = aktuellerEditor->tabstring;
   while ((len--) && (l < w))
      if ((*buf = *zeile++) == TAB)
         do
         {
            *buf++ = ' ';
            l++;
            tab++;
         } while ((*tab) && (l < w));
      else
      {
         buf++;
         tab++;
         l++;
      }

   /* Feststellen, ob Zeile nicht komplett konvertiert: */
   if (len >= 0)
      lastchar = 183;

   /* Zeile bis zum Ende mit lastchar auffuellen: */
   while (l++ <= w)
      *buf++ = lastchar;
}

/************************************
 *
 * printAt(buf,len,x,y)
 *
```

```
 * Gibt Zeichenkette buf an Position
 * (x/y) im Fenster aus. Die Zeichen-
 * kette wird dabei veraendert!
 *
 * buf ^ Zeichenkette.
 * len = deren Laenge.
 * x = Pixel-X-Position.
 * y = Pixel-Y-Position.
 *
 ************************************/

void printAt   (buf,len,x,y)
register UBYTE *buf;
WORD                len;
register UWORD          x,y;
{
   struct RastPort *rp = aktuellerEditor->rp;
   register UWORD cw = aktuellerEditor->cw;
   register UWORD x2;
   UWORD y2 = y + aktuellerEditor->ch - 1;
   UBYTE *ptr;
   ULONG l;

   while (len > 0)
      if (*buf == ' ')
      {
         /* Blanks ausgeben: */
         x2 = x - 1;

         while ((*buf == ' ') && len--)
         {
            buf++;
            x2 += cw;
         }

         SetAPen(rp,BGPEN);
         RectFill(rp,(ULONG)x,(ULONG)y,(ULONG)x2,(ULONG)y2);
         SetAPen(rp,FGPEN);

         x = ++x2;
      }
      else
      {
         /* Also Text ausgeben: */
         Move(rp,(ULONG)x,(ULONG)y + aktuellerEditor->bl);
         ptr = buf;

         if (CONTROLCODE(*buf))
         {
            SetAPen(rp,CTRLPEN);

            while (CONTROLCODE(*buf) && len--)
               *buf++ |= 64;
            l = buf - ptr;
```

```
            Text(rp,ptr,l);

            SetAPen(rp,FGPEN);
         }
         else
         {
            while (!CONTROLCODE(*buf) && (*buf != ' ')
                                          && len--) buf++;
            l = buf - ptr;
            Text(rp,ptr,l);
         }
         x += cw*l;
      }
}

/************************************
 *
 * printLine(zeile,y)
 *
 * Gibt Zeile auf dem Bildschirm an
 * Pixel-Position y aus.
 *
 * zeile ^ Zeile-Struktur.
 * y = Pixel-Position.
 *
 ************************************/

void printLine        (zeile,y)
register struct Zeile *zeile;
register UWORD                y;
{
   static UBYTE buf[MAXBREITE];
   register UWORD w;

   convertLineForPrint(zeile+1,zeile->len,w =
                              aktuellerEditor->wcw,buf);
   printAt(buf,w,aktuellerEditor->xoff,y);
}

/*********************************
 *
 * printAll
 *
 * Gibt ganzen Bildschirm neu aus.
 *
 *********************************/

void printAll()
{
   register UWORD y,ch,count;
   register struct Zeile **z;
   register struct RastPort *rp;

   y     = aktuellerEditor->yoff;
```

```
    ch    = aktuellerEditor->ch;
    count = aktuellerEditor->wch;
          rp    = aktuellerEditor->rp;

    /* Zeilen ausgeben: */
    for (z = aktuellerEditor->zeilenptr; count-- && (*z != NULL);
                                                  z++, y += ch)
       printLine(*z,y);

    /* Raender rechts und unten loeschen: */
    SetAPen(rp,BGPEN);
    if (aktuellerEditor->yru >= y)
       RectFill(rp,(ULONG)aktuellerEditor->xoff,(ULONG)y,
              (ULONG)aktuellerEditor->xrr,
                           (ULONG)aktuellerEditor->yru);
    if ((aktuellerEditor->xrr >= aktuellerEditor->xrl)
       && (y > aktuellerEditor->yoff))
       RectFill(rp,(ULONG)aktuellerEditor->xrl,
                   (ULONG)aktuellerEditor->yoff,
                   (ULONG)aktuellerEditor->xrr,(ULONG)y-1);
    SetAPen(rp,FGPEN);
}
```

Auch hier wieder einige Bemerkungen zu den obigen Funktionen und deren Besonderheiten:

- In der Funktion initWindowSize wird bei der Größe des Zeichensatzes eine Sicherheitsabfrage auf Null gemacht. Dies kann zwar eigentlich nicht auftreten, aber da später durch diese Werte dividiert wird, sollte man sich den Luxus einer Abfrage gönnen, man weiß ja nie...

- Je nachdem, ob das Fenster vergrößert oder verkleinert wurde, werden zusätzliche Zeiger berechnet bzw. die überflüssigen Zeiger auf Null gesetzt. Dabei wird die Funktion nextLine verwendet, damit wir es um so einfacher haben, wenn wir Folding implementieren. Dann brauchen wir nur die Funktionen nextLine und prevLine zu ändern, und alle weiteren Funktionen, die auf diese beiden Funktionen zugreifen, funktionieren ebenfalls mit Folding.

- In convertLineForPrint wird zuerst das Zeilenende begutachtet und dementsprechend die Variable lastchar gesetzt. Sie enthält das Zeichen, mit dem der Rest der Zeile aufgefüllt wird. Die Variable tab dient dabei als Zeiger auf das Zeilenende. Beachten Sie dabei, daß hinter dem ei-

gentlichen Text der Zeile mindestens ein weiteres Zeichen ausgegeben wird, nämlich ein Blank oder ein Punkt. Wenn das Fenster also 70 Zeichen breit ist und die Zeile ebenso, so werden trotzdem nur 69 Zeichen angezeigt. In die siebzigste Spalte kommt dann der Punkt, da die Zeile ja nicht komplett ins Fenster paßte.

- In printAt enthalten x2 und y2 jeweils die Koordinaten der rechten unteren Ecke für das Rechteck, das im Falle von Blanks ausgegeben werden muß. Dabei muß zuvor die Zeichenfarbe auf die Hintergrundfarbe gesetzt werden, da wir sonst ein weißes Rechteck erhalten.
- Beim Move-Befehl wird die Baseline des Zeichensatzes auf die Y-Koordinate addiert, damit der Text richtig positioniert wird. Die X-Koordinate wird jeweils auf die nächste Ausgabeposition gesetzt.
- Die vielen Typumwandlungen sind notwenig, da wir stets LONG-Werte an die Betriebssystemfunktionen übergeben müssen, sonst aber nach Möglichkeit mit Wörtern rechnen, da deren Bearbeitung schneller vonstatten geht.
- In der Funktion printLine definieren wir den Puffer als static, da er sonst unnötigerweise Platz auf dem Stack belegen würde. Zwar belegen auch andere lokale Variablen Platz auf dem Stack, aber nicht 80 Bytes auf einmal.
- Haben Sie die Funktion RectFill schon mal mit Koordinaten aufgerufen, bei denen die linke Ecke rechts von der rechten Ecke lag bzw. die obere Ecke unter der unteren Ecke? Damit können Sie Ihren Amiga schnell, sicher und bunt "abschießen", weswegen die Abfrage in printAll sehr wichtig ist. Wegen der Definition von xrl, xrr und yru kann es nämlich vorkommen, daß die Ränder negative Breiten haben, nämlich genau dann, wenn die Breite bzw. Höhe des Fensters ganzzahlig durch die Breite bzw. Höhe eines Zeichens teilbar ist. In diesem Fall darf kein Rand ausgegeben werden!

So weit, so gut. Fehlen nur noch die Änderungen in der Testumgebung. Zuerst deklarieren wir ein paar externe Funktionen:

```
struct Zeile *nextLine();
void printAll();
```

Dann schreiben wir die Funktion print so um, daß sie uns die Werte der Zeiger liefert, die in dem Feld zeilenptr festgehalten werden:

```
void print()
{
   register struct Zeile **z;
   register UWORD n;

   for (n = 0, z = aktuellerEditor->zeilenptr;
        n <= aktuellerEditor->wch; n++, z++)
      printf("Zeile %d an Adresse %6lx.\n",n,*z);
}
```

Neu hinzu kommt eine Funktion zur Initialisierung des zeilenptr-Feldes. Diese werden wir so später nicht im eigentlichen Programm brauchen, so daß wir sie in die Testumgebung ausgelagert haben:

```
void init_zeilenptr()
{
   register struct Zeile *z,**zptr;
   register UWORD n;

   for (n = 0, z = (struct Zeile *)
                        &(aktuellerEditor->zeilen.head),
        zptr = aktuellerEditor->zeilenptr; n < MAXHOEHE; n++)
      *zptr++ = (z = nextLine(z));
}
```

Und zu allerletzt ändern wir noch etwas im eigentlichen Testprogramm. Nach dem Einfügen und Löschen von Zeilen muß jetzt das zeilenptr-Feld neu initialisiert werden. Ferner muß bei dem Befehl p (print) auch die Funktion printAll aufgerufen werden. Die entsprechenden Zweige der Switch-Case-Anweisung sehen nun so aus:

```
case 'a':
   input();
   init_zeilenptr();
   break;
case 'p':
   print();
   printAll();
```

```
   break;
case 'd':
   loesche_zeile();
   init_zeilenptr();
   break;
```

Nachdem Sie die Make-Datei um die folgende Zeile und die Objektdefinition (OBJ=...) um src/Ausgabe.o erweitert haben, haben Sie es auch schon geschafft:

```
src/Ausgabe.o: src/Ausgabe.c src/Editor.h pre/Editor.pre
```

Wenn Sie den Editor nicht extra abtippen wollen, so finden Sie ihn ebenso wie die Quelltexte im Verzeichnis V0.3 der Diskette zum Buch. Starten Sie den Editor, und geben Sie mehrere Zeilen ein (mit a = Append). Verlassen Sie anschließend die Testumgebung durch Escape. Danach können Sie das Fenster verschieben, vergrößern, verkleinern usw. und somit ausprobieren, ob der Inhalt des Fensters wirklich immer korrekt ausgegeben wird.

Da wir bei der Ausgabe sehr viel über Effizienz gesprochen haben und uns davor auch schon den Assembler-Quelltext angesehen haben, den der C-Compiler "ausgestoßen" hat, wollen wir diese Kenntnisse nun dazu benutzen, unser Programm noch etwas effizienter zu machen.

Sie erinnern sich sicher noch, daß der C-Compiler den Ausdruck "*var++" etwas ungünstig übersetzt hatte. Lassen wir das Inkrement (++) weg und bearbeiten dies extra, so erzeugt der C-Compiler bereits einen besseren Code. Übersetzen Sie dazu das folgende kleine Testprogramm so, daß Sie dessen Assembler-Quelltext erhalten:

```
main()
{
   register char *ptr;
   char string[10];
   register short n;

   for (n = 0, ptr = string; n < 10; n++)
      *ptr++ = 0;

   for (n = 0, ptr = string; n < 10; n++, ptr++)
      *ptr = 0;
}
```

In der ersten For-Schleife wird der Zeiger ptr bei der Zuweisung inkrementiert, während er in der zweiten Schleife extra inkrementiert wird. Übersetzen Sie das Prograämmchen mit folgender Anweisung:

```
cc test.c -a -t
```

Sie erhalten den Assembler-Quelltext mit dem Namen Test.asm, der bei unserem Aztec-C-Compiler folgendermaßen aussieht:

```
;                       main()
;                       {
   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
;                          register char *ptr;
;                          char string[10];
;                          register short n;
;
;                          for (n = 0, ptr = string; n < 10; n++)
   move.l #0,d4
   lea -10(a5),a0
   move.l a0,a2
   bra .7
.6
;                             *ptr++ = 0;
   move.l a2,a0
   add.l #1,a2
   clr.b (a0)
.4
   add.w #1,d4
.7
   cmp.w #10,d4
   blt .6
.5
;
;                       for (n = 0, ptr = string; n < 10; n++, ptr++)
   move.l #0,d4
   lea -10(a5),a0
   move.l a0,a2
   bra .11
.10
;                             *ptr = 0;
   clr.b (a2)
.8
   add.w #1,d4
   add.l #1,a2
.11
   cmp.w #10,d4
   blt .10
```

```
.9
;                       )
.12
   movem.l (sp)+,.3
   unlk a5
   rts
.2 equ -10
.3 reg d4/a2
   public .begin
   dseg
   end
```

Schauen Sie sich insbesondere die beiden Schleifenkörper an, also für die erste Schleife von Label .6 bis Label .7 und für die zweite Schleife von Label .10 bis Label .11. Sie sehen selbst, daß die erste Schleife hier vier Befehle braucht, während die zweite mit drei Befehlen auskommt. Dabei ist besonders wichtig, daß diese Befehle so oft durchlaufen werden, wie der Schleifenzähler vorgibt. Dieser Befehl wirkt sich also um so stärker aus, je öfter die Schleife durchlaufen wird.

Sie sehen, es lohnt sich, den Assembler-Quelltext einmal genauer unter die Lupe zu nehmen. Zwar geben sich die Compiler-Hersteller alle Mühe, ihr Produkt so effizient wie möglich zu machen, doch besser als von Hand erstellter Assembler-Code kann ein Compiler nicht sein. Im obigen Testprogramm würde mindestens noch ein weiterer Befehl wegfallen, so daß nur noch ein CLR.B (A2)+ übrigbleiben würde. Auch der Schleifenzähler würde anders realisiert werden, aber das steht auf einem anderen Blatt.

Wir, also die Autoren, sind nun in einer Zwickmühle. Einerseits möchten wir Ihnen zeigen, wie man möglichst optimale Programme schreibt, andererseits sind optimale Programme meist schlechter lesbar oder weniger einsichtig, so daß Sie vielleicht Schwierigkeiten haben, sie zu verstehen. Sie wissen ja, daß es vor allem in C keine Probleme bereitet, unlesbare Programme zu schreiben. Daher werden wir weiterhin einen Kompromiß zwischen Lesbarkeit und Effizienz eingehen.

4.3.7 Der Cursor

Um den Text zu verändern, müssen wir jede Textstelle erreichen, also den Cursor überall hinbewegen können. Wir werden daher im folgenden unseren Editor dahingehend erweitern, daß man den Cursor sowohl über die Cursor-Tasten der Tastatur als auch mit der Maus bewegen kann. Erreicht der Cursor dabei den Rand des Fensters, so muß dessen Inhalt gescrollt werden, damit man auch außerhalb des Fensters liegende Textstellen erreichen kann. Dies gilt vor allem für die vertikale Bewegung des Cursors, da die zu bearbeitenden Texte meist aus mehr Zeilen bestehen als in unser Fenster passen. Aber auch horizontales Scrolling muß möglich sein, da es durchaus sein kann, daß einzelne Zeilen breiter sind als das Editorfenster, vor allem, wenn wir das Fenster verkleinert haben.

Bei der Cursor-Bewegung müssen wir das Folding berücksichtigen. Sie erinnern sich, daß wir bereits festgestellt haben, daß zwei Zeilen, die im Editorfenster direkt untereinander liegen, im Text nicht zwingendermaßen direkt aufeinander folgen müssen. Enthält eine Zeile eine Faltenanfangsmarkierung, so sind alle folgenden Zeilen bis zur dazugehörigen Faltenendemarkierung unsichtbar, und die im Fenster nächste Zeile ist die, die auf die Faltenendemarkierung folgt. Das bedeutet, daß wir die Zeilenposition des Cursors nicht direkt über seine Position im Fenster bestimmen können. Ebenso wie bei dem Feld zeilenptr brauchen wir auch hier wieder ein Feld, das für jede Zeile, die im Editorfenster sichtbar ist, deren Zeilennummer bereithält.

Befindet sich der Cursor auf einer Zeile, so erhalten wir seine Zeilenposition über dieses Feld, das wir zeilennr nennen werden. Zusätzlich zu der realen Zeilenposition benötigen wir noch die Position bezüglich des Fensters, damit wir auf das Feld zeilennr zugreifen können. Damit wir die Mühe nicht ohne Grund auf uns nehmen, werden wir auch Folding bereits implementieren. Dies hat den Vorteil, daß wir die Funktionen dahingehend testen können, ob sich diese auch mit Folding korrekt verhalten.

Das nächste Problem sind die Tabulatoren. Der Editor Z, der zum Aztec-C-Compiler mitgeliefert wird, handhabt Tabulatoren

so, daß der Cursor entsprechend viele Spalten überspringt, wenn er über Tabulatoren bewegt wird. Dadurch hängt die Cursor-Bewegung von dem Zeichen ab, auf dem sich der Cursor gerade befindet. Dies ist auch der Grund, warum sich der Cursor nicht hinter eine Zeile setzen läßt, sondern immer nur auf das letzte Zeichen der Zeile. Wir wollen den Cursor aber völlig frei positionierbar lassen, auch wenn uns das ein paar kleine Probleme bei der Eingabe von Zeichen bringt. (Was passiert, wenn man einen Tabulator mit einem anderen Zeichen überschreibt? Wird der restliche Text nach links gerückt?) Dies bedeutet aber, daß die horizontale Positionierung des Cursors relativ simpel wird.

Während es mit dem vertikalen Scrolling keine großen Probleme geben wird, sieht dies beim horizontalen Scrolling schon ein klein wenig anders aus. Wenn der Inhalt des Fensters nach rechts gescrollt wird, so müssen wir den linken Rand neu ausgeben, wofür sich unsere Funktion printAll eignet. Wir müssen ihr nur vormachen, daß das Fenster lediglich ein Zeichen breit ist. Dabei tritt das Problem auf, daß das letzte Zeichen bei der Ausgabe immer entweder ein Blank oder ein Punkt ist, welcher dann davon zeugt, daß die Zeile breiter als das Fenster ist (Variable lastchar in der Funktion convertLineForPrint). Das wiederum bedeutet, daß, wenn wir nur um eine Spalte scrollen, lauter Punkte in der ersten Spalte ausgegeben werden würden. Wir müssen an dieser Stelle zugeben, daß wir dieses Problem auch erst beim Ausprobieren bemerkt haben. Wir werden es umgehen, indem wir eine Spalte mehr konvertieren lassen, als wir eigentlich bräuchten, und dann bei der Ausgabe einfach das letzte Zeichen vergessen. Ein weiteres Problemchen besehtt darin, daß wir auch den rechten Rand neu ausgeben lassen müssen, da das letzte Zeichen, wie bereits erwähnt, ja immer ein Blank oder ein Punkt sein muß. Verschieben wir den Fensterinhalt aber nach rechts, so stehen in der letzten Spalte irgendwelche Zeichen, die wir überschreiben müssen.

Die Neuausgabe des rechten Randes tritt auch dann auf, wenn wir nach links scrollen. Wenn wir auch dazu die Funktion printAll nehmen wollen, so müssen wir dem Editor nicht nur vorgaukeln, daß das Fenster nur ein (bzw. zwei) Zeichen breit ist, sondern auch, daß es einen sehr breiten linken Rand hat,

damit unser Text wirklich am rechten Rand erscheint. Wir müssen dabei eine Spalte mehr ausgeben als gescrollt wurde, da die letzte Spalte ja eh nur Blanks und Punkte enthielt, die wir wieder überschreiben müssen.

Dann könnten wir doch gleich eine neue Funktion schreiben, die eine Ausgabe ermöglicht, bei der die erwähnten Probleme nicht auftreten, meinen Sie? Klar, das könnten wir schon. Nur, wenn wir die Ausgabe erweitern, so daß beispielsweise C-Befehle fett gedruckt werden, so müßten wir diese Änderung in beiden Ausgabefunktionen vornehmen. Wenn wir dann vielleicht noch ein paar Funktionen zur Ausgabe einbauen, so verlieren wir bald die Übersicht, wo überall Änderungen vorzunehmen sind. Solange wir unsere bisherigen Funktionen nicht übermäßig verkomplizieren, wollen wir uns daher auf diese beschränken und sie den Umständen entsprechend erweitern.

Was das Scrolling anbelangt, so sind wir mit unseren Überlegungen nun fertig, und wir können uns dem Folding zuwenden. Als provisorische Faltenanfangsmarkierung definieren wir eine Zeile, die mit "/*#FOLD:" anfängt, während das Faltenende durch eine Zeile markiert wird, die mit "/*#ENDFD" beginnt. Beide Zeichenfolgen müssen in der momentanen Programmversion der Einfachheit halber direkt am Anfang der Zeile stehen, also ohne vorangestellte Blanks. Sie haben sicherlich erkannt, daß die Markierungen mit den beiden Zeichen /* beginnen, die in C einen Kommantar einleiten. Dies ist notwendig, da der C-Compiler diese Zeilen überlesen muß. Später werden wir die Markierungen variabel machen, so daß auch die Programmierer anderer Sprachen die Faltentechnik nutzen können.

Damit Sie erkennen, was in einer Falte liegt, können Sie hinter das /*#FOLD: einen Kommentar setzen, aus dem hervorgeht, welchen Zweck der Programmtext hat, der in der Falte liegt. Ist eine Falte verdeckt, so sehen Sie von dieser nur die Faltenanfangsmarkierung, also das /*#FOLD: mit dem Kommentar. Um sich nun den Inhalt dieser Falte ansehen zu können, bewegen Sie den Cursor auf diese Zeile und drücken Ctrl-F (F wie Falte). Nun sehen Sie nur noch das, was in der Falte liegt. Da Sie Falten beliebig verschachteln können, müssen wir dies bei der Aus-

gabe berücksichtigen. Zuerst definieren wir den Level jeder Zeile, also die Tiefe der Falte, in der die Zeile liegt. Die erste Zeile im Text erhält den Level Null. Nach einer Faltenanfangsmarkierung wird der Level um Eins erhöht, und nach einer Faltenendemarkierung dementsprechend um eins vermindert. Im folgenden Beispiel stehen die Falten-Level jeweils ganz links, und die Zeilen sind entsprechend ihrem Falten-Level eingerückt:

```
0   Zeile 1
0   /*#FOLD: Zeile 2 */
1      Zeile 3
1      /*#FOLD: Zeile 4 */
2         Zeile 5
2         /*#ENDFD Zeile 6*/
1      Zeile 7
1      /*#ENDFD Zeile 8 */
0   Zeile 9
0   /*#FOLD: Zeile 10 */
1      /*#FOLD: Zeile 11 */
2         Zeile 12
2         /*#ENDFD Zeile 13 */
1      Zeile 14
1      /*#ENDFD Zeile 15 */
0   Zeile 16
```

In unserer Editor-Struktur definieren wir zwei Variablen, minfold und maxfold. Im Fenster werden nur die Zeilen ausgegeben, deren Level zwischen minfold und maxfold liegt. Auch läßt sich der Cursor nur innerhalb dieser Zeilen bewegen, so daß Sie mit dem Cursor eine Falte nicht verlassen können, es sei denn, Sie verlassen diese Falte explizit, indem Sie Ctrl-E (E wie Exit) drücken. Da es aber sinnvoll sein kann, daß man mehrere Level gleichzeitig sieht, wirkt Ctrl-F auf zweierlei Weise:

- Befindet sich der Cursor auf einer Faltenanfangsmarkierung, so werden minfold und maxfold auf den Level der folgenden Zeile gesetzt, so daß Sie nur noch die entsprechende Falte sehen.
- Andernfalls wird nur maxfold um eins erhöht, so daß Sie zusätzlich zur momentanen Falte auch noch alle Falten sehen, deren Level eins höher liegen; Sie falten den Text also quasi wieder auseinander.

Ctrl-E dagegen wirkt immer gleich: Es vermindert zuerst maxfold um eins. Wenn dann minfold größer als maxfold ist, so wird

es auf maxfold gesetzt. Damit haben Sie alle Möglichkeiten, sich den Text anzusehen. Anhand des Beispieltextes sieht dies wie folgt aus:

- Am Anfang sind minfold und maxfold Null, so daß sich Ihnen der Text so darstellt:

```
0   Zeile 1
0   /*#FOLD: Zeile 2 */
0   Zeile 9
0   /*#FOLD: Zeile 10 */
0   Zeile 16
```

- Sie sehen nur die Zeilen, deren Falten-Level zwischen minfold und maxfold liegen, also null sind. Drücken Sie nun Ctrl-F, während der Cursor auf Zeile 1 steht, so wird der gesamte Text aufgefaltet, und Sie sehen folgendes:

```
0   Zeile 1
0   /*#FOLD: Zeile 2 */
1      Zeile 3
1      /*#FOLD: Zeile 4 */
1      Zeile 7
1      /*#ENDFD Zeile 8 */
0   Zeile 9
0   /*#FOLD: Zeile 10 */
1      /*#FOLD: Zeile 11 */
1      Zeile 14
1      /*#ENDFD Zeile 15 */
0   Zeile 16
```

- Nochmaliges Drücken von Ctrl-F würde den gesamten Text sichtbar machen, da der maximale Falten-Level aller Zeilen zwei ist. Drücken Sie nun zuerst Ctrl-E, damit Sie wieder nur alle Zeilen mit dem Falten-Level null sehen, bewegen Sie den Cursor auf Zeile 2, und drücken Sie Ctrl-F. Sie treten damit in die Falte hinein und sehen nur noch deren Inhalt:

```
1      Zeile 3
1      /*#FOLD: Zeile 4 */
1      Zeile 7
1      /*#ENDFD Zeile 8 */
```

Nun wirkt aber ein aufgefalteter Text eher unübersichtlich, denn woran erkennen Sie, ob die zu einer Faltenanfangsmarkierung gehörende Falte bereits auf dem Bildschirm zu sehen ist oder nicht? Es wirkt sehr verwirrend, wenn man maxfold auf den

maximalen Zeilen-Level gesetzt hat, während minfold auf Null geblieben ist, und man dann dauernd Faltenmarkierungen im Text sieht, obwohl gar keine Falten mehr verdeckt sind! Aus diesem Grund werden wir alle Faltenendemarkierungen und die Faltenanfangsmarkierungen in schwarzer Schrift ausgeben, die zu Falten gehören, deren Inhalt man auf dem Bildschirm sehen kann. Wenn man dann auf eine Faltenanfangsmarkierung in normaler (weißer) Farbe stößt, so kann man sicher sein, daß man in diese Falte noch eintreten kann.

Doch kommen wir nun zur Realisation: Was die Cursor-Bewegung anbelangt, so brauchen wir vier Funktionen, die den Cursor in jede Richtung bewegen können:

```
cursorRight, cursorLeft, cursorUp und cursorDown.
```

Diese benötigen keine Parameter und geben TRUE zurück, wenn der Cursor wirklich bewegt wurde, und FALSE, falls nicht, der Cursor also zum Beispiel bereits am Zeilenende war. Im Moment haben wir zwar noch keine Verwendung für diesen Rückgabewert, aber wenn wir den Editor programmierbar machen, kann es sinnvoll sein zu wissen, ob der Cursor wirklich bewegt worden ist. Dies kann man dann z.B. als Abbruchkriterium für Schleifen verwenden. Diese Funktionen zur Cursor-Bewegung rufen gegebenenfalls Funktionen auf, die den Inhalt des Editorfensters in die entsprechende Richtung scrollen. Diese Funktionen heißen analog:

```
scrollLeft, scrollRight, scrollDown und scrollUp.
```

Es ist dabei zu beachten, daß cursorRight die Funktion scrollLeft aufruft; dies gilt für die anderen Funktionen analog. Als Parameter wird nur die Anzahl der Zeilen bzw. Spalten übergeben, um die gescrollt werden soll. Der Parametertyp ist UWORD. Alle Scroll-Funktionen sind vom Typ void; sie liefern also kein Ergebnis zurück.

Aufrufen müssen wir diese Funktionen auch noch, und zwar nachdem die korrespondierende Taste gedrückt worden ist. Wir benötigen also eine Funktion, an die wir die Zeichen übergeben, die uns der Tastaturtreiber geschickt hat, und die für uns die

Cursor-Funktionen aufruft. In dieser Funktion muß auch das Ein- und Ausfalten behandelt werden, da dies ja auch über die Tastatur erfolgt:

```
void handleKeys(buf,len)
UBYTE          *buf;
UWORD                len;
```

Die ganze Zeit haben wir nun von der Cursor-Bewegung gesprochen, sind aber noch nicht auf die Idee gekommen, den Cursor auch anzeigen zu lassen. Dies wird uns die Funktion Cursor abnehmen, die den Cursor im COMPLEMENT-Modus ausgibt. Wenn wir diese Funktion einmal vor dem Wait-Befehl in unserer Hauptschleife und einmal danach aufrufen, so sehen wir den Cursor immer dann, wenn unser Programm nichts zu tun hat; andernfalls ist der Cursor verschwunden und kann somit nicht von irgendwelchen Operationen beeinträchtigt werden.

```
void Cursor()
```

Damit hätten wir mal wieder alle wichtigen Funktionen beisammen für unser neues Modul, so daß wir direkt an die Implementierung gehen können, nachdem wir unsere Datei Editor.h entsprechend erweitert haben:

```
#define FOLDPEN 2L

#define ZEILENPTR(n) aktuellerEditor->zeilenptr[n]
#define ZEILENNR(n) aktuellerEditor->zeilennr[n]

#define ZLF_FSE 64
#define ZLF_FOLD 63
```

FOLDPEN ist die Farbe, in der Faltenmarkierungen ausgegeben werden, also normalerweise Schwarz. Die beiden Makros ZEILENPTR und ZEILENNR sollen lediglich den Zugriff auf Elemente des entsprechenden Feldes vereinfachen. Statt:

```
aktuellerEditor->zeilenptr[n]
```

schreiben Sie einfach:

```
ZEILENPTR(n)
```

Die beiden letzten Defines gehören zu der Datenstruktur Zeile, genauer zu den Flags in der Zeile-Struktur. Wenn das Flag ZLF_FSE gesetzt ist, so handelt es sich bei der Zeile um eine Faltenmarkierung, wobei egal ist, ob es sich um eine Faltenanfangs- oder Faltenendemarkierung handelt (FSE = Fold Start-End). ZLF_FOLD dagegen gibt den Falten-Level der Zeile an, also Null für alle Zeilen, die zu keiner Falte gehören. Beachten Sie dabei bitte, daß die Faltenanfangsmarkierung selbst nicht mit zur Falte gehört! Der maximale Falten-Level ist in unserem Fall 63, was sicher mehr als großzügig ist.

Unsere Editor-Struktur wurde ebenfalls erweitert, so daß wir diese hier nochmal ganz abdrucken:

```
struct Editor
{
 struct Editor *succ;
 struct Editor *pred;
 struct Window *window;
 struct BList block;
 struct ZList zeilen;
 UBYTE puffer[MAXBREITE];
 UBYTE tabstring[MAXBREITE];
 UWORD anz_zeilen;
 struct Zeile *aktuell;
 struct Zeile *zeilenptr[MAXHOEHE];
 UWORD zeilennr[MAXHOEHE];
 UWORD toppos,leftpos;
 UWORD xpos,ypos;
 UWORD wdy;
 UWORD changed:1;
 UWORD insert:1;
 struct RastPort *rp;
 UWORD xoff,yoff;
 UWORD xscr,yscr;
 UWORD cw,ch;
 UWORD wcw,wch;
 UWORD xrl,xrr,yru,bl;
 UWORD minfold;
 UWORD maxfold;
};
```

Neu sind dabei folgende Elemente:

zeilennr[ ]

Enthält die Nummern aller Zeilen, die im Fenster sichtbar sind (siehe auch zeilenptr).

leftpos

Wenn im Fenster nach links gescrollt wird, so entspricht die erste Spalte des Fensters nicht mehr der ersten Spalte des Textes, sondern einer der folgenden Spalten. leftpos gibt die Nummer der Spalte an, die direkt links vom Fenster liegt. Normalerweise ist leftpos also null, was bedeutet, daß die erste Spalte, die im Fenster sichtbar ist, die Nummer Eins trägt.

wdy

Ist die bereits erwähnte Zeilenposition des Cursors, bezogen auf das Fenster. Dieser Wert liegt zwischen null und (aktuellerEditor->wch -1).

xscr, yscr

Diese beiden Variablen geben den rechten/unteren Rand des Fensters an, der für das Scrollen benötigt wird. In der Abbildung zur Textausgabe entspricht xscr = (xrl -1) und yscr = (y - 1) (jeweils in Pixeln gemessen).

minfold, maxfold

Geben den minimalen und maximalen Falten-Level an, der bestimmt, welche Falten auf dem Bildschirm erscheinen und welche nicht.

Nach dieser Vorarbeit wollen wir jetzt das neue Modul Cursor implementieren, das alle zuvor beschriebenen Funktionen zum Cursor beinhaltet:

```
<src/Cursor.c>

/*************
 *
 * Includes:
 *
 *************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"
```

```
/***********
 *
 * Defines:
 *
 ***********/

#define CSI 0x9B
#define CUU 'A'
#define CUD 'B'
#define CUF 'C'
#define CUB 'D'
#define SU 'S'
#define SD 'T'

#define CFOLD 6
#define CENDF 5

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

void SetDrMd(),RectFill(),printAll();
struct Zeile *nextLine(),*prevLine();

/*********************
 *
 * Externe Variablen:
 *
 *********************/

extern struct Editor *aktuellerEditor;
extern UWORD spaltenDec;

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

/************************************
 *
 * restoreZeilenptr:
 *
 * Restauriert das zeilenptr/nr-Feld
 * fuer alle Zeilen im Fenster plus
 * der naechsten Zeile.
 * Die erste Zeile muss allerdings
 * korrekt sein!
 *
 ************************************/
```

```
void restoreZeilenptr()
{
   register UWORD n;
   register struct Zeile **zptr,*z;
   register UWORD *pnr;
   UWORD znr;

   for (n = aktuellerEditor->wch,
                  zptr = aktuellerEditor->zeilenptr,
        pnr = aktuellerEditor->zeilennr,
        z = *zptr++, znr = *pnr++; n; n--)
   {
      *zptr++ = (z = nextLine(z,&znr));
      *pnr++ = znr;
   }
}

/**********************************
 *
 * Cursor:
 *
 * Setzt bzw. loescht den Cursor an
 * der aktuellen Position.
 *
 **********************************/

void Cursor()
{
   register struct RastPort *rp;
   register LONG x,y;
   register LONG cw,ch;

   SetDrMd(rp = aktuellerEditor->rp,COMPLEMENT);

   x = (aktuellerEditor->xpos - aktuellerEditor->leftpos - 1)
      *(cw = aktuellerEditor->cw) + aktuellerEditor->xoff;
   y = aktuellerEditor->wdy
      *(ch = aktuellerEditor->ch) + aktuellerEditor->yoff;

   RectFill(rp,x,y,x + cw-1,y + ch-1);

   SetDrMd(rp,JAM2);
}

/*******************************
 *
 * scrollRight(anz):
 *
 * Scrollt den Fensterinhalt um
 * anz Zeichen nach rechts.
 *
 * anz = Anzahl der Zeichen.
 *
 *******************************/
```

```
void scrollRight(anz)
register UWORD   anz;
{
   register UWORD y,ch,count;
   register struct Zeile **z;
   UWORD wcw,xoff,leftpos;

   aktuellerEditor->leftpos -= anz;

   if (anz >= aktuellerEditor->wcw)
      printAll();
   else
   {
      /* Scrollen: */
      ScrollRaster(aktuellerEditor->rp,
            -anz * (LONG)aktuellerEditor->cw,0L,
            (LONG)aktuellerEditor->xoff,
                                 (LONG)aktuellerEditor->yoff,
            (LONG)aktuellerEditor->xscr,
                                 (LONG)aktuellerEditor->yscr);

      /* nun printAll() ueberlisten: */
      wcw = aktuellerEditor->wcw;
      aktuellerEditor->wcw = anz + 1;
      spaltenDec = 1;

      y  = aktuellerEditor->yoff;
      ch  = aktuellerEditor->ch;
      count = aktuellerEditor->wch;

      /* Zeilen ausgeben: */
      for (z = aktuellerEditor->zeilenptr; count--
                                       && (*z != NULL);
           z++, y += ch)
         printLine(*z,y);

      spaltenDec = 0;

      /* Rechten Rand nochmals ausgeben, wegen letztem Zeichen */
      leftpos = aktuellerEditor->leftpos;
      xoff  = aktuellerEditor->xoff;
      aktuellerEditor->wcw    = 2;
      aktuellerEditor->leftpos += (y = wcw - 2);
      aktuellerEditor->xoff  += aktuellerEditor->cw * y;

      y  = aktuellerEditor->yoff;
      count = aktuellerEditor->wch;

      /* Zeilen ausgeben: */
      for (z = aktuellerEditor->zeilenptr;
                              count-- && (*z != NULL);
           z++, y += ch)
         printLine(*z,y);
      aktuellerEditor->wcw  = wcw;
```

```
      aktuellerEditor->leftpos= leftpos;
      aktuellerEditor->xoff = xoff;
   }
}

/*******************************
 *
 * scrollLeft(anz):
 *
 * Scrollt den Fensterinhalt um
 * anz Zeichen nach links.
 *
 * anz = Anzahl der Zeichen.
 *
 *******************************/

void scrollLeft(anz)
register UWORD  anz;
{
   register UWORD y,ch,count;
   register struct Zeile **z;
   UWORD wcw,xoff,leftpos;

   aktuellerEditor->leftpos += anz;

   if (anz >= aktuellerEditor->wcw)
      printAll();
   else
   {
      /* Scrollen: */
      ScrollRaster(aktuellerEditor->rp,
             anz * (LONG)aktuellerEditor->cw,0L,
             (LONG)aktuellerEditor->xoff,
                                  (LONG)aktuellerEditor->yoff,
             (LONG)aktuellerEditor->xscr,
                                  (LONG)aktuellerEditor->yscr);

      /* nun printAll() ueberlisten: */
      wcw  = aktuellerEditor->wcw;
      leftpos = aktuellerEditor->leftpos;
      xoff  = aktuellerEditor->xoff;
      aktuellerEditor->wcw    = anz + 1;
      aktuellerEditor->leftpos += (y = wcw - anz - 1);
      aktuellerEditor->xoff  += aktuellerEditor->cw * y;

      y  = aktuellerEditor->yoff;
      ch  = aktuellerEditor->ch;
      count = aktuellerEditor->wch;

      /* Zeilen ausgeben: */
      for (z = aktuellerEditor->zeilenptr;
                              count-- && (*z != NULL);
          z++, y += ch)
         printLine(*z,y);
```

```
      aktuellerEditor->wcw  = wcw;
      aktuellerEditor->leftpos= leftpos;
      aktuellerEditor->xoff = xoff;
   }
}

/*******************************
 *
 * scrollUp(anz):
 *
 * Scrollt den Fensterinhalt um
 * anz Zeilen nach oben.
 * Es muessen aber noch anz
 * Zeilen existieren!
 *
 * anz = Anzahl der Zeilen.
 *
 *******************************/

void scrollUp (anz)
register UWORD anz;
{
   register struct Zeile *z,**zptr,**zold;
   register UWORD n,wch = aktuellerEditor->wch,y;
   UWORD znr,*pnr,*oldnr;

   if (anz >= wch)
   {
      /* Fenster neu ausgeben, ohne zu Scrollen */
      /* Zuerst die oberste Zeile suchen: */
      n = wch;
      if (anz > wch)
      {
         z = ZEILENPTR(n);
         znr = ZEILENNR(n);
         while (++n < anz)
            z = nextLine(z,&znr);
      }
      else
      {
         z = ZEILENPTR(n - 1);
         znr = ZEILENNR(n - 1);
      }

      /* z zeigt nun vor die oberste Zeile des Fensters */
      for (n = 0, zptr = aktuellerEditor->zeilenptr,
           pnr = aktuellerEditor->zeilennr; n <= wch; n++)
      {
         *zptr++ = (z = nextLine(z,&znr));
         *pnr++ = znr;
            }
```

```
      /* Nun noch Fenster neu ausgeben: */
      printAll();
   }
   else
   {
      /* Scrollen: */
      ScrollRaster(aktuellerEditor->rp,
            0L,anz * (LONG)aktuellerEditor->ch,
            (LONG)aktuellerEditor->xoff,
                                 (LONG)aktuellerEditor->yoff,
            (LONG)aktuellerEditor->xscr,
                                 (LONG)aktuellerEditor->yscr);

      /* zeilenptr/nr neu berechnen: */
      zptr = aktuellerEditor->zeilenptr;
      zold = &(ZEILENPTR(anz));
      pnr = aktuellerEditor->zeilennr;
      oldnr = &(ZEILENNR(anz));
      for (n = 0; n <= wch - anz; n++)
      {
         *zptr++ = *zold++;
         *pnr++  = *oldnr++;
      }

      z = *--zold;
      zold = zptr - 1;   /* merken fuer Ausgabe! */
      znr = *--oldnr;
      oldnr = pnr - 1;
      while (n <= wch)
      {
         *zptr++ = (z = nextLine(z,&znr));
         *pnr++ = znr;
         n++;
      }

      /* Zeilen neu ausgeben: */
      for (n=anz, y=aktuellerEditor->
                              yoff+(wch-anz)*aktuellerEditor->ch;
          (n && (*zold != NULL)); n--, zold++,
                              y += aktuellerEditor->ch)
         printLine(*zold,y);
   }

   aktuellerEditor->toppos += anz;
}

/*******************************
 *
 * scrollDown(anz):
 *
 * Scrollt den Fensterinhalt um
 * anz Zeilen nach unten.
 * Es muessen aber noch anz
 * Zeilen existieren!
```

```
 *
 * anz = Anzahl der Zeilen.
 *
 ******************************/

void scrollDown (anz)
register UWORD   anz;
{
   register struct Zeile *z,**zptr,**zold;
   register UWORD n,wch = aktuellerEditor->wch,y;
   UWORD znr,*pnr,*oldnr;

   if (anz >= wch)
   {
      /* Fenster neu ausgeben, ohne zu Scrollen */
      /* Zuerst rueckwaerts nach oberster Zeile suchen: */
      for (z = ZEILENPTR(0), znr = ZEILENNR(0),
                              n = anz - wch; n; n--)
         z = prevLine(z,&znr);

      /* z zeigt auf erste Zeile nach Fensterunterkante. */
      for (n = 0, zptr = &(ZEILENPTR(wch)),
                                    pnr = &(ZEILENNR(wch));
           n <= wch; n++)
      {
         *zptr-- = z;
         *pnr-- = znr;
         z = prevLine(z,&znr);
      }

      /* Nun noch Fenster neu ausgeben: */
      printAll();
   }
   else
   {
      /* Scrollen: */
      ScrollRaster(aktuellerEditor->rp,
            0L,-anz * (LONG)aktuellerEditor->ch,
            (LONG)aktuellerEditor->xoff,
                                 (LONG)aktuellerEditor->yoff,
            (LONG)aktuellerEditor->xscr,
                                 (LONG)aktuellerEditor->yscr);

      /* zeilenptr neu berechnen: */
      zptr = &(ZEILENPTR(wch));
      zold = &(ZEILENPTR(wch - anz));
      pnr = &(ZEILENNR(wch));
      oldnr = &(ZEILENNR(wch - anz));
      for (n = 0; n <= wch - anz; n++)
      {
         *zptr-- = *zold--;
         *pnr-- = *oldnr--;
      }
      for (z = *++zold, znr = *++oldnr, n = anz; n; n--)
```

```
      {
         *zptr-- = (z = prevLine(z,&znr));
         *pnr-- = znr;
      }

      /* Zeilen neu ausgeben: */
      for (n = anz, y = aktuellerEditor->yoff;
                                 (n && (*zold != NULL));
           n--, zold++, y += aktuellerEditor->ch)
         printLine(*zold,y);
   }

   aktuellerEditor->toppos -= anz;
}

/*****************************************
 *
 * cursorLeft:
 *
 * Bewegt den Cursor um eine Stelle
 * nach links. Gibt FALSE zurueck,
 * wenn Cursor bereits in erster Spalte.
 *
 *****************************************/

BOOL cursorLeft()
{
   if (aktuellerEditor->xpos > 1)
   {
      aktuellerEditor->xpos--;
      if (aktuellerEditor->xpos <= aktuellerEditor->leftpos)
         scrollRight((UWORD)1);

      return (TRUE);
   }
   else
      return (FALSE);
}

/*****************************************
 *
 * cursorRight:
 *
 * Bewegt den Cursor um eine Stelle
 * nach rechts. Gibt FALSE zurueck,
 * wenn Cursor bereits in letzter Spalte.
 *
 *****************************************/

BOOL cursorRight()
{
   if (aktuellerEditor->xpos < MAXBREITE)
   {
      aktuellerEditor->xpos++;
```

```
      if (aktuellerEditor->xpos
          >= (aktuellerEditor->leftpos + aktuellerEditor->wcw))
         scrollLeft((UWORD)1);

      return (TRUE);
   }
   else
      return (FALSE);
}

/*****************************************
 *
 * cursorUp:
 *
 * Bewegt Cursor um eine Zeile nach oben.
 * Gibt FALSE zurueck, falls Cursor
 * bereits in erster Zeile.
 *
 *****************************************/

BOOL cursorUp()
{
   UWORD hilf;

   if (aktuellerEditor->wdy)
      aktuellerEditor->wdy--;
   else
      if (prevLine(ZEILENPTR(0),&hilf))
         scrollDown((UWORD)1);
      else
         return (FALSE);

   aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);

   return (TRUE);
}

/*****************************************
 *
 * cursorDown:
 *
 * Bewegt Cursor um eine Zeile nach unten.
 * Gibt FALSE zurueck, falls Cursor
 * bereits in letzter Zeile.
 *
 *****************************************/

BOOL cursorDown()
{
   if (ZEILENPTR(aktuellerEditor->wdy + 1))
   {
      if (++aktuellerEditor->wdy >= aktuellerEditor->wch)
      {
         scrollUp((UWORD)1);
```

```
            aktuellerEditor->wdy--;
        }
        aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);

        return (TRUE);
    }
    else
        return (FALSE);
}

/*********************************
 *
 * halfPageUp:
 *
 * Bewegt den Cursor um eine halbe
 * Seite nach oben. Der Text wird
 * analog um eine halbe Seite nach
 * unten gescrollt.
 * Gibt FALSE zurueck, falls Cursor
 * bereits in erster Zeile.
 *
 *********************************/

BOOL halfPageUp()
{
    register UWORD max,n;
    register struct Zeile *z;
    UWORD hilf;

    max = aktuellerEditor->wch / 2;
    for (n = 0, z = ZEILENPTR(0); n < max; n++)
        if (!(z = prevLine(z,&hilf)))
            break;

    if (n)
    {
        scrollDown(n);
        aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);

        return (TRUE);
    }
    else
        return (FALSE);
}

/*********************************
 *
 * halfPageDown:
 *
 * Bewegt den Cursor um eine halbe
 * Seite nach unten. Der Text wird
 * analog um eine halbe Seite nach
 * oben gescrollt.
 * Gibt FALSE zurueck, falls Cursor
```

```
 * bereits in letzter Zeile.
 *
 *********************************/

BOOL halfPageDown()
{
   register UWORD max,n;
   register struct Zeile *z;
   UWORD hilf;

   max = aktuellerEditor->wch / 2;
   for (n = 0, z = ZEILENPTR(aktuellerEditor->wch - 1);
                                                n < max; n++)
      if (!(z = nextLine(z,&hilf)))
         break;

   if (n)
   {
      scrollUp(n);
      aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);

      return (TRUE);
   }
   else
      return (FALSE);
}

/*******************************
 *
 * handleKeys(buf,len):
 *
 * Behandelt Tastatur-Messages.
 *
 * buf ^ Puffer mit Zeichen.
 * len = Anzahl der Zeichen.
 *
 *******************************/

void handleKeys(buf,len)
register UBYTE *buf;
register WORD       len;
{
   register UBYTE first;
   register struct Zeile *z;

   while (len > 0)
   {
      first = *buf;
      buf++;
      len--;
      if ((first == CSI) && (len > 0))
      {
         len--;
         switch (*buf++)
```

```
   {
      case CUU:
         cursorUp();
         break;
      case CUD:
         cursorDown();
         break;
      case CUF:
         cursorRight();
         break;
      case CUB:
         cursorLeft();
         break;
      case SU:
         halfPageDown();
         break;
      case SD:
         halfPageUp();
         break;

      default:
         buf--;
         len++;
         break;
   }
}
else if ((first == CFOLD)&&(aktuellerEditor->
                                    maxfold < ZLF_FOLD))
{
   aktuellerEditor->maxfold++;
   if (z = ZEILENPTR(aktuellerEditor->wdy))
      if ((z = z->succ)->succ)
         /* Wenn es eine nachfolgende Zeile gibt: */
         if (((ZEILENPTR(aktuellerEditor->wdy)->
                                          flags & ZLF_FOLD)
             < (z->flags & ZLF_FOLD))
            && ((z->flags & ZLF_FOLD)==
                              aktuellerEditor->maxfold))
         {
            /* Wenn dieser Anfang einer neuen Falte ist: */
            aktuellerEditor->minfold =
                           aktuellerEditor->maxfold;
            ZEILENPTR(0) = z;
            ZEILENNR(0)  = ZEILENNR
                            (aktuellerEditor->wdy) + 1;
            aktuellerEditor->wdy = 0;
         }

   restoreZeilenptr();
   printAll();
   aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);
}
else if ((first == CENDF) && (aktuellerEditor->maxfold))
{
```

```
            aktuellerEditor->maxfold--;
            if (aktuellerEditor->minfold > aktuellerEditor->maxfold)
               aktuellerEditor->minfold = aktuellerEditor->maxfold;

            if (((ZEILENPTR(0)->flags & ZLF_FOLD)>
                                          aktuellerEditor->maxfold)
            || ((ZEILENPTR(0)->flags & ZLF_FOLD)<
                                          aktuellerEditor->minfold))
            {
               ZEILENPTR(0) = prevLine(ZEILENPTR(0),&(ZEILENNR(0)));
               aktuellerEditor->wdy = 0;
            }

            restoreZeilenptr();
            printAll();
            aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);
         }
         else
            putchar(first);
      }
   }
```

Und nun wieder ein paar aufmunternde Worte zu den Funktionen im einzelnen und ein paar Besonderheiten im speziellen:

- Die Defines bestimmen die Zeichen, über die die Cursor-Bewegung und das Falten ausgelöst werden. Die Bezeichnungen sind dem Amiga-DOS-Manual entnommen. Bei der Abfrage in handleKeys muß beachtet werden, daß die Zeichenfolgen für die Cursor-Bewegung zwei (bis drei) Zeichen lang sind: jeweils ein CSI gefolgt von dem in den Defines angegebenen Zeichen.

- Die Funktion restoreZeilenptr berechnet das zeilenptr- und das zeilennr-Feld neu, wobei sie davon ausgeht, daß die oberste Zeile korrekt ist. Diese Funktion wird benötigt, um nach dem Eintritt in eine Falte den Fensterinhalt neu ausgeben zu können.

- Die Scroll-Funktionen erlauben es, um mehr als eine Zeile bzw. ein Zeichen zu scrollen. Es darf sogar um mehr Zeilen bzw. Zeichen gescrollt werden, als auf den Bildschirm passen. Dadurch lassen sich die vertikalen Scroll-Funktionen später gut dafür verwenden, eine bestimmte Textstelle aufzusuchen.

- Statt die Funktion printAll in den Scroll-Funktionen zu verwenden, wird eine for-Schleife benutzt, weil printAll auch die Ränder löscht, was für das Scrollen nicht nötig ist und nur Zeit kosten würde.
- Bei scrollUp und scrollDown muß das zeilenptr-/nr-Feld neu berechnet werden. Um dabei nicht das ganze Feld neu berechnen zu müssen, werden die Inhalte, die nach dem Scrollen noch gebraucht werden, entsprechend verschoben, eben so, wie auch die Zeilen verschoben (gescrollt) worden sind.
- Bei cursorUp und cursorDown muß wegen des Folding getestet werden, ob die nächste bzw. die vorige Zeile überhaupt in dem momentanen Level existiert. Dies wird von den Funktionen nextLine und prevLine übernommen, die entsprechend erweitert werden müssen. So wird diesen Funktionen ein Zeiger auf die aktuelle Zeilennummer übergeben, die dann entsprechend erhöht bzw. vermindert wird.
- Um vertikales Scrolling um mehr als eine Zeile testen zu können, wurden die Funktionen halfPageUp und halfPageDown eingebaut, die den Text jeweils um eine halbe Fensterseite nach oben bzw. unten scrollen. Die Position des Cursors bleibt dabei unverändert, zumindest die Position bezüglich des Fensters.
- Wenn Ctrl-F gedrückt wird, so muß der Editor feststellen, ob er sich auf einer Faltenanfangsmarkierung befindet. Eine solche ist dadurch definiert, daß der Level der aktuellen Zeile kleiner als der Level der folgenden Zeile ist, vorausgesetzt, es gibt eine folgende Zeile. Die folgende Zeile wird hier nicht mit nextLine bestimmt, da wir die nächste Zeile im Text brauchen, ohne Beachtung der Faltung. Wenn dann noch der Level der folgenden Zeile gleich dem maximalen Falten-Level plus eins entspricht, so handelt es sich bei der aktuellen Zeile um eine Faltenanfangsmarkierung, da die nächste Zeile nicht sichtbar ist.
- Beim Austritt aus einer Falte muß sichergestellt sein, daß die oberste Zeile nicht zu der Falte gehört, die gerade

verlassen wurde, bevor restoreZeilenptr aufgerufen wird. Andernfalls stimmt der Fensterinhalt nicht.

Alles klar? Na fein! Dann können wir ja die anderen Module entsprechend erweitern. Beginnen wollen wir mit dem Hauptmodul Editor, in dem zwei neue Funktionen deklariert werden müssen

```
void handleKeys(),Cursor();
```

Natürlich müssen die Funktionen nextLine und prevLine an das Folding angepaßt werden:

```
struct Zeile *nextLine(zeile,pnr)
register struct Zeile *zeile;
register UWORD              *pnr;
{
   register struct Zeile *z;
   register UWORD minfold = aktuellerEditor->minfold;
   register UWORD maxfold = aktuellerEditor->maxfold;

   if (z = zeile)
      if (z->succ)
         do
         {
            if ((z = z->succ)->succ)
               if ((z->flags & ZLF_FOLD) < minfold)
               {
                  /* Nachfolgende Zeile liegt ausserhalb Falte */
                  z = NULL;
                  break;
               }
               else
                  *pnr += 1;
            else
            {
               /* keine nachfolgende Zeile */
               z = NULL;
               break;
            }
         }
         while ((z->flags & ZLF_FOLD) > maxfold);
      else
         z = NULL;

   return (z);
}
```

Die Funktion prevLine sieht genauso aus, nur daß ->succ überall durch ->pred ersetzt wird:

```
struct Zeile *prevLine(zeile,pnr)
register struct Zeile *zeile;
register UWORD              *pnr;
{
   register struct Zeile *z;
   register UWORD minfold = aktuellerEditor->minfold;
   register UWORD maxfold = aktuellerEditor->maxfold;

   if (z = zeile)
      if (z->pred)
         do
         {
            if ((z = z->pred)->pred)
               if ((z->flags & ZLF_FOLD) < minfold)
               {
                  /* Vorige Zeile liegt außerhalb der Falte */
                  z = NULL;
                  break;
               }
               else
                  *pnr += 1;
            else
            {
               /* keine vorige Zeile */
               z = NULL;
               break;
            }
         }
         while ((z->flags & ZLF_FOLD) > maxfold);
      else
         z = NULL;

   return (z);
}
```

In der Funktion OpenEditor werden die zusätzlichen Elemente der Editor-Struktur initialisiert:

```
ed->leftpos = 0;
ed->wdy     = 0;
ed->minfold = 0;
ed->maxfold = 0;

/* zeilenptr/nr-Array initialisieren: */
for (n = 0, zptr = ed->zeilenptr, pnr = ed->zeilennr;
     n < MAXHOEHE; n++, zptr++, pnr++)
{
   *zptr = NULL;
   *pnr = 1;
}
```

Im Hauptprogramm ändern wir die Variablen: Statt SHORT 1 fügen wir zwei neue hinzu:

```
ULONG mouseX,mouseY;
```

Auch die Hauptschleife bekommt ein anderes Gesicht, weswegen wir diese nochmal komplett auflisten:

```
do
{
   /* Cursor setzen: */
   Cursor();

   signal = Wait(1L << edUserPort->mp_SigBit);

   /* Cursor wieder loeschen: */
   Cursor();

   while (imsg = GetMsg(edUserPort))
   {
      class     = imsg->Class;
      code      = imsg->Code;
      qualifier = imsg->Qualifier;
      iaddress  = imsg->IAddress;
      mouseX    = imsg->MouseX;
      mouseY    = imsg->MouseY;

      ReplyMsg(imsg);

      /* Event bearbeiten: */
      switch (class)
      {
         case RAWKEY:
            if (! (code & IECODE_UP_PREFIX))
            {
               inputEvent.ie_Code  = code;
               inputEvent.ie_Qualifier = qualifier;
               if ((inputLen = RawKeyConvert(
                    &inputEvent,inputPuffer,MAXINPUTLEN,NULL)
                   ) >= 0)
                  handleKeys(inputPuffer,inputLen);
            }
            break;

         case MOUSEBUTTONS:
            {
               register WORD x,y;

               if (mouseX <= aktuellerEditor->xoff)
                  x = 0;
               else
                  x = (mouseX - aktuellerEditor->xoff)
```

```
                    / aktuellerEditor->cw;
            if (++x >= aktuellerEditor->wcw)
               x = aktuellerEditor->wcw - 1;
            x += aktuellerEditor->leftpos;
            if (x > MAXBREITE)
               x = MAXBREITE;

            if (mouseY <= aktuellerEditor->yoff)
               y = 0;
            else
               y = (mouseY - aktuellerEditor->yoff)
                  / aktuellerEditor->ch;

            if ((y < aktuellerEditor->wch)
               && (aktuellerEditor->zeilenptr[y]))
            {
               aktuellerEditor->xpos= x;
               aktuellerEditor->wdy = y;
               aktuellerEditor->ypos=
                              aktuellerEditor->zeilennr[y];
            }
         }

      case NEWSIZE:
         initWindowSize(aktuellerEditor);

         /* Pruefe, ob Cursor noch im Fenster! */
         if (aktuellerEditor->xpos > aktuellerEditor->leftpos
                                    +aktuellerEditor->wcw)
            aktuellerEditor->xpos = aktuellerEditor->leftpos
                                  + aktuellerEditor->wcw;

         if (aktuellerEditor->wdy >= aktuellerEditor->wch)
            aktuellerEditor->ypos = aktuellerEditor->zeilennr
             [(aktuellerEditor->wdy  = aktuellerEditor->wch - 1)];

         break;

      case REFRESHWINDOW:
         BeginRefresh(aktuellerEditor->window);
         printAll();
         EndRefresh(aktuellerEditor->window,TRUE);
         break;

      case CLOSEWINDOW:
         running = FALSE;
         break;

      default:
         printf("Nicht bearbeitbarer Event: %lx\n",class);

    } /* of case */
  } /* of while (GetMsg()) */
} while (running);
```

Hinzugekommen ist außer den bereits besprochenen Dingen die Abfrage der MOUSEBUTTONS, so daß Sie den Cursor auch mit der Maus setzen können, wie Sie es von Textverarbeitungen gewöhnt sind. Während sich im Speicher-Modul nichts geändert hat, müssen wir im Ausgabe-Modul noch die Ausgabe der Faltenmarkierungen in einer anderen Farbe einbauen. Doch zuerst eine neue globale Variable, die wir für printLine benötigen, um die letzte Spalte nicht mit auszugeben:

```
/*********************
 * Globale Variablen:
 *********************/

UWORD spaltenDec = 0;
```

In der Funktion initWindowSize muß nun außer dem zeilenptr-Feld auch das zeilennr-Feld neu berechnet werden, wozu zwei zusätzliche Variablen definiert werden müssen:

```
UWORD znr,*pnr;
```

Die Berechnung des Feldes sieht nun wie folgt aus:

```
/* Nun ggf. zeilenptr-Feld restaurieren: */
if (newh != ed->wch)
{
   if (newh < ed->wch)
   {
      /* Ueberfluessige Zeiger auf Null setzen: */
      zptr = ed->zeilenptr + newh + 1;
      for (n = newh; n < ed->wch; n++)
         *zptr++ = NULL;
   }
   else
   {
      zptr = ed->zeilenptr + ed->wch;
      z    = *zptr++;
      pnr  = &(ed->zeilennr[ed->wch]);
      znr  = *pnr++;
      for (n = ed->wch; n < newh; n++)
      {
         *zptr++ = (z = nextLine(z,&znr));
         *pnr++  = znr;
      }
   }
   ed->wch = newh;
}
```

Auch die Funktion convertLineForPrint bleibt nicht von Änderungen verschont. Hier muß berücksichtigt werden, daß die Ausgabe nicht in der ersten Spalte beginnen kann, sondern in der Spalte, die durch leftpos bestimmt wird.

Wir machen es uns jetzt einfach und setzen einen Zähler (skip) auf leftpos, und jedesmal, wenn wir ein neues Zeichen nach buf schreiben wollen, testen wir zuerst, ob skip bereits null ist.

Ist dies der Fall, so wird das Zeichen geschrieben, andernfalls wird skip um eins vermindert. Wir überspringen damit die ersten skip-Zeichen:

```
register UWORD skip = aktuellerEditor->leftpos;
```

Die Bestimmung von lastchar bleibt gleich, aber die Konvertierung wird entsprechend geändert:

```
/* Zeile konvertieren: */

tab = aktuellerEditor->tabstring;
while ((len--) && (l < w))
   if ((*buf = *zeile++) == TAB)
      do
      {
         tab++;
         if (skip)
            skip--;
         else
         {
            l++;
            *buf++ = ' ';
         }
      } while ((*tab) && (l < w));
   else
   {
      tab++;
      if (skip)
         skip--;
      else
      {

         l++;
         buf++;

      }
   }
```

Der Rest der Funktion bleibt unverändert. Als nächstes betrachten wir die Funktion printAt, an die zusätzlich zu den bisherigen Parametern noch die Zeichenfarbe übergeben werden muß. Diese ist entweder FGPEN oder FOLDPEN, und sie ist deswegen nötig, weil sonst nach einem Blank die Schriftfarbe nicht mehr stimmt, wenn es sich bei der Zeile um eine Faltenmarkierung handelt. Denn für Blanks muß die Vordergrundfarbe geändert und danach wieder zurückgesetzt werden:

```
void printAt   (buf,len,x,y,fgpen)
register UBYTE *buf;
WORD                len;
register UWORD          x,y;
ULONG                       fgpen;
```

Dementsprechend wird überall in dieser Funktion

```
SetAPen(rp,FGPEN);
```

durch

```
SetAPen(rp,fgpen);
```

ersetzt. Die Funktion printLine wird nun noch so erweitert, daß sie Faltenmarkierungen erkennt und die Schriftfarbe dementsprechend setzt. Dabei muß aber am Funktionsende wieder auf FGPEN geschaltet werden, damit wir uns an jeder beliebigen Stelle im Programm darauf verlassen können, daß die Vordergrundfarbe FGPEN ist:

```
void printLine        (zeile,y)
register struct Zeile *zeile;
register UWORD                y;
{
   static UBYTE buf[MAXBREITE];
   register UWORD w;
   register struct Zeile *z;
   register ULONG pen;

   convertLineForPrint(zeile+1,zeile->len,w =
                              aktuellerEditor->wcw,buf);

   /* Falls Anfang/Ende-Markierung einer Falte => FOLDPEN */
   if (zeile->flags & ZLF_FSE)
   {
      if ((z = zeile->succ)->succ)
      {
```

```
            if ((z->flags & ZLF_FOLD) <= aktuellerEditor->maxfold)
            {
               SetAPen(aktuellerEditor->rp,FOLDPEN);
               pen = FOLDPEN;
            }
            else
               pen = FGPEN;
         }
         else
         {
            SetAPen(aktuellerEditor->rp,FOLDPEN);
            pen = FOLDPEN;
         }

         printAt(buf,w - spaltenDec,aktuellerEditor->xoff,y,pen);

         /* Alte Schreibfarbe wieder einstellen: */
         if (pen != FGPEN)
            SetAPen(aktuellerEditor->rp,FGPEN);
      }
      else
         printAt(buf,w - spaltenDec,aktuellerEditor->xoff,y,FGPEN);
}
```

So, das war alles bezüglich des Ausgabe-Moduls. Fehlen nur noch die Änderungen in der Testumgebung, und wir können mit dem Testen beginnen. So muß die Zeilennummer, die von der Funktion print ausgegeben wird, nun aus dem zeilennr-Feld geholt werden:

```
void print()
{
   register struct Zeile **z;
   register UWORD n,*pnr;

   for (n = 0, z = aktuellerEditor->zeilenptr,
        pnr = aktuellerEditor->zeilennr;
        n <= aktuellerEditor->wch; n++, z++, pnr++)
      printf("Zeile %d an Adresse %6lx.\n",*pnr,*z);
}
```

In den Funktonen input und loesche_zeile sollte die Anzahl der Zeilen entsprechend angepaßt werden. Bei input, wenn die Eingabe erfolgreich war, mittels:

```
aktuellerEditor->anz_zeilen++;
```

und bei loesche_zeile:

```
      aktuellerEditor->anz_zeilen--;
```

Die Funktion init_zeilenptr muß nun zusätzlich die Falten-Level aller Zeilen bestimmen:

```
void init_zeilenptr()
{
   register struct Zeile *z,**zptr;
   register UWORD n,*pnr,fold = 0;
   UWORD znr;

   for (z = aktuellerEditor->zeilen.head; z->succ; z = z->succ)
   {
      z->flags = ((z->flags & ~ZLF_FOLD) | (fold & ZLF_FOLD));
      if (z->len >= 8)
      {
         if ((*((ULONG *)(z+1))=='/*#F')&&
                                    (*((ULONG *)(z+1)+1)=='OLD:'))
         {
            fold++;
            z->flags |= ZLF_FSE;
         }
         if ((*((ULONG *)(z+1))=='/*#E')&&
                                    (*((ULONG *)(z+1)+1)=='NDFD'))
         {
            fold--;
            z->flags |= ZLF_FSE;
         }
      }
   }

   for (n = 0, z = aktuellerEditor->zeilen.head,
        zptr = aktuellerEditor->zeilenptr,
        pnr = aktuellerEditor->zeilennr, znr = 1;
        n < MAXHOEHE; n++)
   {
      *zptr++ = z;
      *pnr++ = znr;
      z = nextLine(z,&znr);
   }
}
```

Die Abfrage auf Faltenmarkierung sieht zwar etwas "wild" aus, funktioniert aber, und zwar schneller als mit der String-Compare-Funktion aus der Standard-Bibliothek, da immer vier Zeichen auf einmal miteinander verglichen werden (ULONG = 4 Bytes). Im Moment handelt es sich ja eh noch um unsere Testumgebung, da können wir uns solche Spielereien durchaus erlauben. Später, wenn wir beliebige Strings als Faltenmarkierung zulassen, werden wir die Funktion strcmp oder strncmp verwen-

den, da dann nicht mehr sichergestellt ist, daß die entsprechende Zeichenkette genau acht Zeichen lang ist.

Nachdem wir zu guter Letzt noch einen printAll-Aufruf an das Ende der Funktion Test gesetzt haben, der dafür sorgt, daß der Fensterinhalt stimmt, wenn wir die Testumgebung verlassen, sind wir mit unseren Umbauarbeiten fertig und können uns wieder dem Testen widmen. Im Verzeichnis V0.4 finden Sie diesmal nicht nur die Quelltexte und den compilierten Editor, sondern auch eine Testdatei namens TestText. Starten Sie den Editor mit:

```
Editor <TestText
```

Dann sparen Sie sich das mühselige Eingeben der einzelnen Zeilen. Wenn die Datei geladen ist, wird die Testumgebung automatisch verlassen, da die Testdatei mit dem ESCAPE-Zeichen endet, und Sie können den Editor ausgiebig ausprobieren, inklusive Folding, da im Testtext auch einige Falten definiert sind.

Nun sollten wir uns so langsam mit dem Debugger vertraut machen, damit wir ihn im Falle eines Falles auch optimal einsetzen können. Zu diesem Zweck folgende Übungsaufgabe: Es soll die tatsächliche Y-Position des Cursors ausgegeben werden, und zwar vom Debugger.

Zuerst compilieren wir den Editor erneut, aber diesmal, indem wir "make Debug" aufrufen. Die Make-Datei sorgt dann dafür, daß das fertige Programm auch Informationen über die Variablennamen, also eine sogenannte Symboltabelle enthält. Diese erleichtert uns das Auffinden von bestimmten Stellen des Programms ganz erheblich. Nach dem Compilieren starten wir den Debugger mit:

```
SYS3:bin/db
```

Dieser lädt eine Datei namens .dbinit, in der meist nur der Befehl al steht. Wartet der Debugger nach dem Starten nicht darauf, daß Sie einen neuen Task starten, was in der Infozeile des Debugger-Fensters angezeigt wird, so müssen Sie al eingeben. Dann starten Sie den Editor, wie Sie es zuvor zum Ausprobieren getan haben. Der Debugger merkt, daß ein neues Programm ge-

startet wurde, und fängt dieses ab. Dann überprüft er, ob das Programm auch eine Symboltabelle enthält und lädt diese gegebenenfalls.

Nachdem alles erfolgreich geladen worden ist, zeigt der Debugger den ersten Befehl an, in unserem Fall:

```
jmp .begin
```

Wir wollen den Debugger nun dazu bringen, die Y-Position des Cursors anzuzeigen. Da dies nur dann notwendig ist, wenn sich diese geändert hat, lassen wir diese anzeigen, nachdem der Benutzer den Cursor bewegt hat. Und wo wird der Cursor bewegt? In den Funktionen cursorUp und cursorDown. Zwar gibt es noch ein paar andere Stellen, aber fürs erste wollen wir uns mit diesen beiden bescheiden. Um die Y-Position ausgeben zu können, müssen wir wissen, an welcher Speicherposition diese liegt. Wir wissen, daß die Y-Position Element der Editor-Struktur ist, und wir wissen, daß sie von den Funktionen cursorUp und cursorDown geändert wird. Folglich sehen wir uns diese Funktionen genauer an. Geben Sie dem Debugger folgende Anweisung:

```
u cursorUp
```

Der Debugger zeigt nun den Speicher in disassemblierter Form ab der Adresse cursorUp an, also ab dem Anfang unserer Funktion cursorUp. Bei uns sah dies wie folgt aus:

```
_cursorUp       link    a5,#-2
_cursorUp+4     movea.l _aktuellerEditor,a0
_cursorUp+8     tst.w   252(a0)
_cursorUp+c     beq.s   _cursorUp+18
_cursorUp+e     movea.l _aktuellerEditor,a0
_cursorUp+12    subq.w  #1,252(a0)
_cursorUp+16    bra.s   _cursorUp+40
_cursorUp+18    pea     -2(a5)
_cursorUp+1c    movea.l _aktuellerEditor,a0
_cursorUp+20    move.l  ca(a0),-(a7)
_cursorUp+24    jsr     _prevLine
_cursorUp+28    addq.w  #8,a7
_cursorUp+2a    tst.l   d0
_cursorUp+2c    beq.s   _cursorUp+3a
_cursorUp+2e    move.w  #1,-(a7)
_cursorUp+32    jsr     _scrollDown
_cursorUp+36    addq.w  #2,a7
_cursorUp+38    bra.s   _cursorUp+40
```

```
_cursorUp+3a        moveq   #0,d0
_cursorUp+3c        unlk    a5
_cursorUp+3e        rts
_cursorUp+40        movea.l _aktuellerEditor,a0
_cursorUp+44        moveq   #0,d0
_cursorUp+46        move.w  252(a0),d0
_cursorUp+4a        asl.l   #1,d0
_cursorUp+4c        movea.l d0,a1
_cursorUp+4e        adda.l  _aktuellerEditor,a1
_cursorUp+52        movea.l _aktuellerEditor,a6
_cursorUp+56        move.w  1ca(a1),250(a6)
_cursorUp+5c        moveq   #1,d0
_cursorUp+5e        bra.s   _cursorUp+3c
```

Sie wissen ja noch, daß die Y-Position aus dem Feld zeilennr geholt worden ist. Wir suchen also ein Programmstück, in dem zuerst auf ein Feld zugegriffen wird und dann ein Wert in ein Element der Editor-Struktur geschrieben wird. Dieses Stück finden wir ab Adresse _cursorUp+40: Zuerst wird nach A0 ein Zeiger auf die aktuelle Editor-Struktur gebracht. Dann wird D0 mit wdy geladen und dieses als Index für das zeilennr-Feld benutzt (deswegen die Multiplikation mit zwei (ASL)). Ab Adresse _cursorUp+56 wird dann der Wert aus dem Feld in die Variable ypos geschrieben. Das bedeutet, daß ypos 250 (hex) Bytes vom Anfang der Struktur entfernt ist.

Um herausfinden zu können, welche Variable in C durch welche Speicherstelle in Assembler repräsentiert wird, brauchen Sie allerdings etwas Erfahrung in Assembler. Eine andere Möglichkeit, an diese Werte zu kommen, besteht darin, in der Editor-Struktur die Elemente entsprechend ihres Platzbedarfs in Bytes abzuzählen. Nachdem wir nun wissen, wo ypos steckt, müssen wir uns eine Stelle überlegen, an die wir den Breakpoint setzen, der dafür sorgt, daß die Y-Position ausgegeben wird. Nachdem die Y-Position geändert worden ist springt, das Programm an die Adresse _cursorUp+3c. An dieser Adresse steht ein UNLK-Befehl, mit dem alle Unterprogramme in C enden (zumindest beim Aztec). An diese Adresse setzen wir auch den Breakpoint:

```
bs cursorUp+3c ;pd 250+*aktuellerEditor;g
```

Das pd bedeutet Print Decimal; der Debugger soll den Inhalt der angegebenen Speicherstelle dezimal ausgeben. Die Speicherstelle entspricht der Y-Position des aktuellen Editors, also 250 Bytes

hinter der Adresse, auf die aktuellerEditor zeigt. Das g bedeutet, daß der Debugger den Editor sofort wieder starten soll. Wir wollen den Editor ja nicht wirklich unterbrechen, sondern nur die Y-Position ausgegeben haben. Den nächsten Breakpoint setzen wir in das Unterprogramm cursorDown, und zwar auch wieder ans Ende. Geben Sie also:

```
u cursorDown
```

ein, und suchen Sie den UNLK-Befehl, indem Sie solange u eingeben, bis Sie ihn gefunden haben. In unserem Fall lag er an Adresse _cursorDown+66. Setzen Sie den Breakpoint wie gehabt:

```
bs cursorDown+66 ;pd 250+*aktuellerEditor;g
```

Damit wären unsere Vorbereitungen beendet, und wir können den Editor mittels g (Go) starten. Alles verläuft zunächst ganz normal... bis Sie CursorUp (also die Taste auf der Tastatur) drücken. Dann wird das Debugger-Fenster nach vorne geholt, aktiviert, und der Editor wird sofort wieder gestartet, was Sie daran sehen, daß der Cursor im Debugger-Fenster nicht hinter einem Fragezeichen steht. Die Sache hat nur einen Haken: Das Editorfenster ist nicht mehr vorne und, was schon tragischer ist, nicht mehr aktiviert!

Verkleinern Sie das Debugger-Fenster nun so, daß es das Editorfenster nicht mehr stört, aber noch groß genug ist, daß die Y-Position ausgegeben werden kann. Das Editorfenster vergrößern Sie analog soweit, daß man das Debugger-Fenster noch ganz sehen kann. Nun aktivieren Sie das Editorfenster wieder, indem Sie entweder den Fensterrahmen oder direkt den Cursor anklicken; in allen anderen Fällen würde ja der Cursor an eine andere Position gesetzt werden.

Sie müssen nun jedesmal, nachdem Sie CursorUp oder CursorDown gedrückt haben, das Editorfenster wieder aktivieren, mit anderen Worten: reaktivieren. Das ist zwar unschön, aber erträglich. Bewegen Sie den Cursor nun über ein paar Zeilen, auch über Falten, und prüfen Sie, ob die angezeigte Position plausibel ist. Gehen Sie auch in Falten hinein, und scrollen Sie ein wenig. Wenn Sie davon genug haben, so gehen Sie wieder an den Text-

anfang, und prüfen Sie, ob dieser immer noch die Zeilennummer Eins hat. Sie werden dabei sehr wahrscheinlich feststellen, daß dies nicht der Fall ist, aber vielleicht ist Ihnen bereits aufgefallen, daß in der Berechnung der Y-Position noch ein Fehler steckt.

Überlegen wir uns dazu, bei welcher Gelegenheit der Fehler aufgetreten ist: Solange wir im Fenster blieben und in keine Falten eingetreten sind, hat alles noch tadellos funktioniert. Das ist kein Wunder, da die Zeilennummern in dem Feld zeilennr abgelegt waren. Erst als diese geändert werden mußten, trat der Fehler auf. Dies bedeutet zumindest, daß es an der Funktion nextLine, die die Zeilennummern berechnet hat, wahrscheinlich nicht liegt, da sie ja die Zeilennummern initialisiert hat. Nun gibt es zwei Möglichkeiten zur Fehlersuche: Entweder man schaut sich den Quelltext ein paarmal scharf an, oder man testet so lange alle Möglichkeiten aus, bis der Fehler soweit eingekreist ist, daß man schon ziemlich genau sagen kann, an welcher Stelle dieser liegt.

Wir werden es zuerst mit scharfem Hinsehen versuchen. Laden Sie dazu das Modul Cursor.c in einen Editor, und sehen Sie sich die Programmstelle an, an der das Folding bewerkstelligt wird. Sie werden vermutlich keinen Fehler entdecken können; zumindest konnten wir keinen finden, aber das will ja nichts heißen. Was haben wir noch beim Testen getan? Gescrollt! Sehen Sie sich also die Funktionen scrollUp und scrollDown an. Diese scheinen jedoch auch keinen Fehler zu enthalten. Bevor wir endlos testen, werfen wir noch einen Blick auf die Funktion prevLine, die von der Funktion scrollDown benutzt wird. Wenn Sie genau hinsehen, so bemerken Sie, daß in dieser Funktion die Zeilennummer ebenso wie in nextLine um Eins erhöht wird (*pnr++), statt um Eins vermindert zu werden (*pnr--).

Ändern Sie dies, speichern Sie die Datei ab und lassen den Editor erneut compilieren. Der Editor im Verzeichnis V0.4 hat übrigens auch diesen Fehler, damit Sie daran den Debugger ausprobieren können. Ist der Fehler behoben, können Sie den Editor erneut testen, so Sie Lust dazu haben.

4.3.8 Texteingabe

Im folgenden wollen wir den Editor einsatzfähig machen. Wir werden Funktionen einbauen, die es uns ermöglichen, einen Text zu editieren. Dazu brauchen wir Funktionen, die es uns erlauben, Zeichen einzugeben und zu löschen. Ferner müssen sich auch Zeilen einfügen und löschen lassen. Überlegen wir uns nun, wie diese Funktionen arbeiten sollen.

Vor dem Einfügen von Zeichen muß die entsprechende Zeile in den Puffer der Editor-Struktur kopiert werden, so sie sich noch nicht dort befindet. Dann werden alle Zeichen nach hinten geschoben, die unterhalb oder rechts vom Cursor stehen, und das neue Zeichen unterhalb des Cursors eingefügt. Allerdings gilt dies nur für den Einfügemodus. Befindet sich der Editor im Überschreibmodus, so wird das Zeichen einfach nur in den Text geschrieben. Das Zeichen, daß sich zuvor dort befand, wird dadurch gelöscht. Anschließend wird der Cursor noch um eine Stelle nach rechts bewegt. Dabei gilt es zu beachten, das keine Zeichen rechts aus der Zeile herausgeschoben werden. In einem solchen Fall wird das Zeichen nicht eingefügt, statt dessen lassen wir den Bildschirm zur Warnung einmal blinken.

Eine Sonderstellung beim Einfügen nehmen die Sonderzeichen ein, namentlich CR (Carriage Return = 13), LF (Line Feed = 10), BS (Back Space = 8), DEL (Delete = 127) und TAB (Tabulator = 9). Bei CR und LF muß die Zeile an der Cursor-Position aufgesplittet und eine neue Zeile eingefügt werden. Bei BS und DEL werden dagegen Zeichen gelöscht, wobei bei BS auch noch berücksichtigt werden muß, ob sich der Editor im Einfüge- oder Überschreibmodus befindet (aktuellerEditor->insert), ebenso wie beim Einfügen von Zeichen. Bei TAB werden, abhängig von der Cursor-Position, gleich mehrere Zeichen eingefügt.

Überhaupt müssen wir uns bei Tabulatoren so langsam darauf einigen, wie wir sie behandeln. Bei der Cursor-Positionierung wollten wir uns ja nicht von diesen stören lassen, also liegt es nahe, daß wir dies auch nicht beim Editieren tun. Stellt sich aber die Frage, wie wir unseren Text mit Tabulatoren formatieren. Da wir mit unserem Editor vorwiegend Programmtexte ein-

geben wollen, bei denen es meist egal ist, ob und wie wir diese formatieren (der Aztec überliest Tabulatoren genauso wie Blanks), können wir folgendermaßen vorgehen:

- Wenn wir eine Zeile in den Puffer kopieren, wandeln wir alle Tabulatoren in eine entsprechende Anzahl von Blanks um, so wie wir es bereits bei der Funktion convertLine-ForPrint getan haben.
- Soll die Zeile wieder aus dem Puffer zurück in die Zeile geschrieben werden, so wandeln wir alle Blanks wieder in Tabulatoren um, soweit dies möglich ist.

Somit haben wir stets eine Formatierung mit möglichst geringem Platzbedarf. Der einzige Nachteil dieser Methode ist, daß wir Tabulatoren nicht gezielt bzw. nicht mehrere Blanks hintereinander setzen können, die wirklich auch als Blanks abgespeichert werden statt als Tabulatoren. Diesem können wir abhelfen, indem wir ein Flag namens aktuellerEditor->tabs einführen. Ist dieses auf Eins gesetzt, so werden Blanks möglichst optimal in Tabulatoren umgewandelt, andernfalls werden Tabulatoren bei der Aus- und Eingabe als ganz normale Sonderzeichen behandelt (Ctrl-I), und es wird keine Konvertierung vorgenommen.

Nun müssen wir den Puffer irgendwann wieder in die Zeile-Struktur zurückschreiben. Dies tun wir genau dann, wenn wir mit dem Cursor die aktuelle Zeile verlassen. Zu diesem Zweck schreiben wir eine Funktion, die aus der Hauptschleife des Editors jedesmal aufgerufen wird, nachdem eine Message bearbeitet wurde, und die den Pufferinhalt in die Zeile zurückschreibt, falls der Cursor diese Zeile verlassen hat. Wir brauchen also noch eine Variable namens pufypos, die die Zeilennummer der Zeile enthält, die sich gerade im Puffer befindet, bzw. die Null ist, falls sich keine Zeile im Puffer befindet. Auf die Zeile-Struktur der Zeile, die sich im Puffer befindet, zeigt die Variable aktuell der Editor-Struktur, die wir als Indiz dafür nehmen, ob sich gerade eine Zeile im Puffer befindet. Wir dürfen übrigens nicht vergessen, das Flag ZLF_USED in der Zeile-Struktur zu setzen, wenn wir eine Zeile in den Puffer kopieren. Dieses Flag brauchen wir, um bei der Ausgabe festzustellen, ob sich eine Zeile gerade im Puffer befindet.

Damit können wir nun die Funktionen konzipieren, aus denen sich unser neues Modul "Edit" zusammensetzen wird:

```
void getLineForEdit(zeile,zeilennummer)
struct Zeile      *zeile;
UWORD                   zeilennummer;
```

Diese Funktion kopiert die Zeile zeile in den Puffer und setzt alle entsprechenden Variablen. Zum Konvertieren verwenden wir die Funktion convertLineForPrint, da diese ohnehin genau das macht, was wir benötigen. Freilich müssen wir aktueller-Editor->leftpos vorher auf Null setzen, damit auch wirklich alle Zeichen konvertiert werden und später wieder den ursprünglichen Wert zurückschreiben. Allerdings müssen wir diese Funktion noch etwas erweitern, da wir die Länge der konvertierten Zeile benötigen, die wir in aktuellerEditor->puflen abspeichern, und da wir auch lastchar kennen müssen, das wir uns in aktuellerEditor->puflastchar merken. Beim Löschen von Zeichen muß nämlich das letzte Zeichen im Puffer durch lastchar ersetzt werden, damit diese auch korrekt aussieht, wenn wir die Zeile auf dem Bildschirm ausgeben.

```
BOOL saveLine(zeile)
struct Zeile *zeile;
```

Speichert die im Puffer befindliche Zeile wieder in die Zeile-Struktur ab, auf die aktuellerEditor->aktuell zeigt. Dabei muß eventuell eine neue Zeile beschafft werden, falls die Länge der ursprünglichen Zeile von der neuen Länge abweicht. Falls die alte oder die geänderte Zeile eine Faltenmarkierung darstellt, so müssen gegebenenfalls die Level aller folgenden Zeilen entsprechend angepaßt werden. Falls, aus welchem Grund auch immer, die Zeile nicht zurückgeschrieben werden konnte, so gibt die Funktion FALSE zurück.

```
void saveIfCursorMoved()
```

Diese Funktion wird in der Hauptschleife des Editors nach der Abarbeitung jeder Message aufgerufen. Stellt diese Funktion fest, daß der Cursor die aktuelle Eingabezeile verlassen hat, so ruft sie die Funktion saveLine auf.

Die folgenden Funktionen stellen die eigentlichen Funktionen zur Textbearbeitung dar. Dadurch, daß diese als einzelne Funktionen implementiert werden, läßt sich später eine Kommandosprache um so einfacher realisieren, da die einzelnen Befehle einfach nur die entsprechenden Funktionen aufrufen. Jede dieser Funktionen gibt FALSE zurück, falls die Funktion nicht ausgeführt werden konnte.

```
BOOL undoLine()
```

Wenn sich eine Zeile im Puffer befindet, so werden alle Verweise gelöscht, und die ursprüngliche Zeile wird wieder sichtbar gemacht.

```
BOOL deleteChar()
```

Löscht das Zeichen unter dem Cursor und schiebt den rechten Rest der Zeile um ein Zeichen nach links. Dies geschieht im Programm beim Drücken der Delete-Taste.

```
BOOL backspaceChar()
```

Löscht das Zeichen links vom Cursor und schiebt den rechten Rest der Zeile um ein Zeichen nach links, wenn der Editor im Einfügemodus arbeitet. Andernfalls wird einfach nur ein Blank links vom Cursor über den Text geschrieben. Der Cursor wird um eine Position nach links gesetzt.

```
BOOL insertChar(c)
UBYTE c;
```

Fügt das Zeichen c an der Cursor-Position in den Text ein, wenn sich der Editor im Einfügemodus befindet. Arbeitet der Editor dagegen im Überschreibmodus, so wird das Zeichen unter dem Cursor durch das neue Zeichen ersetzt. Diese Funktion bearbeitet auch die Tabulatoren.

```
BOOL insertLine(c)
UBYTE c;
```

Diese Funktion trennt die aktuelle Zeile an der momentanen Cursor-Position auf und fügt eine neue Zeile in den Text ein. Das Zeichen c ist entweder CR oder LF, je nachdem, welche

Taste gedrückt worden ist. Der Cursor wird auf den Anfang der nächsten Zeile gesetzt.

In diese Funktion bauen wir Auto-Indent ein. Auto-Indent bedeutet, daß der Cursor nach Drücken von Return nicht in der ersten Spalte der neuen Zeile steht, sondern unter dem ersten Zeichen der darüber liegenden Zeile. Dies gilt auch dann, wenn eine Zeile durch Return aufgesplittet wird und sich der Cursor mitten in der Zeile befand. Über das Flag autoindent der Editor-Struktur kann man Auto-Indent auch abschalten.

```
BOOL deleteLine()
```

Löscht die Zeile, in der sich der Cursor befindet. Aufpassen muß man hierbei, wenn man sich in einer Falte befindet, die am Textende steht. Löscht man hier alle Zeilen, so kommt es dazu, daß noch weitere Zeilen existieren, diese aber nicht auf dem Bildschirm dargestellt werden, da deren Falten-Level außerhalb von minfold und maxfold liegen. In diesem Fall vermindert diese Funktion minfold solange, bis eine Zeile gefunden wird, die dann auch in der obersten Zeile des Fensters angezeigt wird. Ansonsten arbeitet die Funktion so, daß sie versucht, den Cursor auf der aktuellen Position zu lassen, das heißt, daß normalerweise der untere Teil des Fensterinhalts nach oben gescrollt wird, nachdem die Zeile gelöscht worden ist. Aber das Scrolling ist ohnehin eine Sache für sich, weswegen wir es auch erst nach der Auflistung der Funktionen erläutern werden, damit Sie sich dies dann anhand des Programmtextes vor Augen führen können.

Aufgerufen werden alle genannten Funktionen von der Funktion handleKeys aus, die wir ja bereits für die Cursor-Steuerung verwendet haben. Hier nun der Quelltext unseres neuen Moduls:

```
<src/Edit.c>

/*************
 *
 * Includes:
 *
 *************/

#include <exec/types.h>
#include <intuition/intuition.h>
```

```
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define CR 13
#define LF 10
#define TAB 9

/* Laenge von Foldanfang und -ende: */
#define FSE_LEN 10
/* Anzahl Zeichen, die signifikant fuer Faltenmarkierung sind: */
#define FSE_SIG 8

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

UWORD convertLineForPrint(),strncmp(),recalcTopOfWindow();
void loescheZeile(),Insert(),printAll(),restoreZeilenptr();
void DisplayBeep(),printLine(),AddHead(),scrollRight(),scrollUp();
void ScrollRaster();
struct Zeile *neueZeile(),*nextLine(),*prevLine();
BOOL cursorLeft(),cursorRight(),cursorDown(),cursorHome();

/**********************
 *
 * Externe Variablen:
 *
 **********************/

extern struct Editor *aktuellerEditor;

/**********************
 *
 * Globale Variablen:
 *
 **********************/

UBYTE foldAnfang[] = "/*#FOLD:*/";
UBYTE foldEnde[]   = "/*#ENDFD*/";

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/
```

```
/*********************************
 *
 * getLineForEdit(zeile,znr)
 *
 * Kopiert Zeile zum Editieren
 * in den Puffer.
 *
 * zeile ^ Zeile.
 * znr = Zeilennummer der Zeile.
 *
 *********************************/

void getLineForEdit   (zeile,znr)
register struct Zeile *zeile;
register UWORD               znr;
{
   register UWORD leftpos;

   /* Leftpos retten: */
   leftpos = aktuellerEditor->leftpos;
   aktuellerEditor->leftpos = 0;

   /* Zeile konvertieren: */
   aktuellerEditor->puflen =
      convertLineForPrint(zeile+1,zeile->len,MAXBREITE + 1,
         aktuellerEditor->puffer,&aktuellerEditor->puflastchar);

   /* Zeile als "benutzt" markieren: */
   zeile->flags |= ZLF_USED;
   aktuellerEditor->aktuell = zeile;
   aktuellerEditor->pufypos = znr;

   /* Leftpos wieder herstellen: */
   aktuellerEditor->leftpos = leftpos;
}

/*********************************
 *
 * cursorOnText:
 *
 * Stellt sicher, dass sich der
 * Cursor auf Text befindet.
 *
 *********************************/

void cursorOnText()
{
   register struct Zeile **zptr;
   register UWORD l;

   /* Sicherstellen, dass Cursor auch wirklich auf Text steht: */
   zptr = &(ZEILENPTR(l = aktuellerEditor->wdy));
   while ((*zptr == NULL) && l)
   {
```

```
      zptr--;
      l--;
   }
   aktuellerEditor->wdy  = l;
   aktuellerEditor->ypos = ZEILENNR(l);
}

/************************
 *
 * deconvertLine():
 *
 * Dekonvertiert Zeile.
 *
 * buf ^ Ziel-Puffer.
 * puffer^ Quell-Puffer.
 * puflen= dessen Laenge.
 *
 ************************/

UWORD deconvertLine(buf, puffer,puflen)
UBYTE              *buf,*puffer;
UWORD                           puflen;
{
   register UBYTE *p1,*p2,*tab,*fb = NULL;
   register WORD l;

   /* Zeile dekonvertieren: */
   p1 = puffer;
   p2 = buf;
   tab= aktuellerEditor->tabstring;
   l = puflen;

   if (aktuellerEditor->tabs)
      /* Blanks wieder in Tabulatoren verwandeln: */
      while (l)
      {
         if ((*p2 = *p1++) == ' ')
         {
            if (fb == NULL)
               fb = p2;
         }
         else
         {
            if (fb)
               fb = NULL;
         }

         /* fb zeigt auf erstes Blank, sofern seitdem keine  */
         /* anderen Zeichen kopiert worden sind. (first blank) */
         if ((! *++tab) && (fb))
         {
            *fb = TAB;
            fb++;
            p2 = fb;
```

```
            }
            else
               p2++;

            l--;
         }
      else
         while (l)
         {
            *p2 = *p1;
            p1++;
            p2++;
            l--;
         }

      /* Blanks am Ende loeschen: */
      if (aktuellerEditor->skipblanks)
         while ((p2 > buf) && ((*(p2-1) == ' ') || (*(p2-1) == TAB)))
            p2--;

      return ((UWORD)(p2 - buf));
}

/****************************************
 *
 * getFoldInc(zeile)
 *
 * Bestimmt, ob Zeile eine Faltenanfangs-
 * oder -Endemarkierung ist und gibt
 * dementsprechend 1 oder -1 zurueck.
 * Andernfalls wird 0 zurueckgegeben.
 *
 * zeile ^ entsprechende Zeile.
 *
 ****************************************/

UWORD getFoldInc(zeile)
struct Zeile    *zeile;
{
   register UBYTE *p1;
   register UWORD l;

   /* Faltenmarkierung am Anfang einer Zeile! */
   p1 = (UBYTE *)(zeile + 1);
   l = zeile->len;
   while (l && ((*p1 == ' ') || (*p1 == TAB)))
   {
      p1++;
      l--;
   }

   /* Bestimme Art der Faltenmarkierung (-> l): */
   /* 1 = Anfang,                               */
   /* -1 = Ende und                             */
```

```
   /* 0 = keine Faltenmarkierung.                */
   if (l >= FSE_SIG)
      if (strncmp(p1,foldAnfang,FSE_SIG) == 0)
         l = 1;
      else if (strncmp(p1,foldEnde,FSE_SIG) == 0)
         l = -1;
      else
         l = 0;
   else
      l = 0; /* keine Faltenmarkierung */

   return (l);
}

/*******************************
 *
 * saveLine(zeile)
 *
 * Speichert Zeile wieder ab.
 *
 * zeile ^ alte Zeile-Struktur.
 *
 *******************************/

BOOL saveLine(zeile)
struct Zeile *zeile;
{
   static UBYTE buf[MAXBREITE + 2];  /* +2 fuer CR und LF */
   register UBYTE *p1,*p2,*tab,*fb = NULL;
   register WORD l,len;
   struct Zeile *pred,*old,**zptr;

   /* Nur zur Sicherheit: */
   if (aktuellerEditor->aktuell == NULL)
      return (FALSE);

   /* Zeile dekonvertieren: */
   p2 = buf + (len = deconvertLine(buf,aktuellerEditor->puffer,
                                   aktuellerEditor->puflen));

   /* Stelle fest, ob Zeile durch CR oder LF beendet, */
   /* und erhoehe len entsprechend. */
   p2 = buf + len;
   if (l = zeile->len)
   {
      p1 = ((UBYTE *)(zeile + 1)) + zeile->len - 1;
      if (*p1 == CR)
      {
         len++;
         *p2 = CR;
      }
      else if (*p1 == LF)
      {
         len++;
```

```
      if ((l > 1) && (*--p1 == CR))
      {
         len++;
         *p2++ = CR;
      }
      *p2 = LF;
   }
}

/* ggf. neue Zeile beschaffen: */
if (EVENLEN(len) != EVENLEN(zeile->len))
{
   old = zeile;
   if (zeile = neueZeile(len))
   {
      Insert(&aktuellerEditor->zeilen,zeile,old);
      zeile->flags = old->flags & ~ZLF_USED;
      loescheZeile(old);

      /* Nun eventuelle Zeiger umsetzen: */
      for (l = 0, zptr = aktuellerEditor->zeilenptr;
           l <= aktuellerEditor->wch; l++, zptr++)
         if (*zptr == old)
         {
            *zptr = zeile;
            break;
         }
   }
   else
      return (FALSE);
}
else
{
   zeile->len   = len;
   zeile->flags &= ~ZLF_USED;
}

/* Zeile zurueckschreiben: */
p1 = (UBYTE *)(zeile + 1);
p2 = buf;
l = len;

while (l)
{
   *p1 = *p2;
   p1++;
   p2++;
   l--;
}

/* Folding ggf. rekonstruieren: */
/* War alte oder ist neue Zeile eine Faltenmarkierung? */
l = getFoldInc(zeile);
if ((zeile->flags & ZLF_FSE) || l)
```

```
{
   /* Zunaechst das ZLF_FSE-Flag entsprechend l setzen: */
   if (l)
      zeile->flags |= ZLF_FSE;
   else
      zeile->flags &= ~ZLF_FSE;

   /* Bestimme nun, wie der Falten-Level der nachfolgenden */
   /* Zeilen geändert werden muß:          level += l.     */
   if ((old = zeile->succ)->succ)
   {
      if (((zeile->flags+1) & ZLF_FOLD) ==
                              (old->flags & ZLF_FOLD))
         /* Alte Zeile war Faltenanfang: */
         l -= 1;
      else if ((zeile->flags &ZLF_FOLD)==
                              ((old->flags+1) &ZLF_FOLD))
         /* Alte Zeile war Faltenende: */
         l += 1;

      if (l)
      {
         /* Falten-Level aller folgender Zeilen ändern: */
         while (old->succ)
         {
            old->flags = (old->flags & ~ZLF_FOLD)
                       | ((old->flags + l) & ZLF_FOLD);
            old = old->succ;
         }

         restoreZeilenptr();
         printAll();
         cursorOnText();
      }
   }
}

/* Zeile zur Sicherheit ausgeben: */
/* Zuerst Ypos bestimmen:    */
for (l = 0, zptr = aktuellerEditor->zeilenptr;
     l <= aktuellerEditor->wch; l++, zptr++)
   if (*zptr == zeile)
   {
      printLine(zeile,aktuellerEditor->yoff +
                              aktuellerEditor->ch*l);
      break;
   }

/* Verweise loeschen: */
aktuellerEditor->aktuell = NULL;
aktuellerEditor->puflen  = 0;
aktuellerEditor->pufypos = 0;
```

```
   /* Text wurde veraendert! */
   aktuellerEditor->changed = 1;

   return (TRUE);
}

/*************************************
 *
 * saveIfCursorMoved:
 *
 * Speichert Zeile im Puffer wieder ab,
 * falls der Cursor bewegt worden ist.
 *
 *************************************/

void saveIfCursorMoved()
{
   register UWORD pufypos;

   if (pufypos = aktuellerEditor->pufypos)
      if (aktuellerEditor->ypos != pufypos)
         if (! saveLine(aktuellerEditor->aktuell))
            DisplayBeep(NULL);
}

/*****************************************
 *
 * undoLine:
 *
 * Macht die Aenderungen in der aktuellen
 * Zeile wieder rueckgaengig, sofern
 * der Cursor diese noch nicht wieder
 * verlassen hat.
 *
 *****************************************/

void undoLine()
{
   register struct Zeile *z;

   if (z = aktuellerEditor->aktuell)
   {
      aktuellerEditor->aktuell = NULL;
      aktuellerEditor->pufypos = 0;
      aktuellerEditor->puflen  = 0;

      z->flags &= ~ZLF_USED;

      /* Zeile ausgeben: */
      printLine(z,aktuellerEditor->yoff
                  + aktuellerEditor->ch*aktuellerEditor->wdy);
   }
}
```

```
/*************************************************
 *
 * getPufferPointer(pptr,pinc,prest):
 *
 * Initialisiert Zeiger auf aktuellerEditor->puffer.
 * Der Puffer wird ggf. initialisiert.
 * Gibt FALSE zurueck, falls Puffer nicht
 * initialisiert werden konnte.
 *
 * pptr ^ Zeiger in Puffer auf Cursor-Position.
 * pinc ^ Offset vom Pufferanfang.
 * prest^ puflen - inc.
 *
 *************************************************/

BOOL getPufferPointer(pptr,pinc, prest)
UBYTE               **pptr;
UWORD                    *pinc,*prest;
{
   register UBYTE *ptr;
   register WORD inc,rest;

   /* Puffer ggf. initialisieren: */
   if (aktuellerEditor->aktuell == NULL)
   {
      register UWORD wdy;

      if (ZEILENPTR(wdy = aktuellerEditor->wdy))
          getLineForEdit(ZEILENPTR(wdy),ZEILENNR(wdy));
      else
         if (aktuellerEditor->anz_zeilen == 0)
         {
            /* Erzeuge allererste Zeile: */
            register struct Zeile *z;

            if (z = neueZeile((UWORD) 1))
            {
               *((UBYTE *)(z + 1)) = LF;
               AddHead(&aktuellerEditor->zeilen,z);
               ZEILENPTR(0) = z;
               ZEILENNR(0)  = (aktuellerEditor->anz_zeilen = 1);
               getLineForEdit(ZEILENPTR(0),ZEILENNR(0));
            }
            else
               return (FALSE);
         }
         else
            /* Cursor steht nicht auf Zeile! */
            return (FALSE);
   }

   /* Bestimme Zeiger auf Cursor-Position: */
   inc = aktuellerEditor->xpos - 1;
   ptr = aktuellerEditor->puffer + inc;
```

```
   rest= aktuellerEditor->puflen - inc;

   /* Was, wenn inc > puflen? */
   if (inc < MAXBREITE)
   {
      if (rest < 0)
      {
         register UBYTE *p;

         p = aktuellerEditor->puffer + aktuellerEditor->puflen;
         aktuellerEditor->puflen -= rest;
         while (rest)
         {
            *p = ' ';
            p++;
            rest++;
         }
      }
   }
   else
      return (FALSE);

   *pptr = ptr;
   *pinc = inc;
   *prest= rest;
   return (TRUE);
}

/************************************
 *
 * deleteChar:
 *
 * Loescht Zeichen unter dem Cursor.
 * Gibt FALSE zurueck, falls kein
 * Buchstabe mehr.
 *
 ************************************/

BOOL deleteChar()
{
   UBYTE *ptr;
   UWORD inc,rest;
   register UBYTE *p1,*p2;
   register UWORD l;

   if (! getPufferPointer(&ptr,&inc,&rest))
      return (FALSE);

   if (rest)
   {
      p1 = ptr + 1;
      p2 = ptr;
      l = --rest;
      while (l)
```

```
      {
         *p2 = *p1;
         p1++;
         p2++;
         l--;
      }
      *p2 = aktuellerEditor->puflastchar;

      /* Laenge - 1 */
      aktuellerEditor->puflen--;

      printLine(aktuellerEditor->aktuell,aktuellerEditor->yoff
                  + aktuellerEditor->ch*aktuellerEditor->wdy);

      return (TRUE);
   }
   else
      return (FALSE);
}

/*****************************
 *
 * backspaceChar:
 *
 * Loescht Zeichen vor Cursor.
 * Gibt FALSE zurueck, wenn
 * Cursor am Zeilenanfang.
 *
 *****************************/

BOOL backspaceChar()
{
   UBYTE *ptr;
   UWORD inc,rest;
   register UBYTE *p1,*p2;
   register UWORD l;

   if (! getPufferPointer(&ptr,&inc,&rest))
      return (FALSE);

   if (inc)
   {
      if (aktuellerEditor->insert)
      {
         p1 = ptr;
         p2 = --ptr;
         l = rest;
         while (l)
         {
            *p2 = *p1;
            p1++;
            p2++;
            l--;
         }
```

```
            *p2 = aktuellerEditor->puflastchar;

            /* Laenge - 1 */
            aktuellerEditor->puflen--;
            inc--;
         }
         else
         {
            *--ptr = ' ';
            inc--;
            rest++;
         }

         cursorLeft();
         printLine(aktuellerEditor->aktuell,aktuellerEditor->yoff
                     + aktuellerEditor->ch*aktuellerEditor->wdy);

         return (TRUE);
      }
      else
         return (FALSE);
}

/*****************************************
 *
 * insertChar(c)
 *
 * Fuegt Zeichen in Puffer ein.
 * Gibt FALSE zurueck, falls Zeichen nicht
 * eingefuegt werden konnte.
 *
 * c = Zeichen.
 *
 *****************************************/

BOOL insertChar(c)
register UBYTE  c;
{
   UBYTE *ptr;
   WORD inc,rest;
   WORD xadd = 1;

   if (! getPufferPointer(&ptr,&inc,&rest))
      return (FALSE);

   if ((c == TAB) && (aktuellerEditor->tabs))
   {
      if (aktuellerEditor->insert)
      {
         /* Blanks einfuegen: */
         register UBYTE *p1,*p2;
         register UWORD anz,l;
```

```
        p1  = aktuellerEditor->tabstring +
                                aktuellerEditor->xpos - 1;
        l  = MAXBREITE - aktuellerEditor->xpos;
        anz = 1;
        while (l && (*++p1))
        {
           xadd++;
           l--;
           anz++;
        }

        /* anz = Anzahl der Blanks, die eingefuegt werden: */
        if (aktuellerEditor->puflen + anz > MAXBREITE)
           anz = MAXBREITE - aktuellerEditor->puflen;

        /* Rest der Zeile verschieben: */
        p1 = ptr + rest;
        p2 = p1 + anz;
        l = rest;
        while (l)
        {
           *--p2 = *--p1;
           l--;
        }

        /* Blanks einfuegen: */
        p1 = ptr;
        l = anz;
        while (l)
        {
           *p1 = ' ';
           p1++;
           l--;
        }

        aktuellerEditor->puflen += anz;
        rest += anz;
     }
     else
     {
        /* Nur Cursor bewegen: */
        register UBYTE *tab;
        register UWORD max;

        tab = aktuellerEditor->tabstring +
                                aktuellerEditor->xpos - 1;
        max = MAXBREITE - aktuellerEditor->xpos;
        while (max && (*++tab))
        {
           xadd++;
           max--;
        }
     }
}
```

```
   else
   {
      /* sonst als normales Zeichen: */
      if (aktuellerEditor->insert)
         if (aktuellerEditor->puflen < MAXBREITE)
         {
            register UBYTE *p1,*p2;
            register UWORD l;

            p1 = ptr + rest;
            p2 = p1;
            l = rest;
            while (l)
            {
               *p2 = *--p1;
               p2--;
               l--;
            }
            *ptr = c;

            /* Laenge + 1 */
            aktuellerEditor->puflen++;
            rest++;
         }
         else
            /* Spaeter hier Word-Wrap */
            return (FALSE);
      else
      {
         *ptr = c;
         if (rest = 0)
         {
            aktuellerEditor->puflen++;
            rest = 1;
         }
      }
   }

   /* Cursor bewegen: */
   while (xadd)
   {
      if (! cursorRight()) break;
      xadd--;
   }

   /* Zeile ausgeben: */
   printLine(aktuellerEditor->aktuell,aktuellerEditor->yoff
                + aktuellerEditor->ch*aktuellerEditor->wdy);

   return (TRUE);
}
```

```
/*************************
 *
 * insertLine(c):
 *
 * Fuegt neue Zeile hinter
 * aktueller Zeile ein.
 * Gibt FALSE zurueck, falls
 * Fehler aufgetreten ist.
 *
 * c = Zeilenende (CR/LF).
 *
 *************************/

BOOL insertLine(c)
UBYTE           c;
{
   static UBYTE buf[MAXBREITE + 1];
   UBYTE *ptr;
   register UBYTE *p1,*p2;
   WORD inc,rest,len;
   UWORD autoindent;
   register UWORD l;
   register struct Zeile *z;

   /* Nur neue Zeile einfuegen, wenn aktuelle Zeile
      keine Faltenmarkierung!!! */
   if ((z = aktuellerEditor->aktuell) == NULL)
      z = ZEILENPTR(aktuellerEditor->wdy);

   if (!z || !(z->flags & ZLF_FSE))
   {
      /* Zeile in Puffer holen: */
      if (! getPufferPointer(&ptr,&inc,&rest))
         return (FALSE);

      /* Auto-Indent bestimmen: */
      if (aktuellerEditor->autoindent)
      {
         p1 = aktuellerEditor->puffer;
         l = inc;
         while (l && ((*p1 == ' ') || (*p1 == TAB)))
         {
            p1++;
            l--;
         }

         if (l)
            autoindent = p1 - aktuellerEditor->puffer + 1;
         else
            /* Zeile besteht nur aus Spaces: */
            autoindent = 1;
      }
      else
         autoindent = 1;
```

```
/* Zeile dekonvertieren: */
p2 = buf + (len = deconvertLine
                    (buf,aktuellerEditor->puffer,inc));
*p2= c;
len++;

if (z = neueZeile(len))
{
   z->flags = (aktuellerEditor->aktuell->flags & ~ZLF_USED);

   /* Zeile, in der Cursor steht, abspeichern: */
   p1 = (UBYTE *)(z + 1);
   p2 = buf;
   l = len;

   while (l)
   {
      *p1 = *p2;
      p1++;
      p2++;
      l--;
   }

   /* Zeile einbinden: */
   Insert(&aktuellerEditor->zeilen,z,
          aktuellerEditor->aktuell->pred);
   aktuellerEditor->anz_zeilen++;
   aktuellerEditor->changed = 1;

   /* Rest der Puffer-Zeile nach vorne schieben: */
   /* Dabei muss autoindent beachtet werden! */
   p1 = aktuellerEditor->puffer;
   l = autoindent;
   while (--l)
      p1++;    /* Alter Inhalt bleibt erhalten! */

   p2 = ptr;
   l = rest;
   while (l)
   {
      *p1 = *p2;
      p1++;
      p2++;
      l--;
   }
   aktuellerEditor->puflen = p1 - aktuellerEditor->puffer;
   /* Rest mit lastchar fuellen: */
   l = inc + 1 - autoindent;
   while (l)
   {
      *p1 = aktuellerEditor->puflastchar;
      p1++;
      l--;
   }
```

```
/* Stelle fest, ob neue Zeile Faltenmarkierung: */
if (l = getFoldInc(z))
{
   register struct Zeile *old;
   register UWORD fold;

   z->flags |= ZLF_FSE;

   /* Falten-Level aller folgender Zeilen ändern: */
   if ((old = z->succ)->succ)
      while (old->succ)
      {
         old->flags = (old->flags & ~ZLF_FOLD)
                    | ((old->flags + l) & ZLF_FOLD);
         old = old->succ;
      }

   /* ggf. minfold und maxfold anpassen: */
   if ((l > 0) || ((z->flags & ZLF_FOLD) > 0))

   {
      fold = (z->flags & ZLF_FOLD) + l;
      if (aktuellerEditor->minfold > fold)
         aktuellerEditor->minfold = fold;
      if (aktuellerEditor->maxfold < fold)
         aktuellerEditor->maxfold = fold;

      ZEILENPTR(aktuellerEditor->wdy) = z;
      aktuellerEditor->wdy =
                recalcTopOfWindow(aktuellerEditor->wdy);
   }
}

/* Falls Cursor auf oberster Zeile
        => zeilenptr[0] aendern: */
if (aktuellerEditor->wdy == 0)
   ZEILENPTR(0) = z;

restoreZeilenptr();

/* Fenster neu ausgeben: */
aktuellerEditor->xpos = autoindent;
if (aktuellerEditor->xpos <= aktuellerEditor->leftpos)
{

   /* Falls waagerechtes Scrolling noetig: printAll */
   aktuellerEditor->leftpos = aktuellerEditor->xpos - 1;
   if (++aktuellerEditor->wdy >= aktuellerEditor->wch)
   {
      ZEILENPTR(0) = ZEILENPTR(1);
      ZEILENNR(0)  = ZEILENNR(1);
      restoreZeilenptr();
      aktuellerEditor->wdy--;
   }
```

```
      aktuellerEditor->ypos =
              ZEILENNR(aktuellerEditor->wdy);

      printAll();
   }
   else
      if (l)
      {
         /* falls Faltenmarkierung:
                Ganzes Fenster neu ausgeben:*/
         if (++aktuellerEditor->wdy >= aktuellerEditor->wch)
         {
            ZEILENPTR(0) = ZEILENPTR(1);
            ZEILENNR(0)  = ZEILENNR(1);
            restoreZeilenptr();
            aktuellerEditor->wdy--;
         }
         aktuellerEditor->ypos =
                 ZEILENNR(aktuellerEditor->wdy);

         printAll();
      }
      else
      {
         /* Es wird gescrollt: */
         if (++aktuellerEditor->wdy >= aktuellerEditor->wch)
         {
            /* Cursor war auf letzter Zeile: */
            register UWORD nr;

            scrollUp((UWORD) 1);
            nr = (--aktuellerEditor->wdy) - 1;
            aktuellerEditor->ypos=
                            ZEILENNR(aktuellerEditor->wdy);
            printLine(ZEILENPTR(nr),aktuellerEditor->yoff
                                  + aktuellerEditor->ch*nr);
         }
         else
         {
            /* Cursor steht irgendwo mitten im Fenster: */
            register UWORD nr;

            if (ZEILENPTR((nr = aktuellerEditor->wdy) + 1))
            {
               /* Rest des Fensters nach unten schieben: */
               ScrollRaster(aktuellerEditor->rp,
                            0L,(LONG)-aktuellerEditor->ch,
                            (LONG)aktuellerEditor->xoff,
                            (LONG)aktuellerEditor->yoff
                            + aktuellerEditor->wdy
                             *aktuellerEditor->ch,
                            (LONG)aktuellerEditor->xscr,
                            (LONG)aktuellerEditor->yscr);
            }
```

```
                aktuellerEditor->ypos=ZEILENNR
                                (aktuellerEditor->wdy);

                printLine(ZEILENPTR(nr),aktuellerEditor->yoff
                                + aktuellerEditor->ch*nr);
                nr--;
                printLine(ZEILENPTR(nr),aktuellerEditor->yoff
                                + aktuellerEditor->ch*nr);
             }
          }

        return (TRUE);
     }
     else
        return (FALSE);
  }
  else
  {
     cursorHome();
     cursorDown();

     return (FALSE);
  }
}

/*****************************
 *
 * deleteLine:
 *
 * Loescht die Zeile,
 * auf der der Cursor steht.
 * Gibt FALSE zurueck, falls
 * die Zeile nicht geloescht
 * werden konnte.
 *
 *****************************/

BOOL deleteLine()
{
  register struct Zeile *zeile,*next,*prev;
  register UWORD l = 0;
  UWORD nextnr,prevnr;

  if (zeile = ZEILENPTR(aktuellerEditor->wdy))
  {
     /* Verweise auf Zeile loeschen: */
     if (zeile == aktuellerEditor->aktuell)
     {
        aktuellerEditor->aktuell = NULL;
        aktuellerEditor->pufypos = 0;
        aktuellerEditor->puflen  = 0;
     }
```

```
aktuellerEditor->changed = 1;
aktuellerEditor->anz_zeilen--;

/* Folding ggf. rekonstruieren: */
if (zeile->flags & ZLF_FSE)
{
   register struct Zeile *z;

   /* Falten-Level aller folgender Zeilen ändern: */
   if ((z = zeile->succ)->succ)
   {
      if (((zeile->flags+1) & ZLF_FOLD) ==
                              (z->flags & ZLF_FOLD))
         /* Zeile war Faltenanfang: */
         l = -1;
      else if ((zeile->flags&ZLF_FOLD)==
                           ((z->flags+1)&ZLF_FOLD))
         /* Zeile war Faltenende: */
         l = 1;
      else
         l = 0;

      if (l)
         while (z->succ)
         {
            z->flags = (z->flags & ~ZLF_FOLD)
                     | ((z->flags + l) & ZLF_FOLD);
            z = z->succ;
         }
   }
}

/* Naechste und vorige Zeile merken: */
prevnr= ZEILENNR(aktuellerEditor->wdy);
prev  = prevLine(zeile,&prevnr);
nextnr= ZEILENNR(aktuellerEditor->wdy);
next  = nextLine(zeile,&nextnr);

/* Sicherheitsabfrage: */
while ((prev==NULL) && (next==NULL) &&
                       (aktuellerEditor->minfold))
{
   l = 1;       /* Fenster komplett neu ausgeben */
   aktuellerEditor->minfold--;

   prevnr= ZEILENNR(aktuellerEditor->wdy);
   prev = prevLine(zeile,&prevnr);
   nextnr= ZEILENNR(aktuellerEditor->wdy);
   next = nextLine(zeile,&nextnr);

   if (prev || next) break;
   if (aktuellerEditor->minfold) continue;

   if ((next = aktuellerEditor->zeilen.head)->succ == NULL)
```

```
      next = NULL;
   else
      nextnr = 1;

   break;
}

/* Zeile loeschen: */
loescheZeile(zeile);

/* Zeiger auf aktueller Zeile neu setzen fuer recalc: */
if (next)
{
   ZEILENPTR(aktuellerEditor->wdy) = next;
   ZEILENNR(aktuellerEditor->wdy)  = nextnr;
}
else
{
   ZEILENPTR(aktuellerEditor->wdy) = prev;
   ZEILENNR(aktuellerEditor->wdy)  = prevnr;
}

/* Nun Fensterinhalt restaurieren: */
if (l)
{
   /* Fenster auf jeden Fall ganz neu ausgeben: */
   aktuellerEditor->wdy =
                  recalcTopOfWindow(aktuellerEditor->wdy);
   restoreZeilenptr();
   printAll();
   cursorOnText();
}
else
{
   /* Mit Scrolling: */
   register UWORD oldwdy;

   oldwdy = aktuellerEditor->wdy;
   aktuellerEditor->wdy =
                  recalcTopOfWindow(aktuellerEditor->wdy);
   restoreZeilenptr();

   if (next)
   {
      /* Unteren Rest des Fensters nach oben schieben: */
      ScrollRaster(aktuellerEditor->rp,
                   0L,(LONG)aktuellerEditor->ch,
                   (LONG)aktuellerEditor->xoff,
                   (LONG)aktuellerEditor->yoff +
                   aktuellerEditor->wdy*aktuellerEditor->ch,
                   (LONG)aktuellerEditor->xscr,
                   (LONG)aktuellerEditor->yscr);

      printLine(ZEILENPTR(aktuellerEditor->wch - 1),
```

```
                aktuellerEditor->yoff +
                aktuellerEditor->ch*(aktuellerEditor->wch - 1));
        }
        else
            if (oldwdy == aktuellerEditor->wdy)
            {
                /* Oberen Teil des Fensters nach unten scrollen: */
                ScrollRaster(aktuellerEditor->rp,
                             0L,(LONG)-aktuellerEditor->ch,
                             (LONG)aktuellerEditor->xoff,
                             (LONG)aktuellerEditor->yoff,
                             (LONG)aktuellerEditor->xscr,
                             (LONG)aktuellerEditor->yoff
                                   + (aktuellerEditor->wdy + 1)
                                     *aktuellerEditor->ch - 1);

                printLine(ZEILENPTR(0),aktuellerEditor->yoff);
            }
            else
                printLine(ZEILENPTR(oldwdy),aktuellerEditor->yoff
                                  + aktuellerEditor->ch*oldwdy);

        cursorOnText();
    }

    return (TRUE);
  }
  else
    return (FALSE);
}
```

Hier nun ein paar Bemerkungen zu den Funktionen:

- Am Anfang des Moduls sind als globale Variablen die Zeichenketten definiert, die eine Faltenmarkierung darstellen. Dabei gibt FSE_SIG die Anzahl der signifikanten Zeichen für die Faltenmarkierung an; in unserem Fall wäre also /*#FOLD: signifikant für den Faltenanfang (fold-Anfang) und /*#ENDFD für das Faltenende (foldEnde). Diese Zeichenketten lassen sich beliebig ändern, so daß sich der Editor auch an andere Programmiersprachen anpassen läßt, also z.B. ;#FOLD: in Assembler. Das Define FSE_LEN gibt die gesamte Länge der Zeichenketten an, wobei, wie auch bei der Anzahl der signifikanten Stellen, diese für Faltenanfangs- und Faltenendemarkierung gleich sein muß (der Einfachheit halber). Die gesamte Länge werden wir später noch gebrauchen können, wenn wir das Wegfalten von Textstücken automatisieren. Dann muß der

Editor eine Zeile vor und eine nach dem Faltentext einfügen, die dann die Faltenmarkierungen darstellen. Damit Sie nicht das Kommentarende vergessen, wird dieses gleich mit eingefügt.

- In der Funktion getLineForEdit können Sie bereits die Änderung der Funktion convertLineForPrint erkennen. Sie gibt nun die Länge der konvertierten Zeile zurück. Als fünften Parameter müssen Sie nun einen Zeiger auf eine UBYTE-Variable übergeben, in der dann lastchar abgespeichert wird.

- Neu ist die Funktion cursorOnText, die überprüft, ob sich der Cursor noch auf einer existierenden Zeile befindet, oder ob er bereits hinter das Textende (Faltenende) bewegt wurde. Im zweiten Fall wird der Cursor auf die unterste im Fenster sichtbare Zeile gesetzt.

- Die Funktion deconvertLine stellt das Gegenstück zur Funktion convertLineForPrint aus dem Modul Ausgabe dar. Das Erzeugen von Tabulatoren geschieht dabei folgendermaßen: An jedem Tabulator (tab = 0) überprüft die Funktion, ob sich vor diesem Blanks befinden (fb != 0). Ist dies der Fall, so werden alle Blanks durch ein Tabulator (TAB) ersetzt.

- Eine weitere Option des Editors finden Sie ebenfalls in der Funktion convertLineForPrint. Wenn das Flag aktueller-Editor->skipblanks gesetzt ist, so werden alle Blanks, die am Ende einer Zeile stehen, gelöscht, da sie meist überflüssig sind. Wenn wir jedoch Fließtext mit unserem Editor eingeben wollen, so dürfen diese nicht gelöscht werden, weswegen dieses Feature abschaltbar ist.

- Die Funktion getFoldInc bestimmt, ob eine Zeile eine Faltenmarkierung darstellt, und gibt Null zurück, falls dies nicht der Fall ist. Andernfalls erhalten wir eine 1, wenn es sich bei der Zeile um eine Faltenanfangsmarkierung handelt, und -1 für das Faltenende. Diese Werte wurden mit Rücksicht auf die Funktion saveLine gewählt, auf die wir jetzt zu sprechen kommen.

- Die Funktion saveLine hat einiges zu tun, um die Falten-Level richtig einzustellen, wenn die Zeile eine Faltenmarkierung war.

Dazu folgende Überlegungen: War die Zeile vor der Änderung keine Faltenmarkierung und ist sie danach ein Faltenanfang, dann muß der Falten-Level aller folgenden Zeilen um eins erhöht werden.

War die Zeile vor der Änderung ein Faltenanfang und ist sie danach keine Faltenmarkierung mehr, so muß der Falten-Level aller folgenden Zeilen um eins vermindert werden.

Dieses gilt analog für das Faltenende, wobei nur Erhöhung und Verminderung vertauscht werden müssen. Wenn die Zeile jedoch sowohl vor als auch nach der Änderung eine Faltenmarkierung war, so wird die Sache kompliziert. Aber nur so lange, bis wir eine Wertetabelle aufstellen, aus der hervorgeht, um welchen Wert der Falten-Level aller folgenden Zeilen erhöht werden muß, in Abhängigkeit davon, ob die Zeile vor oder nach der Änderung eine Faltenmarkierung war oder nicht. Stellen Sie diese Tabelle doch einmal selbst auf, und Sie werden feststellen, daß folgende Vorgehensweise zum korrekten Ergebnis führt: Man bestimme mit der Funktion getFoldInc, ob es sich bei der geänderten Zeile um eine Faltenmarkierung handelt, und merke sich den Rückgabewert. Dann tue man dasselbe mit der Zeile vor der Änderung, wobei wir dafür nicht die Funktion getFoldInc benutzen, sondern den Falten-Level der Zeile mit dem der nächsten Zeile vergleichen. In diesem Fall nehmen wir jedoch den negativen Wert bezüglich getFoldInc, also -1 für Anfang und +1 für Ende. Nun addieren Sie beide Werte, und Sie erhalten den Wert, um den die Falten-Level aller folgenden Zeilen erhöht werden müssen.

Soviel zum Thema Folding. Ein Flag, das wir schon früher eingeführt haben, damit wir feststellen können, ob wir bereits etwas an unserem Text geändert haben, ist das changed-Flag aus der Editor-Struktur. Wenn wir mit saveLine eine Zeile endgültig geändert haben (vorher können wir dies ja noch durch Undo rückgängig machen), müssen wir dieses Flag setzen, damit es seinen Zweck erfüllen kann.

Ganz am Ende der Funktion wird die Zeile dann nochmals ausgegeben, da es ja sein kann, daß Blanks am Ende abgeschnitten worden sind, was vor allem bei Zeilen ohne Zeilenende zu einer Änderung der Darstellung auf dem Bildschirm führen würde.

- Bei der Funktion getPufferPointer handelt es sich um eine Hilfsfunktion, die eine Zeile in den Editorpuffer kopiert, oder besser, konvertiert. Dabei initialisiert sie drei Variablen, die das Editieren vereinfachen, und zwar einen Zeiger auf das Zeichen, auf dem sich der Cursor befindet, die Anzahl der Zeichen links vor dem Cursor und die Länge des Restes der Zeile. Befindet sich der Cursor jedoch hinter dem letzten Zeichen der Zeile, so werden so viele Leerzeichen eingefügt, daß zumindest direkt links vom Cursor ein Zeichen steht. Unter dem Cursor selbst braucht keines zu sein, da dort z.B. von der Funktion insertChar ein Zeichen eingefügt werden würde.
- Zur Funktion insertChar wäre noch zu sagen, daß Tabulatoren im Überschreibmodus nur den Cursor nach rechts bewegen, nicht aber den Text ändern, also auch keine Blanks über den Text schreiben.
- Die Funktion insertLine erlaubt es nicht, eine Faltenmarkierung durch Drücken von Return aufzuteilen. Dies liegt daran, daß man aus einer alten Zeile zwei neue (geänderte) erhält. Und dann die Falten-Level anzupassen... Deswegen haben wir uns erlaubt, diesen Sonderfall auszuschließen, was aber nicht heißen soll, daß Sie sich nicht an die Arbeit machen sollten, um diese Lücke zu füllen. Aber das bleibt Ihnen überlassen. Sehr wohl ist es erlaubt, eine Faltenmarkierung neu einzugeben und diese Zeile mit Return zu beenden, aber in diesem Fall ist das Ändern der Falten-Level der folgenden Zeilen auch harmlos.

Eine andere Sache ist die Ausgabe des Textes im Fenster. Dadurch, daß wir eine neue Zeile eingefügt haben, stimmt der Fensterinhalt ja nun nicht mehr. Wir könnten die Funktion printAll benutzen, aber das wäre viel zu langsam. Also müssen wir scrollen! In der insertLine sind wir dabei folgendermaßen vorgegangen:

Wenn der Fensterinhalt ohnehin horizontal gescrollt werden muß oder wenn die neu eingefügte Zeile eine Faltenmarkierung ist, so geben wir den Fensterinhalt doch mit printAll neu aus. Dies wäre zwar beim horizontalen Scrollen nicht nötig, aber wenn wir erst den Fensterinhalt horizontal und dann nochmal vertikal scrollen, so ist printAll auch nicht sehr viel langsamer. Andernfalls wird vertikal gescrollt, und zwar der Teil des Textes ab der Cursor-Position nach unten. Befand sich der Cursor aber schon in der untersten Zeile, so wird der gesamte Fensterinhalt nach oben gescrollt.

- Auch in der Funktion deleteLine muß gescrollt werden, hier bleibt der Cursor aber auf der aktuellen Zeile stehen, und der Rest des Textes wird nach oben gescrollt. Erst wenn der Cursor auf der letzen Zeile stand, wird der obere Teil des Textes nach unten gescrollt. Befindet sich auch kein Text mehr oberhalb des Fensters, so wird die Y-Position des Cursors vermindert.

Ein weiteres Problem in dieser Funktion tritt auf, wenn der Cursor in einer Falte am Textende steht (minfold > 0) und dort alle Zeilen gelöscht werden. Dann kommt der Editor nämlich nicht mehr an die Zeilen über dieser Falte heran, weswegen die Funktion deleteLine in einem solchen Fall so lange minfold vermindert, bis sie eine Zeile findet.

Die Faltenmarkierung können Sie auch mit deleteLine löschen. Dann werden die Falten-Level aller folgenden Zeilen entsprechend umgerechnet.

So weit, so gut. Die Änderungen an den anderen Modulen sind diesmal relativ umfangreich ausgefallen, weswegen wir gleich mit Editor.h beginnen wollen. Hier haben wir aus der Editor-Struktur die Variable toppos gestrichen, da unser Feld zeilennr dessen Funktion bereits erfüllt hat. Neu hinzugekommen sind dagegen folgende Elemente:

```
UBYTE puflastchar;
UWORD puflen,pufypos;
UWORD tabs       : 1;
UWORD skipblanks : 1;
UWORD autoindent : 1;
```

Als nächstes kommt das Modul Ausgabe.c an die Reihe, in dem wir unter anderem die Funktion convertLineForPrint anpassen müssen. Hier zunächst der geänderte Funktionskopf:

```
UWORD convertLineForPrint(zeile,len,w,buf,lc)
UBYTE                    *zeile;
register WORD                   len;
register UWORD                      w;
register UBYTE                       *buf;
UBYTE                                    *lc;
```

Ferner benötigen wir noch eine weitere lokale Variable:

```
UWORD realLen;
```

In dieser Variablen halten wir die tatsächliche Länge der konvertierten Zeile fest. Die folgende Konvertierung bleibt so bestehen, wie sie war. Aber bevor der Rest der Zeile mit lastchar gefüllt wird, müssen wir uns die Länge der Zeile merken:

```
/* Tatsaechliche Laenge der Zeile retten: */
realLen = l - 1;
```

Nun wird der Rest der Zeile mit lastchar gefüllt, und danach geben wir die tatsächliche Länge zurück:

```
*lc = lastchar;
return (realLen);
```

Die Änderungen in der Funktion printLine, die wir nun vornehmen müssen, sind so groß, daß wir diese Funktion komplett auflisten. Diese Funktion gibt nun den Puffer aus, wenn das ZLF_USED-Flag der Zeile gesetzt war. Existierte die Zeile gar nicht (zeile = Null), so wird der entsprechende Fensterbereich gelöscht, was wir uns bereits bei der Funktion deleteLine zunutze gemacht haben:

```
void printLine       (zeile,y)
register struct Zeile *zeile;
register UWORD                y;
{
   static UBYTE buf[MAXBREITE];

   if (zeile)
   {
      register struct Zeile *z;
```

```
register UBYTE *p1,*p2;
register UWORD w;
register ULONG pen;
UBYTE lc;

/* Pruefe, ob Zeile gerade bearbeitet wird: */
if (zeile->flags & ZLF_USED)
{
/* Puffer nach buf kopieren, da printAt diesen veraendert! */
   if (MAXBREITE > aktuellerEditor->leftpos)
   {
      w = MAXBREITE - aktuellerEditor->leftpos;
      p2 = aktuellerEditor->puffer +
                           aktuellerEditor->leftpos;

      p1 = buf;
      while (w)
      {
         *p1 = *p2;
         p1++;
         p2++;
         w--;
      }
   }
   else
      p1 = buf;

   /* Nun Rest mit lastchar fuellen: */
   p2 = buf + (w = aktuellerEditor->wcw);
   lc = aktuellerEditor->puflastchar;
   while (p1 < p2)
   {
      *p1 = lc;
      p1++;
   }
}
else
   convertLineForPrint(zeile+1,zeile->len,
                       w = aktuellerEditor->wcw,buf,&lc);

/* Falls Anfang/Ende-Markierung einer Falte => FOLDPEN */
if (zeile->flags & ZLF_FSE)
{
   if ((z = zeile->succ)->succ)
   {
      if ((z->flags & ZLF_FOLD) <= aktuellerEditor->maxfold)
      {
         SetAPen(aktuellerEditor->rp,FOLDPEN);
         pen = FOLDPEN;
      }
      else
         pen = FGPEN;
   }
   else
```

```
        {
           SetAPen(aktuellerEditor->rp,FOLDPEN);
           pen = FOLDPEN;
        }

        printAt(buf,w - spaltenDec,aktuellerEditor->xoff,y,pen);

        /* Alte Schreibfarbe wieder einstellen: */
        if (pen != FGPEN)
           SetAPen(aktuellerEditor->rp,FGPEN);
     }
     else
        printAt(buf,w - spaltenDec,
                    aktuellerEditor->xoff,y,FGPEN);
  }
  else
  {
     /* Falls keine zeile => Im Fenster loeschen: */
     struct RastPort *rp = aktuellerEditor->rp;

     SetAPen(rp,BGPEN);
     RectFill(rp,(ULONG)aktuellerEditor->xoff,
                 (ULONG)y,
                 (ULONG)aktuellerEditor->xscr,
                 (ULONG)y + aktuellerEditor->ch - 1);
     SetAPen(rp,FGPEN);
  }
}
```

Auch das Modul Cursor mußte sich einige Änderungen gefallen lassen. Neu ist die Funktion recalcTopOfWindow, die von einer beliebigen Zeilenposition ausgehend die oberste Zeile des Fensters bestimmt und in den Feldern zeilenptr und zeilennr abspeichert. Zweck dieser Funktion ist es, nach einer Änderung des Textes oder von minfold und maxfold das zeilenptr-Feld so zu initialisieren, daß der Cursor möglichst auf seiner alten Position stehen bleibt.

```
UWORD recalcTopOfWindow(y)
register UWORD          y;
{
   register struct Zeile *z,*zp;
   register UWORD znrp;
   UWORD znr,oldy = y;

   z  = ZEILENPTR(y);
   znr = ZEILENNR(y);
   while (y)
   {
      zp  = z;
```

```
      znrp = znr;
      if ((z = prevLine(z,&znr)) == NULL)
         break;
      y--;
   }

   if (y)
   {
      ZEILENPTR(0) = zp;
      ZEILENNR(0)  = znrp;
   }
   else
   {
      ZEILENPTR(0) = z;
      ZEILENNR(0)  = znr;
   }

   return (oldy - y);
}
```

Diese Funktion wird vor allem beim Austreten aus einer Falte und beim Löschen von Zeilen verwendet, damit der Cursor in der aktuellen Zeile stehenbleibt und der Benutzer ihn nicht erst großartig suchen muß. Zurückgegeben wird von dieser Funktion die neue Zeilenposition, die normalerweise mit der alten Zeilenposition übereinstimmt. Nun kann es aber vor allem nach dem Austreten aus einer Falte vorkommen, daß der Cursor relativ weit unten im Fenster stand, aber nicht soviel Text sichtbar ist. Dann muß der Cursor entsprechend nach oben verschoben werden, wofür die neue Position direkt von dieser Funktion übernommen werden kann.

Die Programmstücke zum Behandeln von Falten haben wir aus der Funktion handleKeys ausgelagert und in eigene Funktionen gepackt, um sie übersichtlicher zu machen:

```
/****************************
 *
 * enterFold:
 *
 * Tritt in eine Falte ein.
 *
 ****************************/

void enterFold()
{
   register struct Zeile *z;
```

```
   if (aktuellerEditor->maxfold < ZLF_FOLD)
   {
      aktuellerEditor->maxfold++;
      if (z = ZEILENPTR(aktuellerEditor->wdy))
         if ((z = z->succ)->succ)
            /* Wenn es eine nachfolgende Zeile gibt: */
            if (((ZEILENPTR(aktuellerEditor->wdy)->
                                    flags & ZLF_FOLD)
                 < (z->flags & ZLF_FOLD))
               && ((z->flags & ZLF_FOLD) ==
                                    aktuellerEditor->maxfold))
            {
               /* Und wenn diese Anfang einer neuen Falte ist: */
               aktuellerEditor->minfold =
                                 aktuellerEditor->maxfold;
               ZEILENPTR(0) = z;
               ZEILENNR(0)  = ZEILENNR(aktuellerEditor->wdy) + 1;
               aktuellerEditor->wdy = 0;
            }

      aktuellerEditor->wdy = recalcTopOfWindow
                           (aktuellerEditor->wdy);
      restoreZeilenptr();
      printAll();
      aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy);
   }
}

/*****************************
 *
 * exitFold:
 *
 * Tritt aus einer Falte aus.
 *
 *****************************/

void exitFold()
{
   register UWORD wdy;

   if (aktuellerEditor->maxfold)
   {
      aktuellerEditor->maxfold--;
      if (aktuellerEditor->minfold > aktuellerEditor->maxfold)
         aktuellerEditor->minfold = aktuellerEditor->maxfold;

      wdy = aktuellerEditor->wdy;
      if (ZEILENPTR(wdy) == NULL)
         aktuellerEditor->wdy = wdy = 0;

      if (ZEILENPTR(wdy))
         if (((ZEILENPTR(wdy)->flags & ZLF_FOLD)
              > aktuellerEditor->maxfold)
```

```
            || ((ZEILENPTR(wdy)->flags & ZLF_FOLD)
                < aktuellerEditor->minfold))

            ZEILENPTR(wdy) =
                    prevLine(ZEILENPTR(wdy),&(ZEILENNR(wdy)));

        aktuellerEditor->wdy = recalcTopOfWindow(wdy);
        restoreZeilenptr();
        printAll();
        cursorOnText();
    }
}
```

In der letzten Funktion sehen Sie auch, wie die Funktion recalcTopOfWindow eingesetzt wird. Zusätzlich zu den bisherigen Funktionen zur Cursor-Bewegung brauchen wir noch eine, die den Cursor auf den Anfang der Zeile setzt, da wir diese in der Funktion insertLine bereits verwendet haben:

```
BOOL cursorHome()
{
   aktuellerEditor->xpos = 1;
   if (aktuellerEditor->leftpos)
      scrollRight(aktuellerEditor->leftpos);

   return (TRUE);
}
```

Bevor wir nun handleKeys, das ebenfalls geändert wurde, auflisten, müssen wir noch einige Defines definieren, auf die wir in der Funktion zurückgreifen:

```
#define CHOME 'A'
#define UNDO '?'
#define CFOLD 6
#define CENDF 5
#define TABMODE 20
#define WRITEMODE 15
#define AUTOINDENT 1
#define DELLINE 2
#define DEL 127
#define BS 8
#define LF 10
#define CR 13
```

Doch nun zur Funktion handleKeys, die wir komplett auflisten, da sich dort einiges getan hat:

```
void handleKeys(buf,len)
register UBYTE *buf;
register WORD       len;
{
   register UBYTE first;

   while (len > 0)
   {
      first = *buf;
      buf++;
      len--;
      if ((first == CSI) && (len > 0))
      {
         len--;
         switch (*buf++)
         {
            case CUU:
               cursorUp();
               break;
            case CUD:
               cursorDown();
               break;
            case CUF:
               cursorRight();
               break;
            case CUB:
               cursorLeft();
               break;
            case SU:
               halfPageDown();
               break;
            case SD:
               halfPageUp();
               break;
            case ' ':
               if (len > 0)
               {
                  len--;
                  switch (*buf++)
                  {
                     case CHOME:
                        cursorHome();
                        break;
                     default:
                        if (! insertChar((UBYTE)CSI))
                           DisplayBeep(NULL);
                        buf -= 2;
                        len += 2;
                  }
                  break;
               }
               else
                  goto noCSI;
            case UNDO:
```

```
                if ((len > 0) && (*buf == '~'))
                {
                   buf++;
                   len--;
                   undoLine();
                   break;
                }
                /* Sonst: default */
             default:
noCSI:
                /* Keine Kommando-Sequenz! */
                if (! insertChar((UBYTE)CSI))
                   DisplayBeep(NULL);

                buf--;
                len++;
                break;
          } /* switch (first) */
       }
       else
          switch (first)
          {
             case CFOLD:
                enterFold();
                break;
             case CENDF:
                exitFold();
                break;
             case TABMODE:
                aktuellerEditor->tabs = !aktuellerEditor->tabs;
                printAll();
                break;
             case WRITEMODE:
                aktuellerEditor->insert = !aktuellerEditor->insert;
                break;
             case AUTOINDENT:
                aktuellerEditor->autoindent=
                                   !aktuellerEditor->autoindent;
                break;
             case DELLINE:
                if (! deleteLine())
                   DisplayBeep(NULL);
                break;
             case DEL:
                if (! deleteChar())
                   DisplayBeep(NULL);
                break;
             case BS:
                if (! backspaceChar())
                   DisplayBeep(NULL);
                break;
             case LF:
             case CR:
                if (! insertLine(first))
```

```
                DisplayBeep(NULL);
              break;
            default:
              if (! insertChar(first))
                DisplayBeep(NULL);

          } /* switch (first) */
      }   /* while (len) */
}
```

In der ersten Switch-Case-Anweisung werden wie bisher die CSI-Sequenzen behandelt. Ungünstig ist, daß diese CSI-Sequenzen nicht eine feste Länge haben, sondern daß einige zwei, andere drei Zeichen lang sind. In dieser Anweisung liegt eine weitere Switch-Case-Anweisung, die alle Sequenzen, die mit CSI ' ' beginnen, bearbeitet. Dies ist im Moment nur CSI ' ' 'A', was über Shift-Cursor-links abgeschickt wird. Wir können dies aber später noch erweitern, auch um selbst definierte Sequenzen.

Die zweite große Switch-Case-Anweisung behandelt die Control-Zeichen, die im Moment noch einige Funktionen aufrufen. Später werden wir einen Teil davon über die Kommandozeile aufrufen. Doch nun zunächst ein Überblick über die Tastenbelegung des Editors, damit Sie diesen dann gleich auch ausprobieren können:

Tasten	Kommentar
Cursor-Tasten	Bewegen Cursor um ein Zeichen/Zeile.
Shift-Cursor-hoch/runter	Scrolle Text um halbe Seite nach oben bzw. nach unten.
Shift-Cursor-links	Setze Cursor auf Zeilenanfang.
Help	Undo.
Ctrl-A	Schalte Auto-Indent an/ab.
Ctrl-B	Lösche Zeile, auf der der Cursor steht.
Ctrl-E	Trete aus einer Falte aus.
Ctrl-F	Trete in eine Falte ein.
Ctrl-O	Schalte zwischen Einfüge- und Überschreibmodus um.
Ctrl-T	Schalte Tabulatoren ein/aus.
Delete	Wie gewohnt.
Backspace	Wie gewohnt.

Die Funktionen, die aus dem neuen Modul Edit stammen, müssen Sie noch deklarieren, was wir jetzt fast vergessen hätten:

```
void undoLine();
void cursorOnText();
BOOL insertChar(),deleteChar(),backspaceChar(),
                                insertLine(),deleteLine();
```

Im Modul Speicher haben wir die Funktion loescheZeile dahingehend geändert, daß nun der Inhalt des zeilenptr- und zeilennr-Feldes nicht mehr geändert wird und daß die Zeile nicht aus der Liste aller Zeilen gelöscht wird; wir haben also folgende Befehle ersatzlos gestrichen:

```
/* Zeile aus Liste entfernen: */
Remove(zeile);

/* eventuelle Zeiger auf diese Zeile auf Null setzen: */
if (aktuellerEditor->aktuell == zeile)
   aktuellerEditor->aktuell = NULL;
for (n = 0, zptr = aktuellerEditor->zeilenptr;
     n <= aktuellerEditor->wch; n++, zptr++)
   if (*zptr == zeile)
   {
        *zptr = NULL;
        break;
   }
```

Dies war nötig, damit die Funktion deleteLine einwandfrei funktionieren kann. Nun noch ein paar Änderungen im Hauptmodul. In der Funktion OpenEditor müssen die zusätzlichen Elemente der Editor-Struktur auch initialisiert werden:

```
ed->puflen     = 0;
ed->pufypos    = 0;
ed->puflastchar= ' ';
ed->tabs       = 1;
ed->skipblanks = 1;
ed->autoindent = 1;
```

Ferner dürfte Ihnen schon beim Testen des Cursors aufgefallen sein, daß der Cursor vor allem beim horizontalen Scrollen nachläuft, das heißt, daß Sie die Taste längst losgelassen haben, der Cursor läuft aber immer noch weiter. Dieser Effekt tritt bei allen anderen Funktionen, die den Cursor bewegen oder Text einfügen, auch auf, aber erst, wenn Sie den Tastatur-Repeat auf sehr schnell stellen, also in Preferences den Schieber dafür ganz nach rechts schieben.

Dieser Effekt läßt sich nun relativ einfach unterbinden, indem wir, nachdem wir einen Tastendruck bearbeitet haben, überprüfen, ob schon wieder neue ins Haus stehen. Ist dies der Fall und haben alle RAWKEY-Messages das REPEAT-Flag gesetzt, so wird die erste Message entfernt. Das REPEAT-Flag wird vom Betriebssystem gesetzt, wenn eine RAWKEY-Message wegen des Tastatur-Repeats auftritt, weil der Benutzer die Taste gedrückt gehalten hat. Messages, bei denen das REPEAT-Flag nicht gesetzt ist, werden nicht gelöscht, da es unter Umständen sein könnte, daß der Benutzer Tasten schneller drückt, als unser Editor diese bearbeiten kann. In diesem Fall darf der Editor nicht einfach welche verschlucken.

Nun wird so lange die aktuelle Message gelöscht, bis nur noch eine übrig ist. Diese wird dann beim nächsten Durchlauf durch die Hauptschleife bearbeitet. Prinzipiell könnten wir auch die erste Message löschen, aber die Cursor-Bewegung wäre dann etwas ruckelig oder oszillierend, wie Sie es z.B. bei manchen Textverarbeitungen beobachten können. Den Vorteil unserer gleichmäßigen Cursor-Bewegung erkaufen wir uns allerdings mit dem kleinen Nachteil, daß unser Cursor schlimmstenfalls ein Zeichen weiter läuft als geplant.

Die RAWKEY-Abfrage sieht nun wie folgt aus:

```
case RAWKEY:
   {
      register struct IntuiMessage *im1,*im2;

      if (! (code & IECODE_UP_PREFIX))
      {
         inputEvent.ie_Code  = code;
         inputEvent.ie_Qualifier = qualifier;
         if ((inputLen = RawKeyConvert(
                        &inputEvent,inputPuffer,MAXINPUTLEN,NULL)
             ) >= 0)
           handleKeys(inputPuffer,inputLen);
      }

      /* Nachlaufen verhindern: */
      im1 = (struct IntuiMessage *)edUserPort->mp_MsgList.lh_Head;
      while (im2 = (struct IntuiMessage *)
                   im1->ExecMessage.mn_Node.ln_Succ)
      {
         if (im2->ExecMessage.mn_Node.ln_Succ == NULL) break;
```

```
        if (im1->Class != RAWKEY) break;
        if (!(im1->Qualifier & IEQUALIFIER_REPEAT)) break;
        if (im2->Class != RAWKEY) break;
        if (!(im2->Qualifier & IEQUALIFIER_REPEAT)) break;

        /* Message entfernen: */
        im1 = GetMsg(edUserPort);
        ReplyMsg(im1);

        im1 = (struct IntuiMessage *)
                    edUserPort->mp_MsgList.lh_Head;
    }

    break;
}
```

Als letztes zu erwähnen wäre, daß hinter der Switch-Case-Anweisung noch folgende Funktion aufgerufen werden muß:

```
saveIfCursorMoved();
```

Diese Funktion muß als void deklariert werden. Wenn Sie jetzt noch in der Make-Datei zur Liste der Objektdateien (OBJ) Edit.o hinzunehmen und folgende Zeile ans Ende der Make-Datei anfügen:

```
src/Edit.o: src/Edit.c src/Editor.h /pre/Editor.pre
```

sind wir auch schon fertig. Alle Quelltexte und den bereits übersetzten Editor finden Sie im Verzeichnis V0.5 auf der Diskette zum Buch. Beginnen Sie also gleich mit dem Ausprobieren. Sie haben diesmal ziemlich viel zu tun, da durch die vielen Optionen einiges zu testen ist. Einen Fehler, der uns bei der Programmentwicklung untergekommen ist, haben wir auch diesmal wieder für Sie im Programm gelassen. Aber suchen Sie doch erstmal selbst danach.

Haben Sie ihn gefunden? Wenn man im Überschreibmodus am Ende einer Zeile Zeichen anfügen will, so vergißt der Editor diese, sobald man die Zeile wieder verläßt. Nehmen wir an, wir haben uns den Quelltext bereits mehrfach angesehen, können aber keine fehlerhafte Stelle entdecken. Hier hilft dann nur noch der Debugger. Starten Sie also den Debugger DB und danach den Editor, so, wie wir es bereits zuvor beschrieben haben:

```
sys3:bin/db
Editor <TestText
```

Beachten Sie dabei, daß der Editor mittels "make Debug" erzeugt worden sein sollte, so wie der Editor, der sich im Verzeichnis V0.5 der Diskette zum Buch befindet. Der Debugger fängt den Editor dann wie gewohnt ab. Nun gilt es zu überlegen, wie wir weiter vorgehen. Der Fehler tritt auf, wenn wir ein Zeichen am Ende einer Zeile anfügen wollen, also voraussichtlich in der Funktion insertChar. Setzen Sie also zunächst einen Breakpoint auf diese Funktion:

```
bs insertChar
```

und starten Sie dann den Editor mittels g (Go). Aktivieren Sie das Editorfenster, und drücken Sie nun Ctrl-O, um in den Überschreibmodus zu gelangen. Suchen Sie sich dann eine passende Zeile aus, am besten eine, in der noch gar nichts steht. Bewegen Sie den Cursor dorthin, und drücken Sie eine Taste, also z.B. das a. Der Editor wird unterbrochen, und der Debugger zeigt Ihnen den Beginn des Unterprogramms insertChar an. Jetzt brauchen Sie nur noch den Fehler zu finden! Die Ursache dafür könnte z.B. darin liegen, daß die Länge des Puffers nicht korrekt erhöht wird, wenn Sie ein Zeichen anfügen wollen. Am besten, Sie gehen nun im Einzelschrittmodus durch diese Funktion, vielleicht fällt Ihnen dabei ja etwas auf.

Zuerst kommt die übliche Initialisierung der Funktion mit LINK und MOVEM. Dann wird das einzufügende Zeichen nach Register D4 geschrieben, es handelt sich ja um eine Registervariable. Weiterhin wird die lokale Variable -a(A5) mit dem Wert Eins gefüllt, was darauf schließen läßt, daß es sich dabei um xadd handelt. Dann wird getPufferPointer aufgerufen, was Sie bitte im Debugger mittels t überspringen, anstatt den Befehl s zu benutzen. Im Gegensatz zum Einzelschrittbefehl s führt t nämlich einen Befehl komplett aus, also wird bei einem Unterprogrammaufruf mittels BSR oder JSR das gesamte Unterprogramm ausgeführt, bevor der Debugger wieder die Kontrolle über das Programm erhält.

Nun sehen wir uns die Werte an, die in die Variablen ptr, inc und rest geschrieben wurden:

```
p2dX a5-8
```

Wie kommen wir denn nun darauf? Vor dem Aufruf der Funktion getPufferPointer wurden drei Adressen auf den Stack gelegt, die die Adressen der drei gesuchten Variablen sind:

```
-8(a5)
-6(a5)
-4(a5)
```

Da in C Variablen beim Funktionsaufruf immer in der Reihenfolge auf dem Stack liegen, in der sie definiert wurden, können wir folgende Zuordnung treffen:

```
rest = -8(a5)
inc  = -6(a5)
ptr  = -4(a5)
```

Diese wollen wir nun ausgeben lassen. Dazu ist es am einfachsten, ab Adresse -8(a5) zuerst zwei Wörter und dann ein Langword ausgeben zu lassen, was in der Syntax des Debuggers p2dX heißt, also Print zweimal Dezimal und einmal Hexadezimal-Langwort, gefolgt von der Anfangsadresse. Die Werte für inc und rest sind beide Null, wenn Sie den Cursor auf eine leere Zeile bewegt hatten. Der Wert von ptr ist uninteressant, aber lassen Sie sich doch mal zeigen, worauf ptr zeigt:

```
db *(a5-4)
```

Sie sehen nun eine Zeile im Debugger-Fenster, die aus einer Adresse, einem Gleichheitszeichen und lauter "20" besteht. Der ptr zeigt also auf einen String, der nur aus Blanks (= $20) besteht. Doch lassen Sie den Debugger nun weiter einzelschrittweise laufen, denn bis jetzt war ja alles richtig. Es folgt das Testen des Rückgabewertes, und dann wird überprüft, ob D4 den Wert 9 enthält. D4 ist das einzufügende Zeichen, und 9 entspricht einem Tabulator. Da Sie aber einen normalen Buchstaben gedrückt haben, springt das Programm weiter.

Jetzt wird ein Wert aus der Editor-Struktur geholt und davon das erste Bit ausmaskiert und getestet. War dies Null, so befindet sich der Editor im Überschreibmodus, und der Editor springt wieder weiter. Sie haben hoffentlich das Treiben des Debuggers mit dem Programmtext verglichen und wissen nun, wo wir sind. Wenn Sie sich das Programm von dieser Stelle ab disassemblieren lassen, so sollten Sie folgendes sehen:

```
_insertChar+184    movea.l -4(a5),a0
_insertChar+188    move.b  d4,(a0)
_insertChar+18a    clr.w   -8(a5)
_insertChar+18e    beq.s   _insertChar+19e
_insertChar+190    movea.l _aktuellerEditor,a0
_insertChar+194    addq.w  #1,76(a0)
_insertChar+198    move.w  #1,-8(a5)
_insertChar+19e
```

Dieses kurze Programmstück entspricht folgenden Zeilen, die in unserer Funktion insertChar für das Einfügen von Zeichen im Überschreibmodus sorgen und die Sie am Ende der Funktion finden:

```
*ptr = c;
if (rest = 0)
{
   aktuellerEditor->puflen++;
   rest = 1;
}
```

Zuerst wird also das Zeichen an die Stelle abgespeichert, auf die ptr zeigt. Dann sollte rest auf Null getestet werden, statt dessen wird die Speicherstelle -8(a5) auf Null gesetzt. Und hier liegt auch der Fehler. Statt (rest = 0) muß es (rest == 0) heißen. Diese Eigenart von C, Zuweisungen quasi überall hinschreiben zu können, also auch in Abfragen, erweist sich hier als Stolperstein. Aber damit muß man leben. Außerdem lassen sich dadurch auch recht trickreiche Befehle realisieren.

Nun können Sie den Fehler korrigieren und das Programm nochmals übersetzen, wenn Sie wollen. Wir wollen uns an dieser Stelle einer Sache zuwenden, deren Sinn und Zweck nicht unbedingt einleuchtet, wenn man sich damit nicht schon näher befaßt hat.

Gemeint sind lokale Variablen in Blöcken. Was ein Block ist, wissen Sie ja: eine Folge von Befehlen, die durch geschweifte Klammern eingefaßt sind. Lokale Variablen definiert man für gewöhnlich am Anfang eines Blocks. Wir haben lokale Variablen immer nur im Zusammenhang mit Funktionen in Programmen gefunden. Es wurden also zu Beginn einer Funktion deren lokale Variablen definiert. Dabei kann man in jedem Block lokale Variablen definieren. Nun fragen Sie wahrscheinlich, und nicht ganz zu Unrecht, welchen Sinn dies haben sollte.

Solange es sich bei den Variablen nicht um Register-Variablen handelt, haben lokale Variablen eigentlich nur den Zweck, die Variablenstruktur etwas übersichtlicher zu gestalten. Das Problem mit Register-Variablen ist, daß Sie nur eine begrenzte Anzahl von Registern zur Verfügung haben, in der Regel 5 bis 7. Dieses läßt sich durch lokale Variablen in Blöcken geschickt umgehen. Dazu definieren Sie Register-Variablen nur in dem Block, in dem Sie diese wirklich benötigen; notfalls packen Sie alle betroffenen Anweisungen in einen neuen Block. Außerhalb des Blockes sind die entsprechenden Register nun frei und können vom C-Compiler anderweitig vergeben werden. Ihr Programm wird dadurch schneller, da mehr Register-Variablen genutzt werden können. Betrachten Sie dazu die beiden Funktionen slow und fast:

```
slow()
{
   register int a,b,c;
   register int a1,b1,c1;
   register int a2,b2,c2;

   for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
      ;

   for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
      ;

}

fast()
{
   register int a,b,c;

   {
```

```
      register int a1,b1,c1;

      for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
         ;
   }

   {
      register int a2,b2,c2;
      for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
         ;
   }

}
```

Diese erfüllen beide denselben Zweck, auch wenn dieser sinnlos ist. Es kommt hier nur darauf an, daß relativ viele Register-Variablen verwendet werden. Compilieren Sie diese Funktionen, so erhalten Sie folgenden Assembler-Quelltext:

```
                        ;slow()
                        ;{
     public _slow
_slow:
     link a5,#.2
     movem.l .3,-(sp)
                        ; register int a,b,c;
                        ; register int a1,b1,c1;
                        ; register int a2,b2,c2;
                     ;
                     ; for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
     move.w d4,d7
     move.w d5,a2
     move.w d6,a3
     bra .7
.6
                        ; ;
.4
     sub.w #1,d7
.7
     tst.w d7
     bgt .6
.5
                     ;
                     ; for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
     move.w d4,-2(a5)
     move.w d5,-4(a5)
     move.w d6,-6(a5)
     bra .11
.10
                        ; ;
.8
     sub.w #1,-2(a5)
```

```
.11
     tst.w -2(a5)
     bgt .10
.9
                              ;
                              ;}
.12
     movem.l (sp)+,.3
     unlk a5
     rts
.2   equ -6
.3   reg d4/d5/d6/d7/a2/a3
                              ;
                              ;fast()
                              ;{
     public _fast
_fast:
     link a5,#.13
     movem.l .14,-(sp)
                             ; register int a,b,c;
                             ;
                        ; {
                        ;  register int a1,b1,c1;
                        ;
                        ;  for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
     move.w d4,d7
     move.w d5,a2
     move.w d6,a3
     bra .18
.17
                              ;   ;
.15
     sub.w #1,d7
.18
     tst.w d7
     bgt .17
.16
                        ; }
                        ;
                        ; {
                        ;  register int a2,b2,c2;
                        ;
                        ;  for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
     move.w d4,d7
     move.w d5,a2
     move.w d6,a3
     bra .22
.21
                              ;   ;
.19
     sub.w #1,d7
.22
     tst.w d7
     bgt .21
```

```
.20
                        ; }
                        ;
                        ;}
.23
    movem.l (sp)+,.14
    unlk a5
    rts
.13 equ 0
.14 reg d4/d5/d6/d7/a2/a3
    dseg
    end
```

Hier kommt es jetzt nur auf die Umsetzung der Zuweisungen an, da man an diesen erkennen kann, welche Variablen Register-Variablen sind und welche nicht. In der Funktion slow sehen Sie in der ersten For-Schleife lauter Register, aber in der zweiten For-Schleife werden die Werte der Register in den Speicher geschrieben. Hier wurden zu viele Register-Variablen definiert, so daß der C-Compiler einige davon als ganz normale Variablen umsetzen mußte. Anders dagegen die Funktion fast: Hier sind alle Variablen Register-Variablen, was sich angenehm auf die Laufzeit dieser Funktion auswirken würde.

Wir haben in den neuen Funktionen, die zu unserem Editor hinzugefügt wurden, von der Möglichkeit, lokale Variablen in Blöcken zu definieren, bereits Gebrauch gemacht. Wir werden dies auch in Zukunft tun, damit der Editor recht flott wird. Experimentieren Sie doch auch mal damit. Vielleicht fallen Ihnen auch noch andere Möglichkeiten ein, ein Programm schneller zu machen, ohne daß der Programmtext unlesbar wird. Das Einbinden von Assembler-Unterprogrammen wäre eine solche Möglichkeit, die sich aber nur für denjenigen eignet, der sich mit Assembler ein bißchen auskennt. Denn ein Vorteil von Assembler ist, daß Sie jederzeit mit 15 Registern jonglieren und deren Bedeutung für das Programm an jeder Stelle ändern können, sofern der ursprüngliche Inhalt des Registers nicht mehr benötigt wird!

4.3.9 Kommandozeile

Die Möglichkeit, Texte abzuspeichern und zu laden, haben wir bis jetzt an unserem Editor schmerzlich vermißt. Da wir dies nicht über Control-Codes veranlassen können - schließlich müssen wir einen Dateinamen angeben -, brauchen wir eine sogenannte Kommandozeile, wie sie auch Ed aufweist.

Ferner wollen wir bei der Gelegenheit auch gleich eine Status- oder Infozeile einbauen, in der der Editor die Cursor-Position und andere Informationen anzeigt.

Doch zunächst wollen wir uns mit der Kommandozeile beschäftigen. Diese soll in etwa so funktionieren wie die von Ed, die Aktivierung soll durch Escape geschehen und sie soll vorwiegend Befehle enthalten, die sich aus zwei Buchstaben zusammensetzen. Bei Ed hat uns aber gestört, daß Befehle, die zu einer Gruppe gehören, also z.B. Blockbefehle, nicht immer gleich aufgebaut sind. So lautet der Befehl zum Markieren des Blockanfangs BS, aber zum Einfügen des Blocks in den Text muß man IB eingeben statt BI oder BC (Block Copy). Ed zählt IB zu den Insert-Befehlen, was sicher auch richtig ist, aber über Geschmack läßt sich ja bekanntlich nicht streiten.

Wir wollen unsere Befehle so benennen, daß deren erster Buchstabe Auskunft über die Befehlsgruppe gibt, zu der dieser Befehl gehört. Überlegen wir uns nun, welche Befehlsgruppen uns für unseren Editor einfallen.

Da wären zuerst die bereits erwähnten Block-Befehle. Ferner brauchen wir Befehle zur Cursor-Steuerung, zum Abspeichern und Laden, zum Setzen der Flags, zum Löschen, zum Suchen und Ersetzen, und um das Programm wieder zu verlassen.

Auch brauchen wir Befehle um den Editor programmieren zu können; Sie erinnern sich: Wir sprachen zu Beginn der Programmplanung darüber. Im folgenden nun eine Auflistung der Befehle, die uns eingefallen sind:

ASCII-Befehle

AF ["Dateiname"]	Abspeichern des Textes einer Falte.
AL ["Dateiname"]	Laden eines Textes.
AS ["Dateiname"]	Abspeichern des gesamten Textes.

Block-Befehle

BC (Block Copy)	Block kopieren.
BD (Block Delete)	Block löschen.
BE (Block End)	Blockende markieren.
BF (Block Fold)	Block wegfalten.
BH (Block Hide)	Markierung löschen.
BM (Block Move)	Block verschieben.
BP (Block Paste)	Block aus Puffer kopieren.
BS (Block Start)	Blockanfang markieren.
BX (Block Cut)	Block in Puffer kopieren (ausschneiden).

Cursor-Befehle

CB (Cursor Bottom)	Cursor auf Text-/Faltenende.
CD (Cursor Down)	Cursor um eine Zeile nach unten.
CE (Cursor End)	Cursor hinter Zeilenende setzen.
CH (Cursor Home)	Cursor auf Zeilenanfang setzen.
CL (Cursor Left)	Cursor ein Zeichen nach links.
CN (Cursor Next)	Cursor auf Anfang von nächster Zeile.
CP (Cursor Previous)	Cursor auf Anfang von voriger Zeile.
CR (Cursor Right)	Cursor ein Zeichen nach rechts.
CS (Cursor Start)	Cursor auf Zeilenanfang setzen.
CT (Cursor Top)	Cursor auf Text-/Faltenanfang.
CU (Cursor Up)	Cursor um eine Zeile nach oben.

Delete-Befehle

DB (Delete Back)	Backspace.
DC (Delete Char)	Zeichen löschen (Delete).
DL (Delete Line)	Zeile löschen.

Exchange-Befehle (Ersetzen)

EN ["String1"String2"]	Ersetze nächstes Vorkommen.
EP ["String1"String2"]	Ersetze voriges Vorkommen.

Find-Befehle (Suchen)

FN ["String"]	Suche nächstes Vorkommen von String.
FP ["String"]	Suche rückwärts.

IF-Befehl

IF (Bedingung) Anweisung	

Line-Befehle (Zeilen)

LA "String"	Füge String als neue Zeile hinter (Append) aktueller Zeile in den Text ein.
LI "String"	Füge String als neue Zeile vor (Insert) aktueller Zeile in den Text ein.
LJ (Line Join)	Füge Zeile mit folgender zusammen.
LS (Line Split)	Trenne Zeile an Cursor-Position auf.

Makro-Befehle

M[A-Z] [=Anweisungsfolge]	Makro ausführen bzw. setzen.

Quit-Befehl

Q	Verlasse Editor.

Repeat-Befehl

RP Anweisung	Wiederhole Anweisung bis Abbruch.

Set-Befehle

Sa	Auto-Indent aus.
SA	Auto-Indent ein.
Sb	SkipBlanks aus.
SB	SkipBlanks ein.
Si	Einfügemodus (Insert).
So	Überschreibmodus (Overwrite).
St	Tabulatoren aus.
ST	Tabulatoren ein.

Tabulator-Befehle

TC (Tab Clear)	Lösche Tabulator an Cursor-Position.
TS (Tab Set)	Setze Tabulator an Cursor-Position.

eXit-Befehl

X	Text abspeichern, falls verändert, und Programm beenden.

Undo-Befehl

U	Änderungen in aktueller Zeile rückgängig machen.

Außer bei den Set-Befehlen kommt es übrigens nicht darauf an, ob ein Befehl groß oder klein geschrieben wurde! Wir werden allerdings nicht alle diese Befehle auf einmal einbauen, sondern zunächst nur die wichtigsten: das Abspeichern und Laden von Text sowie die Befehle, für die wir bereits Funktionen programmiert haben, wie z.B. die Cursor-Bewegung.

Um die Befehle eingeben zu können, brauchen wir eine Kommandozeile, die unabhängig von den normalen Editierfunktionen arbeitet. Wir werden dafür ein String-Gadget benutzen, das wir in die untere Kante des Fensters legen. Damit wir es nicht jedesmal mit der Maus aktivieren müssen, lassen wir es vom Editor aktivieren, sobald die ESC-Taste gedrückt wird. Sie werden dann in der Bedienung keinen Unterschied zu dem Editor Ed bemerken, außer dem, daß Sie in der Kommandozeile auch editieren können.

Nun muß der Editor die Befehle der Kommandozeile ausführen, sobald wir diese mit Return bestätigt haben. Damit der Editor dies bemerkt, setzen wir das RELVERIFY-Flag. Der Editor bekommt dann jedesmal eine GADGETUP-Message, wenn Sie im String-Gadget Return drücken. Er muß dann entscheiden, welcher Befehl eingegeben wurde, und die entsprechende Funktion aufrufen. Da wir eine ganze Menge an Befehlen haben, wäre es zu umständlich, die Kommandozeile mit allen bekannten Befehlen zu vergleichen. Aber unsere Befehle sind ja nach Gruppen sortiert, und alle Befehle einer Gruppe beginnen mit demselben Buchstaben. Wir werden also zunächst nach dem ersten Buchstaben der Kommandozeile die Befehlsgruppe bestimmen, zu der der Befehl gehört, und dann eine entsprechende Funktion aufrufen, die bestimmt, um welchen Befehl es sich handelt, und die diesen dann ausführt.

An diese Funktion wird ein Zeiger auf den zweiten Buchstaben des Befehls übergeben, da der erste ja bereits erkannt wurde. Wenn die Funktion den Befehl ausführen konnte, so gibt diese einen Zeiger auf das Zeichen hinter dem Befehl zurück, andernfalls eine entsprechende Fehlermeldung. Aufgerufen werden diese Funktionen von der Funktion executeCommand aus. Diese überspringt nicht nur Blanks und überprüft, ob zwei Befehle

auch wirklich durch ein Semikolon voneinander getrennt wurden, sondern führt Befehle mehrfach aus, wenn eine Zahl vor dem Befehl stand.

Auch ist es erlaubt, mehrere Befehle durch geschweifte Klammern zu klammern und vor diese Klammern eine Zahl zu schreiben. Die eingeklammerten Befehle werden dann so oft ausgeführt, wie die Zahl angibt. Folgende Befehlszeile würde z.B. die aktuelle und weitere neun Zeilen um jeweils drei Zeichen nach links rücken:

```
CS;10{3DC;CD};10CU
```

Anschließend würde der Cursor wieder auf die Ursprungszeile gesetzt werden. Aber diese Funktion muß noch zwei weitere Dinge beachten: Nachdem ein Befehl ausgeführt worden ist, muß die Funktion saveIfCursorMoved aufgerufen werden, da der Cursor ja möglicherweise bewegt worden ist. Ferner muß eine Befehlsfolge abbrechbar sein, damit man im Falle einer Fehlprogrammierung nicht zuviel von seinem Text verliert. Abgebrochen wird, sobald der Benutzer irgendeine Taste gedrückt hat.

Wenden wir uns jetzt der Infozeile zu. Damit diese nicht eine ganze Zeile belegt, werden wir diese neben die Kommandozeile in die untere Fensterkante legen. Damit sie nicht zu breit wird, wollen wir uns auf die wichtigsten Anzeigen beschränken: X- und Y-Position des Cursors, Anzahl der Zeilen, minfold, maxfold und die Flags. Schränken wir die maximale Breite für die einzelnen Zahlen auf vier Zeichen ein (zwei Zeichen für minfold bzw. maxfold), so kommen wir mit 34 Zeichen aus:

```
X=nnnn Y=nnnn #=nnnn [nn,nn] ICATB
```

wobei # für die Anzahl der Zeilen steht. In der eckigen Klammer entspricht die erste Zahl minfold und die zweite maxfold. Bei den Flags werden kleine Buchstaben angezeigt, wenn diese nicht gesetzt sind, und große, wenn doch. Dabei gilt folgende Zuordnung zwischen Buchstaben und Flags:

```
I: insert (0 = Overwrite).
C: changed.
A: autoindent.
T: tabs.
B: skipblanks.
```

Damit sich die Infozeile nicht mit der Kommandozeile überschneidet, wenn das Fenster verkleinert wird, setzen wir die Breite der Kommandozeile relativ zur Fensterbreite. Dann müssen wir die minimale Breite des Fensters allerdings heraufsetzen, da sonst die Kommandozeile eine negative Breite bekommen würde, wenn man das Fenster zu klein machen würde.

Um nun die Werte in der Infozeile auszugeben, können wir nicht jedesmal die gesamte Infozeile neu ausgeben, da dies zu lange dauern würde. Statt dessen schreiben wir Funktionen, die jeweils einen Wert bzw. eine Gruppe von Werten ausgeben. Da es vor allem bei der Ausgabe der Cursor-Position auf Schnelligkeit ankommt, wollen wir diese Gelegenheit nutzen, um ein Unterprogramm in Assembler zu schreiben und in unser Programm einzubinden. Damit das Assembler-Programm nicht allzu schwierig wird, wollen wir die Konvertierung einer Zahl in eine Zeichenkette in Assembler programmieren.

Die C-Bibliothek stellt uns für diesen Zweck zwar die Funktion ftoa zur Verfügung, diese verlangt aber eine Floating-Point-Zahl. Der C-Compiler würde also unsere Integer-Zahl erst in eine Floating-Point-Zahl umwandeln und diese dann konvertieren. Unsere Funktion wird dagegen direkt eine Integer-Zahl konvertieren, wobei wir auch noch festlegen können, wie lang die Zeichenkette maximal sein darf.

Das Einbinden von Assembler-Programmen ist nun beim Aztec kein Problem. Statt des Funktionskopfes schreiben wir zuerst die Compileranweisung #asm an den Anfang einer Zeile. Dann geben wir unser Assembler-Programm ein und beenden dieses mit einem #endasm. Der Compiler weiß dann, daß zwischen #asm und #endasm ein Assembler-Programm liegt und überliest dies bei der Übersetzung. Erst beim nachfolgenden Übersetzen mit dem Assembler, was ja vom C-Compiler automatisch gestartet wird, wird das Assembler-Programm mit übersetzt. Damit wir das Unterprogramm auch von unseren C-Funktionen aus aufru-

fen können, müssen wir die Adresse des Unterprogramms dem Linker bekannt machen. Wollen wir unser Unterprogramm mit dem Namen cn_utoa aufrufen (CoNvert Unsigned TO Ascii), so müssen wir folgende Zeilen an den Anfang des Unterprogramms schreiben:

```
    global _cn_utoa
_cn_utoa:
```

Der Befehl global macht das Label des Unterprogramms _cn_utoa dem Linker bekannt, so daß darauf wie auf eine globale Variable oder eine Funktion zugegriffen werden kann. Der Unterstrich vor dem Funktionsnamen ist deswegen nötig, weil der C-Compiler, wenn er ein C-Programm in ein Assembler-Programm übersetzt, vor alle Bezeichner einen Unterstrich setzt. Wenn Sie sich bereits angesehen haben, was für Assembler-Programme der C-Compiler erzeugt, so haben Sie dies sicher schon bemerkt.

Die Übergabe der Parameter ist dabei so, wie Sie es erwarten würden: Die Parameter, die zuerst definiert werden, liegen oben auf dem Stack. Wenn wir unsere Funktion mit:

```
cn_utoa(string,wert,laenge);
```

aufrufen, so fänden wir die Parameter an folgenden Adressen auf dem Stack, wenn wert und laenge vom Typ UWORD sind:

```
0(SP) ^ Ruecksprungadresse.
4(SP) ^ string.
8(SP) = wert.
10(SP) = laenge.
```

Bei der Verwendung der Register in unserem Assembler-Programm dürfen wir nicht alle Register frei verwenden. Die Register, die der C-Compiler für Verweise auf lokale (A5) und globale (A4) Variablen benutzt, dürfen ebensowenig verändert werden wie die Register, die bei Registervariablen verwendet werden (D4-D7, A2, A3). Aber uns bleiben noch genug Register übrig, vor allem, weil wir für unser Unterprogramm nur vier Stück benötigen. Hier nun unser Assembler-Unterprogramm:

```
#asm
   public _cn_utoa

_cn_utoa:

   ; 4(sp) ^ Puffer,
   ; 8(sp) = Wert,
   ;10(sp) = maximale Laenge.

   lea      4(sp),a0
   move.l   (a0)+,a1
   moveq    #0,d0
   move.w   (a0)+,d0        ; = Wert
   move.w   (a0),d1         ; = maximale Laenge
   lea      0(a1,d1.w),a0   ; ^ Ende des Puffers
   bra.s    .in

.lp:
   divu     #10,d0
   swap     d0
   add.b    #'0',d0         ; d0.w = Wert MOD 10
   move.b   d0,-(a0)
   clr.w    d0
   swap     d0              ; Zero-Flag ist 1, wenn d0.l = 0
.in:
   dbeq     d1,.lp          ; solange bis Wert = 0 oder String voll
   beq.s    .ib             ; Wert = 0 =>
                            ; Rest mit Blanks auffuellen
   bra.s    .r              ; String voll!

.bl:
   move.b   #' ',-(a0)
.ib:
   dbra     d1,.bl
.r:
   rts

#endasm
```

Das Programm arbeitet so, daß der String von hinten gefüllt wird. Dabei wird die Zahl, die konvertiert werden soll, jedesmal durch zehn dividiert, und der Rest, der bei der Division übriggeblieben ist, wird als Ziffer in den String geschrieben. Zum Schluß wird noch der Rest des Strings mit Blanks aufgefüllt, sofern dafür noch Platz ist.

Um die entsprechenden Werte in der Infozeile ausgeben zu können, benötigen wir noch die Position, an der diese ausgegeben werden müssen. Dazu addieren wir auf die Position des Gadgets, zu dem die Infozeile gehört, die Position der IntuiText-Struktur,

die die Infozeile enthält. Damit die Infozeile möglichst schmal wird, werden wir in der IntuiText-Struktur den Zeichensatz festlegen, in dem die Infozeile ausgegeben werden soll. Hier empfiehlt sich Topaz80, der 80 Zeichen pro Zeile erlaubt.

Nun müssen wir bei der Ausgabe der Werte aber auch diesen Zeichensatz wählen. Wenn wir die Intuition-Funktion PrintIText verwenden würden, so wäre dies kein Problem, da wir dann in der IntuiText-Struktur, die diese Funktion verlangt, den Zeichensatz angeben könnten. Allerdings verlangt diese Funktion ein Null-Byte am String-Ende. Besser wäre es, wenn wir die Funktion Text aus der Graphics-Library benutzen, da man bei dieser die Länge der Zeichenkette angeben kann. Sie müssen bedenken, daß wir den Wert auch in den String der Infozeile kopieren müssen, da beim Neu-Ausgeben des Fensters auch dessen Umrandung neu ausgegeben wird. Würden wir nur den neuen Wert mittels "PrintIText" oder "Text" im Fenster über dem alten ausgeben, so erschiene beim Neu-Ausgeben wieder der alte Wert, da sich dieser immer noch im String befände.

Wir werden also den neuen Wert (xpos, ypos, etc.) in die Infozeile kopieren bzw. konvertieren und dann den entsprechenden Ausschnitt der Infozeile mittels "Text" ausgeben. Dazu müssen wir aber zuvor den Zeichensatz mit SetFont einstellen. Und dafür wiederum müssen wir im Hauptprogramm den Zeichensatz mit OpenFont öffnen, damit wir überhaupt auf diesen zugreifen können. Die Funktionen zur Ausgabe der Werte der Infozeile sehen wie folgt aus (Das Assembler-Programm __cn_utoa sollten Sie vor diesen Funktionen in das Modul Ausgabe schreiben!):

```
/*****************************
 *
 * printXpos:
 *
 * Gibt Xpos des Cursors aus.
 *
 *****************************/

void printXpos()
{
   register struct TextFont *oldFont;
   register struct Editor *ae = aktuellerEditor;
```

```
    cn_utoa(ae->gi_Text + INFO_XPOS_INC,
                          ae->xpos,(UWORD)INFO_XPOS_LEN);

    oldFont = ae->rp->Font;
    SetFont(ae->rp,infoFont);
    Move(ae->rp,ae->ip_xpos,ae->ip_topedge);
    Text(ae->rp,ae->gi_Text + INFO_XPOS_INC,(ULONG)INFO_XPOS_LEN);
    SetFont(ae->rp,oldFont);
}

/*****************************
 *
 * printYpos:
 *
 * Gibt Ypos des Cursors aus.
 *
 *****************************/

void printYpos()
{
    register struct TextFont *oldFont;
    register struct Editor *ae = aktuellerEditor;

    cn_utoa(ae->gi_Text + INFO_YPOS_INC,
                          ae->ypos,(UWORD)INFO_YPOS_LEN);

    oldFont = ae->rp->Font;
    SetFont(ae->rp,infoFont);
    Move(ae->rp,ae->ip_ypos,ae->ip_topedge);
    Text(ae->rp,ae->gi_Text + INFO_YPOS_INC,(ULONG)INFO_YPOS_LEN);
    SetFont(ae->rp,oldFont);
}

/*****************************
 *
 * printAnzZeilen:
 *
 * Gibt gesamte Anzahl der
 * Zeilen aus, aus denen der
 * Text besteht.
 *
 *****************************/

void printAnzZeilen()
{
    register struct TextFont *oldFont;
    register struct Editor *ae = aktuellerEditor;

    cn_utoa(ae->gi_Text + INFO_ANZZEILEN_INC,ae->anz_zeilen,
            (UWORD)INFO_ANZZEILEN_LEN);

    oldFont = ae->rp->Font;
    SetFont(ae->rp,infoFont);
    Move(ae->rp,ae->ip_anzzeilen,ae->ip_topedge);
```

```
   Text(ae->rp,ae->gi_Text + INFO_ANZZEILEN_INC,
        (ULONG)INFO_ANZZEILEN_LEN);
   SetFont(ae->rp,oldFont);
}

/*********************************
 *
 * printFold:
 *
 * Gibt minfold und maxfold aus.
 *
 *********************************/

void printFold()
{
   register struct TextFont *oldFont;
   register struct Editor *ae = aktuellerEditor;

   cn_utoa(ae->gi_Text + INFO_MINFOLD_INC,ae->minfold,
           (UWORD)INFO_MINFOLD_LEN);
   cn_utoa(ae->gi_Text + INFO_MAXFOLD_INC,ae->maxfold,
           (UWORD)INFO_MAXFOLD_LEN);

   oldFont = ae->rp->Font;
   SetFont(ae->rp,infoFont);
   Move(ae->rp,ae->ip_minfold,ae->ip_topedge);
   Text(ae->rp,ae->gi_Text +
             INFO_MINFOLD_INC,(ULONG)INFO_MINFOLD_LEN);
   Move(ae->rp,ae->ip_maxfold,ae->ip_topedge);
   Text(ae->rp,ae->gi_Text +
             INFO_MAXFOLD_INC,(ULONG)INFO_MAXFOLD_LEN);
   SetFont(ae->rp,oldFont);
}

/*****************************
 *
 * printFlags:
 *
 * Gibt Ypos des Cursors aus.
 *
 *****************************/

void printFlags()
{
   register struct TextFont *oldFont;
   register struct Editor *ae = aktuellerEditor;
   register UBYTE *fs = ae->gi_Text + INFO_FLAGS_INC;

   if (ae->insert)
      *fs++ = 'I';
   else
      *fs++ = 'O';
   if (ae->changed)
      *fs++ = 'C';
```

```
    else
        *fs++ = 'c';
    if (ae->autoindent)
        *fs++ = 'A';
    else
        *fs++ = 'a';
    if (ae->tabs)
        *fs++ = 'T';
    else
        *fs++ = 't';
    if (ae->skipblanks)
        *fs = 'B';
    else
        *fs = 'b';

    oldFont = ae->rp->Font;
    SetFont(ae->rp,infoFont);
    Move(ae->rp,ae->ip_flags,ae->ip_topedge);
    Text(ae->rp,ae->gi_Text +
                    INFO_FLAGS_INC,(ULONG)INFO_FLAGS_LEN);
    SetFont(ae->rp,oldFont);
}

/*****************************
 *
 * printInfo:
 *
 * Gibt ganze Infozeile aus.
 *
 *****************************/

void printInfo()
{
    printXpos();
    printYpos();
    printAnzZeilen();
    printFold();
    printFlags();
}
```

Alle Funktionen sind eigentlich gleich aufgebaut: Zuerst wird der neue Wert in den String kopiert, dann wird auf den Zeichensatz umgeschaltet und der Teil der Infozeile neu ausgegeben, der geändert wurde. Anschließend wird wieder auf den normalen Zeichensatz zurückgeschaltet. Die Positionen der einzelnen Werte werden dabei einmal aus den Defines und zum anderen aus ein paar neuen Elementen der Editor-Struktur berechnet. Diese neuen Elemente, die die Pixel-Positionen aller Werte beinhalten, werden in initWindowSize initialisiert. Die neuen

Defines werden wir in Editor.h definieren, so daß wir im Modul Ausgabe nur die neue Funktion SetFont deklarieren müssen:

```
void SetFont();
```

Ferner brauchen wir einen Zeiger auf den Zeichensatz, in dem die Infozeile ausgegeben wird und der im Hauptmodul definiert und geöffnet wird:

```
extern struct TextFont *infoFont;
```

In der Funktion initWindowSize müssen nun die Positionen der einzelnen Werte der Infozeile berechnet werden. Dies geschieht vor dem Restaurieren des zeilenptr-Feldes:

```
{
   /* Positionen fuer InfoZeile: */
   register ULONG xinc,xsize;

   xinc = w->Width + ed->G_Info.LeftEdge +
                                    ed->gi_IText.LeftEdge - 1;
   xsize= infoFont->tf_XSize;

   ed->ip_xpos       = xinc + INFO_XPOS_INC * xsize;
   ed->ip_ypos       = xinc + INFO_YPOS_INC * xsize;
   ed->ip_anzzeilen  = xinc + INFO_ANZZEILEN_INC * xsize;
   ed->ip_minfold    = xinc + INFO_MINFOLD_INC * xsize;
   ed->ip_maxfold    = xinc + INFO_MAXFOLD_INC * xsize;
   ed->ip_flags      = xinc + INFO_FLAGS_INC * xsize;
   ed->ip_topedge    = w->Height + ed->G_Info.TopEdge
                     + ed->gi_IText.TopEdge
                     + (ULONG)infoFont->tf_Baseline - 1;
}
```

Damit wären die Änderungen im Modul Speicher abgeschlossen. Die Gadgets für die Info- und die Kommandozeile werden im Hauptmodul Editor definiert. Doch zunächst wollen wir uns das neue Modul Command ansehen:

<src/command.c>

```
/************************************
 *                                  *
 * Modul: Command                   *
 *                                  *
 * Kommando-Interpreter für Editor. *
```

```
 *                                    *
 ************************************/

/************
 *
 * Includes:
 *
 ************/

#include <stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define TAB 9
#define LF 10
#define CR 13

#define UC(c) (((c >= 'a') && (c <= 'z'))? (c & 0xDF) : c)
#define DIGIT(c) ((c >= '0') && (c <= '9'))
#define SKIPBLANKS(str) while (*str == ' ') str++;

/**********************
 *
 * Externe Funktionen:
 *
 **********************/

BOOL cursorLeft(),cursorRight(),
                cursorUp(),cursorDown(),cursorHome();
BOOL cursorEnd(),deleteChar(),deleteLine(),
                              backspaceChar(),insertLine();
BOOL saveLine();
void printAll(),printFlags(),fclose(),Insert(),
                              strncpy(),printAnzZeilen();
void restoreZeilenptr(),SetWindowTitles(),
                            initFolding(),printYpos();
void DisplayBeep(),saveIfCursorMoved(),CopyMem();
struct Zeile *neueZeile(),*prevLine(),*nextLine();
FILE *fopen();
int fwrite(),getc();
UBYTE *getLastWord();
ULONG fseek();
UWORD recalcTopOfWindow();
```

```
/*********************
 *
 * Externe Variablen:
 *
 *********************/

extern struct Editor *aktuellerEditor;
extern struct MsgPort *edUserPort;

/*********************
 *
 * Globale Variablen:
 *
 *********************/

struct jmpEntry
{
   UBYTE c;
   UBYTE *(*fkt)();
};

/***************
 *             *
 * Funktionen: *
 *             *
 ***************/

/*********************************
 *
 * saveASCII(name)
 *
 * Speichert Text unter Namen auf
 * Diskette ab.
 *
 * name ^ Dateinamen.
 *
 *********************************/

BOOL saveASCII(name)
UBYTE         *name;
{
   register FILE *file;
   register struct Zeile *z;

   if (aktuellerEditor->pufypos)
      if (! saveLine(aktuellerEditor->aktuell))
      {
         DisplayBeep(NULL);
         return (FALSE);
      }

   if (file = fopen(name,"w"))
   {
      z = aktuellerEditor->zeilen.head;
```

```
      while (z->succ)
      {
         if (z->len)
            if (fwrite(z + 1,(int)z->len,1,file) != 1)
            {
               /* Fehler! */
               fclose (file);
               return (FALSE);
            }

         z = z->succ;
      }

      aktuellerEditor->changed = 0;
      printFlags();
      fclose (file);
      return (TRUE);
   }
   else
      return (FALSE);
}

/***********************************
 *
 * loadASCII(name)
 *
 * Laedt Text von Disk. Der Text
 * wird hinter der aktuellen Zeile
 * in den Text eingefuegt.
 *
 * name ^ Dateinamen.
 *
 ***********************************/

BOOL loadASCII(name)
UBYTE         *name;
{
   UBYTE buf[MAXBREITE];
   register UBYTE *ptr,*tab;
   int c;
   UWORD rm = aktuellerEditor->rm;
   register FILE *file;
   register UWORD len;
   register struct Zeile *z,*zn;

   if (file = fopen(name,"r"))
   {
      /* Namen in Titelzeile anzeigen: */
      strncpy(aktuellerEditor->filename,name,
              sizeof(aktuellerEditor->filename));
      SetWindowTitles(aktuellerEditor->window,
              aktuellerEditor->filename,-1L);

      if (z = ZEILENPTR(aktuellerEditor->wdy))
```

```
{
   aktuellerEditor->changed = 1;
   printFlags();
}

while (!feof(file))
{
   tab = aktuellerEditor->tabstring;
   ptr = buf;
   len = 0;

   /* Eine Zeile einlesen: */
   while (!feof(file) && (len < rm))
   {
      if ((c = getc(file)) == EOF)
         break;
      else
         *ptr++ = (UBYTE)c;

      if ((c == TAB) && (aktuellerEditor->tabs))
         do
         {
            tab++;
            len++;
         } while ((*tab) && (len < rm));
      else
      {
         tab++;
         len++;

         if (c == LF)
            break;
         else if (c == CR)
            if (!feof(file) && (len < rm))
            {
               if ((*ptr = (UBYTE)getc(file)) == LF)
                  ptr++;
               else
                  /* Dateizeiger zurueck setzen: */
                  fseek(file,-1L,1);

               break;
            }
      } /* of if (TAB) */
   }    /* of while */

   if (len = ptr - buf)
   {
      /* Wordwrap? */
      if (!aktuellerEditor->skipblanks)
         if ((len >= rm) && (c != CR) && (c != LF))
         {
            tab = getLastWord(buf,len);
            fseek(file,(long)tab - ptr,1);
```

```
                len = (ptr = tab) - buf;
             }

          /* Zeile nun abspeichern: */
          aktuellerEditor->lineptr = z;
                          /* wg. Garbage-Collection  */
          if (zn = neueZeile(len))
          {
             z = aktuellerEditor->lineptr;
             zn->len = len;
             CopyMem(buf,zn + 1,(ULONG)len);
             aktuellerEditor->anz_zeilen++;
             Insert(&aktuellerEditor->zeilen,zn,z);
             z = zn;
             aktuellerEditor->lineptr = NULL;
          }
          else
          {
             aktuellerEditor->lineptr = NULL;
             fclose (file);
             return (FALSE);
          }
       }
    }

    /* Nun noch Falten-Level berechnen: */
    initFolding();

    /* Fenster neu ausgeben: */
    if ((ZEILENPTR(0) == NULL) &&
                        (aktuellerEditor->zeilen.head->succ))
    {
       ZEILENPTR(0) = aktuellerEditor->zeilen.head;
       ZEILENNR(0)  = 1;
    }
    restoreZeilenptr();
    printAll();
    printAnzZeilen();

    fclose (file);
    return (TRUE);
  }
  else
    return (FALSE);
}

/*************************************************
 *
 * command-Funktionen:
 *
 * Diese bearbeiten jeweils eine Funktionsgruppe.
 *
 * Alle Funktionen liefern einen Zeiger auf den
 * naechsten Befehl zurueck, falls kein Fehler
```

```
 * aufgetreten ist, andernfalls einen negativen
 * Zeiger auf das fehlerhafte Zeichen.
 *
 * Liefert eine Funktion NULL zurueck, so gilt
 * dies als Abbruchkriterium.
 *
 * Eine -1 beendet das Programm.
 *
 ***********************************************/

UBYTE *commandASCII(str)
register UBYTE     *str;
{
   UBYTE *error = 1L - str,buf[80];
   register UBYTE *name;

   switch (UC(*str))
   {
      case 'L':
         str++;
         SKIPBLANKS(str)
         if (*str == '"')
         {
            /* Hole Dateinamen aus Kommandozeile */
            str++;
            name = buf;
            while ((*name = *str) && (*str != '"')
                  && (name < buf + sizeof(buf) - 1))
            {
               name++;
               str++;
            }

            *name = 0;
            if (*str == '"')
               str++;

            name = buf;
         }
         else
            /* Benutze Dateinamen vom Laden: */
            name = aktuellerEditor->filename;

         if (!loadASCII(name))
            return (error);

         break;
      case 'S':
         str++;
         SKIPBLANKS(str)
         if (*str == '"')
         {
            /* Hole Dateinamen aus Kommandozeile */
            str++;
```

```
            name = buf;
            while ((*name = *str) && (*str != '"')
                  && (name < buf + sizeof(buf) - 1))
            {
               name++;
               str++;
            }

            *name = 0;
            if (*str == '"')
               str++;

            name = buf;
         }
         else
            /* Benutze Dateinamen vom Laden: */
            name = aktuellerEditor->filename;

         if (!saveASCII(name))
            return (error);

         break;

      default:
         return (NULL - str);
   }

   return (str);
}

UBYTE *commandCursor(str)
register UBYTE      *str;
{
   switch (UC(*str))
   {
      case 'R':
         if (!cursorRight())
            return (NULL);
         break;
      case 'L':
         if (!cursorLeft())
            return (NULL);
         break;
      case 'U':
         if (!cursorUp())
            return (NULL);
         break;
      case 'D':
         if (!cursorDown())
            return (NULL);
         break;
      case 'E':
         if (!cursorEnd())
            return (NULL);
```

```
            break;
         case 'S':
         case 'H':
            if (!cursorHome())
               return (NULL);
            break;
         case 'N':
            if (!cursorDown())
               return (NULL);
            if (!cursorHome())
               return (NULL);
            break;
         case 'P':
            if (!cursorUp())
               return (NULL);
            if (!cursorHome())
               return (NULL);
            break;
         case 'T':
            if (!cursorTop())
               return (NULL);
            break;
         case 'B':
            if (!cursorBottom())
               return (NULL);
            break;

         default:
            return (NULL - str);
            }

      return (str + 1);
}

UBYTE *commandDelete(str)
register UBYTE      *str;
{
      switch (UC(*str))
      {
         case 'L':
            if (! deleteLine())
               return (NULL);
            break;
         case 'C':
            if (! deleteChar())
               return (NULL);
            break;
         case 'B':
            if (! backspaceChar())
               return (NULL);
            break;
```

```
      default:
         return (NULL - str);
   }

   return (str + 1);
}

UBYTE *commandLine(str)
register UBYTE    *str;
{
   switch (UC(*str))
   {
      case 'S':
         if (! insertLine((UBYTE) 0))
            return (1 - str);
         break;

      default:
         return (NULL - str);
   }

   return (str + 1);
}

UBYTE *commandQuit(str)
register UBYTE    *str;
{
   return (-1L);
}

UBYTE *commandSet(str)
register UBYTE   *str;
{
   switch (*str)
   {
      case 't':
         aktuellerEditor->tabs = 0;
         printAll();
         printFlags();
         break;
      case 'T':
         aktuellerEditor->tabs = 1;
         printAll();
         printFlags();
         break;
      case 'i':
      case 'I':
         aktuellerEditor->insert = 1;
         printFlags();
         break;
      case 'o':
      case 'O':
         aktuellerEditor->insert = 0;
         printFlags();
```

```
            break;
        case 'b':
            aktuellerEditor->skipblanks= 0;
            printFlags();
            break;
        case 'B':
            aktuellerEditor->skipblanks= 1;
            printFlags();
            break;
        case 'a':
            aktuellerEditor->autoindent= 0;
            printFlags();
            break;
        case 'A':
            aktuellerEditor->autoindent= 1;
            printFlags();
            break;
        case 'r':
        case 'R':
        {
            register UWORD rm;
            if (*++str == '=')
            {
                str++;
                if (DIGIT(*str))
                {
                    rm = 0;
                    while (DIGIT(*str))
                    {
                        rm = 10*rm + (*str - '0');
                        str++;
                    }

                    if (rm > MAXBREITE)
                        rm = MAXBREITE;
                    else if (rm < 10)
                        rm = 10;
                }
                else
                    /* Fehler */
                    return (NULL - str);
            }
            else
                rm = MAXBREITE;

            aktuellerEditor->rm = rm;
            --str;
            break;
        }

        default:
            return (NULL - str);
    }
```

```
   return (str + 1);
}

UBYTE *commandExit(str)
register UBYTE    *str;
{
   if (aktuellerEditor->changed)
      if (saveASCII(aktuellerEditor->filename))
         return (-1L);
      else
         return (NULL - str);
   else
      return (-1L);
}

UBYTE *executeCommand();      /* Deklaration wegen Rekursion! */

UBYTE *commandBracket(str)
register UBYTE       *str;
{
   register UBYTE *ptr;

   if (ptr = executeCommand(str))
   {
      if (ptr != -1L)
         if (ptr >= 0x80000000)
            /* Bestimme Zeiger auf Befehl hinter '}': */
            ptr = 1L - ptr;
         else
            /* Zeiger auf Fehler: */
            ptr = NULL - ptr;
   }
   else
   {
      /* Suche passendes '}': */
      ptr = str;
      while (*ptr && (*ptr != '}'))
      {
         if (*ptr == '"')
            do
            {
               ptr++;
            } while (*ptr && (*ptr != '"'));

         ptr++;
      }

      if (*ptr == 0)
         ptr = NULL;
      else
         ptr++;
   }
```

```
   return (ptr);
}

/*********************************
 *
 * messageWaiting:
 *
 * Testet, ob Message am UserPort
 * anliegt.
 *
 * Gibt TRUE zurueck, falls ja.
 *
 *********************************/

BOOL messageWaiting()
{
   register struct IntuiMessage *im;

   im = (struct IntuiMessage *)edUserPort->mp_MsgList.lh_Head;
   while (im->ExecMessage.mn_Node.ln_Succ)
   {
      if ((im->Class == RAWKEY)
      && !(im->Code & IECODE_UP_PREFIX)) return (TRUE);
      if (im->Class == NEWSIZE) return (TRUE);
      im = (struct IntuiMessage *)im->ExecMessage.mn_Node.ln_Succ;
   }

   return (FALSE);
}

struct jmpEntry jmpTable[] =
{
   'A',&commandASCII,
   'C',&commandCursor,
   'D',&commandDelete,
   'L',&commandLine,
   'Q',&commandQuit,
   'S',&commandSet,
   'X',&commandExit,
   '{',&commandBracket
};

/*****************************************
 *
 * executeCommand(str)
 *
 * Fuehrt die Befehlsfolge aus,
 * auf die str zeigt. Das Ende
 * wird durch ein Null-Byte markiert.
 *
 * Die Funktion gibt folgendes zurueck:
 * 0:     kein Fehler.
```

```
 * -1:    Programm beenden.
 * sonst: Zeiger auf Fehler im String.
 * bzw.:  negativen Zeiger auf ')'
 *
 ****************************************/

UBYTE *executeCommand(str)
register UBYTE       *str;
{
   register UBYTE *ptr,c;
   register UWORD count,n;
   register UBYTE *(*fkt)();

   while (!messageWaiting() && *str)
   {
      SKIPBLANKS(str)

      if (*str == ')')
         return (NULL - str);

      if (DIGIT(*str))
      {
         count = 0;
         while (DIGIT(*str))
         {
            count = 10*count + (*str - '0');
            str++;
         }

         if (count == 0)
            count = 1;
      }
      else
         count = 1;

      /* Suche Funktion, die Kommando ausfuehrt: */
      c = UC(*str);
      fkt = NULL;
      for (n = 0; n < sizeof(jmpTable)/sizeof
                                        (struct jmpEntry); n++)
         if (c == jmpTable[n].c)
         {
            fkt = jmpTable[n].fkt;
            break;
         }

      if (fkt)
         while (!messageWaiting() && count--)
         {
            if (ptr = (*fkt)(str + 1))
            {
               if (ptr >= 0x80000000)
                  if (ptr == -1L)
```

```
                    /* Programmende */
                    return (ptr);
                 else
                    /* Fehler aufgetreten: */
                    return (NULL - ptr);
              else
                 /* Falls count == 0: Naechsten Befehl, */
                 /* sonst denselben nochmals */
                 if (count == 0)
                    str = ptr;
           }
           else
              /* Abbruch */
              return (NULL);
        }
     else
        /* Unbekanntes Kommando */
        return (str);

     saveIfCursorMoved();

     SKIPBLANKS(str)

     if (*str == ';')
        str++;
     else if (*str == ')')
        return (NULL - str);
     else if (*str)
        return (str);
  }

  return (NULL);
}
```

Abschließend ein paar Bemerkungen zum neuen Modul:

- Da wir die Dateien über die Standard-C-Funktionen fopen, fclose usw. bearbeiten, müssen wir auch die Include-Datei stdio.h einbinden.
- Bei den Defines finden Sie nützliche Makros, um ein Zeichen in einen Großbuchstaben umzuwandeln (UC = Upper Case), um zu entscheiden, ob es sich bei einem Zeichen um eine Ziffer handelt (DIGIT), und um Blanks in einem String zu überspringen (SKIPBLANKS).
- Damit wir zur Unterscheidung der Befehle keine übermäßig große switch-case-Anweisung brauchen, werden wir jeweils den Anfangsbuchstaben des Befehls und die Adresse der entsprechenden Funktion in einem Feld able-

gen. Die Datenstruktur des Feldes ist jmpEntry. Das Feld ist nach den Funktionen für die Befehle definiert, da die Funktionen bereits definiert sein müssen, wenn das Feld initialisiert wird (Die Adressen der Funktionen müssen dem C-Compiler bekannt sein!).

- Die Funktion saveASCII erwartet als Parameter einen Zeiger auf einen Dateinamen und speichert den gesamten Text unter diesem Namen auf Diskette ab. Die Funktion gibt FALSE zurück, wenn ein Fehler aufgetreten ist.
- Gleiches gilt für die Funktion loadASCII, die aber wesentlich komplizierter ist, da ggf. Fließtext in einzelne Zeilen aufgespalten werden muß. Der zu ladende Text wird dabei hinter der aktuellen Zeile in den bereits bestehenden Text eingefügt. Existiert noch kein Text, so wird der Text ganz normal geladen. Die Variable z zeigt dabei jeweils auf die vorige Zeile.

Beim Laden werden solange einzelne Zeichen gelesen, bis ein Zeilenende erreicht oder die Zeile zu breit ist. Da die Funktion getc, die das Zeichen liest, intern mit einem Puffer arbeitet, ist diese Methode kaum langsamer, als wenn wir immer einen ganzen Textblock in einen Puffer laden und daraus die Zeichen holen würden. Die maximale Anzahl beim Laden der Zeichen pro Zeile läßt sich übrigens ändern. Hier wurde die Variable rm neu in die Editor-Struktur eingefügt, die den rechten Rand (Right Margin) angibt. Der Editor Ed hat etwas Ähnliches zu bieten.

Ist das Flag skipblanks auf Null gesetzt, so wird die Zeile am Anfang des letzten Wortes aufgesplittet, es wird also ein sogenanntes Wordwrap durchgeführt. Die Funktion getLastWord, die wir dabei benutzen, wird noch im Modul Edit implementiert werden. Zurück liefert sie einen Zeiger auf den Anfang des letzten Wortes einer Zeichenkette.

Ein Fehler, den das Programm bisher hatte, ist uns erst beim Arbeiten mit dem Editor aufgefallen. Sie erinnern sich sicher an die Funktion garbageCollection aus dem Modul Speicher, die einen Speicherblock reorganisiert. Dabei können allerdings Zeilen verschoben werden, so daß diese Funktion die Variable aktuell und das zeilenptr-Feld dahingehend überprüft und ggf.

einen Zeiger umbiegt. Nun wird aber von der Funktion loadASCII stets ein Zeiger auf die letzte Zeile mitgeführt, der auch überprüft werden müßte. Damit dies geschehen kann, wurde in der Editor-Struktur die Variable lineptr geschaffen, die von der Funktion garbageCollection überprüft wird. Haben wir eine lokale Variable, die auf eine Zeile zeigt, so speichern wir diese in lineptr ab, bevor wir die Funktion neueZeile aufrufen, die ihrerseits die Funktion garbageCollection aufruft. Nach dem Aufruf holen wir den Zeiger wieder aus lineptr und können sicher sein, daß dieser Zeiger noch auf dieselbe Zeile zeigt. Der erwähnte Fehler tritt in der Funktion saveLine auf, in der ebenfalls die Funktion neueZeile aufgerufen wird, während eine lokale Variable einen Zeiger auf eine Zeile enthält. Wir werden dies aber noch korrigieren.

Nachdem alle Zeilen geladen sind, werden mit initFolding die Falten-Level aller Zeilen neu berechnet, und anschließend wird der Fensterinhalt neu ausgegeben.

- Die Funktionen, die die einzelnen Befehle ausführen, heißen commandXYZ, wobei XYZ für die Befehlsgruppe steht. An diese Funktionen wird ein Zeiger auf den zweiten Buchstaben des Befehls übergeben. Zurückgegeben wird wieder ein Zeiger, der normalerweise hinter den Befehl zeigt, aber es gibt auch Sonderfälle: Liefert die Funktion Null zurück, so gilt dies als Abbruchkriterium für den Kommando-Interpreter. Eine -1 dagegen beendet den Editor, was bei den Befehlen X und Q Verwendung findet. Erhalten Sie aber einen negativen Wert zurück, so ist ein Fehler aufgetreten. Sie erhalten einen Zeiger auf den Fehler, wenn Sie den Rückgabewert mit -1 multiplizieren bzw. ihn von Null abziehen.
- Die bereits implementierten Befehle bestehen im wesentlichen aus dem Aufruf einer entsprechenden Funktion, die wir bereits in anderen Modulen stehen haben. Lediglich die ASCII-Befehle sind etwas umfangreicher, da dort ggf. eine Zeichenkette eingelesen werden muß. Bei den Set-Befehlen, die das Setzen der Flags bewerkstelligen, finden Sie auch den Befehl sr=nn, mit dem Sie den rechten Rand

für Wordwrap einstellen können. Dabei darf der Wert nn zwischen 10 und MAXBREITE liegen.

- Wenn Befehle geklammert werden, so wird die Funktion commandBracket aufgerufen, die ihrerseits die Funktion executeCommand aufruft (rekursiv). Die Funktion executeCommand springt zurück, wenn ein Fehler aufgetreten oder eine geschweifte Klammer-Zu gefunden wurde. Im zweiten Fall wird dann die Ausführung hinter dem "}" fortgesetzt. Schwierig wird es, wenn die geklammerten Befehle abgebrochen wurden. In diesem Fall soll die Kommandoausführung hinter der "}" fortgeführt werden. Die Funktion commandBracket sucht in diesem Fall dann nach dem "}" und gibt einen Zeiger auf das darauf folgende Zeichen zurück. Es gibt hier übrigens keine Probleme, wenn der Benutzer die Kommandofolge durch Tastendruck unterbrochen hatte, da dieses Abbruchkriterium ja bestehenbleibt.

- Die Funktion messageWaiting übernimmt die Abfrage, ob der Benutzer die Kommandofolge abbrechen will. Dies ist dann gegeben, wenn er entweder eine Taste gedrückt (vorzugsweise Shift-, Ctrl- oder Alt-Taste, da diese keine Zeichen, sondern nur RAWKEY senden!) oder die Größe des Fensters geändert hat. Im zweiten Fall würde die Ausgabe nämlich nicht mehr stimmen, da einige der Werte, die wir für die Ausgabe in der Funktion initWindowSize berechnet haben, sich geändert haben können.

- Die Funktion executeCommand übernimmt die Ausführung der einzelnen Befehle. Dazu erwartet sie als einzigen Parameter einen Zeiger auf eine Kommandofolge. Sie liefert Null zurück, ähnlich wie die command-Funktionen, wenn kein Fehler aufgetreten ist, -1, um das Programm zu beenden, oder ansonsten einen Zeiger auf den Fehler. Ist der zurückgelieferte Zeiger jedoch negativ, so handelt es sich dabei um einen Zeiger auf ein "}". Wurde die Funktion executeCommand von der Funktion commandBracket aufgerufen, so handelt es sich dabei um das Ende einer geklammerten Kommandofolge. Wurde die Funktion dagegen vom Hauptprogramm aus aufgerufen, so handelt es

sich um einen Fehler, da kein passendes "{" in der Kommandofolge zu finden war.

Da in C Zeiger als vorzeichenlose Werte behandelt werden, können wir nicht auf kleiner als Null abfragen, sondern wir müssen testen, ob der Zeiger größer als 0x80000000 ist, die eigentlich die kleinste negative Zahl ist, wenn man diesen Wert vorzeichenbehaftet betrachten würde.

Verbessern wir nun zunächst den Fehler im Modul Speicher. Dort müssen in der Funktion garbageCollection die beiden folgenden Zeilen hinter die Stelle eingefügt werden, an der der Zeiger aktuellerEditor->aktuell überprüft wird:

```
if (aktuellerEditor->lineptr == z)
   aktuellerEditor->lineptr = fz;
```

Nun zu unserer Include-Datei Editor.h, in der folgende Include-Dateien zusätzlich eingebunden werden müssen:

```
#include <stdio.h>
#include <intuition/intuitionbase.h>
```

Ferner müssen die Positionen der einzelnen Werte der Infozeile und deren Längen als Defines definiert werden:

```
#define INFO_XPOS_INC 2
#define INFO_XPOS_LEN 4
#define INFO_YPOS_INC 9
#define INFO_YPOS_LEN 4
#define INFO_ANZZEILEN_INC 16
#define INFO_ANZZEILEN_LEN 4
#define INFO_MINFOLD_INC 22
#define INFO_MINFOLD_LEN 2
#define INFO_MAXFOLD_INC 25
#define INFO_MAXFOLD_LEN 2
#define INFO_FLAGS_INC 29
#define INFO_FLAGS_LEN 5
```

Die "_INC"-Defines geben an, um wie viele Zeichen der entsprechende Wert vom Anfang der Infozeile entfernt ist, und die "_LEN"-Defines geben die Anzahl der Zeichen an, aus denen der Wert besteht. Die Editor-Struktur wird noch um einige Elemente erweitert:

```
    UBYTE gc_SIBuffer[256];
    struct StringInfo gc_GadgetSI;
    struct Gadget G_Command;
    UBYTE gi_Text[36];
    struct IntuiText gi_IText;
    struct Gadget G_Info;
    ULONG ip_xpos;
    ULONG ip_ypos;
    ULONG ip_anzzeilen;
    ULONG ip_minfold;
    ULONG ip_maxfold;
    ULONG ip_flags;
    ULONG ip_topedge;
    UBYTE filename[80];
    UWORD rm;
    struct Zeile *lineptr;
};
```

Die ersten sechs Elemente enthalten die Strukturen der Gadgets für die Kommando- und die Infozeile. Dann folgen die Positionen der einzelnen Werte der Infozeile, der aktuelle Dateiname, der von der Funktion loadASCII initialisiert wird, der rechte Rand (Right Margin) und der Hilfszeiger auf eine Zeile, der von der Funktion garbageCollection berücksichtigt wird.

Wenden wir uns nun dem Hauptmodul Editor.c zu, in das zunächst eine weitere Include-Datei eingebunden werden muß:

```
#include <intuition/intuitionbase.h>
```

Dann folgen zwei neue Defines, die den Text definieren, den der Editor ausgibt, wenn er vom CLI aus mit "Editor ?" aufgerufen wird:

```
#define UT1 "USAGE: Editor [Flags] <Filename>"
#define UT2 "Flags: [-a][-A][-b][-B][- i|I][- o|O][- r|R[=n]][-t][-T]"
```

Da wir das Laden von Text eingebaut haben, wollen wir den Editor natürlich auch vom CLI aus starten, so daß er direkt den Text lädt. Ferner lassen sich dabei auch die Flags beeinflussen. Statt sa, wie man in der Kommandozeile des Editors eingeben würde, kann man beim Aufruf des Editors vom CLI alle Flags setzen, indem man statt des s ein - vor das Flag setzt, also z.B.: -a. Doch dazu später. Nun erstmal die zusätzlichen externen Funktionen:

```
        void strncpy(),CloseFont(),puts();
        void printXpos(),printYpos(),printInfo(),DisplayBeep();
        struct TextFont *OpenFont();
        BOOL handleKeys(),ActivateGadget(),loadASCII();
        UBYTE *executeCommand(),*commandSet();
```

Beachten Sie bitte, daß die Funktion handleKeys jetzt vom Typ BOOL ist. Den Grund dafür werden Sie erkennen, wenn wir das Modul Cursor geändert haben. Bei den globalen Variablen definieren wir nun die beiden Gagdets, die im unteren Fensterrahmen die Funktion von Kommado- und Infozeile übernehmen. Die Kommandozeile wird dabei durch das String-Gadget G_Command repräsentiert, während für die Infozeile das Boolean-Gadget G_Info zuständig ist:

```
UBYTE gc_SIBuffer[256];
UBYTE gc_UndoBuffer[256];
struct StringInfo gc_GadgetSI =
{
   gc_SIBuffer,
   gc_UndoBuffer,
   0,
   256,
   0,
   0,0,0,0,0,
   0,
   NULL,
   NULL
};

SHORT gc_BorderVectors[4] = {0,0,350,0};
struct Border gc_Border =
{
   -1,-1,
   1,0,JAM1,
   2,
   gc_BorderVectors,
   NULL
};

struct Gadget G_Command =
{
   NULL,
   2,-9,
   -302,9,
   GADGHCOMP | GRELBOTTOM | GRELWIDTH,
   RELVERIFY | BOTTOMBORDER,
   STRGADGET,
   (APTR)&gc_Border,
   NULL,
   NULL,
```

```
    0,
    (APTR)&gc_GadgetSI,
    2,
    NULL
};

SHORT gi_BorderVectors[14] =
                        {-1,0,-1,10,0,10,0,0,298,0,298,1,283,1};
struct Border gi_Border =
{
    0,0,
    1,0,JAM1,
    7,
    gi_BorderVectors,
    NULL
};

struct TextAttr TOPAZ80 =
{
    (STRPTR)"topaz.font",
    TOPAZ_EIGHTY,0,0
};

UBYTE gi_Text[36] = "X=   1 Y=   1 #=   0 [ 0, 0] IcATB";
struct IntuiText gi_IText =
{
    1,0,JAM2,
    4,2,
    &TOPAZ80,
    gi_Text,
    NULL
};

struct Gadget G_Info =
{
    &G_Command,
    -298,-10,
    280,10,
    GADGHNONE | GRELBOTTOM | GRELRIGHT,
    RELVERIFY | BOTTOMBORDER,
    BOOLGADGET,
    (APTR)&gi_Border,
    NULL,
    &gi_IText,
    0,
    NULL,
    1,
    NULL
};

struct NewWindow newEdWindow =
{
    0,0,640,200,
    AUTOFRONTPEN,AUTOBACKPEN,
```

```
    REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW |
                                          NEWSIZE | GADGETUP,
    WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH |
                                WINDOWCLOSE | SIZEBBOTTOM
    | SIMPLE_REFRESH | ACTIVATE,
    NULL,NULL,
    NULL,
    NULL,NULL,
    320,50,640,200,
    WBENCHSCREEN
};

struct TextFont *infoFont = NULL;
```

Das Fenster erlaubt nun auch GADGETUP-Events, damit das Programm eine Nachricht erhält, wenn im String-Gadget die Return-Taste gedrückt wurde und damit eine Kommandofolge zur Ausführung gebracht werden soll. Da wir später mehrere Editorfenster öffnen können wollen, müssen wir Teile dieser Gagdets in die Editor-Struktur kopieren, da keine zwei Fenster dieselben Gadgets benutzen können. Allerdings brauchen wir nicht alle Strukturen zu kopieren. So reicht beispielsweise eine Border-Struktur für alle Fenster, da diese vom Betriebssystem nicht verändert, sondern nur ausgelesen wird. Benötigt werden aber der Puffer für die Kommandozeile SIBuffer, damit wir in zwei Fenstern auch zwei verschiedene Kommandofolgen eingeben können, die StringInfo-Struktur, der Text der Infozeile, die IntuiText-Struktur, die auf diesen zeigt, und natürlich die beiden Gadget-Strukturen.

Die Gadgets und die anderen Strukturen werden in der Funktion OpenEditor in die Editor-Struktur kopiert. Dies geschieht direkt nach Beschaffung des Speichers für die Editor-Struktur, da die Gadgets ja noch in die NewWindow-Struktur eingebunden werden müssen, damit diese auch im Fenster erscheinen. Auch darf nicht vergessen werden, die Zeiger der Strukturen aufeinander zu setzen, da diese nach dem Kopieren in die Editor-Struktur nicht mehr stimmen. Im folgenden nochmals die Funktion OpenEditor, wobei die Initialisierung der restlichen Parameter ausgespart wurde, da diese gleich geblieben ist:

```
struct Editor *OpenEditor()
{
    register struct Editor *ed = NULL;
```

```
register struct Window *wd;
register struct RastPort *rp;
ULONG flags;

/* IDCMPFlags retten, in Struktur auf NULL setzen!
   => eigenen UserPort verwenden! */
flags = newEdWindow.IDCMPFlags;
newEdWindow.IDCMPFlags = NULL;

/* Speicher fuer Editor-Struktur beschaffen: */
if (ed = malloc(sizeof(struct Editor)))
{
   /* Gadgets initialisieren: */
   ed->gc_SIBuffer[0]    = 0;
   ed->gc_GadgetSI       = gc_GadgetSI;
   ed->G_Command         = G_Command;
   ed->gi_IText          = gi_IText;
   ed->G_Info            = G_Info;
   strncpy(ed->gi_Text,gi_Text,sizeof(ed->gi_Text));

   /* Zeiger setzen: */
   ed->gc_GadgetSI.Buffer     = ed->gc_SIBuffer;
   ed->G_Command.SpecialInfo  = (APTR) &(ed->gc_GadgetSI);
   ed->gi_IText.IText         = ed->gi_Text;
   ed->G_Info.NextGadget      = &(ed->G_Command);
   ed->G_Info.GadgetText      = &(ed->gi_IText);
   newEdWindow.FirstGadget    = &(ed->G_Info);
   newEdWindow.Title          = ed->filename;
   ed->filename[0]            = 0;

   /* Fenster oeffnen: */
   if (wd = OpenWindow(&newEdWindow))
   {
      ed->window = wd;
      ed->rp = (rp = wd->RPort);

      /*******************************************
       *                                         *
       * Wie gehabt die Parameter initialisieren! *
       *                                         *
       * (Von den Schreibmodi bis maxfold !!!!!!) *
       *                                         *
       *******************************************/

      ed->rm         = MAXBREITE;

      {
         register UBYTE *ptr;
         register struct Zeile **zptr;
         register UWORD n,*pnr;

         /* zeilenptr/nr-Array initialisieren: */
         for (n = 0, zptr = ed->zeilenptr, pnr = ed->zeilennr;
               n < MAXHOEHE; n++, zptr++, pnr++)
```

```
            {
               *zptr = NULL;
               *pnr  = 1;
            }

            /* Tabulatoren initialisieren: */
            ptr = ed->tabstring;
            *ptr++ = 1;
            for (n = 1; n < MAXBREITE; n++)
               if (n % 3)
                  *ptr++ = 1;
               else
                  *ptr++ = 0;
         }

         {
            register struct Editor *oldae;

            ed->wch = 0;
            initWindowSize(ed);

            oldae           = aktuellerEditor;
            aktuellerEditor = ed;
            printInfo();
            aktuellerEditor = oldae;
         }
      }
      else
      {
         free(ed);
         ed = NULL;
      }
   }

   newEdWindow.IDCMPFlags = flags;
   return (ed);
}
```

Am Ende von OpenEditor wird die Infozeile nochmals ausgegeben, damit diese auch mit den tatsächlichen Werten übereinstimmt. Dazu muß die Variable aktuellerEditor auf den neu geöffneten Editor umgebogen werden, da die Ausgabefunktionen für die Infozeile voraussetzen, daß aktuellerEditor auch wirklich auf den aktuellen Editor zeigt.

Unser Hauptprogramm muß ebenfalls in größerem Umfang geändert werden. Die Funktion main erwartet nun die beiden Parameter argc und argv, die alle Parameter beinhalten, die dem Programm vom CLI aus mitgegeben wurden. Ganz zu Anfang

des Programms, noch bevor die Librarys geöffnet werden, wird nun getestet, ob der Editor mit "Editor ?" aufgerufen wurde. In diesem Fall wird nur die Syntax des Aufrufs ausgegeben und danach das Programm beendet. Zur Ausgabe verwenden wir die Funktion puts, die einen String ausgibt, aber nicht so umfangreich wie die Funktion printf ist.

Nach dem Öffnen der Librarys wird auch der Zeichensatz für die Infozeile geöffnet und ein Zeiger darauf in der Variablen infoFont abgelegt. Bevor der Editor geöffnet wird, stellt das Programm die maximale Höhe des Bildschirms fest und trägt sie in die NewWindow-Struktur ein, so daß das Fenster beim Öffnen die maximale Größe besitzt. Als Zeiger auf den Bildschirm wird die Variable ActiveScreen der IntuitionBase-Struktur verwendet, die auf den gerade aktiven Bildschirm zeigt.

Nachdem das Editorfenster geöffnet ist, wird die Kom#mandozeile ausgewertet, die dem Programm vom CLI aus mitgegeben wurde. Dabei wird vorausgesetzt, daß der Dateiname zu allerletzt angegeben wird. Sonst, und auch im Falle anderer Fehler, wird das Programm beendet, nachdem die Syntax des Programmaufrufes ausgegeben wurde. Der einzige Fehler, der das Programm nicht stört, ist, wenn die Datei nicht gefunden wurde, die der Editor laden soll.

Die Hauptschleife ist im wesentlichen gleich geblieben, die Funktion handleKeys gibt nun aber einen Wert vom Typ BOOL zurück. Ist dieser FALSE, so wird das Programm abgebrochen. In den Programmstücken für MOUSEBUTTON und NEWSIZE-Messages wird nun die Cursor-Position mit printXpos und printYpos ausgegeben, falls diese verändert wurde.

Neu ist die Abfrage für GADGETUP-Messages, über die die Ausführung der Kommandozeile gestartet wird. Tritt dabei ein Fehler auf, so wird der Cursor des Gadgets auf die fehlerhafte Stelle gesetzt und das String-Gadget aktiviert. Da in der GADGETUP-Abfrage nicht überprüft wird, welches Gadget diese Message verursacht hat, können Sie die Kommandozeile ausführen lassen, indem Sie mit der Maus auf die Infozeile klicken.

Dies sorgt dafür, eine bereits eingegebene Kommandofolge nochmals ausführen zu lassen.

Geändert wurde auch der Schluß des Programms. Hier werden alle Variablen, die getestet werden, explizit auf Null gesetzt. Der Grund dafür liegt in einem Hilfsprogramm namens SHELL. Dabei handelt es sich um ein verbessertes CLI, das es erlaubt, Befehle als resident zu definieren. Dies bedeutet, daß der entsprechende Befehl geladen und im Speicher behalten wird, unabhängig davon, wie oft er ausgeführt wird. Der Vorteil ist einmal, daß der Befehl nicht erst geladen werden muß und daß weniger Speicherplatz vergeudet wird, als wenn der Befehl auf die RAM-Disk kopiert werden würde. Denn wenn der Befehl auf die RAM-Disk kopiert wird, so muß er trotzdem noch geladen werden, bevor er ausgeführt werden kann. Ist er dagegen als resident definiert, so befindet er sich nur einmal im Speicher und wird bei Aufruf einfach nur aktiviert.

Allerdings müssen Befehle bzw. Programme, die als resident definiert werden sollen, bestimmte Forderungen erfüllen. So dürfen Sie nicht voraussetzen, daß Variablen initialisiert sind, da deren Inhalt ja noch von einem früheren Aufruf stammen kann. Um dies zu umgehen, setzen wir alle wichtigen Variablen am Ende des Programms wieder auf ihren Anfangswert. Dadurch erreichen wir, daß der Editor jedesmal initialisierte Anfangswerte vorfindet. Doch nun zu unserem neuen Hauptprogramm:

```
main(argc,argv)
int  argc;
UBYTE    *argv[];
{
   register struct Editor *ed;
   BOOL running = TRUE;
   register struct IntuiMessage *imsg;
   ULONG signal,class,mouseX,mouseY;
   UWORD code,qualifier;
   APTR iaddress;

   /* Falls Editor mit "Editor ?" aufgerufen wurde,
                                          nur Text ausgeben */
   if (argc == 2)
      if (*argv[1] == '?')
      {
         puts(UT1);
         puts(UT2);
```

```
      goto Ende;
   }

/* Librarys oeffnen: */
if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
   goto Ende;
if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
   goto Ende;
if ( !(DosBase = OpenLibrary("dos.library",REV)))
   goto Ende;
if ( !(OpenDevice("console.device",-1L,&ioStdReq,0L)))
   ConsoleDevice = ioStdReq.io_Device;
else
   goto Ende;

/* Font fuer Infozeile oeffnen: */
if (! (infoFont = OpenFont(&TOPAZ80))) goto Ende;

/* UserPort oeffnen */
if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

/* Maximale Größe für Fenster bestimmen: */
newEdWindow.Height = newEdWindow.MaxHeight =
     ((struct IntuitionBase *)IntuitionBase)->
                                   ActiveScreen->Height;

/* Editor-Liste initialisieren: */
NewList(&editorList);

/* Erstes Editorfenster oeffnen: */
if (ed = OpenEditor())
{
   AddTail(&editorList,ed);
   aktuellerEditor = ed;
}
else
   goto Ende;

/* Kommandozeile bearbeiten: */
if (argc >= 2)
{
   register UWORD n = 1;

   while (n < argc)
   {
      if (*argv[n] == '-')
      {
         if (commandSet(argv[n] + 1) >= 0x80000000)
         {
            DisplayBeep(NULL);
            break;
         }
      }
      else
```

```
      {
         if (loadASCII(argv[n]))
            printAll();
         else
            DisplayBeep(NULL);

         /* Dateiname nur am Ende erlaubt! */
         n++;
         break;
      }

      n++;
   }

   if (n < argc)
   {
      /* Fehler! */
      puts(UT1);
      puts(UT2);
      goto Ende;
   }
}

do
{
   /* Cursor setzen: */
   Cursor();

   signal = Wait(1L << edUserPort->mp_SigBit);

   /* Cursor wieder loeschen: */
   Cursor();

   while (imsg = GetMsg(edUserPort))
   {
      class       = imsg->Class;
      code        = imsg->Code;
      qualifier   = imsg->Qualifier;
      iaddress    = imsg->IAddress;
      mouseX      = imsg->MouseX;
      mouseY      = imsg->MouseY;

      ReplyMsg(imsg);

      /* Event bearbeiten: */
      switch (class)
      {
         case RAWKEY:
         {
            register struct IntuiMessage *im1,*im2;

            if (! (code & IECODE_UP_PREFIX))
            {
               inputEvent.ie_Code      = code;
```

```
            inputEvent.ie_Qualifier = qualifier;
            if ((inputLen = RawKeyConvert(
                 &inputEvent,inputPuffer,MAXINPUTLEN,NULL)
                     ) >= 0)
               running = handleKeys(inputPuffer,inputLen);
         }

         /* Nachlaufen verhindern: */
         im1 = (struct IntuiMessage *)
               edUserPort->mp_MsgList.lh_Head;
         while (im2 = (struct IntuiMessage *)
                      im1->ExecMessage.mn_Node.ln_Succ)
         {
            if (im2->ExecMessage.mn_Node.ln_Succ == NULL)
                                                   break;
            if (im1->Class != RAWKEY) break;
            if (!(im1->Qualifier & IEQUALIFIER_REPEAT))
                                                   break;
            if (im2->Class != RAWKEY) break;
            if (!(im2->Qualifier & IEQUALIFIER_REPEAT))
                                                   break;

            /* Message entfernen: */
            im1 = GetMsg(edUserPort);
            ReplyMsg(im1);

            im1 = (struct IntuiMessage *)
                  edUserPort->mp_MsgList.lh_Head;
         }

         break;
      }
      case MOUSEBUTTONS:
         {
            register WORD x,y;

            if (mouseX <= aktuellerEditor->xoff)
               x = 0;
            else
               x = (mouseX - aktuellerEditor->xoff)
                  / aktuellerEditor->cw;
            if (++x >= aktuellerEditor->wcw)
               x = aktuellerEditor->wcw - 1;
            x += aktuellerEditor->leftpos;
            if (x > MAXBREITE)
               x = MAXBREITE;

            if (mouseY <= aktuellerEditor->yoff)
               y = 0;
            else
               y = (mouseY - aktuellerEditor->yoff)
                  / aktuellerEditor->ch;

            if ((y < aktuellerEditor->wch)
```

```
            && (aktuellerEditor->zeilenptr[y]))
        {
            aktuellerEditor->xpos = x;
            aktuellerEditor->wdy  = y;
            aktuellerEditor->ypos =
                            aktuellerEditor->zeilennr[y];

            printXpos();
            printYpos();
        }
    }

case NEWSIZE:
    initWindowSize(aktuellerEditor);

    /* Pruefe, ob Cursor noch im Fenster! */
    if (aktuellerEditor->xpos >
                              aktuellerEditor->leftpos
                               +aktuellerEditor->wcw)
    {
        aktuellerEditor->xpos = aktuellerEditor->leftpos
                              + aktuellerEditor->wcw;
        printXpos();
    }

    if (aktuellerEditor->wdy >= aktuellerEditor->wch)
    {
        aktuellerEditor->ypos =
                              aktuellerEditor->zeilennr
            [(aktuellerEditor->wdy=
                             aktuellerEditor->wch - 1)];
        printYpos();
    }

    break;

case REFRESHWINDOW:
    BeginRefresh(aktuellerEditor->window);
    printAll();
    EndRefresh(aktuellerEditor->window,TRUE);
    break;

case GADGETUP:
{
    register UBYTE *ptr;
    register SHORT pos,hw;

    if (ptr = executeCommand
                    (aktuellerEditor->gc_SIBuffer))
        if (ptr == -1L)
            running = FALSE;
        else
        {
            if (ptr >= 0x80000000)
```

```
                    ptr = NULL - ptr;

                /* Cursor in StringGadget auf Fehler setzen */
                 pos = ptr - aktuellerEditor->gc_SIBuffer;
                 aktuellerEditor->gc_GadgetSI.BufferPos = pos;
                 if (pos < aktuellerEditor->
                                    gc_GadgetSI.DispCount)
                    aktuellerEditor->gc_GadgetSI.DispPos = 0;
                 else
                    aktuellerEditor->gc_GadgetSI.DispPos = pos
                       - aktuellerEditor->
                                 gc_GadgetSI.DispCount / 2;

                 /* Gagdet aktivieren: */
                 ActivateGadget(&(aktuellerEditor->G_Command),
                                aktuellerEditor->window,NULL);
                 DisplayBeep(NULL);
              }

              break;
           }

           case CLOSEWINDOW:
              running = FALSE;
              break;

           default:
              printf("Nicht bearbeitbarer Event: %lx\n",class);

        } /* of case */

        saveIfCursorMoved();
     } /* of while (GetMsg()) */
  } while (running);

Ende:
  /* Alle Editorfenster wieder schliessen: */
  while (ed = RemHead(&editorList))
     CloseEditor(ed);

  /* UserPort schliessen: */
  if (edUserPort)
  {
     DeletePort(edUserPort);
     edUserPort = NULL;
  }

  /* Font schliessen: */
  if (infoFont)
  {
     CloseFont(infoFont);
     infoFont = NULL;
  }
```

```
   /* Librarys schliessen: */
   if (DosBase)
   {
      CloseLibrary(DosBase);
      DosBase = NULL;
   }
   if (GfxBase)
   {
      CloseLibrary(GfxBase);
      GfxBase = NULL;
   }
   if (IntuitionBase)
   {
      CloseLibrary(IntuitionBase);
      IntuitionBase = NULL;
   }
}
```

Zum Auswerten der CLI-Kommandozeile wird die Funktion commandSet verwendet, so daß Sie die Flags genauso setzen können wie in der Kommandozeile des Editors, nur daß Sie die einzelnen Befehle durch ein Blank statt des Semikolons trennen und ein - als ersten Buchstaben setzen (statt des s). Ein möglicher Aufruf des Editors wäre:

```
Editor -r=60 -b -t Fliesstext.ASC
```

Hiermit könnte man dann Fließtext editieren, wobei die Textbreite auf 60 Zeichen gesetzt wird. Das Ganze würde zwar nicht genauso einfach funktionieren wie bei einer Textverarbeitung, da uns die Befehle fehlen, um einen Absatz neu zu formatieren, aber immerhin.

Auch im Modul Cursor sind einige Änderungen vorzunehmen. Beginnen wir mit einigen neuen Defines:

```
#define CEND '@'
#define ESC 27
#define TAB 9
#define REPCOM 7
```

Da wir jetzt das Umschalten der Flags über die Kommandozeile vornehmen werden, können wir auf das Umschalten über Control-Codes verzichten, so daß Sie die Defines TABMODE, WRITEMODE und AUTOINDENT löschen können.

Neu sind auch die folgenden externen Funktionen, die wir für die Info- und die Kommandozeile brauchen:

```
void printXpos(),printYpos(),printFold(),printFlags();
BOOL ActivateGadget();
UBYTE *executeCommand();
```

Im folgenden müssen an allen Stellen, an denen die Cursor-Position oder andere Werte der Infozeile geändert werden (cursorLeft, cursorRight, cursorUp, cursorDown, halfPageUp, halfPageDown, cursorHome, enterFold, exitFold), die Aufrufe der entsprechenden Funktionen (printXpos, printYpos, printFold) eingebaut werden. Hinzu kommen drei neue Funktionen zur Cursorbewegung, die den Cursor auf das Ende einer Zeile, an den Textanfang oder an das Textende setzen:

```
/*************************************
 *
 * cursorEnd:
 *
 * Setzt den Cursor auf das Zeilenende.
 *
 *************************************/

BOOL cursorEnd()
{
   register UBYTE *ptr,*tab;
   register UWORD len,pos;
   register struct Zeile *z;

   if (aktuellerEditor->pufypos)
      pos = aktuellerEditor->puflen + 1;
   else
      if (z = ZEILENPTR(aktuellerEditor->wdy))
      {
         /* Zuerst von Laenge CR abziehen: */
         len = z->len;
         ptr = ((UBYTE *)(z + 1)) + len - 1;
         if (len)
            if (*ptr == CR)
               len--;
            else if (*ptr == LF)
               if (--len)
                  if (*--ptr == CR)
                     len--;

         if (aktuellerEditor->tabs)
         {
            ptr = ((UBYTE *)(z + 1));
            tab = aktuellerEditor->tabstring;
```

```
            pos = 1;

            while ((len--) && (pos < MAXBREITE))
               if (*ptr++ == TAB)
                  do
                  {
                     tab++;
                     pos++;
                  } while ((*tab) && (pos < MAXBREITE));
               else
               {
                  tab++;
                  pos++;
               }
         }
         else
            pos = len + 1;
      }
      else
         return (FALSE);

   aktuellerEditor->xpos = pos;
   if (pos >= (aktuellerEditor->leftpos + aktuellerEditor->wcw))
      scrollLeft(pos + 1 - (aktuellerEditor->leftpos
                            + aktuellerEditor->wcw));
   else if (pos <= aktuellerEditor->leftpos)
      scrollRight(aktuellerEditor->leftpos + 1 - pos);

   printXpos();

   return (TRUE);
}

/*****************************
 *
 * cursorTop:
 *
 * Setzt Cursor auf Text- bzw
 * Faltenanfang.
 *
 *****************************/

BOOL cursorTop()
{
   register struct Zeile *z;
   UWORD n;
   register UWORD cnt;

   z   = ZEILENPTR(0);
   n   = ZEILENNR(0);
   cnt = 0;
   while (z = prevLine(z,&n))
      cnt++;
```

```
   if (cnt)
      scrollDown(cnt);
   aktuellerEditor->ypos = ZEILENNR(aktuellerEditor->wdy = 0);
   printYpos();

   return (TRUE);
}

/*****************************
 *
 * cursorBottom:
 *
 * Setzt Cursor auf Text- bzw
 * Faltenende.
 *
 *****************************/

BOOL cursorBottom()
{
   register struct Zeile *z;
   UWORD n;
   register UWORD wch,cnt;

   if (z = ZEILENPTR(wch = aktuellerEditor->wch - 1))
   {
      n   = ZEILENNR(wch);
      cnt = 0;
      while (z = nextLine(z,&n))
         cnt++;

      if (cnt)
         scrollUp(cnt);

      aktuellerEditor->ypos =
               ZEILENNR(aktuellerEditor->wdy = wch);
   }
   else
   {
      aktuellerEditor->ypos =
               ZEILENNR(aktuellerEditor->wdy = wch);
      cursorOnText();
   }

   printYpos();

   return (TRUE);
}
```

Dabei ist cursorEnd noch die komplizierteste Funktion, da das Ende der Zeile über die Tabulatoren berechnet werden muß. Die Funktion cursorTop zählt die Zeilen, die noch oberhalb des Fensters liegen, und scrollt um diesen Wert nach unten. Die

Funktion cursorBottom arbeitet genauso, nur daß hier der Fall abgefangen werden muß, daß bereits in der untersten Zeile des Fensters kein Text mehr sichtbar ist. Sie können diese drei Funktionen nach der Funktion cursorHome einfügen.

Die Funktion handleKeys definieren Sie nun so, daß diese einen Rückgabewert vom Typ BOOL liefert:

```
BOOL handleKeys(buf,len)
```

In der zweiten Switch-Case-Anweisung, in der die Abfrage für alle CSI-Sequenzen stattfindet, die mit CSI ' ' beginnen, fügen Sie hinter der Abfrage für CURSORHOME drei Zeilen ein:

```
case CEND:
   cursorEnd();
   break;
```

In der zweiten großen Switch-Case-Anweisung, in der die Abfragen auf Control-Codes stehen, fügen Sie am Anfang, also vor der Abfrage für CFOLD, die folgenden Zeilen ein:

```
case ESC:
   /* Rest von buf uebertragen: */
   {
      register UBYTE *sibuf;
      register UWORD cnt;

      sibuf = aktuellerEditor->gc_SIBuffer;
      while (len)
      {
         *sibuf++ = *buf++;
         len--;
      }
      *sibuf = 0;

      cnt = (sibuf - aktuellerEditor->gc_SIBuffer);
      aktuellerEditor->gc_GadgetSI.BufferPos = cnt;
      aktuellerEditor->gc_GadgetSI.NumChars  = cnt;
      aktuellerEditor->gc_GadgetSI.DispPos   = 0;
   }

   /* Gagdet aktivieren: */
   if (!ActivateGadget(&(aktuellerEditor->G_Command),
                       aktuellerEditor->window,NULL))
      DisplayBeep(NULL);
   break;
case REPCOM:
```

```
{
   register UBYTE *ptr;
   register SHORT pos,hw;

   if (ptr = executeCommand(aktuellerEditor->gc_SIBuffer))
      if (ptr == -1L)
         return (FALSE);
      else
      {
         if (ptr >= 0x80000000)
            ptr = NULL - ptr;

         /* Cursor in StringGadget auf Fehler setzen */
         pos = ptr - aktuellerEditor->gc_SIBuffer;
         aktuellerEditor->gc_GadgetSI.BufferPos = pos;
         if (pos < aktuellerEditor->gc_GadgetSI.DispCount)
            aktuellerEditor->gc_GadgetSI.DispPos = 0;
         else
            aktuellerEditor->gc_GadgetSI.DispPos = pos
                  - aktuellerEditor->gc_GadgetSI.DispCount / 2;

         /* Gagdet aktivieren: */
         ActivateGadget(&(aktuellerEditor->G_Command),
                        aktuellerEditor->window,NULL);
         DisplayBeep(NULL);

         /* Sofort zurueck: */
         return (TRUE);
      }

   break;
}
```

Im Falle von Escape (ESC) wird der Rest der Zeichen, die von der Tastatur geschickt wurden, in die Infozeile geschrieben, und diese wird dann aktiviert. Allerdings erhält die Übertragung des Restes der Zeichen erst dann Bedeutung, wenn die Tastatur frei belegbar ist. Dann könnten wir eine Taste so belegen, daß diese zuerst ein Escape und dann den Anfang einer Befehlsfolge sendet, die wir dann nur noch mit Return starten müssen.

Die zweite Case-Anweisung behandelt den Fall, daß man Ctrl-G gedrückt hat (REPCOM), was eine wiederholte Ausführung der Kommandozeile bedingt. Im Falle eines Fehlers wird, genau wie in der GADGETUP-Abfrage im Hauptprogramm, der Cursor der Kommandozeile auf den Fehler gesetzt und diese aktiviert.

Löschen müssen Sie die Abfragen auf TABMODE, WRITEMODE und AUTOINDENT, da wir diese Flags nun über die Kommandozeile einstellen können. Dies hat den Vorteil, daß wir nicht mehr aus Versehen die falsche Taste drücken können und daß wir zusätzliche Control-Codes direkt in den Text einfügen können. Denn eine Besonderheit unseres Editors ist, daß wir Texte einlesen können, die aus beliebigen Zeichen bestehen.

Beim Ausprobieren des Editors stellten wir fest, daß die Return-Taste ein CR sendet, daß aber andere Programme des CLI (wie TYPE) nichts mit Zeilen anfangen konnten, die durch ein CR abgeschlossen sind. So lange wir also die Tastatur noch nicht frei belegen können, werden wir immer ein LF als Zeilenende benutzen, damit wir die Texte, die wir mit unserem Editor geschrieben haben, auch von anderen Programmen lesen lassen können. Die Abfrage auf CR und LF sieht damit wie folgt aus:

```
case LF:
case CR:
   if (! insertLine((UBYTE) LF))
      DisplayBeep(NULL);
   break;
```

Damit wären wir mit dem Modul Cursor fertig und können uns nun den letzten Änderungen im Modul Edit zuwenden. Dort deklarieren wir wieder ein paar neue externe Funktionen:

```
void printXpos(),printYpos(),printFold();
void printAnzZeilen(),printFlags();
BOOL cursorEnd();
```

Auch in diesem Modul gilt es, an allen Stellen, an denen Werte der Infozeile geändert werden könnten, Aufrufe der entsprechenden Funktionen (printXpos, printYpos, printFold, printAnzZeilen und printFlags) einzubauen.

In der Funktion saveLine muß ein lokaler Zeiger auf eine Zeile in die Variable lineptr der Editor-Struktur gerettet werden, bevor die Funktion neueZeile aufgerufen wird. Das entsprechende Programmstück sieht danach wie folgt aus:

```
/* ggf. neue Zeile beschaffen: */
if (EVENLEN(len) != EVENLEN(zeile->len))
{
   aktuellerEditor->lineptr = zeile; /* wg. Garbage-Collection */
   if (zeile = neueZeile(len))
   {
      old = aktuellerEditor->lineptr;
      Insert(&aktuellerEditor->zeilen,zeile,old);
      zeile->flags = old->flags & ~ZLF_USED;
      loescheZeile(old);

      /* Nun eventuelle Zeiger umsetzen: */
      for (l = 0, zptr = aktuellerEditor->zeilenptr;
            l <= aktuellerEditor->wch; l++, zptr++)
         if (*zptr == old)
         {
            *zptr = zeile;
            break;
         }

      aktuellerEditor->lineptr = NULL;
   }
   else
   {
      aktuellerEditor->lineptr = NULL;
      return (FALSE);
   }
}
else
{
   zeile->len    = len;
   zeile->flags &= ~ZLF_USED;
}
```

Vergessen Sie bitte nicht, in diese Funktion die Aufrufe von printYpos und printFlags (da changed gesetzt wird) einzubauen. Auch in die Funktion getPufferPointer gehört ein Aufruf von printAnzZeilen, da ggf. eine erste Zeile neu eingerichtet wird. Die Funktionen deleteChar und backspaceChar ändern Sie bitte so ab, daß sie immer TRUE zurückliefern. Die Unterscheidung zwischen TRUE und FALSE hat sich an dieser Stelle leider nicht bewährt. Vor allem beim Ausführen von Befehlsfolgen führte dies oft zum vorzeitigen Abbruch.

In der Funktion insertLine ist eine kleine Änderung nötig, ohne die der Befehl LS (Line Split) nicht richtig funktioniert. Statt das Zeichen, das die Zeile aufgetrennt hat (in der Regel LF), immer in die Zeile zu schreiben und die Länge um Eins zu er-

höhen, werden wir dies ab jetzt nur dann machen, wenn das Zeichen nicht den Wert Null hat:

```
/* Zeile dekonvertieren: */
p2 = buf + (len = deconvertLine(buf,aktuellerEditor->puffer,inc));
if (*p2 = c)
   len++;
```

Dadurch können wir Zeilen auch aufsplitten, ohne daß ein Zeichen eingefügt wird. Am Ende der Funktion müssen eine ganze Reihe von Werten der Infozeile neu ausgegeben werden:

```
printXpos();
printYpos();
printAnzZeilen();
printFlags();
```

Falls minfold und maxfold geändert wurden, so muß an den entsprechenden Stellen auch printFold aufgerufen werden.

Leider ist uns in der Funktion deleteLine ein kleiner Fehler unterlaufen, den wir erst bemerkten, als wir die Infozeile implementiert hatten. Sie erinnern sich, daß wir bereits einmal die Y-Position falsch berechnet hatten und dies nur mit dem Debugger feststellen konnten, da wir zu diesem Zeitpunkt noch keine Infozeile hatten. Etwas Ähnliches ist uns in der Funktion deleteLine passiert, und zwar an den Stellen, an denen die Zeiger auf die nächste und die vorige Zeile bestimmt werden:

```
prevnr = ZEILENNR(aktuellerEditor->wdy);
prev   = prevLine(zeile,&prevnr);
nextnr = ZEILENNR(aktuellerEditor->wdy);
next   = nextLine(zeile,&nextnr);
```

Leider haben wir dabei nicht bedacht, daß die nächste Zeile dieselbe Zeilennummer hat wie die aktuelle Zeile, da die aktuelle Zeile ja gelöscht wird. Daher müssen die beiden Stellen, an denen die obige Befehlsgruppe auftritt, wie folgt abgewandelt werden:

```
prevnr = ZEILENNR(aktuellerEditor->wdy);
prev   = prevLine(zeile,&prevnr);
next   = nextLine(zeile,&nextnr);
nextnr = ZEILENNR(aktuellerEditor->wdy);
```

Daß dabei nextnr nicht initialisiert ist, wenn nextLine aufgerufen wird, stört nicht, da in der Funktion nextLine der Wert von nextnr ohnehin nur um eins erhöht wird.

Neu hinzugekommen ist die Funktion getLastWord, die von einem gegebenen String den Zeiger auf das letzte Wort zurückliefert. Diese Funktion fügen Sie bitte vor der Funktion insertChar ein:

```
/*************************************
 *
 * getLastWord(str,len)
 *
 * Gibt Zeiger auf den Anfang des
 * letzten Wortes des Strings zurueck.
 * Dabei wird vorzugsweise nach einem
 * Wort gesucht, vor dem ein Leer-
 * zeichen steht.
 *
 * str ^ String.
 * len = dessen Laenge.
 *
 *************************************/

UBYTE *getLastWord(str,len)
register UBYTE    *str;
register UWORD         len;
{
   register UBYTE c,*lwb,*lwn,*end;

   lwb = NULL;              /* Last Word mit Blank */
   lwn = NULL;              /* Last Word normal */
   end = str + len - 1;     /* Zeiger auf String-Ende */
   for (str += len; len; len--)
   {
      if ((c = *--str) == ' ')
      {
         lwb = str;
         break;
      }
      else if (! ((c >= 'A') && (c <= 'Z')
                || (c >= 'a') && (c <= 'z')
                || (c >= '0') && (c <= '9')
                || (c >= 192) || (c == '_')))
         if (lwn == NULL)
            lwn = str;
   }

   if (lwb == NULL)
      if (lwn)
         lwb = lwn;
```

```
      else
         lwb = end;
   else
      if (lwn)
         if ((end - lwb) > MAXBREITE / 5)
            if ((end - lwn + 5) < (end - lwb))
               lwb = lwn;

   return (lwb + 1);
}
```

Dabei sucht die Funktion sowohl nach dem letzten Zeichen, vor dem ein Blank steht (lwb = last word blank), als auch nach dem letzten Zeichen, vor dem weder ein Buchstabe noch eine Zahl, ein Unterstrich oder ein internationales Sonderzeichen steht (lwn = last word normal). Normalerweise gibt die Funktion lwb den Vorzug. Ist jedoch lwb zu weit vom String-Ende entfernt (1/5 der maximalen Breite = 16 Zeichen) und wurde auch ein passendes Zeichen für lwn gefunden, so wird lwn dann der Vorzug gegeben, wenn es mindestens fünf Zeichen näher am String-Ende liegt als lwb. Dadurch wird einmal erreicht, daß mit Bindestrich getrennte Wörter am Bindestrich aufgetrennt werden, sofern das erste Teilwort länger als fünf Zeichen ist, daß aber andererseits beispielsweise eine negative Zahl wie "-123" nicht ausgerechnet hinter dem Minuszeichen getrennt wird.

In der Funktion insertChar muß nun zunächst eine kleine Unsauberkeit eliminiert werden: Fügt man Tabulatoren am Anfang der Zeile ein, und erreicht die Zeile die maximale Breite, so werden zwar keine weiteren Tabulatoren mehr eingefügt, aber der Cursor wird trotzdem weiter gesetzt. Um dies abzufangen, wird die Variable xadd zurückgesetzt, wenn die Länge der Zeile die maximale Breite übersteigt. Dazu wurde im folgenden Programmstück die Zeile "xadd -= ..." erweitert:

```
/* anz = Anzahl der Blanks, die eingefuegt werden: */
if (aktuellerEditor->puflen + anz > MAXBREITE)
{
   xadd -= anz + aktuellerEditor->puflen - MAXBREITE;
   anz   = MAXBREITE - aktuellerEditor->puflen;
}
```

Vor dem Einfügen eines Zeichens in die Zeile wird dann noch Wordwrap durchgeführt:

```
/* sonst als normales Zeichen: */
if ((! aktuellerEditor->skipblanks)
   && (aktuellerEditor->xpos >= aktuellerEditor->rm))
{
   /* Word-Wrap nur wenn SkipBlanks == 0 */
   aktuellerEditor->xpos = getLastWord(aktuellerEditor->puffer,
                                       aktuellerEditor->puflen)
                         - aktuellerEditor->puffer + 1;
   if (! insertLine((UBYTE) 0))
      return (FALSE);

   if (! saveLine(aktuellerEditor->aktuell))
      DisplayBeep(NULL);

   cursorEnd();

   if (! getPufferPointer(&ptr,&inc,&rest))
      return (FALSE);
}

if (aktuellerEditor->insert)
   ...
```

Daran schließt sich wieder die ursprüngliche Funktion an, es wird also das Zeichen in die Zeile eingefügt. Am Ende des Moduls Edit fügen Sie dann noch die Funktion initFolding an:

```
/**********************************
 *
 * initFolding:
 *
 * Berechnet die Falten-Level aller
 * Zeilen ab Textanfang neu.
 *
 **********************************/

void initFolding()
{
   register struct Zeile *z;
   register UWORD level,l;

   z = aktuellerEditor->zeilen.head;
   level = 0;
   while (z->succ)
   {
      z->flags = (z->flags & ~ZLF_FOLD) | (level & ZLF_FOLD);
      if (l = getFoldInc(z))
      {
         z->flags |= ZLF_FSE;
         level += l;
      }
```

```
        z = z->succ;
    }
}
```

Damit wären alle Änderungen durchgeführt. Ersetzen Sie nun im Make-File die Testumgebung Test durch das neue Modul Command, und lassen Sie den Editor übersetzen. Quelltext und fertiges Programm finden Sie wie gewohnt auf der Diskette zum Buch, und zwar im Verzeichnis V0.6! Auf den TestText wurde diesmal verzichtet, da Sie nun beliebige Texte laden können. Schauen Sie sich doch mal den Quelltext des neuen Moduls mit dem Editor an:

```
cd V0.6
Editor src/Command.c
```

Wir haben dieses und alle anderen Module mit Falten versehen, wodurch die Übersichtlichkeit nicht unerheblich gesteigert wurde. Sie haben nun auf einen Blick alle Funktionen, die dieses Modul enthält, parat und können sich mit Ctrl-F jede dieser Funktionen genauer ansehen.

An dieser Stelle wollen wir - zumindest in diesem Buch - die Entwicklung des Editors abbrechen. Wir denken, daß wir Ihnen alles Wissenswerte vermittelt haben, das zum Erstellen eines größeren Programms auf dem Amiga wichtig und hilfreich ist. Vielleicht entwickeln Sie den Editor ja weiter; wir werden es auf jeden Fall auch tun. Schließlich haben wir noch viele Ideen, die auf ihre Verwirklichung warten.

4.4 Befehlsübersicht des Editors

1. Direkte Tastenkommandos

Cursor-Tasten

normal	Bewegen den Cursor um Zeichen bzw. Zeile.
Shift rechts/links	Cursor an Zeilenanfang bzw. -ende.
Shift hoch/runter	Cursor halbe Seite hoch bzw. runter.
Delete	Löscht Zeichen unter Cursor.
Backspace	Löscht Zeichen links vom Cursor.
Tab	Tabulator einfügen.
Ctrl-B	Lösche Zeile.
Ctrl-E	Aus Falte heraustreten.
Ctrl-F	In Falte eintreten.
Ctrl-G	Kommandozeile erneut ausführen.
Ctrl-H	Backspace.
Ctrl-I	Tab.
Ctrl-J	Zeile auftrennen (LF).
Ctrl-M	Zeile auftrennen (LF).

2. Befehle für Kommandozeile

AL ["Dateiname"]	Laden eines Textes.
AS ["Dateiname"]	Abspeichern des gesamten Textes.
CB (Cursor Bottom)	Cursor auf Text-/Faltenende.
CD (Cursor Down)	Cursor um eine Zeile nach unten.
CE (Cursor End)	Cursor hinter Zeilenende setzen.
CH (Cursor Home)	Cursor auf Zeilenanfang setzen.
CL (Cursor Left)	Cursor ein Zeichen nach links.
CN (Cursor Next)	Cursor auf Anfang von nächster Zeile.
CP (Cursor Previous)	Cursor auf Anfang von voriger Zeile.
CR (Cursor Right)	Cursor ein Zeichen nach rechts.
CS (Cursor Start)	Cursor auf Zeilenanfang setzen.
CT (Cursor Top)	Cursor auf Text-/Faltenanfang.
CU (Cursor Up)	Cursor um eine Zeile nach oben.
DB (Delete Back)	Backspace.
DC (Delete Char)	Zeichen löschen (Delete).
DL (Delete Line)	Zeile löschen.

LS (Line Split)	Trenne Zeile an Cursor-Position auf.
Q	Verlasse Editor.
Sa	Auto-Indent aus.
SA	Auto-Indent ein.
Sb	SkipBlanks aus.
SB	SkipBlanks ein.
Si	Einfügemodus (Insert).
So	Überschreibmodus (Overwrite).
St	Tabulatoren aus.
ST	Tabulatoren ein.
X	Text abspeichern, falls verändert, und Programm beenden.

Anhang

Anhang A

Die häufigsten Routinen der c.lib

Dieser Anhang gibt einen Überblick über die am häufigsten gebrauchten Routinen der Linker-Library c.lib:

```
atof     1)
atoi     1)
atol     1)
close    3)
creat    3)
getc     4)
getchar  4)
open     3)
printf   4)
puts     4)
read     3)
rename   3)
strlen   2)
strcat   2)
strncat  2)
strcmp   2)
strncmp  2)
write    3)
```

1. Ziffer-String-Konvertierungen

atof — ASCII nach DOUBLE

Syntax

```
Double = atof (String)

duble  Double;
char  *String;
```

Beschreibung

Diese Funktion konvertiert die in einem ASCII-String enthaltene Zahl (z.B. String = "1.23445") in eine Double-Variable.

Parameter

String: Zu konvertierender Ziffern-String.

Ergebnis

Double: Wert des Strings als Double-Variable.

atoi — ASCII nach Integer

Syntax

```
Integer = atoi (String)

    int   Integer;
    char *String;
```

Beschreibung

Diese Routine konvertiert einen Ziffern-String in eine Integer-Zahl.

Parameter

String: Zu konvertierender Ziffern-String.

Ergebnis

Integer: Integer-Repräsentation des Ziffern-Strings.

atol — ASCII nach Long

Syntax

```
Long = atol (String)

long Long;
char *String;
```

Beschreibung

Diese Routine konvertiert einen Ziffern-String in einen Long-Wert.

Parameter

String: Zu konvertierender Ziffern-String

Ergebnis

Long: Long-Repräsentation des Ziffern-Strings.

2. String-Handling

strlen — Länge des Strings berechnen

Syntax

```
Länge = strlen (String)

int   Länge;
char *String;
```

Beschreibung

Diese Routine liefert die Anzahl der Zeichen des angegebenen Strings zurück. Der String muß allerdings durch ein Null-Byte

abgeschlossen worden sein. Dieses Null-Byte wird nicht mitgezählt.

Parameter

String: Adresse des zu untersuchenden Strings.

Ergebnis

Länge: Anzahl der im String enthaltenen Zeichen.

strcat — Verbinde zwei Strings

Syntax

```
NeuerString = strcat (String1, String2)

char *NeuerString;
char *String1;
char *String2;
```

Beschreibung

Diese Funktion verbindet zwei Strings miteinander. String2 wird an String1 angehängt.

Parameter

String1: Adresse des ersten Strings.
String2: Adresse des anzuhängenden Strings.

Ergebnis

NeuerString: Adresse des zusammengesetzten Strings.

strcmp — Vergleich zweier Strings

Syntax

```
Ergebnis = strcmp (String1, String2)

int   Ergebnis;
char *String1;
char *String2;
```

Beschreibung

Diese Funktion vergleicht zwei Strings.

Parameter

String1: Adresse des mit String2 zu vergleichenden Strings.
String2: Adresse des mit String1 zu vergleichenden Strings.

Ergebnis

Ergebnis: Ergebnis gibt an, ob die Strings gleich (Ergebnis = 0) waren, oder ob String1 lexikographisch größer ist als String2 (Ergebnis > 0) oder er lexikographisch kleiner ist als String2 (Ergebnis > 0).

3. File-Handling

close — Schließe File

Syntax

```
Error = close (fd)

    int  Error;
    long fd;
```

Beschreibung

Diese Routine schließt ein File.

Parameter

fd: Filedeskriptor, der von open() oder creat() zurückgeliefert wurde.

Ergebnis

Error: Hat Error den Wert 0, ist alles OK. Bei Error gleich -1 ist ein Fehler aufgetreten.

creat — Erzeuge ein File

Syntax

```
fd = creat (Name)

long  fd;
char *Name;
```

Beschreibung

Diese Funktion öffnet ein File, das nur beschrieben werden kann. Existierte schon ein File gleichen Namens, so wird das alte File gelöscht.

Parameter

Name: Name des zu öffnenden Files.

Ergebnis

fd: Filedeskriptor, der für close(), read() und write() benutzt wird.

open — Öffne File

Syntax

```
fd = open (Name, Modus)

long  fd;
char *Name
long  Modus;
```

Beschreibung

Diese Routine öffnet ein File. Sie können dann dieses File auslesen, beschreiben etc. Allerdings müssen Sie für weitere Operationen immer den Filedeskriptor angeben.

Parameter

Name: Adresse des Namens-Strings.

Modus: Modus gibt an, auf welche Weise das File geöffnet werden soll und wie ein schon mit dem angegebenen Namen existierendes File behandelt werden soll:

O_RDONLY

Das File kann nur gelesen werden. Es muß also schon existieren.

O_WRONLY

Dieses File kann nur beschrieben werden.

O_RDWR

Das File kann sowohl beschrieben als auch gelesen werden.

O_CREAT

Das File wird erzeugt und dann geöffnet.

O_TRUNC

Ein schon mit gleichem Namen existierendes File wird gelöscht und neu geöffnet.

O_EXCL
Wenn schon ein File mit gleichem Namen existiert, verursacht open() einen Fehler.

O_APPEND
Das File wird geöffnet, um an das Ende des bestehenden Files noch Daten anzuhängen.

Die Symbole für Modus werden im Include-File fcntl.h definiert.

Ergebnis
fd: fd ist der Filedeskriptor des Files. Über ihn kann das File mittels read(), write() und close() bearbeitet werden. Hat fd den Wert -1, so ist bei open() ein Fehler aufgetreten.

Siehe auch: read(), write(), close().

read — Lese File

Syntax

```
Anzahl = read (fd, Buffer, AnzBytes)

int   Anzahl;
long  fd;
char *buf;
long  AnzBytes;
```

Beschreibung
Dieser Befehl liest Daten vom File des angegebenen Filedeskriptors.

Parameter
fd: Filedeskriptor des Files, aus dem Daten gelesen werden sollen.

buf: Adresse des Speicherbereichs, in den die gelesenen Daten hineingeschrieben werden sollen.

AnzBytes: Anzahl der Bytes, die aus dem File gelesen werden sollen. Diese Anzahl sollte kleiner oder gleich der Größe des Puffers sein.

Ergebnis

Anzahl: Anzahl gibt die Anzahl der tatsächlich gelesenen Bytes an. Ist dieser Wert gleich 0, wurde das Ende des Files erreicht.

rename — Benenne File um

Syntax

```
Error = rename (AlterName, NeuerName)

int   Error;
char *AlterName;
char *NeuerName;
```

Beschreibung

Dieser Befehl gibt einem existierenden File einen neuen Namen.

Parameter

AlterName: Alter Name des Files.
NeuerName: Name, den das File erhalten soll.

Ergebnis

Error: Hat Error den Wert 0, wurde die Umbenennung erfolgreich ausgeführt. Bei Error gleich -1 kann der Fehler aufgetreten sein, daß ein File mit dem Namen NeuerName schon existiert. Dann wird das E_EXISTS-Flag in der globalen "errno"-Variablen gesetzt, die aufgetretene Fehler näher spezifiziert.

write — Schreibe Daten auf File

Syntax

```
Anzahl = write (fd, Buffer, AnzBytes)

int   Anzahl;
long  fd;
char *Buffer;
long  AnzBytes;
```

Beschreibung

Diese Funktion schreibt AnzBytes Bytes auf das File, das durch den Filedeskriptor näher spezifiziert wird.

Parameter

fd: Filedeskriptor des Files, in das die Daten geschrieben werden sollen.

Buffer: Anfangsadresse der Daten, die geschrieben werden sollen.

AnzBytes: Anzahl der Bytes, die auf das File geschrieben werden sollen.

Ergebnis

Anzahl: Gibt die Anzahl der tatsächlich geschriebenen Bytes an.

4. Ein-/Ausgabe

getchar — Lies Zeichen von 'stdin'

Syntax

```
Zeichen = getchar()

int Zeichen;
```

Beschreibung

Diese Funktion liest das nächste Zeichen vom Standard-Input-Stream. Beim Amiga muß allerdings erst <Return> eingegeben werden, damit getchar() zum Programm zurückkehrt.

Ergebnis:

Zeichen: Eingelesenes Zeichen.

printf — Gib einen String aus

Syntax

```
printf (Kommando_String, Var1, Var2, ...);

char *Kommando_String
...   Var1
...   Var2
.........
```

Beschreibung

Dieser Befehl dient zur Ausgabe von Nachrichten an den Benutzer. Diese werden in das Standard-Output-File (stdout) geschrieben. Meist ist stdout das CLI-Window.

Parameter:

Kommando_String: Neben normalen ASCII-Codes können Sie durch Spezialcodes veranlassen, daß Werte von Variablen ausgedruckt werden können:

%d Ausgabe einer Integer-Variablen in dezimaler Notation.

%o Ausgabe einer Integer-Variablen in oktaler Notation.

%x Ausgabe einer Integer-Variablen in hexadezimal. Notation.

%u Ausgabe einer Integer-Variablen ohne Vorzeichen.

%c Ausgabe eines Zeichens.

%s Ausgabe eines Strings.

%f Ausgabe einer Fließkommavariablen (float oder double).

%e Ausgabe einer Fließkommavariablen (float oder double) in wissenschaftlicher Notation.

%g Ausgabe einer Fließkommavariablen (float oder double) in der Notation (dezimal, Fließkomma (normal), Fließkomma (wissenschaftlich)), die am wenigsten Platz beansprucht.

%% Ausgabe des Prozentzeichens.

Zwischen Prozentzeichen (%) und dem Zeichen, das die auszugebende Variable bestimmt, können noch Formatangaben angegeben werden.

Var1
Var2
... Dies sind die jeweiligen Variablen, die ausgegeben werden sollen. Sie müssen in der Reihenfolge angegeben werden, in der sie im Kommando_String spezifiziert werden.

puts — Glb String auf stdout aus

Syntax

```
puts (String)

char *String;
```

Beschreibung

Diese Routine gibt einen String auf das Standard-Output-File aus. Dabei ist dieser String kein Kommando_String wie bei printf() und wird genauso ausgegeben, wie er angegeben wurde.

Parameter

String: Adresse des auszugebenden Strings.

Anhang B

Lattice-Compiler- und Linker-Optionen

Compiler LC

Der Compiler wird nach folgendem Format aufgerufen:

```
lc [Optionen] Files
```

Der Compiler besteht aus zwei Programmen (lc1 und lc2), die vom Programm lc automatisch aufgerufen werden.

Hinweis: Bei der Angabe der zu compilierenden Files können Wildcards (#?) und Joker (?) verwendet werden. So wird durch den Aufruf von lc #? dafür gesorgt, daß alle C-Sources des aktuellen Verzeichnisses compiliert werden. Nach der Endung .c für C-Sources wird dabei automatisch gesucht.

Hier die Compiler-Optionen:

-a

Mit Hilfe dieser Option legen Sie fest, welche Programmbereiche in den Chip-Memory geladen werden sollen. Welcher Datenbereich in den Chip-Memory geladen werden soll, muß durch einen Buchstaben, der direkt hinter -a angegeben werden muß, festgelegt werden.

b BSS (Block Storage Segment) und uninitialisierte Daten

c Code-Segment

d Daten-Segment

Normalerweise werden alle Segmente ins Fast-RAM geladen. Aber bei der Programmierung der Grafik- und Sound-Chips ist es notwendig, daß alle Daten im Chip-Memory stehen (-abd).

-b

Normalerweise kann man alle Daten eines Programms adressieren. Dies wird möglich durch eine absolute 32-Bit-Adressierung. Wenn man allerdings die Option -b verwendet, wird eine rela-

tive 16-Bit-Adressierung verwendet. Diese Adressierungsart ist zwar schneller als die 32-Bit-Adressierung, legt aber gewisse Beschränkungen auf: Um diese Adressierung zu verwenden, enthält das Register A4 die Basisadresse des Datenbereichs (A4 relative Adressierung). Wenn Sie diese Art der Adressierung bei eigenen Tasks oder Interrupts benutzen wollen, muß dieses Register gerettet und neu gesetzt werden (siehe Option -y). Außerdem macht die Verwendung einer 16-Bit-Adressierung nur Datenbereiche von maximal 64 KB möglich, die im selben Hunk liegen müssen.

-C

Wenn Sie mehrere zu compilierende Files angegeben haben, werden Sie beim Auftreten eines schweren Fehlers gefragt, ob weiter compiliert werden soll. Wenn Sie die Option -C (Achtung: Großschreibung!) verwenden, wird auf die Abfrage verzichtet und automatusch das nächste File compiliert.

-c

Mit Hilfe dieser Option können Sie einige Abweichungen vom ANSI-C-Standard festlegen. Je nach Buchstabe, der nach -c folgt, wird folgende Abweichung festgelegt:

c Es ist erlaubt, Kommentare zu verschachteln.

d Das Zeichen $ darf in Symbolnamen verwendet werden.

m Es ist möglich, Zeichen-Konstanten mit mehr als einem Buchstaben anzugeben (z.B. ab).

s Von identischen Strings wird nur eine Kopie angelegt.

t Wenn bei der Definition eines Zeigers auf eine Struktur festgestellt wird, daß die Struktur nicht definiert wurde, wird normalerweise eine Warnung ausgegeben. Diese Option verhindert dies allerdings, so daß

```
struct ABC *Pointer;
```

erlaubt ist, obwohl ABC nicht definiert wurde.

u Alle char-Deklarationen werden in Unsigned-Char-Deklarationen umgewandelt.

w Diese Option schaltet die Generierung von Warnings ab, wenn bei einer INT-Funktion die Rückgabe des Rückgabewertes mit return() vergessen wurde. INT-Funktionen werden also wie VOID-Funktionen gehandhabt.

-dSYMVBOL[=WERT]

Mit Hilfe dieser Option können Sie nachträglich eine Konstante definieren (#define).

-f

Diese Option veranlaßt den Compiler, die Motorola-Fast-Floating-Point--Routinen (mffp) zu verwenden (bei FLOAT- und DOUBLE-Variablen). Normalerweise werden die IEEE-Routinen benutzt. Beachten Sie, daß IEEE und MFFP nicht kompatibel sind! Bei Verwendung der Option -f muß außerdem die lcmffp.lib beim Linken verwendet werden.

-h

Mit Hilfe dieser Option können Sie festlegen, welches Segment ins Fast-RAM geladen werden soll:

b BSS und uninitialisierte Daten.

c Code-Segment.

d Datensegment.

Mit -hbcd wird dafür gesorgt, daß das ganze Programm ins Fast-RAM geladen wird.

-i

Mit Hilfe dieses Kommandos können Sie nachträglich festlegen, welche Disketten und Verzeichnisse nach Include-Files durchsucht werden sollen (z.B. -idf0:Meine:Incs/). Normalerweise wird das mit der Umgebungsvariable INCLUDE: spezifizierte Verzeichnis durchsucht. Mit der Option -i haben Sie allerdings die Möglichkeit, bis zu 10 weitere Verzeichnisse anzugeben (-iVerz1: -iVerz2: usw.).

-L

Mit Hilfe dieser Option wird veranlaßt, daß der Linker automatisch vom Compiler aufgerufen wird. Standardmäßig werden der Startup-Code (c.o), sowie die Librarys lc.lib und amiga.lib beim Linken verwendet.

Nach -L kann ein weiterer Buchstabe angegeben werden, der die Arbeitsweise des Linkers beeinflußt:

c Es wird die Smallcode-Option des Linkers verwendet.

d Es wird die Smalldata-Option des Linkers verwendet.

f Es wird neben lc.lib und amiga.lib die lcmffp.lib beim Linken verwendet.

m Es wird die lm.lib verwendet (IEEE-Mathematik-Routinen).

v Verbose-Option des Linkers.

Beim Aufruf des Comilers mit der Option -L wird ein File mit der Endung .lnk erzeugt. Dieses File wird zur Übergabe der Optionen an den Linker verwendet. Sie können dieses File verändern (mit dem ED) und den Link-Vorgang einfach durch BLINK xxxx.lnk erneut starten.

-l

Mit Hilfe dieser Option wird der Compiler veranlaßt, alle Variablen außer char und short int an durch 4 teilbaren Adressen beginnen zu lassen (BCPL-Kompatibilität).

-M

Mit Hilfe dieser Option werden nur die Files compiliert, deren Objekt-File älter als das zugehörige Source-File ist (siehe Make beim Aztec). Wie auch beim Make-Utility werden Änderungen in (eigenen) Includes hierbei nicht berücksichtigt. Wenn Sie also ein eigenes Include-File verändert haben, ohne das zugehörige Source-File zumindest in den ED (oder einen anderen Editor) geladen und wieder abgespeichert zu haben, wird diese Änderung nicht berücksichtigt.

-o

Mit Hilfe dieser Option können Sie festlegen, in welches Verzeichnis oder unter welchem Namen das vom Compiler erzeugte Objekt-File geschrieben werden soll (z.B. -odh0: oder -odh0:objekt.o).

-q

Mit Hilfe dieser Option legen Sie fest, wohin die temporären Files des Compilers geschrieben werden sollen (siehe QUAD: Umgebungsvariable).

-r

Mit Hilfe dieser Option wird der Compiler dazu veranlaßt, zur Adressierung die PC-Relative-Adressierungsart zu verwenden. Dies hat den Nachteil, daß Routinen, die angesprungen werden sollen, in einem Bereich von +/-32K vom Aufruf liegen müssen. Diese Adressierungsart ist nützlich für kleinere Programme. Bei größeren Programmen, wo die Zieladressen außerhalb dieses Bereichs liegen, muß eine "Übersetzung" des Sprungs durchgeführt werden, was zur Verlangsamung des Programms führt oder dafür sorgt, daß das Program nicht vom Linker erzeugt werden kann.

-R

Mit Hilfe dieser Funktion können Sie dafür sorgen, daß die vom Compiler erzeugten Objekt-Files direkt in eine eigene Library geschrieben werden. Routinen, die vorher schon in der Library existierten, werden hierbei überschrieben. Nach -R muß der Name der Library erfolgen (-RMeineLib.Lib) - inklusive des Tags .lib.

-s

Normalerweise sind alle Hunks, die vom Linker erzeugt werden und das lauffähige Programm ausmachen, unbenannt. Mit Hilfe dieser Option wird aber dafür gesorgt, daß im Objekt-File die Namen text, data und udate für die Code-Hunks, Data-Hunks und BSS-Hunks vergeben werden, was wiederum dazu führt, daß der Linker diese Namen bei der Programm bzw. Hunkgene-

rierung verwendet. Wenn Sie andere Namen als text, data und udata verwenden wollen, müssen Sie nur die Optionen

```
-sc=Codehunk
-sd=Datahunk
-SB=Bsshunk
```

verwenden. Die Symbole Codehunk, Datahunk und Bsshunk werden dabei von Ihnen durch Strings ersetzt.

-v

Mit Hilfe dieser Funktion wird die Überprüfung des Stacks beim Eintritt in jede Funktion ausgeschaltet.

-x

Alle globalen Variablen werden als externe Variablen deklariert. Somit können von verschiedenen Modulen aus dieselben Variablen verwendet werden.

-y

Wenn Sie diese Option verwenden, wird bei jedem Eintritt in eine Funktion das Register A4 mit der Basisadresse der Variablen geladen (siehe Option -b).

Linker BLINK

Der Linker wird nach folgendem Format aufgerufen:

```
BLINK [FROM] [ROOT] Files [TO File] [WITH File] [VER File] [LIBRARY
oder LIB Files] [XREF File] [Optionen]
```

Wie Sie sehen, können nach einem Schlüsselwort ein oder mehrere Files folgen. Mehrere Files werden dabei entweder durch ein Leerzeichen oder duch ein + getrennt.

Neben den oben beim Aufruf des BLINK angegebenen Schlüsselwörtern erkennt der BLINK noch eine Menge weiterer Schlüsselwörter. Diese werden als Optionen angebenen. Hier eine alphabetische Liste der BLINK-Schlüsselwörter:

ADDSYM

Dieses Schlüsselwort veranlaßt BLINK, einen Symbol-Hunk zu generieren, der von symbolischen Debuggern (DB oder Metascope) verwendet wird, um Variablen, Routinen usw. symbolisch (also mit deren Namen) ansprechen zu können.

BATCH

Alle undefinierten Symbole werden auf 0 gesetzt. Normalerweise unterbricht der Linker seine Arbeit, wenn er auf ein undefiniertes Symbol trifft.

BUFSIZE n

Dieses Schlüsselwort bestimmt, wie groß der I/O-Buffer während des Linkens sein soll. Normalerweise ist der Buffer 488 Bytes groß (ein Datensektor). Wenn Sie diesen Buffer vergrößern, können mehr Daten auf einmal vom Linker gelesen werden. Dies hat den Vorteil, daß der Linker schneller arbeiten kann, da mehr Daten im Speicher stehen und nicht erst von Diskette geladen werden müssen.

CHIP

Mit Hilfe dieses Schlüsselwortes können Sie dafür sorgen, daß alle Hunks des Programms in den Chip-Memory geladen werden.

FAST

Mit Hilfe dieses Schlüsselwortes wird dafür gesorgt, daß alle Hunks in das Fast-RAM des Rechners geladen werden (siehe Chip).

FROM/ROOT

Die nach diesen beiden Schlüsselwörtern stehenden Files bestimmen die Objekt-Files, die zu Beginn des lauffähigen Programms stehen sollen. Zu beachten ist, daß mindestens ein Objekt-File angegeben werden muß. Meistens verwendet man aber mehrere, da das erste Objekt-File immer das File c.o ist, das den Startup-Code enthält, der dafür sorgt, daß die Kommando-Parameter für die Routine main() aufgearbeitet werden, und die Möglichkeit besteht, das Programm mit exit() zu verlassen.

Außerdem sorgt c.o dafür, daß der Stack beim Start und Verlassen des Programms korrekt verwaltet wird.

LIB/LIBRARYS

Die nach diesen Schlüsselwörten definierten Files bestimmen die Libararys, die durchsucht werden sollen. Wie zu Beginn dieses Buches schon erwähnt, werden die Librarys lc.lib und amiga.lib immer verwendet, denn diese enthalten erstens alle C-immanenten Funktionen (strlen(), strcmp() etc.) sowie die Aufrufe der Amiga-Betriebssystemroutinen. Der Unterschied zwischen Objekt-Files und Libararys ist folgender: Objekt-Files werden in Ihrer Gesamtheit zum Programm gelinkt - es werden also auch Routinen eingebunden, die gar nicht angesprungen werden. Librarys hingegen werden nach den noch nicht in den Objekt-Modulen definierten Routinen durchsucht. Zum Programm werden dann nur die tatsächlich verwendeten Routinen gelinkt.

NODEBUG/ND

Mit Hilfe dieses Schlüsselwortes wird die Erzeugung des Hunks HUNK_DEBUG unterdrückt.

NOALVS

Mit Hilfe dieses Schlüsselwortes wird die Erzeugung einer 32-Bit-Sprungtabelle zur Auflösung von 16-Bit-Sprüngen verhindert. Dieses Schlüsselwort wird verwendet, wenn Sie sichergehen wollen, daß ein tatsächlich relokatibles Programm erzeugt wird. Wenn der Linker feststellt, daß ALV (Automatic Load Vectors) erzeugt werden müssen, damit das Programm laufen kann, unterbricht BLINK seine Arbeit.

SMALLCODE/SD

Mit Hilfe dieser Schlüsselwörter wird dafür gesorgt, daß alle Code-Hunks zu einem einzigen zusammengeschlossen werden.

SMALLDATA/SD

Mit Hilfe dieser beiden Schlüsselwörter wird dafür gesorgt, daß alle Daten-Hunks (initialisierte und uninitialisierte (BSS) Daten) zu einem Hunk zusammengefaßt werden.

TO

Das File, das nach diesem Schlüsselwort angegeben wird, bestimmt, welchen Namen das lauffähige Programm erhalten soll. Oder anders gesagt: Dieser File-Name bestimmt, zu welchem File die angegebenen Objekt-Files und Library-Routinen zusammengebunden werden sollen.

VERIFY/VER

Mit Hilfe dieses Schlüsselwortes können Sie ein File festlegen, in das alle Linker-Nachrichten geschrieben werden sollen. Geben Sie dieses Schlüsselwort nicht an, wird der Bildschirm zur Ausgabe benutzt.

VERBOSE

Mit Hilfe dieses Schlüsselwortes können Sie den Linker veranlassen, den Namen jeder Routine auszugeben, die gerade untersucht wird.

XREF

Mit Hilfe dieses Schlüsselwortes legen Sie fest, wohin das Cross-Reference-Listing geschrieben werden soll.

Stichwortverzeichnis

<exec/memory.h> 499

= 128, 144
< 128, 145
> 128, 145
>> 128, 145
? 129, 145

640*512 Punkte 282
720*452 Punkte 263
68000er 496
8 MByte 497

Abfrageroutine 365
Abhängigkeiten 522
Acc 136
ACCESS_READ 164
ACCESS_WRITE 164
Acd 136
Acs 136
Add 119, 137
ADDR 117, 134
Adi 119, 137
Adl 119, 137
Adp 119, 137
Adr 119, 137
Adresse 503
Ae 137
Ai 119
Ak 119
Aktivierungsbereich 428
Al 120
Alert-Nummer 378
Alert-String 379
Alert-Struktur 380
Alert-Text 379

Alert-Typ 378, 382
Alerts 377
Alt 414, 485, 486, 494
Am 120, 137
Amiga-Symbol 432
Amiga-Taste 432
An 120
Ap 120
Aq 120
Ar 121
As 121
ASCII 436, 448
ASCII-Code 430, 475
ASCII-Ziffern 475
Assembler 217
Assembler 453
Assembler-Code 451, 563
Assembler-Modus 494
Assembler-Optionen 100
At 121
Audio.Device 391
Auflösung 260
Augenschmerzen 261
AUSDR 116, 134
Ausgabe aller Files 178
Ausgabe-Window 418
Ausgaben 261
Auto-Indent 634
Automatik-Requester 359
AutoRequest 377
AutoRequester 363, 378
Ax 137
Aztec-DB 112
Aztec-Source-Level-Debugger 129
Aztec-Compiler 17

BACKDROP-Window 210
Backspace-Taste 478
BackspaceChar 633, 645
Baukastensystem 285

Bb ... 121
Bc ... 122, 138
Bd ... 122, 138
Be ... 138
BECKERtext ... 308
BEREICH ... 118, 135
Betriebssystem ... 386
Bh ... 122
Bildschirmzeilen ... 379
Bit-Gruppen ... 408
BitMap ... 209, 265, 267
BitPlanes ... 301
Bl ... 121
Blockstift ... 206
Boolean-Gadgets ... 314
Booten ... 204
Border ... 209
Border-Struktur ... 294, 295, 322, 324
BORDERLESS ... 209
Bq ... 123
Br ... 123, 139
Breitschrift ... 289
Bs ... 123, 139
Bt ... 123, 139
Bu ... 123
Bw ... 121

C ... 139
C-Code ... 451
C-Programm ... 217, 453
C-Quelltext ... 453
C-Quelltext-Editor ... 284, 305
Cancel ... 364
CapsLock-Taste ... 494
Cast-Anweisung ... 221
Casting ... 537
CheckMark ... 207
Checkmark ... 440, 453
Chip-Memory ... 305, 347
Chip-RAM ... 496

CHIP_MEMORY 282
CLI 478
CLI-Editor 483
CLI-Fenster 305
Clipboard.Device 391
Clipping 570
Close-Gadget 210, 213, 270
Close_All() 230, 231, 270
CloseEditor 535
CMDLISTE 119, 135
Cn_utoa 686
CommandASCII 698
CommandBracket 703
CommandCursor 699
CommandDelete 700
CommandExit 703
CommandLine 701
CommandQuit 701
CommandSet 701
Compiler 302
Compiler-Optionen 57
Console.Device 391, 467, 468
ConsoleBase 539
ConsoleReadMsg-Struktur 470
ConsoleWriteMsg-Struktur 470
Container 328
ConvertLineForPrint 574, 583, 661
ConvertSpstToZeile 549, 555, 562
Cs 124
CSI 475, 479
Ctrl 414, 485, 486, 494
Ctrl-C 182
Cursor 599, 603
Cursor-Position 475, 480
CursorBottom 727
CursorDown 598, 610
CursorEnd 725
CursorHome 666
CursorLeft 598, 609
CursorOnText 636

CursorRight 598, 609
CursorTop 726
CursorUp 598, 610
Custom-Chips 497
Custom-Gadget 266, 276
CustomRequester 366
CustomScreen 207, 265

D 124, 140
Da 139
Daten lesen und schreiben 156
Datenleitungen 469
Datenpuffer 477
Datenstrukturen 513
Datentransport 399
Db 124, 140
Dc 124, 140
Dd 124, 140
Debugger 217, 625, 672
DeconvertLine 637
Default-Font 288
Default-Struktur 307
Default-Struktur-Werte 307
Default-Werte 222
Default-Zeichensatz 424
DefaultTitle 274
Deklarieren 538
Delay() 231
DeleteChar 633, 644
DeleteLine 634, 653, 732
Delta-Werte 280
Depth-Arrangement-Gadgets 213
DEVICE STATUS REPORT 480
Df 140
Dg 124, 140
Directorys anzeigen 174
Diskfont.library 288, 289
Dl 124, 140
DMRequest 376, 407
Doppelkasten 294

DOS 216, 477
DOS-Fenster 361
DOS-Window 326, 337, 378
Double-Menu-Requester 375
Drag-Gadget 213
Dringlichkeit 377
Ds 125, 141
Dw 124, 140

E 144
Editor.pre 520
Editor.prelist 520
Editorfenster 485
Effizienz 590
Ein- und Ausschalten 314
Einbinden von Assembler-Programmen 685
Eindimensional 314
Eingebundene Message-Struktur 399
Einzel-Code 491
EnterFold 664
Erstellungsdatum 170
Exec/types.h 537
ExecuteCommand 704
Exit 154
ExitFold 665

Fachliteratur 470
Falten-Level 596, 658
Faltenanfangsmarkierung 595
Faltenende 595
Farbauswahl 453
Farbauswahl-Untermenü 443
Farbbalken 443
Farbe 260
Farbstifte 236
Fast-RAM 496
Fd 141
Fehlerklasse 388
Fehlermeldung 388, 503
Fehlerquelle 386

Fehlersuche ... 629
Fettschrift ... 289
File-Arbeit ... 451
File-Handling ... 152
File-Informationen ... 162
File-Select-Box ... 349f, 359, 369, 417
FileInfoBlock ... 165
FileLock ... 174
Fließtext ... 724
Folding ... 512, 575, 593
FreeSpeicherblock ... 547, 554
Freigaberoutine ... 499
Fu ... 141
Funktions-Offset ... 217, 278
Funktionstabellen ... 149

G ... 141
Gadget-Abfrage ... 270, 404
Gadget-Struktur ... 266, 321
GADGHIGHBITS ... 316
Gameport.Device ... 391
GarbageCollection ... 546, 551, 556, 707
Geisterschrift ... 419
Gesamtlänge ... 217
GetFoldInc ... 638
GetLastWord ... 733
GetLineForEdit ... 632, 636
GetPufferPointer ... 643
Ghosted ... 419
GIMMEZEROZERO ... 211, 570
GIMMEZEROZERO-Window ... 212
Graphics.library ... 285
Grafik-Menüpunkt ... 435
Grafikausgabe ... 261
Grafikprogramme ... 248
Grafikspeicher ... 209, 282
Grafikverwaltung ... 453
Graphics.library ... 392
Grundtyp ... 315
Grundvoraussetzungen ... 261

Guru-Meditation 377, 386
Guru-Nummer 386

HalfPageDown 611
HalfPageUp 611
HAM 261, 265
HandleKeys 599, 612, 666
Hauptprogramm 535
Hauptschleife 537
Hexadezimal 217
Highlighting-Flags 434
HoleZeile 545, 550

IDCMP 391, 468
IDCMP-Flag 325, 364, 376, 398, 411
If-Befehl 479
IFF 436, 448
Image-Struktur 300, 302, 322
Include-File 379
Informationsaustausch 203, 312
Infozeile 684
InitFolding 735
InitWindowSize 575, 582
Inner window 211
Input.Device 391
InsertChar 633, 646
InsertLine 633, 649
Interlace-Screen 283
IntuiMessage-Struktur 398
IntuiText-Struktur 287, 290, 306, 323, 440
Intuition-Library 473
Intuition.library 215, 242, 285
INVERSEVID 286, 295
IoErr 158
IOStdReq 472, 488
IOStdRequest-Struktur 469, 484
ITEMNUM 408

JAM1, JAM2 ..... 286, 295
Joker 180

Keyboard.Device ... 391
KeyMap-Belegung ... 488
KeyMap-Struktur ... 488
Kickstart-ROM ... 496
Klickfelder ... 294, 313
Klickimpuls ... 313
Klickpunkt ... 236
Knobs ... 328
Kommando-Sequenz ... 435, 438, 475
Kommando-Taste ... 429
Kommandozeilen-Parameter ... 149
Kommunikation ... 203
Kommunikationskanal ... 364
KONSTANTE ... 116
Koordinatenauswahl ... 467
Koordinatentabelle ... 296
Kursivschrift ... 289
Kurzschreibweise ... 240

Lattice-Compiler ... 51
Laufwerk ... 417
Ld ... 141
Lese-Funktion ... 472
Link-Pointer ... 423
Linke Maustaste ... 313
Linker-Optionen
Linkpointer ... 273
Ll ... 141
LoadASCII ... 695
Lock ... 164
LoescheZeile ... 561
Lokale Variablen in Blöcken ... 676
Lp ... 141
Ls ... 125

Ma ... 125
Main ... 535, 718
Main() ... 231, 240
Make ... 23, 520, 542
Make-File ... 521

Maus-Cursor 236, 429
Mauskoordinaten 234, 405
Mauspfeil 428
Maustaste 411, 426
Maximalwerte. 234
Maximum 206
Mb 126, 142
Mc 126, 142
Medium 418
Mehrere 510
MEMF_CHIP 497
MEMF_FAST 497
MEMF_PUBLIC 497
Menu-Strip 423
Menu-Struktur 424
MenuItem-Kette 436
MenuItem-Struktur 427, 432, 436
MENUNUM 408
MENUPICK-Flag 407
Menü-Überschrift 429
Menüleiste 426, 436
Menüpunkt 428
Menüs 417
Menütaste 426
MessagePort 529
MessageWaiting 704
Mf 126, 142
Minimalwerte. 234
Minimum 206
Ml 126, 142
Mm 126, 142
MODE_NEWFILE 153
MODE_OLDFILE 153
Modulsystematik 205
Ms 126, 143
MsgPort-Strukturen 400
Multi-Alert 383
Multitasking-Betriebssystem 377
Multitasking-Computer 402

MutualExclude 429, 440
Mw 126, 142

Narrator.Device 391
Nb 126
Nd 126
NeuerBlock 544, 547
NeueZeile 543, 553
Neustart 386
NewCLI-Window 476
NewScreen-Struktur 262, 267, 270
NewWindow-Struktur 216, 220
NextLine 578, 616
No 126
Null-Byte 473
Null-Pointer 402, 497
Numericpad 415
NX 126

Offsets 271, 292
Open 152
Open_All() 230, 231, 270
OpenEditor 534, 579, 714
OptimiereBlock 551, 556
Optimieren 564
Option +H 520
Option +I 520
Öffnen von Dateien 152

PAL-Fernseher 282
PAL-Version 264
Parallel.Device 391
Pointerfeld 306
PowerWindows 450
Präcompilieren 520
Preferences 288
Preferences-Menü 451
PrevLine 579, 617
PrintAll 573, 586
PrintAnzZeilen 689

PrintAt 575, 584, 622
Printer.Device 391
PrintFlags 690
PrintFold 690
PrintInfo 691
PrintLine 574, 586, 622, 661
PrintXpos 688
PrintYpos 689
PropInfo-Struktur 327, 330
Proportional-Gadget 326
Proportionen 314
Protection-Bit 171
Prozessor-Trap 388
Puffern 208

Q 127, 143
Qualifier 414, 487, 493
Quelltext 453, 494

R 127, 143
RastPort 291
RAWKEY 414, 530
RAWKEY-Abfrage 467
RAWKEY-Codes 485, 488
Read 157
Read-Port 468
RecalcTopOfWindow 663
Refresh-Mode 250
Regel 524
REGISTER 116
Register-Variablen 676
Remember-Struktur 503
Rename-Window 406
Repeat-Status 415
Report 488
Requester 358, 359
Requester-Gadget 359
Requester-Struktur 367
Requester-Window 364
RestoreZeilenptr 602

Retry ... 364
Return ... 477
Return-Taste ... 337, 451
RETURN_ERROR ... 155
RETURN_FAIL ... 155
RETURN_OK ... 155
RETURN_WARN ... 155
RMBTRAP ... 411
ROM-Font ... 288

S ... 127, 143
SaveASCII ... 694
SaveIfCursorMoved ... 632, 642
SaveLine ... 632, 639
Schalter in der Kommandozeile ... 147
Schiebebalken ... 206, 326
Schiebekasten ... 315
Schließen von Dateien ... 152
Schriftarten ... 440
Schriftbreiten ... 440
Schrifttyp ... 440, 448
Screen ... 260
Screen-Struktur ... 267, 270, 271, 274
Screen-Titel ... 266, 274
ScrollDown ... 598, 607
Scrollen ... 212
Scrollen ... 481
Scrolling ... 594
ScrollLeft ... 598, 605
ScrollRight ... 598, 603
ScrollUp ... 598, 606
SDB-Optionen ... 130
Select-Box ... 428, 432, 435
SELECT-Flag ... 344
Select-Gadget ... 345
Serial.Device ... 391
SHELL ... 718
Shift ... 414, 481, 485, 486, 494
Signal ... 537
SIMPLE_REFRESH ... 208, 569

Sizing-Gadget 213
SMART_REFRESH 208, 569
Sonderzeichen 512
Source-Code-Utility 449
Speicherausführung 378
Speicherbedarf 208
Speicherbereiche 496
Speicherorganisation 496
Speicherplatz 208
Speicherstueck 555
Speichertyp 503
Speicherverwaltung 495
Speicherverwaltung 509, 543
Sprites 236
Sprungstärke 329
Src/Ausgabe.c 581
Src/command.c 692
Src/Cursor.c 601
Src/Edit.c 634
Src/Editor.c 532
Src/Editor.h 530
Src/Speicher.c 546
Src/Test.c 557
Startposition 295
StdIOs 468
Steuersequenzen 473, 475, 479
Steuertasten 491
String 266, 473, 488
String-Gadget 331, 406, 683
StringInfo-Struktur 331
Struct BList 517
Struct Editor 517, 600
Struct EList 532
Struct FList 516
Struct Speicherblock 515
Struct Speicherstueck 515
Struct Zeile 513
Struct ZList 517
SUBITEM 408
SubItems 438

Subsystem-Liste ... 388
SucheSpeicherstueck ... 548, 554
SUPER_BITMAP ... 208, 212
Switch() ... 326
Systemabsturz ... 426
System-Gadgets ... 206, 212, 313
System-Grafik ... 406
System-Requester ... 358, 360, 364, 377
Systemzeit ... 196

T ... 127, 143
Tabulator ... 451, 594, 630
Tabulatoren ... 511
Tastaturwiederholfunktion ... 539
Tastatur-Codes ... 470
Tastaturbelegung ... 417
Tastatureingaben ... 470
Tastaturtabelle ... 485, 488
Tastenkombination ... 494
TERM ... 116
Text ... 436, 448
Text-Menüpunkt ... 435
TextAttr-Struktur ... 266, 288, 289, 440
Textbearbeitung ... 633
Texteditor ... 205
TEXTOMAT ... 308
Textpuffer ... 472, 498
Textpuffer-Funktionen ... 482
Textverarbeitung ... 208, 241, 248, 261, 425, 436
Timer.Device ... 391
Titel ... 235
Titelleiste ... 211, 265
Titeltext-String. ... 274
Toggle-Gadgets ... 341
Topaz.font ... 289, 290
Trackdisk.Device ... 391

U ... 127, 143
UND-Verknüpfung ... 342
UNDO ... 315

Undo-Puffer 353
UndoLine 633, 642
Untergrund 261
Untermenü 431, 439
Unterstreichen 289
User-Gadget 324, 352

V 128
V0.1 542
V0.2 568
V0.3 590
V0.4 625
V0.5 672
V0.6 736
VANILLAKEY 415
VANILLAKEY 530
VANILLAKEY-Abfrage 467
Variablenfeld 495
Verkettung 236, 509
Version 1.2 242
Version 1.8 276
Verwaltungsaufwand 508
Video-Chip 262, 265

Wahrheitswert 366
Wait() 325
Wait-Status 325, 403
Wiederholfunktion 495
Wildcards 180
Window-Puffer 477
Window-Struktur 233
Window-Titel 211
Wordwrap 734
Workbench 262
Workbench-Diskette 288
Workbench-Menüs 418
Workbench-Screen 262, 265, 314
Workbench-Windows 205
Write 157
Write-Port 468

X 128, 144

Z 144
Zeichen-Codes 473
Zeichenfarbe 306
Zeichensatz 266, 288, 306
Zeichensatzeinstellung 420
Zeichenstift 206, 425
Zugriff auf absolute Speicherstellen 146
Zweidimensional 314
Zwischenspeicherung 208

Von der Hardware über den Betriebssystemkern EXEC bis zum DOS finden Sie in diesem Buch die entscheidenden Informationen zum Amiga. Und zwar so verständlich, daß auch Nicht-Profis die Arbeitsweise des Amiga-Betriebssystems schnell verstehen werden.

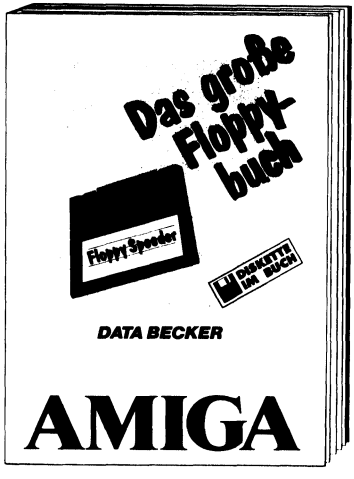


Aus dem Inhalt:

- die Chips des Amiga (68000, CIA, Agnus, Denise, Paula)
- Die Schnittstellen (Video, Audio, RGB, Centronics, seriell, Floppy, Gameport, Expansionport, Tastatur)
- Programmierung der Hardware (Speicherbelegung, Interrupts, Copper, Blitter, Diskcontroller)
- Strukturen des EXEC (Node, List, Libraries, Tasks)
- Funktion des Multitasking (Task-switching, Kommunikation zwischen Tasks, Exceptions, Traps, Speicherverwaltung)
- I/O-Handhabung beim Amiga durch Devices und I/O-Request)
- Interrupt-Handhabung und Verwaltung der Ressourcen
- RESET-feste Programme und Strukturen, Dokumentation der RESET-Routine
- EXEC-Base (Dokumentation und Nutzung der Systemvariablen)
- DOS-Bibliothek (Funktionen, Parameter, Fehlermeldungen)
- Disketten (Bootvorgang, Datenstrukturen, Programmaufbau, IFF-Format)
- Programmstart, Parameter, Aufruf von CLI und Workbench, detaillierte Beschreibung des internen Aufbaus der CLI-Befehle (interne DOS-Bibliothek)
- Ein-/Ausgaben (Tastatur, Bildschirm, Diskette, parallele und serielle Schnittstelle)
- Devices (Trackdisk, Console, Narrator, SER, PAR, PRT, Gameport)

Dittrich, Gelfand, Schemmel
AMIGA Intern
Hardcover, 716 Seiten, DM 69,-
ISBN 3-89011-104-1

Dieses DATA BECKER Floppybuch beschreibt alles Wissenswerte zum Thema Floppy ausführlich und sofort nutzbar. Zusätzlich finden Sie in diesem Buch Insider-Informationen über Kopierschutz, Speed-Routinen und Viren, die in einschlägigen Kreisen für Aufsehen sorgen werden. Daß diese Informationen in einfach zu bedienende Programme umgesetzt wurden, macht dieses Buch zu einer unersetzlichen Hilfe bei der täglichen Arbeit mit dem Amiga.



Aus dem Inhalt:

- Floppy-Operationen unter Workbench und CLI
- BASIC: Laden, Speichern, sequenzielle und relative Dateien
- DOS-Funktionen
- Fileverwaltung: Blocktypen, Boot-Block, Checksummen, File-Header, Hash-Berechnung, Bitmap
- Viren: Boot-Block und Schutz
- Trackdisk-Device: Befehle, Strukturen, Nachrichten
- Trackdisk-Task: Funktion und Aufbau
- Diskettenzugriff ohne DOS: MFM, GCR, Aufbau von Tracks, Block-Header, Datenblock, Prüfsummen, Codierung und Decodierung, Hardware-Register, SYNC, Interrupts
- Diskmonitor
- Fast-Copy: Komplette Diskette in ca. 1 Minute kopieren
- Crunch-Copy: Durch Packroutinen weniger Diskettenwechsel
- Deep-Copy: Kopiert praktisch jeden Kopierschutz, ST- und PC-Disketten
- Floppyspeeder: Superschnelle Trackdisk-Routinen beschleunigen den Diskettenzugriff

Abraham, Gelfand, Grote
Das große Amiga-Floppybuch
557 Seiten, DM 59,-
ISBN 3-89011-180-7

Schreiben Sie Ihre Programme in Maschinenesprache – und Sie werden sehen, wie schnell ein Amiga sein kann. Das nötige Know-how liefert Ihnen dieses Buch: Grundlagen des 68000, das Amiga-Betriebssystem, Druckeransteuerung, Diskettenoperationen, Sprachausgabe, Windows, Screens, Register, Pull-Down-Menüs . . . Aber es wird auch gleich gezeigt, wie man mit den wichtigsten Assemblern arbeitet.



Dittrich
Amiga Maschinensprache
Hardcover, 288 Seiten, DM 49,–
ISBN 3-89011-076-2

Schreiben Sie Ihre Programme in Maschinenesprache – und Sie werden sehen, wie schnell ein Amiga sein kann. Das nötige Know-how liefert Ihnen dieses Buch: Grundlagen des 68000, das Amiga-Betriebssystem, Druckeransteuerung, Diskettenoperationen, Sprachausgabe, Windows, Screens, Register, Pull-Down-Menüs . . . Aber es wird auch gleich gezeigt, wie man mit den wichtigsten Assemblern arbeitet.



Dittrich
Amiga Maschinensprache
Hardcover, 288 Seiten, DM 49,–
ISBN 3-89011-076-2

DAS STEHT DRIN:

Wer den Einstieg in C geschafft hat, wird sich so manches Mal fragen, warum das gut durchdachte Programm nicht funktioniert. Hier helfen fundierte Informationen von Profis weiter: Funktionsweise von Compiler Assembler und Linker (am Beispiel von Aztec 3.6 und Lattice 4.0), komfortable Oberflächen für eigene Programme mit Intuition und ein großes Projekt professionell programmiert – mit diesem Buch bleibt keine Frage offen.

Aus dem Inhalt:

- Funktionsweise des Aztec-Compilers (Compiler, Assembler, Linker)
- Wie funktionieren INCLUDE, DEFINE und CASTS?
- Debugging und Optimierung des Assembler-Sources
- Knifflige Programmierprobleme: Sprungtabellen und dynamische Arrays in C
- Einbinden von Assembler-Source in den C-Source
- Alles über Intuition-Programmierung (Windows, Screens, Pulldown-Menüs, Requester, Gadgets)
- Programmierung des Console-Device
- Ein professioneller Editor zeigt alle Probleme bei der Erstellung größerer Programme
- Richtiger Einsatz von MAKE
- Debugging von C-Programmen mit verschiedenen Hilfsmitteln
- Ein kompletter Text-Editor als Source in mehreren Entwicklungsstufen
- Folding-Technik (Der Editor kann Textteile und Funktionen wegfalten, das erhöht die Übersichtlichkeit enorm.)

UND GESCHRIEBEN HABEN DIESES BUCH:

Bruno Jennrich stellte bereits mit den Büchern „Amiga Intern Band II“ und „Amiga 3-D-Grafikprogrammierung“ seine Programmierkenntnisse unter Beweis. Wolf-Gideon Bleek hat in dem Buch „Amiga Tips & Tricks“ viele Kniffe zur Workbench und zu Intuition vorgestellt. Peter Schulz hat mit dem PROFIMAT AMIGA professionelle Programmierqualitäten bewiesen; mit dem Texteditor in diesem Buch hat er endlich „seinen“ Editor geschrieben.

ISB N 3-89011-191-2 DM +069.00

DM 69,–
ÖS 538,–
sFr 67,–

06900

9 783890 111919

DATA BECKER