Für Amiga 500/1000/2000

Amiga
Das Programmierbuch

Robert A. Peck

Amiga
Das Programmierbuch

Amiga
Das Programmierbuch

Robert A. Peck

DÜSSELDORF · SAN FRANCISCO · PARIS · LONDON

Anmerkungen:

Amiga und AmigaDOS sind eingetragene Warenzeichen von Commodore-Amiga, Inc.
UNIX ist ein eingetragenes Warenzeichen von Bell Laboratories, Inc.

Titel der amerikanischen Ausgabe: Programmer's Guide to the Amiga™
Original © 1987 SYBEX Inc., Alameda, CA 94501

Deutsche Übersetzung: Jörg Anslik, Dirk Schaun

Satz: Edgar Brüggemann
Titelgestaltung: vékony color / tgr – typo-grafik-repro gmbh, Remscheid
Gesamtherstellung: Druckerei Hub. Hoch, Düsseldorf

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch bzw. Programm und anderen evtl. beiliegenden Informationsträgern zu publizieren. SYBEX-Verlag GmbH, Düsseldorf, übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf eine Fehlfunktion von Programmen, Schaltplänen o. ä. zurückzuführen sind, nicht haftbar gemacht werden, auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultiert.

ISBN 3-88745-520-7
1. Auflage 1988
2. Auflage 1988

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Printed in Germany
Copyright © 1988 by SYBEX-Verlag GmbH, Düsseldorf

Inhaltsverzeichnis

Vorwort 15

Kapitel 1: Überblick **19**
Aufbau der Soft- und Hardware-Hierarchie des Amiga 19
Die unterste Stufe der Systemsoftware 20
Die zweite Stufe der Systemsoftware 22
Die dritte Stufe der Systemsoftware 22
Die oberste Stufe der Systemsoftware 24
Was Sie wissen müssen, um den Amiga zu programmieren 25
Funktionsbibliotheken 25
Die Programmierung in C 28
Die Programmierung in 68000-Assembler 28
Die "Include"-Dateien des Compilers 29

Kapitel 2: AmigaDOS **31**
Ausgabe von Text in ein CLI-Fenster 32
Übergabe von Argumenten beim Programmaufruf 33
Die Umleitung der Standard-Ein-/Ausgabe 34
Das Arbeiten mit Files (Dateien) 35
Öffnen einer Datei 36
Schließen einer Datei 36
Lesen aus einer Datei 37
Schreiben in eine Datei 38
Fenster-Ein-/Ausgabe mit Hilfe der AmigaDOS-Dateifunktionen 38
Das Öffnen eines neuen CLI-Fensters 39
Eingaben von einem CLI-Fenster 40
Übergabe von Eingabezeilen ohne Betätigung der <Return>-Taste ... 41
Die Ausgabe von Dateien auf einen Drucker 42
Weitere Funktionen zum Arbeiten mit Dateien 45
Finden der Position des Schreib-/Lesezeigers einer Datei 45
Prüfen, ob Ihr Programm mit dem CLI kommuniziert 46
Eine bestimmte Zeit auf die Eingabe eines Zeichens warten 46

Die Struktur des Inhaltsverzeichnisses (Directory) einer Diskette 47
Arbeiten mit AmigaDOS 49
Funktionsaufruf oder Execute? 50
Verwendung des Zeigers, den die Open-Funktion liefert 50
Die "Zweige" des Directory-Baumes und wie man sie anspricht 51
"Wandern" auf dem Directory-Pfad 52
Auslesen der AmigaDOS-Fehlernummern 54
Anfordern eines "Locks" für Dateien und Verzeichnisse 54
Die Freigabe von Dateien und Verzeichnissen (UnLock) 55
Wechsel von einem Verzeichnis in ein anderes 56
Informationen über eine Datei oder ein Verzeichnis 57
Informationen über weitere Dateien in einem Verzeichnis 60
Rückkehr zum nächsthöheren Verzeichnis 61
Beispielprogramm: Anwendung von Directory-Funktionen ... 61
Bestimmung des aktuellen Verzeichnisses 61
Hilfsfunktionen des AmigaDOS 66
Hilfsfunktionen des CLI 67
Umbenennen einer Datei 67
Löschen einer Datei 68
Anlegen eines neuen Verzeichnisses 69
Schützen einer Datei 69
Dateien oder Verzeichnisse mit einem Kommentar versehen ... 70
Auslesen des Datums 71
Auslesen von diskettenspezifischen Informationen 71
Verschiedene andere Funktionen 74
Feststellen, mit welcher Diskette das System gebootet wurde ... 74
Erzeugung einer Warteschleife 75
Herausfinden, welcher Task welches Gerät verwendet 75
Ändern des Diskettennamens 76

Kapitel 3: Exec 79
Die Struktur von Exec 79
Warum Listen so wichtig sind 80
Einige Funktionen von Exec und ihre Bezeichnungen 81
Tasks und Prozesse 82
Speicherreservierung 83
Eine einfache Speicherreservierung 83
Die Freigabe von reserviertem Speicher 85
Ein Programm zur Speicherreservierung 86
Listen 87
Initialisierung der Kopfzeile einer Liste 87

Signifikanz der "List Nodes" 88
Routinen, die Listen manipulieren 89
Beispielprogramm: Anwendung von Listen-Funktionen 90
Signale 92
Was geschehen kann, wenn ein Signal erzeugt wurde 92
Signifikanz von Signalen beim Multitasking 93
Reservierung eines Signal-Bits 93
Verwendung von Signal-Bits beim Multitasking 94
Direktes Setzen eines Signal-Bits 95
Verwendung von mehreren Signal-Bits 95
Virtuelle Kommunikationsleitungen (Message Ports) 97
Erstellen und Löschen einer Kommunikationsleitung 98
Hinzufügung und Wegnahme einer Kommunikationsleitung 99
Das Auffinden einer Kommunikationsleitung 100
Beispiele zur Verwendung einer Kommunikationsleitung 101
Erkennen von Signalen auf einer Kommunikationsleitung 102
Nachrichten (Messages) 102
Wozu benötigt man Nachrichten? 103
Der Inhalt einer Nachricht 103
Signifikanz von Nachrichten 104
Eigene Nachrichten (Custom Messages) 105
Funktionen zum Arbeiten mit Nachrichten und Kommunikationsleitungen 106
Beispielprogramm: Arbeiten mit Messages und Message-Ports 106
Bibliotheken 106
Die Struktur einer Bibliothek 109
Das "lib_Flags"-Feld 110
Das "lib_Pad"-Feld 111
Das "lib_NegSize"-Feld 111
Das "lib_PosSize"-Feld 112
Die Felder "lib_Version" und "lib_Revision" 112
Das "lib_IdString"-Feld 112
Das "lib_Sum"-Feld 112
Das "lib_OpenCnt"-Feld 113
Öffnen einer Bibliothek 113
Die Namen und Basisadressen der Bibliotheken 114
Verwendung der Funktionen einer Bibliothek 116
Schließen einer Bibliothek 117
Programm zum Öffnen, Benutzen und Schließen einer Bibliothek 117
Virtuelle Geräte (Devices) 118
Vorbereitung und Durchführung einer Ein-/Ausgabeoperation 119

Befehle eines virtuellen Gerätes 120
Das Öffnen eines virtuellen Gerätes 121
Die Bezeichnungen der verfügbaren Devices 122
Die Struktur eines typischen "IORequest"-Blocks 122
Das Mindestmaß an Initialisierung für eine Ein-/Ausgabe 125
Senden eines Befehls an ein virtuelles Gerät 126
Einige Bemerkungen zum "Request Block" 126
SendIO 126
DoIO 127
Weitere Funktionen zur Ein-/Ausgabe 128
Beispielprogramm: Anwendung der Ein-/Ausgabefunktionen 128
Warum verwendet man "Reply Ports"? 131
Stapelverarbeitung von mehreren Ein-/Ausgabeoperationen 131
Zugriff auf eine Bibliotheks-Funktion für ein virtuelles Gerät 132
Das Schließen eines virtuellen Gerätes 133

Kapitel 4: Grafik 135
Das Öffnen eines Fensters auf dem "Workbench-Screen" 136
Definition und Initialisierung eines neuen Fensters 136
Das "IDCMP Flags"-Datenfeld 136
Das "Window Flags"-Datenfeld 138
Öffnen des Fensters 139
Verarbeitung der Ereignisse, die Intuition sendet 140
Auffinden des "RastPort" 140
Ausgabe von Grafik in einem Fenster 142
Auswahl der Farben 143
Der "APen"-Zeichenstift 143
Der "BPen"-Zeichenstift 144
Der "OPen"-Zeichenstift 145
Auswahl des Zeichenmodus 145
Zeichnen der Achsen 146
Bezeichnen der Achsen 147
Zeichnen von Rechtecken 150
Rechtecke, die nur aus Linien zusammengesetzt sind 150
Mit Farben und Mustern gefüllte Rechtecke 150
Rechtecke mit zweifarbigen Mustern 151
Rechtecke mit mehrfarbigen Mustern 153
Allgemeines über Muster 154
Das Zeichnen der Balken relativ zum Ursprung des Koordinatensystems 155

Punktierte Linien 156
Zeichnen von mehreren Linien mit einem einzigen Funktionsaufruf . 157
Das Hauptprogramm 157
Verhindern, daß der Fensterinhalt neu gezeichnet wird 160
Fenster vom Typ "SMART_REFRESH" 161
Fenster vom Typ "SUPER_BITMAP" 162
Erstellen und Öffnen eines eigenen Bildschirms (Custom Screen) 166
Definition des Custom Screens 167
Öffnen des Custom Screens 167
Öffnen eines Fensters auf dem Custom Screen 168
Auswahl der Farben 170
Bestimmung der aktuell verwendeten Farben 172
Anwendung der Füllroutine Flood 173
Füllen von Bereichen mit eigenen Mustern 174
Vorbereitungen zur Arbeit mit AreaMove und AreaDraw 176
Lesen und Setzen beliebiger Bildpunkte (Pixels) 178
Zeichnen einer Landkarte 178
Text 183
Öffnen eines Zeichensatzes 184
Definition der "TextAttr"-Datenstruktur 185
Einbinden des Zeichensatzes in den RastPort 186
Eintragen eines neuen Zeichensatzes in der Zeichensatzliste des Systems 187
Bestimmen, welche Zeichensätze verfügbar sind 187
Die Darstellungsweise des Textes 191
Schriftarten: Fett, Kursiv oder Unterstrichen 192
Löschen und Verschieben (Scrollen) von Bildschirmbereichen 194
Kombination mehrerer Komponenten zu einem Bild 196
Initialisierung einer "BitMap" 196
Initialisierung eines "RastPort" 197
Kopieren der Daten einer BitMap in eine andere 198
Anwendung der Funktionen zum Kopieren von Grafikdaten 200
Kopieren des "unsichtbaren" Hintergrunds 200

Kapitel 5: Die Benutzeroberfläche "Intuition" 215
Die Kommunikation mit Intuition 216
Nachrichten (Messages) von Intuition 219
Der Inhalt einer "IntuiMessage" 219
Das "Class"-Datenfeld 220
Das "Code"-Datenfeld 220

Das "Qualifier"-Datenfeld 220
Das "IAdress"-Datenfeld 221
Die "MouseX"- und "MouseY"-Datenfelder 221
Die "Seconds" und "Micros"-Datenfelder 222
Das "IDCMP Window"-Datenfeld 223
Beispielprogramm: Anwendung einer IDCMP-Routine 223
Entwicklung eines Malprogrammes 224
Die Auswahl des Bildschirms (Screen) 230
Die "New-Screen"-Datenstruktur 230
Die "NewWindow"-Datenstruktur 232
Bestimmen der Position des Mauszeigers 236
Auslesen des Status der linken und rechten Maustaste 238
Die Erstellung und Verwendung von Menüs 239
Die Beziehung zwischen Menüs und Menükomponenten 239
Initialisierung von Menüs 240
Initialisierung von Menükomponenten 244
Text oder Grafik 245
Farbige Grafiken 247
"Checkmarks" 250
"Mutual Exclusion" – Automatisches Freigeben von Menüpunkten 252
Hervorheben von Menüpunkten bei ihrer Anwahl 253
Requester 255
Gadgets 257
Die Position eines Gadgets 257
Die Größe eines Gadgets 257
Das Aussehen eines Gadgets 257
Die Identifikation von Gadgets 258
"Mutual Exclusion" – Automatisches Freigeben von Gadgets 259
Gadgettypen 259
Verfügbare Flags für Gadgets 259
Die Anwahl von Gadgets 260
Gadgets vom Typ "Boolean" 263
Gadgets vom Typ "String" 264
Gadgets vom Typ "Proportional" 266
Menüverarbeitung 269
Gadgetverarbeitung 270
Das Malprogramm 271
Zusätzliche Extras 278
Die Kombination von Text und Grafiken 278
Menükomponenten 279

Kapitel 6: Devices (Virtuelle Geräte) 285
Der Zeitgeber (Timer Device) 286
Das "Console Device" 293
Die Codes der einzelnen Zeichen 294
Eingaben vom Typ "RAWKEY" 298
Kontrollieren des "Console Device" 299
Das "Input-Device" 306
Die Tastatur 306
Der Joystick-/Mausport 306
Tastatur-Spezialitäten 310

Kapitel 7: Animation (Bewegte Grafiken) 313
Simple-Sprites 313
Die "SimpleSprite"-Datenstruktur 314
Reservieren eines Sprites 315
Ändern eines Sprites 316
Die Daten eines Sprites 317
Die Farben eines Sprites 318
Freigabe eines Sprites 320
Beispielprogramm: SimpleSprite 321
"Virtual Sprites" 330
Die Vorteile eines "Virtual-Sprites" 331
Die Nachteile eines "Virtual-Sprites" 331
Die Initialisierung des Gel-Systems 332
"SpriteHead" und "SpriteTail" 333
Reservierte Sprites 333
Die Datenfelder "NextLines" und "LastColors" 336
Die MakeVSprite-Routine 337
Die "VSprite"-Datumstruktur 337
Das Datenfeld "Height" 337
Das Datenfeld "Image" 340
Das Datenfeld "Colors" 340
Die Datenfelder "x" und "y" 341
Das Datenfeld "Width" 341
Das Datenfeld "Depth" 341
Das Datenfeld "Flags" 341
Beispielprogramm: VirtualSprite 341
Blitter Objects (Bobs) 351
Die MakeBob-Routine 351
Das "Image"-Zeiger 351

Die Datenfelder "PlanePick" und "PlaneOnOff" 354
Die PurgeGels-Routine 356
Die Vorteile eines "Bobs" 358
Die Nachteile eines "Bobs" 358
Beispielprogramm: BOBS 359

Kapitel 8: Sound **371**
Die Audio-Hardware 371
Die Kommunikation mit dem Audio-Device 372
Die Audio-Software 374
Reservierung von Kanälen 375
Auf die Reservierung von Kanälen warten 381
Die Verwendung von reservierten Kanälen 382
Das "Abschließen" von Kanälen 383
Setzen einer neuen Priorität 385
Kontrolle der Audio-Ausgabe 386
Audio-Daten 387
Die Definition der Wellenform 388
Die Tonhöhe (Sampling-Period) 389
Audio-Ausgabe löschen 390
Die Plazierung der Audiodaten 390
Lautstärkeregelung 391
Das Audio-Programm 391

Kapitel 9: Multitasking **397**
Tasks 397
Prozesse 398
Der einfache Weg, etwas Neues zu beginnen 399
Ein Task-Beispiel 401
Die Linkdatei für das Task-Beispiel 402
Der Hauptprogramm inklusive eines kleinen Tasks 403
Die InitTask()-Funktion 403
Ein Prozeß-Beispiel 404
Die Linkfiles für das Prozeß-Beispiel 414
Die Prozeß-Programme 415
Kommunikation zwischen einzelnen Tasks 423
Das Aufspüren von Tasks 424
Das Aufspüren von Prozessen 424
Das Aufspüren von Ports 425

Anhang A: Der Texteditor ED **433**

Anhang B: Der Amiga-C-Compiler **441**
Starten des Amiga-C-Compilers 441
Erste Compilerphase 441
Zweite Compilerphase 442
Dritte Compilerphase 443
Zusammenfassung der Compileraufrufe 443
Erstellen und Verwendung einer Make-Datei 444
Inhalt einer Make-Datei 445
Parameterersetzung in einem Execute-File 446
Wie man MAKESIMPLE.A erstellt 449

Stichwortverzeichnis 451

Vorwort

Als wir das "Amiga ROM Kernel Manual" schrieben, unterteilten wir das Software-System des Amiga in verschiedene funktionale Bereiche. Ein Abschnitt des Buches beschäftigt sich mit den Exec-Routinen, ein anderer mit der Grafik. Weiterhin werden Sound, Animation, mathematische Funktionen, virtuelle Geräte (Devices) usw. behandelt. Diese Aufteilung nach funktionalen Gesichtspunkten stellt einen Weg dar, sich mit dem System zu befassen.

Oben genanntes Buch enthält weiterhin Abschnitte, die weiterführende Informationen zu verwendeten Funktionen beinhalten, sowie eine Erläuterung der benutzten Datenstrukturen. Das Ende eines jeden Abschnittes bildet ein funktionstüchtiges Beispielprogramm, das eingetippt und ausprobiert werden sollte.

Das vorliegende Buch führt Sie auf andere Art und Weise in die Programmierung des Amiga ein. Meine Erfahrung mit Anwendergruppen hat gezeigt, daß Beispielprogramme oftmals effektiver als ausführliche Erklärungen sind. Wenn ein verständlich kommentiertes Listing aufzeigt, mit welchen Mitteln man was erreichen kann, so bringt dies sicherlich den größten Nutzen für den Leser; aus diesem Grund werden Sie auch viele Beispielprogramme in diesem Buch vorfinden.

Die Programme sind – soweit dies möglich ist – in sich abgeschlossen und können abgetippt und ausprobiert werden. Einige wenige Programme verwenden jedoch Teile von vorausgegangenen Beispielen; in einem solchen Fall ist es erforderlich, die zuvor compilierten Programmteile mit dem Objektmodul des neuen Programms zu linken. Genauere Informationen hierzu entnehmen Sie bitte dem Anhang B, der sich ausführlich mit den C-Compilern für den Amiga

Die Kombination aus dem "Amiga ROM Kernel Manual", dem "AmigaDOS Developers Manual", dem "Intuition Manual" und diesem Buch liefert einen umfassenden Überblick über das Software-System des Amiga, das Sie ja begreifen und beherrschen wollen.

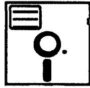
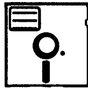
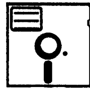
Bei der Lektüre dieses Buches werden Sie auf viele Funktionen der Programmiersprache C treffen, da fast alle Beispielprogramme in diesem Buch in C geschrieben wurden.

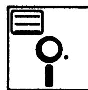
C-Compiler (aber nicht nur sie) greifen bei ihrer Ausführung auf sogenannte "Bibliotheken" zurück, in denen verwendete Funktionen enthalten und definiert sind. Diese Bibliotheken nehmen – da C eine universelle und hardwareunabhängige Sprache ist – die Anpassung an die Hardware des jeweiligen Computers vor. Da für den Amiga jedoch verschiedene C-Compiler existieren, werden in diesem Buch keine compilerspezifischen Funktionen verwendet, um eine möglicherweise notwendige Adaption der Programme einfach zu halten.

Die Entwickler der Amiga-System-Software haben viele Routinen implementiert, die die Programmierung vereinfachen sollen. So wird z.B. ein großer Teil der Hardware des Amiga durch Systemroutinen verwaltet. Sie werden feststellen, daß es nicht erforderlich ist, direkt auf Register der Prozessoren zuzugreifen, wenn Sie Funktionen des Software-Systems verwenden.

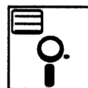
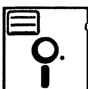
Auf diese Art und Weise ist es in vielen Fällen ungleich einfacher, ein Ziel zu erreichen, als wenn Sie die zuerst genannte Programmiertechnik anwenden. Um eine leichte Verständlichkeit der Beispielprogramme sicherzustellen, sind Inhalte von Datenstrukturen oder Kontrollregistern – soweit vertretbar – nicht in allen Einzelheiten erklärt. Was den Leser interessiert, das ist das Ergebnis bzw. der Effekt, der mit dem jeweiligen Programm erzielt werden kann.

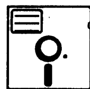
Alle Beispielprogramme in diesem Buch sind sowohl unter der System-Software Version 1.1 als auch unter der überarbeiteten Version 1.2 lauffähig.



1



Kapitel 1

Überblick

Der Amiga besitzt neben dem Prozessor "68000" noch eine Reihe anderer Chips ("Coprozessoren"), die den Hauptprozessor entlasten sollen. Die Aufgaben dieser Coprozessoren sind vielfältig: Einige sind für die Ein-/Ausgabe, etwa von der Tastatur oder bei Diskettenoperationen, zuständig, andere kümmern sich um die Verarbeitung und Darstellung von Grafik auf dem Bildschirm.

Alle diese Hardwarekomponenten können direkt über ihre Register angesprochen werden; es ist jedoch um ein Vielfaches einfacher, die Routinen und Funktionen des Betriebssystems (der System-Software) zu verwenden, um einen gewünschten Effekt zu erzielen.

Diese Routinen – der Amiga stellt Ihnen über 300 zur Verfügung – stellen einen bequemen Weg dar, auf die Hardware und spezielle Eigenschaften des Amiga zuzugreifen.

Aufbau der Soft- und Hardware-Hierarchie des Amiga

Die Abb. 1.1 zeigt ein Blockdiagramm der Amiga Systemsoftware. Diese ist entweder – beim Modell 1000 – auf Diskette abgelegt und muß bei jedem Einschalten des Rechners geladen werden, oder sie ist – bei den Modellen 500 und 2000 – in ROM-Bausteinen abgelegt und steht direkt nach dem Einschalten des Rechners zur Verfügung.

Die Abbildung zeigt Ihnen, wie die einzelnen Komponenten der Systemsoftware aufeinander aufbauen. Zwischen der gemeinsamen Basis der Systemsoftware – der Hardware – und ihrer obersten Stufe (Workbench, CLI usw.) liegen die verschiedenen Bestandteile des Systems. Jeder einzelne Baustein der Systemsoftware kann vom Programmierer direkt über die Routinen des Betriebssystems erreicht werden; er hat also eine Vielzahl von Ansatz- bzw. Einstiegspunkten. Welche Komponente er wählt, um einen bestimmten Effekt zu erzielen, hängt davon ab, welche Art Programm entwickelt werden soll.

Die unterste Stufe der Systemsoftware

Die Anpassung und die Kommunikation zwischen Hard- und Software wird von einer Anzahl Systemroutinen vorgenommen, die unter dem Oberbegriff "Exec" zusammengefaßt sind.

Exec überwacht die Verwendung bzw. Nutzung des 68000-Prozessors und teilt seine Rechenleistung und -zeit zwischen einzelnen Programmen (Tasks) auf, die auf diese Art und Weise scheinbar gleichzeitig ablaufen können. Diese Fähigkeit des Amiga, mehrere Programme zur selben Zeit abarbeiten zu können (Multitasking), stellt eine der Stärken dieses Computers dar.

Exec übernimmt weiterhin die Reservierung von Speicherplatz, der von einzelnen Programmen benötigt oder angefordert wird, verarbeitet Interrupts der Hardware, der Programme oder des Systems. Weiterhin verwaltet Exec sogenannte Listen, in denen Programme aufgeführt sind, die momentan ablaufen oder auf ein bestimmtes Ereignis warten. Es existieren zusätzlich Listen, die Auskunft über den noch freien Speicher geben; Listen, in denen noch nicht abgearbeitete Nachrichten (Messages) vermerkt sind, sowie Listen der registrierten Eingaben, etwa mit der Maus oder der Tastatur.

Neben Exec befinden sich noch andere Bausteine der Systemsoftware auf der untersten Stufe: die Devices (virtuelle Geräte). Devices sind keine Peripheriegeräte in diesem Sinne; sie sind Bestandteil der Software, nicht "greifbar" – eben "virtuell".

Diese virtuellen Geräte kümmern sich um Diskettenoperationen, die Tastatur sowie um die Schnittstellen und den Maus-/Joystickport. Weitere "Geräte"

kontrollieren die Audio-Hardware und die Grafik. Die Grafiksoftware des Systems kommuniziert direkt mit der Hardware und stellt dem Programmierer eine Vielzahl von Funktionen zur Verfügung, mit denen man die Darstellung von Grafik auf mancherlei Art und Weise kontrollieren und beeinflussen kann.

Weiterhin finden sich hier Routinen zur Farb- und Musterauswahl sowie zur Kontrolle und Manipulation von Screens (virtuellen Bildschirmen), deren Auflösung (Gesamtzahl der Bildpunkte) und grafischen Objekten (Sprites, Bobs usw.).

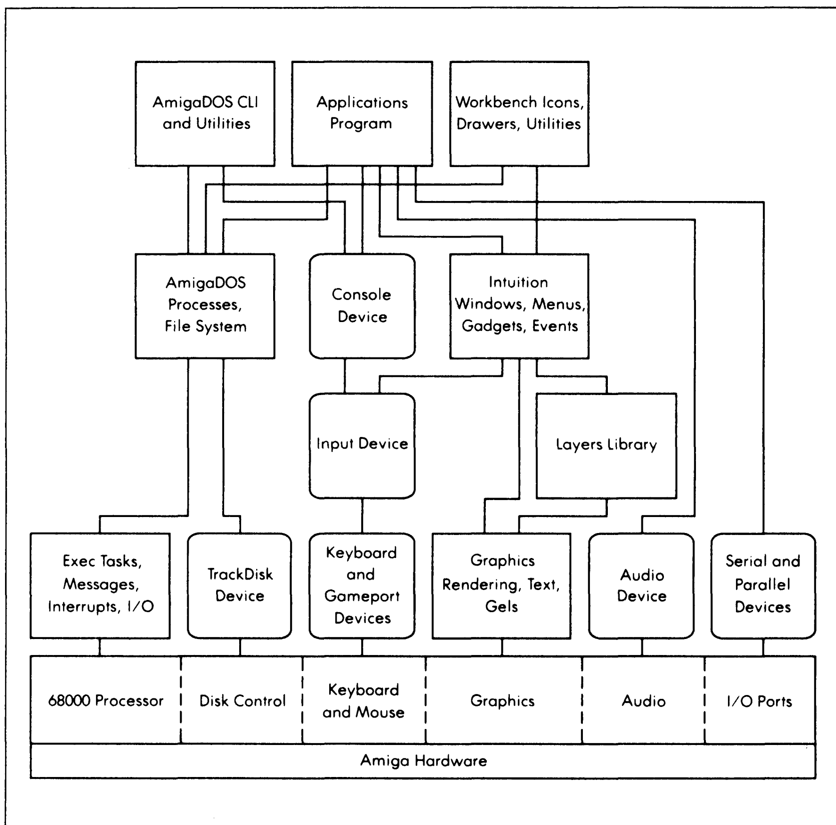


Abb. 1.1: Die Hierarchie des Amiga-Software-Systems

Die zweite Stufe der Systemsoftware

Die nächsthöhere Stufe der Systemsoftware wird von zwei Komponenten gebildet: dem "Input Device" (virtuelles Eingabegerät) und der "Layers Library", die für die Darstellung bzw. Wiederherstellung von Fensterinhalten, Requestern usw. zuständig ist.

Das "Input Device" läuft als eigenständiges Programm im System ab, mit der Aufgabe, die verschiedenen Eingabeereignisse von der Tastatur und der Maus (oder des Joysticks) zu einem einzigen Eingabestrom zu vereinigen. Dieser wird entweder – wie aus der Abbildung ersichtlich ist – an die Benutzeroberfläche "Intuition" und ihre Komponenten weitergeleitet, oder er dient dazu, mit einem "Console Device" (eine Art Terminal) zu kommunizieren.

Die "Layers Library" befindet sich oberhalb des eigentlichen Grafiksystems. Die Aufgabe dieser "Library" (Bibliothek) ist es, eine einzelne Grafikebene in mehrere unabhängige Bereiche zu unterteilen (Grundprinzip der Fenstertechnik). Sollten sich Bereiche überlappen, dann übernimmt die "Layers Library" die Wiederherstellung der zuvor von anderen Bereichen überdeckten Abschnitte.

Zusätzlich findet der Programmierer hier Funktionen zum Erstellen, Löschen und Verschieben von Grafikbereichen, als auch zum Manipulieren der Darstellungsprioritäten und des Inhalts derselben.

Die dritte Stufe der Systemsoftware

AmigaDOS, Intuition (die Benutzeroberfläche des Amiga) und "Console Device" (eine Art Terminal) bilden die dritte Stufe des Software-Systems. AmigaDOS ist für Diskettenoperationen jedweder Art zuständig. Es kommuniziert mit Exec, um die Multitaskingfähigkeit auch bei Diskettenzugriffen sicherzustellen. Dies schließt die Aufteilung der Arbeitsleistung der zuständigen Prozessoren mit ein.

AmigaDOS stellt auch das Dateisystem des Computers zur Verfügung, mit dem auf einfache Art und Weise Dateien angesprochen, ausgelesen, geschrieben, gelöscht usw. werden können.

Intuition bildet die grafische Schnittstelle zwischen Computer und Anwender. Intuition ist für die Darstellung von Screens (virtuellen Bildschirmen) zuständig, die die Basis für jede weitere Darstellungs- oder Kommunikationsform (Fenster, Requester usw.) bilden.

Es können mehrere Screens gleichzeitig dargestellt werden, wobei die Auflösung (Gesamtzahl aller Bildpunkte) der Screens unterschiedlich sein kann. Intuition stellt verschiedene Auflösungsmodi zur Verfügung: Die horizontale Auflösung kann 320 oder 640 Bildpunkte (Pixel) betragen; die vertikale liegt bei 200 oder 400 Bildpunkten (im PAL-Format 256 oder 512 Pixel).

Screens mit einer Breite von 640 Pixeln (MED-RES) können maximal 16 Farben gleichzeitig beinhalten; Screens mit einer horizontalen Auflösung von 320 Bildpunkten (LO-RES) können 32 Farben haben, wobei bei allen beiden Arten der Programmierer aus einer Palette von 4096 Farbtönen wählen kann.

Intuition ist aber nicht nur der Name der Benutzeroberfläche; es existiert auch eine Funktionsbibliothek mit gleichem Namen (auf Bibliotheken wird später in diesem Kapitel genauer eingegangen).

Die Funktionen dieser Bibliothek helfen dem Programmierer bei der Erstellung einer grafischen Benutzeroberfläche für seine Programme. Mit ihrer Hilfe werden Fenster erzeugt und dargestellt, Menüs kreiert, aus denen der Anwender Optionen auswählen kann, Requester und Gadgets verwaltet usw.

Wie bereits erwähnt, kommuniziert Intuition mit dem "Input Device", jenem virtuellen Eingabegerät, das Eingaben von der Tastatur oder durch die Maus zu einem einzigen Eingabestrom vereint. Intuition nimmt eine weitere Verarbeitung dieses Eingabestroms vor, indem die Ereignisse nach ihrer Art unterteilt werden.

Der Programmierer entscheidet, welche dieser Ereignisse für sein Programm interessant sind und welche nicht. Intuition sendet grundsätzlich jedes Ereignis an ein Programm, das eine virtuelle Kommunikationsleitung (Message Port) beinhaltet und Eingaben über diese Leitung erwartet. Dem Programmierer bleibt es überlassen, welche Ereignisse sein Programm "sehen" soll und welche nicht; Intuition nimmt von sich aus keine Filterung vor, sondern nur eine Unterteilung in Arten.

Das "Console Device" bildet in Kombination mit einem Fenster eine Art Terminal, wie sie zur Kommunikation zwischen Anwender und einem Großrech-

ner verwendet werden. Dieses virtuelle Gerät des Systems verhält sich genauso wie ein "richtiges" Terminal; es sendet und empfängt Zeichen, die für das gewählte Fenster bestimmt sind.

Damit es im System nicht zum großen Durcheinander kommt, sorgt Intuition auch dafür, daß immer nur ein Fenster zur Zeit aktiv ist. Tastatureingaben werden also immer nur an das Fenster gesendet, das gerade aktiv ist (sofern es darauf konfiguriert ist, derartige Eingaben zu verarbeiten).

Die oberste Stufe der Systemsoftware

Die letzte Stufe des Software-Systems wird von drei Bausteinen gebildet, wie Abb. 1.1 zeigt: einem Anwendungsbeispiel (Programm), der "Workbench" (dem "Arbeitstisch" des Amiga) und dem CLI (Command Line Interpreter, etwa: Befehlszeilen-Interpreter). Jeder dieser drei Bestandteile kommuniziert auf seine eigene Art und Weise mit dem Anwender.

Das CLI ist so etwas wie die "traditionelle" Schnittstelle zwischen Maschine und Mensch: Eingaben werden von der Tastatur erwartet und nach dem Betätigen der <Return>-Taste direkt verarbeitet. Das CLI ist nichts anderes als ein Programm, das Eingabezeilen des Benutzers in Befehle übersetzt, die der Computer dann ausführen kann.

Auf der Workbench, mit der der frischgebackene Amiga-Anwender sicherlich als erstes konfrontiert wird, werden Eingaben auf andere Art und Weise erwartet: Die Kommunikation mit dem Computer spielt sich größtenteils mit Hilfe der Maus ab. Mit ihr wählt der Anwender kleine grafische Symbole (Icons) aus, um ein Programm zu starten oder sich den Inhalt einer Diskette anzusehen.

Der Nachteil der Workbench – es können keine Befehle direkt an den Computer übergeben werden – wird durch den hohen Bedienungskomfort jedoch weitgehend wettgemacht.

Die dritte Komponente der obersten Stufe – ein Anwendungsbeispiel – stellt die Art der Kommunikation zwischen Mensch und Maschine dar, die der Programmierer des Programms vorgesehen hat. Das Spektrum reicht – wie bereits beschrieben – von Intuition-gestützten Programmen bis hin zu einfachen Hilfsprogrammen, die vom CLI aus gestartet und bedient werden.

Was Sie wissen müssen, um den Amiga zu programmieren

Was Sie zur Programmierung wissen müssen, hängt natürlich davon ab, welche Art von Programm Sie entwickeln wollen. Wollen Sie beispielsweise die Sound- und Grafikeigenschaften des Rechners ausnutzen, dann müssen Sie wissen, auf welche Art und Weise die zugehörigen Datenstrukturen initialisiert und verwendet werden, wie man die betreffenden Funktionsbibliotheken öffnet und anspricht und welche Funktionen für welchen Zweck gebraucht werden.

Oder wollen Sie das Multitasking des Amiga ausnutzen, z.B. um eine Datei auf dem Drucker auszugeben, während Sie eine andere mit einem Texteditor bearbeiten und zusätzlich über Modem Informationen aus einer Datenbank abrufen?

In diesem Fall brauchen Sie Informationen darüber, wie man eigene Tasks (Programme) und Prozesse startet und beendet bzw. wie man unter AmigaDOS das Multitasking ausnutzt.

Allgemein gilt jedoch – egal, welches Ziel Sie sich gesetzt haben – daß das Verständnis der Arbeitsweise des CLI, des Exec sowie der für Ihr Programmprojekt notwendigen Bausteine der Systemsoftware nicht nur hilfreich, sondern ein Muß ist. Ohne dieses Wissen ist es ungleich schwieriger, die Möglichkeiten des Amiga auszunutzen.

Funktionsbibliotheken

Bei der Beschreibung der verschiedenen Stufen der Systemsoftware wurde an mehreren Stellen der Begriff "Bibliothek" verwendet. Eine Bibliothek ist ein Zusammenschluß von Funktionen (Routinen), die auf verschiedene Weise miteinander in Beziehung stehen können.

Die Entwickler des Amiga haben diesen Weg – das Zusammenfassen von Routinen in Bibliotheken – gewählt, damit Programme auch unter verschiedenen Versionen der Systemsoftware lauffähig sind. Um diese Kompatibilität zu garantieren, mußte sichergestellt werden, daß Programme die Systemroutinen immer finden können, egal wie sehr sich das System geändert hat.

Aus diesem Grund wurde eine einheitliche Bibliotheksstruktur geschaffen, die, neben anderen Dingen, eine Sprungtabelle und die Adressen der Funktionen enthält. Der Aufbau einer solchen Struktur sieht wie folgt aus:

```
        EINTRAG n DER SPRUNGTABELLE FÜR FUNKTION n
        ... ...
        EINTRAG 3 DER SPRUNGTABELLE FÜR FUNKTION 3
        EINTRAG 2 DER SPRUNGTABELLE FÜR FUNKTION 2
        EINTRAG 1 DER SPRUNGTABELLE FÜR FUNKTION 1
LibBase <Basisadresse der Bibliothek>
        <Rest der Bibliotheksstruktur>
```

Beim Einschalten des Amiga werden die einzelnen Bibliotheken in einen geschützten RAM-Bereich, den "Kickstart-Speicher", eingelesen und zusätzlich ins allgemein zugängliche RAM kopiert, wo sie dann später, falls erforderlich, modifiziert werden können. Nach Beendigung des Einlesens bzw. Kopierens wird vom Betriebssystem nach Bibliotheken gesucht. Jede gefundene Bibliothek wird einer Liste hinzugefügt, in der alle verfügbaren Bibliotheken enthalten sind.

Wird der Amiga eingeschaltet, dann werden die Bibliotheken irgendwo im Speicher abgelegt; sie sind fast nie im gleichen Speicherabschnitt zu finden. Aus diesem Grund müssen Programme, die auf Funktionen von Bibliotheken zugreifen, spezielle Variablen definieren: die Basisadressen der Bibliothek(en) nämlich.

Um dies zu veranschaulichen, folgt nun ein kurzes Beispielprogramm, das die Art und Weise der Programmierung erläutern soll:

```
#include "intuition/intuition.h"

struct IntuitionBase *IntuitionBase;    /* Zeiger auf die Basisadresse */

main()
{
  if(!(IntuitionBase=(struct IntuitionBase *)
      OpenLibrary("intuition.library",0)))
      {
        printf("Intuition läßt sich nicht öffnen");
        exit(FALSE);
      }

  /* Rest des Programms...  */
}
```

Beim Compilieren eines Programms (siehe Anhang A) wird beim Vorgang des "Linkens" eine Datei eingebunden (je nach Compiler heißt diese Datei anders, z.B. "c32.lib" oder "amiga.lib"), die dem compilierten Programm spezielle Codefragmente hinzufügt, damit es die Bibliotheksfunktionen verwenden kann.

Die Aufgabe dieser Codefragmente beim Aufruf einer Bibliotheksfunktion ist aus dem folgenden zu ersehen:

- Die Registerinhalte des Prozessors werden zwischengespeichert, damit die Programmausführung nach der Beendigung der Funktion fortgeführt werden kann.

- Ein Register wird mit der Basisadresse der betreffenden Bibliothek geladen.

- Erwartet die Funktion die Übergabe von Parametern, dann werden diese ebenfalls in Register geladen.

- Anhand der Sprungtabelle und der Basisadresse der Bibliothek wird die Lage der Funktion im Speicher bestimmt und die Funktion aufgerufen.

- Registerinhalte werden beim Eintritt in die Funktion wiederhergestellt.

- Wenn die Funktion ein Resultat liefert, wird dieses an das Hauptprogramm übergeben.

Sollten Sie vergessen, die Basisadresse einer Bibliothek in Ihrem Programm zu definieren, dann erhalten Sie beim Linken eine Fehlermeldung, die etwa so aussehen kann:

```
Undefined symbol: _<name der basisadresse>
```

Solange der Linker noch Fehlermeldungen liefert, haben Sie Glück. Sollten Sie die Basisadresse einer Bibliothek definiert haben, jedoch eine Funktion aus ihr verwenden, bevor Sie die Bibliothek geöffnet haben, dann wird Ihnen der Amiga auf seine Art ("Guru Meditation"...) klarmachen, daß derartige Programmiertechniken bei ihm nicht sehr beliebt sind.

Haben Sie jedoch alles richtig gemacht, dann sollten Sie in der Lage sein, die gewünschte Funktion der Bibliothek anzusprechen. Der Anhang A des "Amiga

ROM Kernel Manual" enthält die Namen aller Basisadressen sowie alle Funktionen mit Angabe der Bibliotheken, in denen sie zu finden sind.

Die Bibliotheken "dos.library" und "exec.library" werden automatisch geöffnet, egal, welchen Compiler Sie verwenden. Sie brauchen in Ihrem Programm also keine Basisadressen anzugeben und die Bibliothek zu öffnen, wenn Sie eine Funktion der "exec.library" oder "dos.library" verwenden wollen. Weitere Informationen zu Bibliotheken finden Sie im Kapitel 3.

Die Programmierung in C

Die Beispielprogramme in diesem Buch sind fast alle in C geschrieben. Zum einen, weil für den Amiga mehrere gute Compiler für diese Programmiersprache erhältlich sind, zum anderen sind Datenstrukturen in C sehr leicht und anschaulich zu programmieren. Ein großer Teil des Betriebssystems des Amiga wurde in C geschrieben; sicherlich ein Beweis für die Leistungsfähigkeit dieser Sprache.

Wenn Sie in C, Pascal oder einer anderen höheren Programmiersprache Ihre Programme entwickeln, dann wird beim Compilieren automatisch zusätzlicher Programmcode eingebunden, der die Anpassung und Parameterübergabe von Ihrem Programm an die Hardware oder das Betriebssystem des Amiga übernimmt.

Die Programmierung in 68000-Assembler

Wenn Sie in Assembler programmieren, dann müssen Sie zunächst einmal wissen, wie der Amiga die Register des Prozessors 68000 handhabt. Die Register D0, D1, A0 und A1 dienen nur zur temporären Aufnahme von Werten. Daß die Registerinhalte beim Aufruf von Systemroutinen erhalten bleiben, darauf sollte man sich jedoch nicht verlassen; Registerinhalte werden vom System nur bei allen anderen Registern zwischengespeichert.

Funktionen, die nach ihrer Ausführung ein Resultat liefern, geben dieses über das Register D0 an das Hauptprogramm zurück. Liefert eine Funktion mehr als einen Wert, dann sollten Sie zuvor ein Feld oder eine Datenstruktur bereitstellen, wo die Funktion ihre Resultate ablegen kann.

Ein Register des Prozessors hat beim Amiga eine Sonderfunktion: A6. Über dieses Register werden keine Parameter oder Resultate übergeben; es beinhaltet vielmehr die Adresse einer Sprungtabelle, in der die aktuellen Adressen von Systemfunktionen aufgeführt sind. Aus diesem Grund wird dieses Register auch "SysBase" genannt.

Wird eine Funktion aufgerufen, dann sucht sich Exec die Startadresse der Funktion aus der Sprungtabelle. Der Programmierer kann die Sprungtabelle ändern, damit Systemroutinen anders als gewohnt ablaufen, oder er fügt an einer Stelle ein Programm ein, das zusammen mit den normalen Systemroutinen abgearbeitet wird. Auf diese Art und Weise kann eine eventuell notwendige Fehlersuche in einem Programm erheblich erleichtert werden.

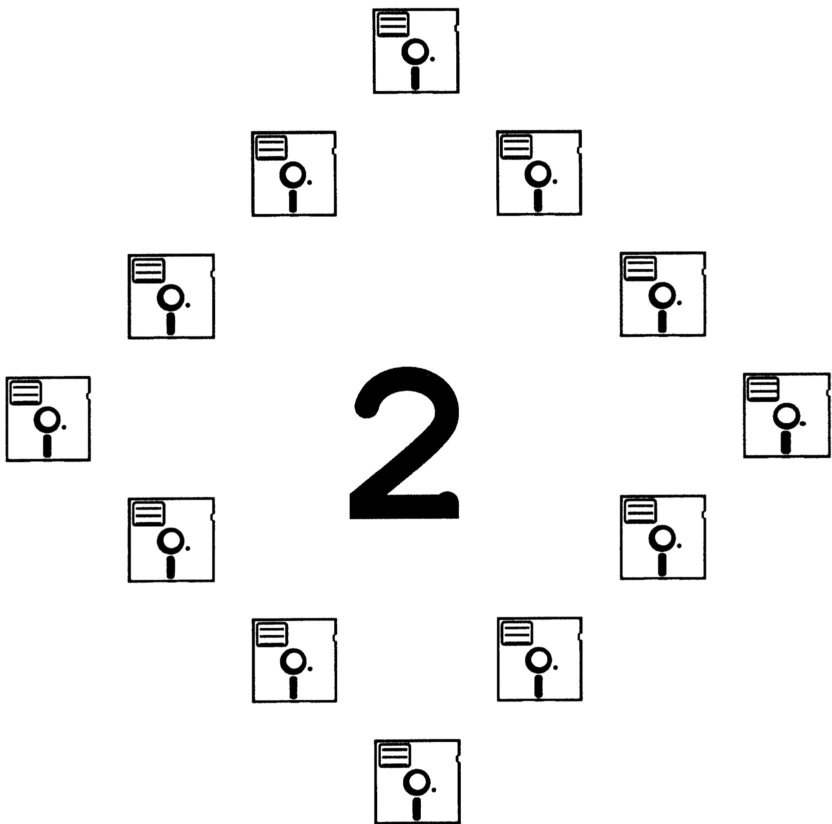
Die "Include"-Dateien des Compilers

Auf der Compiler- oder Assemblerdiskette befinden sich sogenannte "Include-Files". Beim Assembler haben diese Dateien die Endung ".i", bei C-Compilern finden Sie den Zusatz ".h".

In diesen Dateien sind Systemkonstanten und Datenstrukturen definiert, zu denen Sie in diesem Buch vielfach Hinweise und Erläuterungen finden werden. Die Dateien als solche sind nicht in diesem Buch vorhanden. Kommentierte Listings der Dateien finden Sie im "Amiga ROM Kernel Manual". Die Funktion dieses Buches soll sein, Informationen zu vermitteln, ohne tiefgreifende Erklärungen zu liefern.

Viele der Systemroutinen sind auf den folgenden Seiten erklärt. Dieses Buch sagt Ihnen, was Sie tun müssen, bevor Sie eine bestimmte Routine verwenden können bzw. welche Routine Sie benötigen, um einen bestimmten Effekt zu erzielen. Der Besitz des "Amiga ROM Kernel Manual" kann nützlich sein, falls Sie die Beispielprogramme dieses Buches modifizieren wollen.

Viel Erfolg!


2

Kapitel 2

AmigaDOS

Die grafische Benutzeroberfläche des Amiga, mit der Sie zum Beispiel beim Arbeiten mit der Workbench zu tun haben, stellt nur einen Weg dar, den Computer zu bedienen.

Die andere Möglichkeit ist die Nutzung des CLI, das der "traditionellen" Schnittstelle zwischen Ihnen und einem Computer entspricht: Sie geben Befehlszeilen ein, und der Rechner führt diese dann aus. Dieses Kapitel beschäftigt sich mit den Funktionen des CLI, geht aber auch auf die Programmierung und Nutzung von AmigaDOS ein. Hier der Inhalt des Kapitels im Überblick:

- Ausgabe von Text in ein CLI-Fenster
- Übergabe von Argumenten beim Programmaufruf
- Eingaben vom CLI
- Öffnen und Schließen von Dateien
- Lese- und Schreibzugriff auf Dateien
- andere Dateifunktionen
- Öffnen von Fenstern des Typs "RAW" oder "CON" für Ein-/Ausgabeoperationen
- Nutzung der AmigaDOS-Befehle und AmigaDOS-Hilfsfunktionen

Alle Beispielprogramme in diesem Kapitel müssen vom CLI aus gestartet werden, weil jedes Informationen und Statusanzeigen in dem CLI-Fenster ausgibt, von dem aus es aufgerufen wurde. Es gibt immer zwei Möglichkeiten, ein Programm auf dem Amiga zu starten: Man kann es zum einen vom CLI aus aufrufen, zum anderen kann man das Icon des Programms – sofern eins vorhanden ist – von der Workbench aus mit der Maus aktivieren.

Um auf eine Routine der Systemsoftware zugreifen zu können, muß man zuvor die Bibliothek öffnen, in der sich die betreffende Routine befindet (siehe Kapitel 1). Wie dies im einzelnen gemacht wird, sehen Sie in den folgenden Kapiteln. In diesem Kapitel ist ein Aufruf der Funktion OpenLibrary, die das Öffnen einer Bibliothek vornimmt, nicht erforderlich, da die "dos.library", in der sich die Routinen des AmigaDOS befinden, vom Compiler geöffnet wird; dies gilt auch für die Exec-Bibliothek.

Die Routinen der anderen Bibliotheken, die später in diesem Buch behandelt werden, können erst dann verwendet werden, wenn die betreffende Bibliothek zuvor geöffnet wurde. Wie dies geschieht, erfahren Sie in den folgenden Kapiteln. Sollten Sie später einmal Änderungen am Compiler bzw. Linker vornehmen und Ihre eigenen Bibliotheken verwenden, dann sollten Sie nicht vergessen, die DOS-Bibliothek öffnen zu lassen, falls Sie beabsichtigen, AmigaDOS-Funktionen zu verwenden.

Ausgabe von Text in ein CLI-Fenster

Es hat sich eingebürgert, daß das erste Programm, das Sie in einem Buch zu einer Programmiersprache finden, den Text "Hallo!" auf dem Bildschirm ausgibt, um Sie grundsätzlich mit der Sprachstruktur bekannt zu machen. Auf dem Amiga ist dies, wie Sie gleich sehen werden, leicht zu realisieren:

```
/* Hallo.c */

main()
{
  printf("Hallo!\n");
}
```

Informationen zum Compilieren von Programmen finden Sie im Anhang A. Wenn Sie dieses Programm compilieren und starten, wird der Text "Hallo!"

im aktuellen CLI-Fenster ausgegeben und das Programm beendet. Haben Sie das Beispiel unter dem Namen "hallo" compilieren lassen, dann bringen Sie es vom CLI aus zur Ausführung, indem Sie den Programmnamen "hallo" eintippen und die <Return>-Taste drücken.

Übergabe von Argumenten beim Programmaufruf

Einer der Vorteile von C ist, daß bereits beim Aufruf eines Programms Argumente so übergeben werden können, wie dies auch unter UNIX oder MS-DOS möglich ist. Die notwendigen Parameter, die dem Programm übergeben werden sollen, werden einfach, durch <Space> getrennt, hinter dem Programmnamen angegeben.

Haben Sie beispielsweise ein Programm mit dem Namen "argecho" geschrieben, dann sieht ein Aufruf des Programms z.B. so aus:

```
argecho argument1 argument2 argument3
```

Bei der Programmausführung werden dann zwei Variablen initialisiert: "argc" (vom Typ INT) beinhaltet die Anzahl der Parameter (in unserem Beispiel den Wert 3), "argv[]" (vom Typ CHAR) beinhaltet die einzelnen Argumente in Form von Zeichenketten. Listing 2.1 zeigt ein Beispielprogramm, das die ihm übergebenen Argumente ausgibt.

```
/* argecho.c */

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

     for (i=0; i<argc; i++)
          printf("Argument %d = %s\n",i,argv[i]);
}
```

Listing 2.1: Beispielprogramm zur Übergabe von Argumenten

Die Umleitung der Standard-Ein-/Ausgabe

Bei der Eingabe von Befehlszeilen können Sie – wie dies auch unter UNIX möglich ist – Symbole verwenden, um dem CLI mitzuteilen, daß die Ein- oder Ausgabe nicht mehr durch das aktuelle CLI-Fenster geschehen soll.

Diese Umleitung geschieht durch die Symbole "<" und ">", die direkt auf den eigentlichen Befehl folgen müssen. Nachfolgend finden Sie einige Beispiele, die die Umleitung der Standard-Ein-/Ausgabe veranschaulichen sollen. "Standard" bedeutet in diesem Zusammenhang nur, daß keine eigenen E/A-Routinen verwendet werden.

Wir verwenden das bereits besprochene Beispielprogramm "argecho" und ein fiktives Programm mit Namen "magicopy". Das erste Beispiel liest ein Zeichen aus einer Datei und schreibt es in eine andere Datei:

```
magicopy quelldatei zieldatei
```

Von der Quelldatei wird also nur eine Kopie mit Namen "zieldatei" erstellt, weiter geschieht nichts. Das nächste Beispiel schreibt die Argumente, die dem Programm "argecho" übergeben wurden, nicht in das aktuelle CLI-Fenster, sondern in eine Datei mit Namen "echodatei".

```
argecho > echodatei argument1 argument2 argument3
```

Das Symbol zur Umleitung der Ausgabe und der Name der Zieldatei werden von AmigaDOS aus der Eingabezeile gefiltert; "argecho" erhält also nur die drei folgenden Argumente. In der Zieldatei befinden sich nach der Ausführung von "argecho" die drei übergebenen Argumente.

Bei der Verwendung von Symbolen zur Umleitung der Ein-/Ausgabe benutzt AmigaDOS intern ein sogenanntes "file handle" (einen Zeiger, der beim Öffnen einer Datei erzeugt wird), um die Daten bzw. den "Datenstrom" der Ein-/Ausgabe verarbeiten zu können. Wenn Ihr Befehl oder Ihr Programm beendet wurde, dann verwendet AmigaDOS diesen Dateizeiger, um die Datei automatisch zu schließen.

Sie als Programmierer müssen sich nicht darum kümmern, auf welche Art die Ein- oder Ausgabe umgelenkt wird; Sie geben lediglich an, woher bzw. wohin

die Daten kommen, den Rest erledigt AmigaDOS. Die Termini stdin (Standardeingabe) und stdout (Standardausgabe) sind für den Compiler bereits vordefiniert. Wenn Sie diese Definitionen nicht ändern, dann beziehen sich die Ein- und Ausgaben immer auf das aktuelle CLI-Fenster.

Im nächsten Abschnitt erfahren Sie mehr über die "file handles" und wie man sie benutzt, um mit einer Datei (o.ä.) arbeiten zu können.

Das Arbeiten mit Files (Dateien)

Ein "file handle" ist ein Zeiger auf eine Datenstruktur, in der Informationen über eine Datei zu finden sind. AmigaDOS liefert diesen Zeiger, wenn Sie eine Datei zum Lesen oder Schreiben öffnen. Der Zeiger wird für verschiedene Dinge verwendet: zur Umleitung der Ein-/Ausgabe und beim Lesen, Schreiben und Schließen der Datei.

Die C-Funktion printf wird dazu verwendet, Zeichen zum aktuellen Ausgabegerät (meistens der Bildschirm) zu senden. Es gibt jedoch noch eine zweite Möglichkeit, Ausgaben im aktuellen CLI-Fenster zu machen: durch das Öffnen des Fensters mit Hilfe eines "file handle".

Bitte beachten Sie, daß die Zeiger, die die "normalen" C-Funktionen liefern (z.B. der Zeiger der Funktion fopen), anders als die der AmigaDOS-Funktionen (z.B. Open) sind. Sie können also nicht den Zeiger einer Standard-C-Funktion in einer AmigaDOS-Funktion verwenden und umgekehrt. Sie müssen sich entscheiden, ob Sie die Dateifunktionen des AmigaDOS oder die der Standard-C-Bibliothek verwenden. Die Funktionen der Standardbibliothek machen es leichter, Ihre Programme an andere Rechner anzupassen; die AmigaDOS-Funktionen hingegen können Ihnen helfen, Programme zu schreiben, die auf dem Amiga effizienter laufen.

In den vielen Büchern zur Programmiersprache C werden Sie Erläuterungen zu den Standard-Ein-/Ausgabefunktionen, z.B. putchar und getchar, finden. In diesem Buch beschäftigen wir uns mit den AmigaDOS-Funktionen, z.B. Read und Write.

Öffnen einer Datei

Um mit einer Datei arbeiten zu können, benötigt man ein "file handle", also einen Zeiger auf die Datenstruktur, die Informationen über die betreffende Datei enthält. Diesen Zeiger erhält man, wenn man die AmigaDOS-Funktion Open verwendet. Ein Aufruf der Funktion sieht so aus:

```
struct FileHandle   *MeineDatei;
char *dateiname;
int zugriffsart;

MeineDatei = Open(dateiname,zugriffsart);
```

Wie Sie sehen, benötigt die Open-Funktion zwei Parameter. Der erste Parameter (dateiname) ist ein Zeiger auf den String, der den Namen der zu öffnenden Datei enthält; man kann jedoch auch den Dateinamen – in Anführungszeichen – direkt angeben. Der zweite Parameter ("zugriffsart") kann die Werte "MODE_OLDFILE" oder "MODE_NEWFILE" annehmen. Die Werte dieser symbolischen Konstanten sind in der Datei "dos.h", die sich auf Ihrer Compilerdiskette befindet, definiert. "MODE_OLDFILE" öffnet eine bereits vorhandene Datei zum Lesen oder Schreiben und positioniert den Schreib-/Lesezeiger auf den Anfang der Datei.

"MODE_NEWFILE" kreiert zuerst eine Datei mit dem Namen "dateiname" und öffnet diese dann ebenfalls zum Lesen oder Schreiben, wobei auch hier der Schreib-/Lesezeiger auf den Anfang der Datei weist. Wenn Sie den Parameter "MODE_NEWFILE" auf eine bereits existierende Datei anwenden, dann wird diese – sofern sie nicht dagegen geschützt wurde – gelöscht und eine leere Datei mit gleichem Namen erzeugt.

Kann eine Datei aus irgendwelchen Gründen nicht geöffnet werden, dann liefert die Open-Funktion einen "leeren" ("NULL"-) Zeiger. Auf diese Art und Weise kann man feststellen, ob der Open-Befehl erfolgreich ausgeführt wurde; dies ist immer dann der Fall, wenn der gelieferte Zeiger ungleich "NULL" ist.

Schließen einer Datei

Sind alle Arbeiten mit einer Datei ausgeführt worden, dann kann sie mit der Funktion Close geschlossen werden. Sie sollten immer sicherstellen, daß alle Dateien, die Sie in einem Programm verwenden, vor dem Beenden des Pro-

gramms geschlossen werden. Beim Amiga können maximal 17 Dateien zur gleichen Zeit geöffnet sein. Sollten Sie versuchen, noch eine Datei zu öffnen, so können recht merkwürdige Dinge geschehen, zumal AmigaDOS keine Fehlermeldungen ausgibt, wenn die maximale Anzahl geöffneter Dateien überschritten werden sollte.

Hier finden Sie den Aufruf der Close-Funktion:

```
struct FileHandle   *MeineDatei;

Close(MeineDatei);
```

Close benötigt einen Parameter – den Zeiger, den die Open-Funktion liefert. Close beendet den Zugriff auf eine Datei und gibt zusätzlich noch alle Zeichen aus, die eventuell intern zwischengespeichert wurden.

Lesen aus einer Datei

Um Zeichen aus einer Datei lesen zu können, wird die Funktion Read verwendet. Aufgerufen wird diese Funktion wie folgt:

```
struct FileHandle   *MeineDatei;
char *Puffer;
int zeichen_zaehler,zeichen_total;

zeichen_total = Read(MeineDatei,Puffer,zeichen_zaehler);
```

"MeineDatei" ist wieder der von der Open-Funktion gelieferte Zeiger, der der Read-Funktion sagt, aus welcher Datei die Zeichen gelesen werden sollen. Zusätzlich geben Sie noch an, wie viele Zeichen Sie auslesen wollen ("zeichen_zaehler") und wo diese abgelegt werden sollen (Puffer).

Als Returnwert liefert Read die Anzahl der Zeichen, die gelesen wurden. Geht alles gut, dann stimmt "zeichen_total" mit dem Wert von "zeichen_ zaehler" überein. Hat "zeichen_total" den Wert Null, dann wurden keine Zeichen gelesen. Bei einem Wert von –1 ist ein Fehler aufgetreten. Welche Art Fehler es war, das können Sie mit der Funktion IoErr feststellen, die später in diesem Kapitel behandelt wird.

Die Read-Funktion wird ebenfalls von der Bibliotheksfunktion getchar des Amiga verwendet.

Schreiben in eine Datei

Die Funktion Write dient dazu, Zeichen in eine Datei zu schreiben. Nachfolgend sehen Sie den Aufruf der Funktion:

```
struct FileHandle   *MeineDatei;
char *Puffer;
int zeichen_zaehler,zeichen_total;

zeichen_total = Write(MeineDatei,Puffer,zeichen_zaehler);
```

Die Bedeutung der Parameter ist äquivalent zu der der Read-Funktion. "MeineDatei" sagt der Funktion, in welche Datei geschrieben werden soll, "Puffer" und "zeichen_zaehler" enthalten die zu schreibenden Zeichen bzw. die Gesamtanzahl zu schreibender Zeichen.

Write liefert als Resultat die Menge der tatsächlich geschriebenen Zeichen. Im Normalfall stimmen "zeichen_total" und "zeichen_zaehler" überein; im Fehlerfall hat "zeichen_total" den Wert –1.

Eine genauere Bestimmung der Fehlerart kann mit Hilfe der Funktion IoErr vorgenommen werden. Die Funktion putchar der Standardbibliothek verwendet ebenfalls die Write-Funktion.

Fenster-Ein-/Ausgabe mit Hilfe der AmigaDOS-Dateifunktionen

Das folgende Beispielprogramm benutzt das aktuelle CLI-Fenster, um den Text "Hallo Magic!" auszugeben. Allerdings geschieht die Ausgabe auf eine andere Art und Weise als im allerersten Beispielprogramm ("hallo.c"), wo wir die Funktion printf verwendet haben.

Der Dateiname "*", der der Open-Funktion übergeben wird, bedeutet soviel wie "alle weiteren Ein-/Ausgabeoperationen beziehen sich auf das aktuelle CLI-Fenster". Hier finden Sie das Programm:

```
#include "libraries/dosextens.h"                /* In dieser Datei ist
                                                   "FileHandle" definiert */

main()
{
  struct FileHandle   *dos_fh;

  dos_fh=Open("*",MODE_OLDFILE);                /* "Console" öffnen */
  fprintf(dos_fh,"%Ls","Hallo Magic!\n");       /* Funktion der
                                                   Standardbibliothek */
  Close(dos_fh);
}
```

Das Öffnen eines neuen CLI-Fensters

Anstatt den Text in das bestehende CLI-Fenster ausgeben zu lassen, kann man natürlich auch ein neues Fenster öffnen, in dem dann die Ausgabe erfolgt. Zu diesem Zweck braucht man nur den Parameter "dateiname" für die Open-Funktion zu modifizieren. Das folgende Programm ist eine etwas abgeänderte Version des zuvor beschriebenen:

```
#include "libraries/dosextens.h"
extern sruct FileHandle *Open();
main()
{
  struct FileHandle *dos_fh;

  dos_fh = Open("CON:10/10/500/150/Neues Fenster",MODE_NEWFILE);
  Write(dos_fh,"Hallo Magic!\n",13);
  Delay(6*50);                                  /* 6 Sekunden Pause */
  Close(dos_fh);
}
```

Das neue Fenster ist vom Typ "CON". Das bedeutet, daß sich der Amiga bei der Ein-/Ausgabe mit diesem Fenster wie ein Terminal verhält (mehr darüber im nächsten Abschnitt). Der Ursprung des neuen Fensters (seine linke obere Ecke) liegt in den Koordinaten 10/10; die Breite des Fensters beträgt 500 Bildpunkte (Pixel), seine Höhe 150 Pixel. "Magic Window" ist der Name des Fensters.

Die AmigaDOS-Funktion Delay ist nichts weiter als eine Warteschleife. Der notwendige Parameter gibt an, wie lange das Programm angehalten werden soll (in fünfzigstel Sekunden). "MODE_NEWFILE" wird verwendet, da es sich um ein neues Fenster handelt, das erst geöffnet werden muß.

Eingaben von einem CLI-Fenster

Das CLI-Fenster, das Sie für die Textausgabe geöffnet haben, kann natürlich auch dazu verwendet werden, um Eingaben zu machen. Das neue Fenster ist nach dem Öffnen automatisch aktiv (natürlich nur so lange, bis Sie mit der Maus ein anderes aktivieren).

Das bedeutet, daß die Eingaben, die Sie mit der Tastatur machen, direkt an das aktive Fenster geschickt werden. Das nachfolgende Listing 2.2 zeigt eine abgeänderte Version des vorhergehenden Beispielprogramms; es erwartet die Eingabe einer Zeile und gibt diese nach Betätigung der <Return>-Taste wieder auf dem Bildschirm aus.

Beim Ausprobieren des Programms werden Sie feststellen, das nur darstellbare Zeichen in diesem Fenster ausgegeben werden können. Weiterhin wird Ihre Eingabezeile erst ausgegeben, nachdem Sie <Return> gedrückt haben. Kurz gesagt: AmigaDOS nimmt bereits während der Eingabe eine Filterung der eingehenden Zeichen vor.

Aufgrund dieses Auswahlverfahrens durch AmigaDOS können Sie auch die vorhandenen einfachen Editierfunktionen nutzen, z.B. <Backspace>, um das zuletzt eingegebene Zeichen zu löschen. Die komplette Zeile kann mit der Tastenkombination <CTRL><X> gelöscht werden. Wenn Sie die Eingabe durch Drücken der <Return>-Taste beenden, dann wird – wie bei anderen Rechnern auch – der Befehl direkt ausgeführt.

```
#include "libraries/dosextens.h"

main()
{
   struct FileHandle   *dos_fh;
   int zaehler;
   char benutzer_eingabe[256];

   dos_fh = Open("CON:10/10/500/150/Eingabefenster",MODE_NEWFILE);
   Write(dos_fh,"Bitte geben Sie eine Zeile ein (RETURN = Ende) :\n",50);
   zaehler = Read(dos_fh,benutzer_eingabe,255);
   benutzer_eingabe[zaehler] = '\0';
   printf("Sie haben %d Zeichen eingegeben,
           nämlich die folgenden:\n",zaehler);
   printf("%s\n",benutzer_eingabe);
   Close(dos_fh);
}
```

Listing 2.2: Eingabe einer Zeile im aktuellen CLI-Fenster

Übergabe von Eingabezeilen ohne Betätigung der <Return>-Taste

Im vorigen Beispiel konnten Sie eine Zeile im aktuellen CLI-Fenster (vom Typ "CON") eingeben, wobei schon während der Eingabe eine Filterung der Zeichen vorgenommen wurde. Die Ausgabe der Zeile erfolgte erst nach dem Drücken der <Return>-Taste; die Zeichen wurden von Ihnen "blind" eingegeben.

Öffnen Sie ein Fenster nicht als "CON", sondern als "RAW", dann können Sie die Zeichen, die Sie eingeben, direkt auf dem Bildschirm sehen. Listing 2.3 enthält ein weiteres Beispielprogramm, in dem jede Tastatureingabe direkt an das CLI übergeben wird.

Zusätzlich werden noch die Codes der einzelnen Tasten in hexadezimaler Notation ausgegeben. Drücken Sie eine Taste, die kein darstellbares Zeichen hervorruft (z.B. die Funktions- oder die Cursortasten), dann wird mehr als ein Wert ausgegeben.

Bei Fenstern vom Typ "RAW" werden grundsätzlich alle Eingaben akzeptiert; eine Filterung durch AmigaDOS findet nicht statt. Beendet wird das Programm durch Drücken der Taste "Q".

Auf dem Amiga erhalten Sie immer zwei Werte beim Betätigen einer Taste: einen beim Niederdrücken, den anderen beim Loslassen der Taste. Ihr Programm kann also unterschiedlich reagieren, je nachdem, welchen Wert Sie abfragen. Unter AmigaDOS können Sie allerdings von dieser Möglichkeit keinen Gebrauch machen.

Dies ist nur möglich, wenn Sie mit einer Fähigkeit von Intuition arbeiten, die sich "IDCMP" (Intuition Direct Communication Message Port; etwa: direkte Kommunikationsleitung zwischen Ihrem Programm und Intuition) nennt.

Über IDCMP können Sie noch andere Dinge abfragen, z.B. ob die Maus bewegt wurde, ob Maustasten gedrückt wurden, ob eine Diskette aus einem Laufwerk entfernt wurde usw. Genauere Informationen zum IDCMP finden Sie in Kapitel 5.

```
#include "exec/types.h"
#include "libraries/dosextens.h"
#define QUIT 0x51  /* Das große (geshiftete) 'Q' */

main()
{
  struct FileHandle *dos_fh;
  char benutzereingabe[256];
  int zaehler,j;

  dos_fh = Open("RAW:10/10/500/150/Eingabefenster",MODE_NEWFILE);

  for(;;)
     {
       zaehler = Read(dos_fh,benutzereingabe,255);
       benutzereingabe[zaehler] = '\0';

       printf("Sie haben %d Zeichen eingegeben.\n",zaehler);
       /* Wenn der Anwender schnell tippt, dann werden mehrere
          Eingaben innerhalb eines einzelnen Intervalls
          übermittelt. */

       printf("Folgende Zeichen wurden eingegeben: ");

       for(j=0;j<zaehler;printf("%lx ",benutzereingabe[j++]));
       printf("\n");
       if(benutzereingabe[0] == QUIT) break;
     }

  Close(dos_fh);
}
```

Listing 2.3: Eingabe in einem Fenster vom Typ "RAW"

Die Ausgabe von Dateien auf einen Drucker

Sie können unter AmigaDOS Dateien oder Eingabezeilen auf einen Drucker ausgeben lassen, der entweder am seriellen oder parallelen Port (Anschlußbuchse) angeschlossen ist. AmigaDOS stellt Ihnen dafür drei Möglichkeiten zur Verfügung:

SER: Hiermit sprechen Sie den seriellen Anschluß des Amiga an.

PAR: Der parallele Ausgang wird hiermit angesprochen.

PRT: Mit dieser Möglichkeit wird der Drucker so angesprochen, wie Sie es mit dem Programm "Preferences" eingestellt haben.

Die oben aufgeführten Bezeichnungen sind feststehende Gerätenamen. Um ein Gerät für die Ausgabe zu öffnen, können diese Bezeichnungen als Dateiname in der Funktion Open verwendet werden; AmigaDOS liefert dann wie gewöhnlich einen Zeiger (file handle), der genauso verwendet wird, wie das bereits in den Abschnitten zur Arbeit mit Dateien erläutert wurde.

Listing 2.4 ist ein Beispielprogramm, mit dem eine Textdatei auf den Drucker ausgegeben wird. Das CLI-Äquivalent zu diesem Programm sieht wie folgt aus:

```
JOIN dateil datei2 datein AS name
TYPE name PRT:
```

Im Beispielprogramm wird bei der Open-Funktion der Gerätename "PRT:" verwendet. Das bedeutet, der Drucker wird so angesprochen, wie dies im Programm "Preferences" eingestellt wurde. "SER:" oder "PAR:" wären als Gerätenamen auch erlaubt, wobei auch bei "SER:" die eingestellten Preferences-Werte verwendet werden (Baudrate, Datenbits, Parität usw.). Beim Gerätenamen "PAR:" verwendet der Amiga einen Hardware-Handshake (Übertragungsprotokoll), der die Kommunikation zwischen Computer und Drucker steuert.

Da mehrere mögliche Gerätenamen existieren, können Sie den Drucker gemäß seinen Fähigkeiten ansprechen. "PRT:" übersetzt Kontrollsequenzen für den Drucker so, wie dies mit Preferences eingestellt wurde. Bei den beiden anderen Gerätenamen erfolgt keine Umwandlung; Sie können auf diese Art und Weise auch Drucker steuern, die im Programm Preferences nicht implementiert sind.

Kapitel 6 enthält genauere Informationen darüber, wie man den Drucker steuert und wie man ihn direkt – ohne AmigaDOS-Funktionen – anspricht.

```
/* printem.c */

#include "libraries/dosextens.h"
extern struct FileHandle *Open();
main(argc,argv)
int argc;
char *argv[];
{
  struct FileHandle   *fh,*fh2;
  int datasize,n=1;
  char puffer[256];

  if(argc<2)
     {
       printf("Programmaufruf: printem <dateiname> [dateiname2...]\n");
       exit(0);
     }

  fh = Open("PRT:",MODE_OLDFILE);
  if(!fh) exit(20);    /* Drucker wurde nicht geöffnet */

  while(argc>1)
       {
         fh2 = Open(argv[n],MODE_OLDFILE);


         if(!fh2) printf("Datei nicht gefunden: %s\n",argv[n]);
         else
           for(;;)
             {
               datasize = Read(fh2,puffer,256);
               Write(fh,puffer,datasize);
               if(datasize<256) break;  /* Zuwenig Zeichen eingelesen */
               /* An dieser Stelle sollte eine Prüfung auf EOF erfolgen */
             }


         Close(fh2);
         n++; argc--;
       }


  Close(fh);
}
```

Listing 2.4: Das "printem"-Programm

Weitere Funktionen zum Arbeiten mit Dateien

Nachfolgend finden Sie noch einige andere Funktionen, die beim Arbeiten mit Dateien eingesetzt werden können:

Seek	Positioniert oder liest den Schreib-/Lesezeiger einer Datei
IsInteractive	Prüft, ob Ihr Programm mit dem CLI kommuniziert
WaitForChar	Wartet für eine bestimmte Zeit auf eine Eingabe

Finden der Position des Schreib-/Lesezeigers einer Datei

Wenn Sie eine Datei öffnen, dann ist der Zeiger (file handle), den die Open-Funktion liefert, immer auf den Anfang (das erste Byte) der Datei gerichtet. Die Funktion Seek bietet die Möglichkeit, die aktuelle Position dieses Zeigers innerhalb der Datei zu finden bzw. den Zeiger neu zu positionieren. Die neue Position können sie auf drei verschiedene Arten angeben: relativ zum Anfang der Datei, relativ zum Dateiende oder relativ zur aktuellen Position des Zeigers. Die Dateigröße wird in Bytes angegeben – ein Byte bedeutet einen Schritt vorwärts oder rückwärts in der Datei.

Ein Aufruf der Funktion erfolgt auf diese Weise:

```
struct FileHandle   *MeineDatei;
int position,aktuelle_position,relativer_startpunkt;

aktuelle_position = Seek(MeineDatei,position,relativer_startpunkt);
```

Um Daten an das Ende einer Datei anzuhängen, ruft man die Funktion so auf:

```
aktuelle_position = Seek(MeineDatei,0,OFFSET_END);
```

Um die aktuelle Position des Schreib-/Lesezeigers zu erfahren, ohne ihn in der Datei zu bewegen, ruft man Seek wie folgt auf:

```
aktuelle_position = Seek(MeineDatei,0,OFFSET_CURRENT);
```

Um den Zeiger 10 Bytes weiter vorwärts zu bewegen (relativ zur aktuellen Position), verwendet man den folgenden Aufruf:

```
aktuelle_position = Seek(MeineDatei,10,OFFSET_CURRENT);
```

Ruft man Seek wie nachfolgend beschrieben auf, dann positioniert man den Zeiger wieder auf den Anfang der Datei:

```
aktuelle_position = Seek(MeineDatei,0,OFFSET_BEGINNING);
```

Prüfen, ob Ihr Programm mit dem CLI kommuniziert

Mit der Funktion IsInteractive können Sie feststellen, ob die Standard-Eingabe Ihres Programms aus dem CLI erfolgt. Liefert die Funktion einen Wert ungleich Null, dann wurde Ihr Programm vom CLI aus gestartet; ist der Wert gleich Null, dann wurde es von der Workbench oder mit der Funktion Execute aufgerufen. Wird ein Programm vom CLI aus gestartet, dann steht immer ein Fenster zur Kommunikation mit dem Anwender zur Verfügung – das Programm ist "interaktiv", d.h. es kann Daten mit dem Benutzer austauschen.

Wird es auf eine andere Weise aufgerufen, dann ist unter Umständen kein Fenster zum Datenaustausch vorhanden; die Eingabe ist in diesem Fall nicht interaktiv. Sie können auf diese Art prüfen, ob Eingaben auch wirklich möglich sind – ansonsten wartet Ihr Programm vielleicht auf eine Eingabe, die niemals erfolgen kann, da kein Fenster zum Datenaustausch vorhanden ist. Ein Aufruf der Funktion sieht so aus:

```
int status;

status = IsInteractive();
```

Eine bestimmte Zeit auf die Eingabe eines Zeichens warten

Wenn Ihr Programm über den CLI kommunizieren kann (wenn er interaktiv ist), dann können Sie die Funktion WaitForChar einsetzen, um für einen bestimmten Zeitraum auf die Eingabe eines Zeichens zu warten. Sollten Sie die Funktion verwenden, wenn kein Fenster oder Eingabepfad vorhanden ist,

dann wartet Ihr Programm bis zum nächsten Warmstart des Rechners; es "hängt sich auf".

Die Funktion WaitForChar kann nützlich sein, um den Anwender wiederholt aufzufordern, eine Eingabe zu machen, falls er innerhalb des definierten Zeitraums nichts eingeben sollte. Oder Sie können Ihr Programm beenden, falls keine Eingabe erfolgt ist.

Die Funktion benötigt beim Aufruf zwei Parameter: den Zeiger, den die Open-Funktion liefert, und den Wert, der angibt, wie lange auf die Eingabe gewartet werden soll (dieser Wert wird in fünfzigstel Sekunden angegeben). Der Returnwert, den die Funktion liefert, ist gleich Null, falls innerhalb des angegebenen Zeitraums keine Eingabe erfolgt ist, oder er ist ungleich Null, d.h. es wurde eine Eingabe vom Anwender gemacht.

Die Funktion WaitForChar gibt einen kleinen Einblick in das Multitasking auf dem Amiga. Während Ihr Programm für den angegebenen Zeitraum "schläft", d.h. auf eine Eingabe wartet, können andere Programme ungestört weiterlaufen.

Die Struktur des Inhaltsverzeichnisses (Directory) einer Diskette

Das von AmigaDOS verwaltete Dateisystem hat einen hierarchischen Aufbau, den man sich am besten im Vergleich mit einem Schreibtisch klarmachen kann. Die "Schubladen" der Diskette sind Unterverzeichnisse des Haupt- ("Root"-) Verzeichnisses, die in den "Schubladen" enthaltenen "Papiere" sind die Dateien.

Es existiert also nicht nur ein großes Verzeichnis, sondern es können beliebig viele Unterverzeichnisse zur Ablage von Dateien generiert werden. Auf diese Art wird der Aufbau des Diskettenverzeichnisses um ein Vielfaches übersichtlicher, da Unterverzeichnisse wiederum andere Unterverzeichnisse (sogenannte "Subdirectories") enthalten können.

Wenn Sie sich das Inhaltsverzeichnis einer Diskette ansehen, dann versieht AmigaDOS alle Namen, die zu einem Unterverzeichnis gehören, mit dem Zusatz "(dir)" hinter dem betreffenden Eintrag.

Das bereits erwähnte "Root-Directory" ist das Basisverzeichnis der Diskette. Alle weiteren Unterverzeichnisse bauen direkt oder indirekt auf dieser Basis auf. AmigaDOS verwendet den Doppelpunkt ":" als Namen für das Root-Directory. Wenn Sie sich in einem Unterverzeichnis befinden, dann gelangen Sie mit dem Befehl "CD :" wieder zurück zum Hauptverzeichnis der Diskette. Ein Beispiel: Lautet der aktuelle Pfad (als "Pfad" bezeichnet man den "Weg", den man vom Hauptverzeichnis gehen muß, um zu einem Unterverzeichnis zu gelangen) "df0:test/magic/ceee", dann führt "CD :" zum Hauptverzeichnis des Laufwerks "df0:".

Das Hauptverzeichnis ist die oberste Stufe des Dateisystems einer Diskette. Mit Hilfe des Befehls "CD" ("Change Directory", etwa: Wechsel auf neues Verzeichnis) kann ein Unterverzeichnis der Diskette zum aktuellen Verzeichnis erklärt werden, auf das sich dann alle folgenden Dateibefehle beziehen.

Wenn Sie im Hauptverzeichnis ein Unterverzeichnis mit Namen "Tests" haben, dann ist nach Eingabe des Befehls "CD Tests" dieses Unterverzeichnis das aktuelle "Arbeitsverzeichnis".

Ein erneutes Ausführen des Befehls zum Auflisten des Diskettenverzeichnisses (o.ä.) zeigt nun nicht mehr den Inhalt des Hauptverzeichnisses an, sondern nur den des Verzeichnisses "Tests".

Befindet sich "Tests" auf einer Diskette, die in einem externen Laufwerk verwaltet wird, dann kann man entweder den Laufwerksnamen (CD df1:Tests) oder den Namen der Diskette (CD MeineDisk:Tests) angeben, um auf dieses Verzeichnis zu wechseln. Wird der Diskettenname verwendet, dann prüft AmigaDOS vor dem Ausführen des Befehls, ob sich eine Diskette mit dem angegebenen Namen in einem Laufwerk befindet. Ist dies nicht der Fall, so wird die korrekte Diskette angefordert. Der Name einer Diskette wird nach dem Formatieren festgelegt. Der Amiga benennt grundsätzlich alle neu formatierten Disketten mit dem Namen "Empty", der allerdings von der Workbench oder vom CLI aus geändert werden kann.

Wenn Sie sich schon länger mit dem Amiga beschäftigen, dann war der letzte Abschnitt sicherlich nicht viel mehr als eine Wiederholung für Sie. Trotzdem bildet er die Grundinformation, die zum Arbeiten mit dem Dateisystem notwendig ist; insbesondere, wenn Sie die Dateifunktionen von eigenen Programmen aus nutzen wollen. Die folgenden Abschnitte gehen näher auf die Struktur einer Diskette ein und zeigen, wie man mit Diskettenverzeichnissen arbeitet.

Arbeiten mit AmigaDOS

Das Eingeben von Befehlen im CLI ist nur eine Möglichkeit, um mit AmigaDOS zu arbeiten. Die zweite besteht darin, diese Befehle in eigenen Programmen zu verwenden, z.B. um eine Datei zu löschen oder umzubenennen. Es stehen Ihnen alle Befehle zur Verfügung, die Sie auch vom CLI aus benutzen können. AmigaDOS stellt Ihnen dafür eine Funktion mit Namen "Execute" zur Verfügung, die komplette Befehlszeilen verarbeitet; so, wie dies auch geschieht, wenn Sie eine Zeile im CLI eingeben.

Mit der Verwendung der Funktion Execute sind zwei Bedingungen verbunden:

- Der Befehl "Run" muß sich im aktuellen C-Verzeichnis der Diskette befinden.
- Der auszuführende Befehl muß entweder im Hauptverzeichnis oder im C-Verzeichnis der Diskette vorhanden sein.

Das folgende Beispiel zeigt eine Anwendung der Execute-Funktion. Beachten Sie bitte, daß der Kommando-String das Symbol ">" enthält, um die Ausgabe in eine Datei umzulenken.

```
/* execute.demo.c */

main()
{
  int erfolg;

  erfolg = Execute("dir >df0:ausgabedatei",0,0);
  if(erfolg == FALSE) printf("Fehler bei Ein-/Ausgabe: %d\n",IoErr());
}
```

Dieses Programm gibt das Inhaltsverzeichnis der Diskette nicht auf dem Bildschirm, sondern in eine Datei namens "ausgabedatei" aus. Um sich den Inhalt der Datei anzusehen, können Sie "TYPE df0:ausgabefile" eingeben; der Dateiinhalt wird dann im aktuellen CLI-Fenster ausgegeben.

Wie Sie im Beispiel sehen können, benötigt die Execute-Funktion drei Parameter. Der erste ist die Befehlszeile, die ausgeführt werden soll. Sie können hier optional eine Umleitung der Ein-/Ausgabe vornehmen, indem Sie die Symbole ">" oder "<" verwenden. ">" bestimmt, wohin die Ausgabe erfolgen soll; "<" definiert den Eingabepfad.

Der zweite und dritte Parameter beinhaltet jeweils einen Zeiger auf die Datenstruktur einer Datei (file handle), der besagt, wohin die Ein- oder Ausgabe erfolgen soll.

Haben diese Parameter den Wert Null, dann verwendet AmigaDOS als Standard-Ein-/Ausgabe die Pfade, die auch für das aktuell ausgeführte Programm gelten. Ein Funktionsaufruf der Form Execute("dir",0,0); gibt das Verzeichnis der Diskette also im aktuellen CLI-Fenster aus.

Funktionsaufruf oder Execute?

Anstatt die Funktion Execute zu verwenden, können Sie natürlich auch direkt die AmigaDOS-Funktionen verwenden, die zur Arbeit mit Dateien zur Verfügung stehen.

Der Vorteil dieses Verfahrens liegt darin, daß weder der Befehl "Run", noch der auszuführende Befehl (Rename, Dir, Delete usw.) auf der Diskette vorhanden sein muß. Der Nachteil bei der direkten Verwendung der AmigaDOS-Funktionen besteht darin, daß Sie Dateinamen (o.ä.) nicht signifikant abkürzen können, wie das beim Arbeiten mit Execute möglich ist.

Verwendung des Zeigers, den die Open-Funktion liefert

Sie haben bereits erfahren, daß man den Zeiger, den AmigaDOS beim Öffnen einer Datei liefert, zum Arbeiten mit dieser Datei weiterverwenden kann, z.B. um die Ein- oder Ausgabe umzuleiten. Hier noch einmal ein Beispiel zur Umleitung der Ausgabe:

```
DIR >magic.list
```

Das Inhaltsverzeichnis der Diskette wird nun nicht mehr auf dem Bildschirm, sondern in eine Datei mit dem Namen "magic.list" ausgegeben; die Standard-Ausgabe wurde umgeleitet.

Das Beispielprogramm in Listing 2.5 zeigt Ihnen, wie die Umleitung mit Hilfe des Zeigers erfolgt, den Sie von der Open-Funktion als Returnwert erhalten.

```
/* execute.demo2.c */

#include "libraries/dosextens.h"
extern struct FileHandle *Open();
main()
{
  struct FileHandle   *MeineAusgabeDatei;
  int erfolg;

  MeineAusgabeDatei = Open("df0:magic.list",MODE_NEWFILE);

  if(!MeineAusgabeDatei)
       {
         printf("Fehler bei Ein-/Ausgabe: %d\n",IoErr());
         exit(FALSE);
       }

  erfolg = Execute("dir",0,MeineAusgabeDatei);

  if(!erfolg) printf("Ein-/Ausgabefehler: %d\n",IoErr());
  Close(MeineAusgabeDatei);
}
```

Listing 2.5: Beispiel zur Verwendung des Dateizeigers in der *Execute*-Funktion

Die "Zweige" des Directory-Baumes und wie man sie anspricht

Das vorige Beispielprogramm zeigt Ihnen, wie Sie einen Befehl mit Execute ausführen können, der sich auf das aktuelle Verzeichnis der Diskette bezieht. Manchmal ist es jedoch erforderlich, in ein anderes Verzeichnis zu wechseln, z.B. um den Anwender auf diese Weise aufzufordern, eine andere Diskette einzulegen.

AmigaDOS bietet die Möglichkeit, von eigenen Programmen aus so mit dem Dateisystem zu arbeiten, wie Sie dies auch vom CLI aus tun können. Diese Möglichkeit wird als "Lock" (deutsch: Schloß) bezeichnet. Mit einem "Lock" sagen Sie AmigaDOS, daß sich alle weiteren Dateizugriffe auf das Inhaltsverzeichnis beziehen, für das Sie ein "Lock" definieren.

Ein Lock kann für jedes Verzeichnis angewendet werden, egal, ob es sich dabei um das Root- oder ein Unterverzeichnis handelt. Die Möglichkeit des "Verschließens" von Verzeichnissen ist auf einem Multitasking-System unab-

dingbar, da mehrere Programme gleichzeitig Dateioperationen durchführen können. Wenn Sie zum Beispiel ein Lock für ein Unterverzeichnis definieren, dann kann dieses Verzeichnis nicht von einem anderen Programm gelöscht werden, was ohne diese Möglichkeit fatale Folgen haben könnte.

Damit AmigaDOS Dateien und Verzeichnisse korrekt verwalten kann, müssen Sie natürlich vor dem Beenden Ihres Programms alle Locks, die Sie angefordert haben, wieder freigeben. Andere Programme gehen sonst davon aus, daß bestimmte Files oder Verzeichnisse noch durch Ihr Programm bearbeitet werden, obwohl Ihr Programm schon längst beendet ist.

Das Programm in Listing 2.6 ist ein Beispiel zur Definition und Verwendung eines Locks. Das Beispiel liefert das gleiche Resultat wie das CLI-Kommando

```
CD <neues_verzeichnis>
```

bezieht sich jedoch nur auf Ihr Programm; das Verzeichnis, in dem der CLI arbeitet, wird nicht geändert.

Beim Ausführen des Beispielprogramms wird also nicht das aktuelle Verzeichnis auf dem Bildschirm ausgegeben, sondern das C-Verzeichnis der Diskette. Nach dem Compilieren des Programms sollten Sie zuerst einmal den CLI-Befehl "dir" ausführen lassen, um das aktuelle Verzeichnis ausgeben zu lassen, in dem der CLI arbeitet. Danach starten Sie das Beispielprogramm, das den Inhalt des C-Verzeichnisses ausgibt. Ein Ausführen des Befehls "CD" zeigt Ihnen, daß das aktuelle CLI-Verzeichnis nicht geändert wurde.

Das Beispielprogramm läuft also unabhängig vom CLI; der Wechsel in das C-Verzeichnis der Diskette und alle weiteren Zugriffe erfolgen nur von Ihrem Programm aus.

"Wandern" auf dem Directory-Pfad

AmigaDOS stellt Ihnen zum Arbeiten mit Verzeichnissen bzw. der Diskettenstruktur einige Funktionen zur Verfügung:

IoErr	Gibt die Fehlernummer der letzten Ein-/Ausgabeoperation aus
Lock	Definiert für Ihr Programm einen Directory-Pfad

```
/* my.cdir.c */

#include "libraries/dosextens.h"
extern struct FileLock *Lock(), *CurrentDir();
main()
{
  struct FileLock *lock,*oldlock;
  struct FileHandle *meine_datei;
  int erfolg;

  /* Zeiger auf das c-Verzeichnis der Diskette in DF0: definieren */

  lock = Lock("df0:c",ACCESS_READ);
  if(!lock)
     {
       printf("Funktion Lock nicht erfolgreich durchgeführt!\n");
       exit(20);
      }

  /* Falls ein gültiger Zeigerwert geliefert wurde:
     ins Verzeichnis wechseln */

  oldlock = CurrentDir(lock);

  /* Einen DOS-Befehl durchführen, Ausgabe in geöffnete Datei umleiten */

  erfolg = Execute("dir >dir.file",0,0);
  if(!erfolg)
     {
       printf("Execute nicht ausgeführt! Fehlernummer: %ld\n",IoErr());
       oldlock = CurrentDir(oldlock); /* Zurück zum Hauptverzeichnis */
       UnLock(lock);
       exit(40);
      }
  oldlock = CurrentDir(oldlock);

  UnLock(lock);    /* Lock freigeben */
}
```

Listing 2.6: Beispiel für einen eigenen Directory-Befehl

CurrentDir	Wechsel in ein bestimmtes Verzeichnis
Examine	Initialisiert eine Datenstruktur mit Informationen über ein bestimmtes Verzeichnis
ExNext	Initialisiert eine Datenstruktur mit Infomationen über Dateien in einem bestimmten Verzeichnis
ParentDir	Führt zurück zum Hauptverzeichnis der Diskette

Auslesen der AmigaDOS-Fehlernummern

Wenn Sie AmigaDOS-Funktionen verwenden, dann sieht ein Aufruf der Funktion meistens so aus:

```
erfolg = Funktion(parameter);
```

oder

```
zeiger = Funktion(parameter);
```

Enthalten "erfolg" oder "zeiger" den Wert Null, dann wurde die Funktion nicht erfolgreich abgeschlossen. Um festzustellen, warum eine Funktion nicht ausgeführt wurde, stellt AmigaDOS die Funktion IoErr zur Verfügung, die die Fehlernummer der letzten Ein-/Ausgabeoperation liefert. Ein Aufruf der Funktion IoErr erfolgt auf diese Weise:

```
fehlernummer = IoErr();
```

Eine Erklärung der Fehlernummern finden Sie in der Datei "libraries/dos.h", die sich auf Ihrer Compiler-Diskette befindet.

Anfordern eines "Locks" für Dateien und Verzeichnisse

Hier ein Beispielaufruf der Funktion Lock:

```
MeinLock = Lock(pfadname,zugriffsart);
```

Die Funktion Lock benötigt zwei Parameter. Der erste Parameter enthält den Pfad, auf dem AmigaDOS das Verzeichnis oder die Datei erreicht, für die Sie ein Lock definieren wollen.

Dieser Pfad kann den Diskettennamen enthalten, den Namen eines beliebigen Verzeichnisses oder den "Null-String" (""). Anwendungen und Beispiele, die den "Null-String" verwenden, finden Sie später in diesem Kapitel.

Der zweite Parameter "zugriffsart" ist wieder eine symbolische Konstante, die ebenfalls in der Datei "dos.h" definiert ist. Gültig für den zweiten Parameter ist entweder die Angabe "ACCESS_READ" bzw. "SHARED_LOCK" oder "ACCESS_WRITE" bzw. "EXCLUSIVE_LOCK".

Verwenden Sie "SHARED_LOCK", dann können andere Programme auch weiterhin mit dem Verzeichnis oder der Datei arbeiten. Benutzen Sie hingegen den Modus "EXCLUSIVE_LOCK", dann ist es anderen Programmen von AmigaDOS her "verboten", auf ein so spezifiziertes Verzeichnis oder eine Datei zuzugreifen; Sie haben somit exklusiven Zugriff und das Verzeichnis oder die Datei für andere Prozesse "verschlossen".

Beim Aufruf der Lock-Funktion erhalten Sie einen Zeiger als Returnwert. Er weist auf eine "FileLock"-Datenstruktur, die Informationen enthält, die später von AmigaDOS verwendet werden, um auf eine Datei zuzugreifen (falls Sie ein Lock für eine Datei angefordert haben).

Haben Sie ein Verzeichnis "verschlossen", dann benutzt AmigaDOS den Inhalt der Datenstruktur, um Dateien in diesem Verzeichnis anzusprechen. Wenn Sie feststellen wollen, ob eine Datei existiert, dann sollten Sie versuchen, für diese Datei ein Lock anzufordern. Diese Methode ist oftmals schneller, als wenn Sie die Open-Funktion dafür verwenden.

Liefert die Funktion Lock einen Zeiger, dessen Wert ungleich Null ist, dann existiert die Datei, und Sie können versuchen, diese zu öffnen. Ist der Wert des Zeigers jedoch Null, dann ist diese Datei nicht vorhanden.

Die Freigabe von Dateien und Verzeichnissen (UnLock)

Haben Sie Dateien oder Verzeichnisse einer Diskette mit der Funktion Lock "abgeschlossen", dann müssen Sie diese vor Beendigung Ihres Programms wieder freigeben, damit andere Prozesse wieder Zugriff haben. Sie tun dies mit der Funktion "UnLock", die wie folgt aufgerufen wird:

```
UnLock(MeinLock);
```

Der notwendige Parameter für die Funktion ist der Zeiger, den Sie nach dem Aufruf der Funktion Lock erhalten. Denken Sie daran: Vor dem Austieg aus Ihrem Programm müssen Sie alle "Locks" freigeben, damit AmigaDOS weiterhin korrekt arbeiten kann.

Wechsel von einem Verzeichnis in ein anderes

Die Funktion CurrentDir dient dazu, von einem Verzeichnis in ein anderes zu wechseln. Als Returnwert erhalten Sie einen Zeiger auf die Datenstruktur des Verzeichnisses, in dem Sie sich gerade befinden, damit Sie später wieder dorthin zurückkehren können.

Die Funktion wird folgendermaßen aufgerufen:

```
altes_verzeichnis = CurrentDir(neues_verzeichnis);
```

Der Parameter "neues_verzeichnis" ist der Zeiger, den die Funktion Lock liefert. Die Funktion CurrentDir ist äquivalent zum CLI-Befehl "CD", der jedoch in ein neues Verzeichnis wechselt, ohne das alte zwischenzuspeichern. Die folgenden Beispiele haben den gleichen Effekt:

```
erfolg = Execute("cd df0:c",0,0);
```

und

```
neues_verzeichnis = Lock("df0:c",ACCESS_READ);
altes_verzeichnis = CurrentDir(neues_verzeichnis);
```

Der Vorteil der zweiten Methode liegt darin, daß der Befehl "Run" nicht im C-Verzeichnis der aktuellen Diskette vorhanden sein muß.

Noch ein Hinweis: Nur die "Locks", die die Funktion Lock liefert, sollten später mit UnLock wieder freigegeben werden; verwenden Sie UnLock niemals dazu, die "Locks" der Funktion CurrentDir freizugeben. AmigaDOS unterscheidet zwischen diesen beiden Arten von Locks, da der Zeigerwert der Funktion CurrentDir von AmigaDOS in anderer Weise verwaltet wird als der der Funktion Lock.

Versuchen Sie dennoch, UnLock auf den Returnwert der Funktion CurrentDir anzuwenden, dann ist AmigaDOS nicht mehr in der Lage, die betreffende Diskette oder das Verzeichnis anzusprechen. Also die Freigabe durch Unlock nur bei den Returnwerten der Funktion Lock anwenden!

Informationen über eine Datei oder ein Verzeichnis

Jede Datei und jedes Verzeichnis besitzt eine Datenstruktur, die Informationen verschiedener Art enthält (Größe der Datei, das Datum des ersten Zugriffs usw.). Um an diese Informationen zu kommen, können Sie die AmigaDOS-Funktion Examine verwenden. Hier der Aufruf:

```
erfolg = Examine(MeinLock,Startadresse_der_Datenstruktur);
```

Der erste Parameter ist der Zeiger, den die Lock-Funktion liefert, der zweite zeigt auf den Anfang einer Datenstruktur ("FileInfoBlock"), in der die Informationen abgelegt werden sollen.

Nach dem erfolgreichen Aufruf der Funktion enthält diese Datenstruktur Informationen darüber, ob es sich bei dem angegebenen "Lock" um eine Datei oder ein Verzeichnis handelt, die Größe (falls es eine Datei ist) in Bytes und Blöcken usw. Die Komponenten der Datenstruktur sind im Beispielprogramm Listing 2.7 einzeln aufgeführt.

```
/* exam.example.c */

#include "libraries/dos.h"
#include "exec/memory.h"
extern struct FileLock *Lock();
long rmask = ((long)('r')<<24);
long brmask = ((long)(' ')<<24);
long wmask = ((long)('w')<<16);
long bwmask = ((long)(' ')<<16);
long emask = ((long)('e')<<8);
long bemask = ((long)(' ')<<8);
long dmask (long)('d');
long bdmask (long)(' ');

char *months[]={"","January","February","March","April","May","June",
"July","August","September","October","November","December"} ;
long n ;
int m, d, y ;
```

Listing 2.7: Beispielprogramm zur Verwendung der "Examine"-Funktion (Teil 1)

```
ShowDate(v)
        long *v;
{
           n = v[0] - 2251 ;
           y = (4 * n + 3) / 1461 ;
           n -= 1461 * y / 4 ;
           y += 1984 ;
           m = (5 * n + 2) / 153 ;
           d = n - (153 * m + 2) / 5 + 1 ;
           m += 3 ;
           if (m > 12)
           {
                      y++ ;
                      m -= 12 ;
           }
           printf("%s %d, %d\n", months[m], d, y) ;
        return(0);
}


struct
{
  long pmask;
  char stringnull;
} maskout;   /* Struktur zur Generierung der Protektions-Bits */


main()
{
  struct FileInfoBlock *fib;
  struct FileLock *lock;
  int erfolg,p;


  /* Überprüft werden soll die "dir" - datei im c-Verzeichnis */


  fib = (struct FileInfoBlock *)AllocMem(sizeof(struct FileInfoBlock),
          MEMF_CLEAR);


  lock = Lock("df0:c/dir",ACCESS_READ);
  if(lock)
```

Listing 2.7: Beispielprogramm zur Verwendung der "Examine"-Funktion (Teil 2)

```
    {
      erfolg = Examine(lock,fib);

      if(erfolg)
         {
           printf("Die Datei %s ",&fib->fib_FileName[0]);
           if(fib->fib_DirEntryType>0)
              printf("ist ein Verzeichnis.\n");
           else printf("ist eine Datei.\n");

           /* Prüfen, ob die Datei geschützt ist */

          p = fib->fib_Protection;
          maskout.pmask=0;
          maskout.stringnull='\0';
          /* Nullbyte = Ende einer Zeichenkette */

          if(p&FIBF_READ) maskout.pmask|=brmask;
          else maskout.pmask|=rmask;

          if(p&FIBF_WRITE) maskout.pmask|=bwmask;
          else maskout.pmask|=wmask;

          if(p&FIBF_EXECUTE) maskout.pmask|=bemask;
          else maskout.pmask|=emask;

          if(p&FIBF_DELETE) maskout.pmask|=bdmask;
          else maskout.pmask|=dmask;

          printf("Gesetzte Protektionsbits: %s\n",&maskout);
          printf("Dateigröße in Bytes: %ld\n",fib->fib_Size);
          printf("Dateigröße in Blocks: %ld\n",fib->fib_NumBlocks);
          printf("Kemmentarzeile der Datei: %s\n",fib->fib_Comment);

          /* Weiterhin existiert eine Datumsangabe im FileInfoBlock.
             Wie man das Datum und die Uhrzeit ausliest, sehen Sie
             später in diesem Kapitel. */

          printf("Letzte Änderung der Datei: ");
          ShowDate(&(fib->fib_Date));
         }
      UnLock(lock);
    }
}
```

Listing 2.7: Beispielprogramm zur Verwendung der "Examine"-Funktion (Schluß)

Wenn Sie sich beim Ausführen der Examine-Funktion im Root-Verzeichnis der Diskette befinden, dann beinhaltet das Feld "FileName" den Diskettennamen. Haben Sie zwei Disketten mit gleichem Namen, dann können Sie auf diese Art durch das Auslesen des "DateStamps" (Datum und Uhrzeit des ersten Zugriffs) eine Unterscheidung zwischen beiden Disketten vornehmen.

Das Programm in Listing 2.7 gibt nur die Informationen aus, die für den Programmierer auch wirklich interessant sind. Es existieren noch weitere Datenfelder im "FileInfoBlock", mit denen AmigaDOS jedoch nur intern arbeitet.

Der "FileInfoBlock" wird angelegt im Beispielprogramm mit der Funktion AllocMem. Der Grund hierfür liegt in der Tatsache, daß AmigaDOS die Startadresse der Datenstruktur an einer geraden Speicheradresse erwartet. AllocMem verwendet zur Reservierung von Speicher grundsätzlich Werte vom Typ LONG (32 Bit) – somit wird die von AmigaDOS geforderte Bedingung erfüllt.

Informationen über weitere Dateien in einem Verzeichnis

Liefert die Funktion Examine als "FileName" den Namen eines Verzeichnisses, dann können Sie mit Hilfe der Funktion ExNext die Datenstrukturen der in diesem Verzeichnis vorhandenen Dateien oder Verzeichnisse abrufen. Sie rufen ExNext wie folgt auf:

```
erfolg = ExNext(MeinLock,Startadresse_der_Datenstruktur);
```

Die verwendeten Parameter entsprechen denen der Examine-Funktion. Bevor Sie ExNext verwenden können, müssen Sie zuvor Examine aufrufen, da der "FileInfoBlock", den diese Funktion liefert, von ExNext zur Bestimmung des nächsten Verzeichnisses oder der nächsten Datei verwendet wird.

Die Datenstruktur wird dann nach dem gleichen Muster gefüllt, wie dies bei der Funktion Examine der Fall ist. Ist kein weiteres Verzeichnis bzw. keine weitere Datei vorhanden, dann liefert ExNext als Returnwert einen Null-Zeiger.

Rückkehr zum nächsthöheren Verzeichnis

Wenn Sie für eine Datei oder ein Verzeichnis ein "Lock" erhalten haben, dann liefert die Funktion ParentDir einen Zeiger auf das Verzeichnis, in dem die Datei oder das Unterverzeichnis enthalten ist.

Sie können auf diese Art und Weise schrittweise zum Hauptverzeichnis der Diskette zurückkehren. Haben Sie dieses erreicht, dann liefert die Funktion ParentDir einen Null-Zeiger. Der Aufruf der Funktion ist nachfolgend beschrieben:

```
uebergeordnetes_verzeichnis = ParentDir(MeinLock);
```

Beispielprogramm: Anwendung von Directory-Funktionen

In Listing 2.8 finden Sie ein Programm, das in etwa die gleiche Funktion erfüllt wie der CLI-Befehl "dir opt a", mit dem Sie sich den Inhalt aller Verzeichnisse einer Diskette auf dem Bildschirm ausgeben lassen können.

Der Unterschied zwischen diesem CLI-Befehl und dem Beispielprogramm liegt darin, daß das Programm die Namen von Dateien und Verzeichnissen nicht sortiert auf dem Bildschirm ausgibt.

Sie können das Programm aus einem beliebigen Verzeichnis aus aufrufen; es arbeitet unabhängig vom aktuellen CLI-Verzeichnis.

Bestimmung des akutellen Verzeichnisses

Um festzustellen, welches Verzeichnis der Diskette das aktuell benutzte ist, können Sie eine spezielle Fähigkeit des AmigaDOS ausnutzen: Wenn Sie beim Aufruf der Lock-Funktion einen leeren String ("") als Parameter angeben, dann erhalten Sie als Returnwert der Funktion einen Zeiger auf das aktuelle Arbeitsverzeichnis.

```
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"
extern struct FileLock *Lock(), *Duplock(), *CurrentDir();
main(argc,argv)
{
  struct FileLock *oldlock;

oldlock = Lock ("", ACCESS_READ);
if(oldlock != 0)
  followthread(oldlock,0);
else
  printf("Kein Lock für das aktuelle Verzeichnis
          möglich!\n");
  printf("\n");
}   /* Ende von main() */

/* Die folgende Funktion "followthread" gibt den
   kompletten Pfad des angewählten Verzeichnisses aus. */

followthread(lock,tab_level)
struct FileLock *lock;
int tab_level;
{
  struct FileInfoBlock *m;
  struct FileLock *oldlock,*newlock,*ignoredlock;
  int erfolg,i;

  /* Nichts ausgeben, falls das Ende des Pfades erreicht
     wurde */

  if(!lock) return();

  /* Speicher für den FileInfoBlock reservieren */

 m = (struct FileInfoBlock *)AllocMem(sizeof(FileInfoBlock),
          MEMF_CLEAR);

  erfolg = Examine(lock,m);
  /* Der erste Aufruf von Examine initialisiert den
     FileInfoBlock. Wurde das Programm vom Root-Verzeichnis
     der Diskette aus aufgerufen, dann enthält der Block
```

Listing 2.8: Das Programm "dir opt a" (Teil 1)

```
      zusätzlich den Diskettennamen. Das Programm gibt daher
      nur das Resultat der Funktion ExNext aus, da sonst
      alle Dateinamen doppelt ausgegeben werden, wenn ExNext
      und Examine zusammen verwendet werden. */

   while(erfolg)
   {
     if(m->fib_DirEntryType>0)  /* Verzeichnis: alle
                                   Dateinamen ausgeben */
        {
          newlock = Lock(&m->fib_FileName[0],ACCESS_READ);

          /* Falls das Lock gültig ist, wird in das neue
             Verzeichnis gewechselt; der vorhergehende
             Verzeichnisname wird jedoch zwischengespeichert,
             damit wieder dorthin zurückgekehrt werden
             kann. */

          oldlock = CurrentDir(newlock);

          /* Rekursiver Aufruf der Funktion "followthread" */
          followthread(newlock,tab_level+1);

          /* Nachdem der gesamte Inhalt eines
             Unterverzeichnisses ausgegeben wurde, wird an
             dieser Stelle der Programmablauf fortgesetzt. */

          ignoredlock = CurrentDir(oldlock);
         }

      erfolg = ExNext(lock,m);
      if(erfolg)
         {
           printf("\n");
           for(i=0;i<tab_level;i++) printf("\t");
                                  /* Ausgabe einrücken */
           printf("%ls",&m->fib_FileName[0]);
           if(m->fib_DirEntryType>0) printf(" [dir]);
         }
    if(lock) UnLock(lock);
    FreeMem(m,sizeof(struct FileInfoBlock));
}     /* Ende der Funktion followthread() */
```

Listing 2.8: Das Programm "dir opt a" (Schluß)

Diesen Zeiger (Lock) können Sie dann wie gewohnt mit allen anderen Funktionen verwenden, die ein Lock benutzen, z.B. um schrittweise zum Hauptverzeichnis der Diskette zurückzukehren. Das folgende Beispielprogramm sucht den Namen des aktuellen Verzeichnisses und wechselt danach zum nächsthöheren (übergeordneten) Verzeichnis.

Das Beispielprogramm in Listing 2.9 gibt Informationen über das aktuelle Arbeitsverzeichnis auf dem Bildschirm aus. Bevor das Programm beendet wird, erfolgt ein Aufruf der Funktion UnLock, mit der das mit der Funktion Lock "abgeschlossene" Verzeichnis wieder für andere Prozesse freigegeben wird.

Denken Sie daran: Sie müssen jede Datei und jedes Verzeichnis, das Sie für Ihr Programm in Anspruch nehmen, vor dem Ende des Programms wieder freigeben, damit AmigaDOS weiterhin korrekt arbeiten kann. Gleiches gilt für reservierten Speicher und alle anderen Dinge, die Systemkomponenten temporär für sich in Anspruch nehmen.

Das Programm in Listing 2.9 ruft die Funktion FollowPath rekursiv auf. Die Aufgabe dieser Funktion ist es, schrittweise auf dem Pfad der Diskette zurückzugehen, bis das Hauptverzeichnis erreicht wird. Jedes Unterverzeichnis wird mit einem "/" markiert; das Hauptverzeichnis wird mit einem ":" gekennzeichnet.

Wird das Root-Directory erreicht, dann erhalten Sie einen Zeiger auf den Namen der aktuell angesprochenen Diskette. Um dieses Programm auszuprobieren, sollten Sie es unter dem Namen "MeinPfad" im C-Verzeichnis Ihrer Diskette ablegen. Ist dies geschehen, dann geben Sie bitte die folgenden CLI-Befehle ein:

```
CD df0:
MAKEDIR magic
CD magic
MAKEDIR ceee
CD ceee
CD
```

Nach dem letzten CD-Befehl erhalten Sie die Meldung "df0:magic/ceee". Starten Sie jetzt das Beispielprogramm mit "MeinPfad" und <Return>, dann wird "DISKETTENNAME:magic/ceee" auf dem Bildschirm ausgegeben, wobei "DISKETTENNAME" der Name Ihrer aktuell verwendeten Diskette ist.

```
/* mybranch.c */

#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"
extern struct FileLock *Lock(), *Duplock(), *ParentDir();
main()
{
  struct FileLock *mylock,*oldlock;

  oldlock = Lock("",ACCESS_READ);    /* Lock für aktuelles
                                        Verzeichnis */
  if(oldlock)
     {
       printf("Pfad zum aktuellen Verzeichnis: ");
       followpath(oldlock,0);
     }

  else printf("Kann kein Lock für das aktuelle Verzeichnis definieren!\n");
}   /* Hinweis: UnLock erfolgt in der Funktion followpath! */

followpath(lock,printslash)
struct FileLock *lock;
int printslash;
{
  struct FileInfoBlock *myinfo;
  struct FileLock *newlock;
  int erfolg,fehler;

  /* Nichts ausgeben, falls das Ende des Pfades erreicht wurde */
  if(lock) return();

  myinfo = (struct FileInfoBlock *)AllocMem(sizeof(struct FileInfoBlock),
                    MEMF_CLEAR);

  if(!myinfo)
    {
     printf("Nicht genügend Speicher für den FileInfoBlock vorhanden!\n");
     return();
    }

  /* Prüfen, ob dem aktuellen Verzeichnis ein anderes übergeordnet ist */
  newlock = ParentDir(lock);
  fehler = IoErr();
  /* Enthält die Fehlernummer, falls die Ein-/Ausgabe nicht durchgeführt
     werden kann (z.B. wenn die Diskette entfernt wurde) */
```

Listing 2.9: Ausgabe des aktuellen Pfadnamens (Teil 1)

```
  if(newlock == 0 && fehler != 0) printf("E/A Fehler: %ld\n",fehler);

  /* Rekursiver Aufruf der Funktion followpath */
  followpath(newlock,1);

  erfolg = Examine(lock,myinfo);
  if(erfolg)
     {
       printf("%s",myinfo->fib_FileName[0]);
       if(!newlock) printf(":");
       else
             {
               if(printslash) printf("/");
               UnLock(lock);
              }
     }

  if(myinfo) FreeMem(myinfo,sizeof(struct FileInfoBlock));
}   /* Ende der Funktion followpath() */
```

Listing 2.9: Ausgabe des aktuellen Pfadnamens (Schluß)

Hilfsfunktionen des AmigaDOS

Nachfolgend finden Sie weitere AmigaDOS-Funktionen, die Sie direkt aus Ihrem Programm heraus verwenden können:

Rename	Umbenennen einer Datei oder eines Verzeichnisses
DeleteFile	Löschen einer Datei oder eines Verzeichnisses
CreateDir	Anlegen eines neuen Verzeichnisses
SetProtection	Schützen einer Datei oder eines Verzeichnisses
SetComment	Dateien oder Verzeichnisse mit einem Kommentar versehen
DateStamp	Auslesen des aktuellen Datums und der Zeit
Info	Ausgabe von diskettenspezifischen Informationen

Es existieren noch weitere Funktionen, die speziell für das Arbeiten auf einem Multitasking-System geschaffen wurden. Nähere Informationen hierzu sind in Kapitel 3 enthalten.

Hilfsfunktionen des CLI

Für die oben aufgeführten AmigaDOS-Funktionen sind äquivalente CLI-Befehle auf Ihrer Workbench-Diskette vorhanden, z.B. "RENAME" (Rename), "MAKEDIR" (CreateDir) oder "PROTECT" (SetProtection).

In diesem Abschnitt finden Sie jeweils drei verschiedene Arten, eine solche Funktion aufzurufen. Damit Sie zwischen CLI-Befehlen und den AmigaDOS-Funktionen unterscheiden können, sind alle CLI-Befehle groß geschrieben. Beginnt ein Beispiel also mit einem groß geschriebenen Wort, dann können Sie es so im CLI verwenden; beginnt ein Beispiel mit dem Returnwert einer Funktion, dann handelt es sich um ein Programmfragment, das Sie so in Ihren Programmen verwenden können.

Umbenennen einer Datei

Ein Aufruf der AmigaDOS-Funktion Rename sieht folgendermaßen aus:

```
int erfolg;
char *alter_name,*neuer_name;

erfolg = Rename(alter_name,neuer_name);
```

Die beiden Parameter "alter_name" und "neuer_name" sind Zeiger auf Zeichenketten, die jeweils den alten bzw. den neuen Namen der Datei oder des Verzeichnisses beinhalten.

Erhalten Sie als Returnwert der Funktion einen Wert gleich Null, dann wurde die Funktion nicht ausgeführt. Mit einem Aufruf der Funktion IoErr können Sie feststellen, warum die Funktion nicht erfolgreich beendet wurde.

Noch ein Hinweis: Sie können Rename auch dazu verwenden, eine Datei von einem Verzeichnis in ein anderes zu kopieren, solange sich Quell- und Zielverzeichnis auf der gleichen Diskette befinden. Geben Sie als alten Namen z.B. "df0:testdatei" und als neuen Namen "df0:knarz/programme/testdatei37" an, dann ist die Datei "test", die sich zuvor im Hauptverzeichnis der Diskette befand, fortan unter dem Namen "testdatei37" im Verzeichnis "programme" zu finden.

Um eine Datei vom CLI aus umzubenennen, geben Sie folgende Zeile ein:

```
RENAME <alter_name> <neuer_name>
```

Sie können natürlich auch die AmigaDOS-Funktion Execute verwenden:

```
erfolg = Execute("rename <alter_name> <neuer_name>",0,0);
```

Eine weitere Alternative ist:

```
erfolg = Rename("<alter_name>","<neuer_name>");
```

Löschen einer Datei

Dateien können mit Hilfe der Funktion DeleteFile gelöscht werden, die folgendermaßen aufgerufen wird:

```
int erfolg;
char *dateiname;

erfolg = DeleteFile(dateiname);
```

Der Parameter "dateiname" ist ein Zeiger auf eine Zeichenkette, die den Pfad enthält, auf dem AmigaDOS die Datei erreichen kann. Der Dateiname oder der Pfad kann jedoch auch direkt – eingeschlossen in Anführungszeichen – eingegeben werden:

```
erfolg = DeleteFile("<dateiname>");
```

Hat "erfolg" den Wert Null, dann wurde die Funktion nicht korrekt ausgeführt. Ein Aufruf der Funktion IoErr gibt nähere Informationen über den aufgetretenen Fehler.

Vom CLI aus wird eine Datei wie folgt gelöscht:

```
DELETE <dateiname>
```

Mit der Funktion Execute sieht der Aufruf so aus:

```
erfolg = Execute("delete <dateiname>",0,0);
```

Anlegen eines neuen Verzeichnisses

Die Funktion CreateDir wird wie folgt ausgeführt:

```
struct FileLock   *MeinLock;
char *verzeichnisname;

MeinLock = CreateDir(verzeichnisname);
```

oder

```
MeinLock = CreateDir("verzeichnisname");
```

Im CLI legen Sie ein Verzeichnis mit dem Befehl "MAKEDIR <verzeichnis>" an. Die Alternative mit der Funktion Execute lautet:

```
erfolg = Execute("makedir <verzeichnis>",0,0);
```

Auch hier können Sie wieder die Funktion IoErr verwenden, falls CreateDir als Returnwert Null liefert.

Schützen einer Datei

Mit der Funktion SetProtection können Sie eine Datei gegen unbeabsichtigtes Löschen schützen. Hier der Funktionsaufruf:

```
int erfolg,bit_maske;
char *dateiname;

erfolg = SetProtection(dateiname,bit_maske);
```

Beim Parameter "bit_maske" werden von AmigaDOS nur die unteren vier Bits verwendet. Diese vier Bits bilden die Maske der Form "RWED" (Read, Write, Execute, Delete). Ist ein Bit dieser Maske gesetzt, dann kann die damit verbundene Funktion nicht auf die betreffende Datei angewendet werden.

AmigaDOS benutzt in seiner gegenwärtigen Version nur den Parameter "D", mit dem eine Datei gegen Löschen geschützt werden kann. Die restlichen drei Parameter "READ" (Lesen), "WRITE" (Schreiben) und "EXECUTE" (Ausführen) können zwar gesetzt werden, werden aber von AmigaDOS beim Prüfen der Maske ignoriert. Sie können diese Maske jedoch von eigenen Programmen aus verwenden, um z.B. zu entscheiden, ob eine Datei ausgeführt werden darf oder nicht.

Beachten Sie bitte, daß der CLI-Befehl "List", mit dem Sie u.a. die Maske einer Datei ausgeben lassen können, die gesetzten Bits nicht anzeigt. Wird also z.B. die Maske "R--D" ausgegeben, dann kann die Datei gelesen und gelöscht, aber nicht beschrieben oder ausgeführt werden. Die Maske der List-Funktion ist also das genaue Gegenteil der Maske, die Sie beim Aufruf der SetProtection-Funktion angeben.

Vom CLI aus wird eine Datei wie folgt geschützt:

```
PROTECT <datei> DW
```

"datei" wird mit diesem Befehl gegen Löschen und Beschreiben geschützt.

Mit der Execute-Funktion schützen Sie eine Datei auf folgende Art:

```
erfolg = Execute("protect <datei> dw",0,0);
```

Eine weitere Möglichkeit zum Schützen einer Datei zeigt folgendes Beispiel:

```
/* RWED = 1111 binär.........-W-D = 0101 binär = 5 dezimal */

erfolg = SetProtection("<dateiname>",5);
```

Dateien oder Verzeichnisse mit einem Kommentar versehen

AmigaDOS bietet Ihnen die Möglichkeit, Dateien oder Verzeichnisse mit einem Kommentar zu versehen, der z.B. einen Hinweis zur Benutzung eines Programms enthalten könnte. Die Funktion SetComment wird wie folgt ausgeführt:

```
int erfolg;
char *dateiname,*kommentar;

erfolg = SetComment(dateiname,kommentar);
```

bzw.

```
erfolg = SetComment("dateiname","Dies ist ein Kommentar");
```

Der CLI-Befehl lautet:

```
FILENOTE <dateiname> "Kommentar für eine Datei"
```

Schließlich noch der Aufruf durch die Execute-Funktion:

```
erfolg = Execute("filenote <dateiname> \"Und noch'n Kommentar\"",0,0);
```

Auslesen des Datums

Die Funktion DateStamp liefert das aktuelle Datum. AmigaDOS verwaltet Uhrzeit und Datum in einem speziellen Format, so daß erst einige Umwandlungen vorgenommen werden müssen, bevor Sie ein "lesbares" Datum auf dem Bildschirm ausgeben können.

Tatsächlich existieren in der Standardbibliothek Ihres Compilers Funktionen, die diese Umwandlungen für Sie vornehmen, jedoch sind diese nicht Bestandteil der AmigaDOS-Bibliothek. Ein Aufruf der Funktion DateStamp hat die Form:

```
DateStamp(v);
```

Der Parameter "v" ist dabei die erste von drei "LONG" (32 Bit) Variablen, die nach dem Aufruf der Funktion das aktuelle Datum und die Uhrzeit enthalten.

Das Beispielprogramm in Listing 2.10 zeigt die Verwendung der Funktion DateStamp sowie die Umwandlung der internen Formate. Das Programm stammt von Tom Rokicki, wurde jedoch um die Ausgabe der aktuellen Uhrzeit erweitert. Beiden sei an dieser Stelle noch einmal für ihre Arbeit gedankt.

Auslesen von diskettenspezifischen Informationen

Wenn Sie die Funktion Lock auf eine Datei oder ein Verzeichnis angewendet haben, dann können Sie den Returnwert der Funktion dazu verwenden, Informationen über die Diskette zu bekommen, auf der sich die Datei oder das Verzeichnis befindet.

Die Funktion Info dient zum Auslesen dieser Informationen. Verwechseln Sie diese Funktion bitte nicht mit dem CLI-Befehl gleichen Namens, der ja Informationen über alle angeschlossenen Datenspeicher ausgibt. Der Aufruf der Info-Funktion hat die Form:

```
struct Lock   *MeinLock;
struct InfoData   *MeineDatenstruktur;
int erfolg;

erfolg = Info(MeinLock,MeineDatenstruktur);
```

```
char monate[] =
{
  "","Januar","Februar","März","April","Mai","Juni","Juli","August",
  "September","Oktober","November","Dezember"
};

struct DateStamp    MagicTime;
long n;
int m,d,y;
char TimeBuffer[50],DateBuffer[50];

main()
{
  long v[3];

  GetDate(v);
  GetTime();
  printf("Datum und Uhrzeit: %s, %s\n",DateBuffer,TimeBuffer);
}

GetDate(v)
long *v;
{
  DateStamp(v);

  n = v[0]-2251;
  y = (4*n+3)/1461;
  n -= 1461*y/4;
  y += 1984;
  m = (5*n+2)/153;
  d = n-(153*m+2)/5+1;
  m += 3;
  if(m>12)
     {
      y++;
      m -= 12;
     }

  sprintf(DateBuffer,"%02d. %s %d",d,monate[m],y);
}

GetTime()
{
  register short h,m;

  DateStamp(&MagicTime);
  h = MagicTime.ds_Minute/60;
  m = MagicTime.ds_Minute%60;

  sprintf(TimeBuffer,"%02d:%02d:%02d",h,m,
              MagicTime.ds_Tick/TICKS_PER_SECOND);
}
```

Listing 2.10: Anwendung der "DateStamp"-Funktion

```
/* info.example.c */

#include "libraries/dos.h"
#include "exec/memory.h"

main()
{
  struct InfoData *id;
  struct Lock *lock;
  struct
  {
    long pmask;
    char stringnull;
  } maskout;
  int erfolg;

  maskout.stringnull = '\0';

  /* Informationen über die Diskette, auf der der Befehl DIR vorhanden ist */
  id = (struct InfoData *)AllocMem(sizeof(struct InfoData),MEMF_CLEAR);
  lock = Lock("df0:c/dir",ACCESS_READ);
  if(lock)
     {
       erfolg = Info(lock,id);
       if(erfolg)
          {
            if(id->id_DiskType == -1) printf("Keine Disk im Laufwerk!\n");
            else
               {
                     printf("Fehler auf der Diskette: %ld\n",id->id_NumSoftErrors);
                     printf("Diskette liegt im Laufwerk %ld\n",id->id_UnitNumber);
                     printf("Diskettenstatus: ");
                     if(id->id_DiskState == ID_WRITE_PROTECTED)
                          printf("schreibgeschützt\n");
                     else if(id->id_DiskState == ID_VALIDATED)
                          printf("Lesen/Schreiben\n");
                     else if(id->id_DiskState == ID_VALIDATING)
                          printf("Nicht validiert!\n");
                     printf("Freie Blocks: %ld\n",id->id_NumBlocks);
                     printf("Belegte Blocks: %ld\n",id->id_NumBlocksUsed);
                     printf("Byted pro Block: %ld\n"id->id_BytesPerBlock);
                     printf("Art der Diskette: \n");
                     maskout.pmask = id->id_DiskType;
                     printf(&maskout);
                     if(!id->id_InUse) printf("Diskette nicht benutzt\n");
                     else printf("Diskette wird benutzt\n");
                    }
               }
          }
}   /* Ende des Programmes */
```

Listing 2.11: Die Info-Funktion und ihre Anwendung

Das Beispielprogramm in Listing 2.11 gibt Informationen über die Diskette auf dem Bildschirm aus, die den CLI-Befehl "dir" im C-Verzeichnis hat, also z.B. Ihre Workbench-Diskette.

Beachten Sie, daß auch in diesem Programm mit der Funktion AllocMem gearbeitet wird, um Speicher für "InfoData" zu reservieren. Die Verwendung dieser Funktion ist erforderlich, da AmigaDOS die Startadresse von "InfoData" an einer geraden Speicherstelle erwartet.

Verschiedene andere Funktionen

Die Funktionen, die in diesem Abschnitt behandelt werden, haben zum größten Teil keine CLI-Äquivalente, sondern sind bereits in CLI-Befehle eingebunden, z.B. in den Befehl "List".

Hier eine Auflistung der behandelten Funktionen bzw. ihrer Resultate:

- Feststellen, mit welcher Diskette das System gebootet wurde

- Delay: Dient zum Aufbau von Warteschleifen

- DeviceProc: Prüfen, welcher Task mit welchem Gerät arbeitet

- Umbenennen einer Diskette

Feststellen, mit welcher Diskette das System gebootet wurde

Wenn Sie der Examine-Funktion einen Leerstring ("") übergeben, dann können Sie auf diese Weise erfahren, welche Diskette als erstes nach dem Einschalten des Rechners eingelegt wurde. Es ist ganz egal, wie oft Sie Disketten wechseln oder den "Assign"-Befehl benutzen; der Amiga "weiß" bis zum Ausschalten des Rechners, mit welcher Diskette zum ersten Mal gebootet wurde.

Das Beispielprogramm in Listing 2.12 verwendet die Examine-Funktion in der oben beschriebenen Form, um den Namen der Diskette festzustellen, mit der Sie nach dem Einschalten gebootet haben.

```
/* bootname.c */

#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"

main()
{
  struct FileInfoBlock *myinfo;
  int erfolg;

  myinfo = (struct FileInfoBlock *)AllocMem(sizeof(struct FileInfoBlock),
                   MEMF_CLEAR);

  erfolg = Examine(0,myinfo);
  if(erfolg) printf("%s\n",&(myinfo->fib_FileName[0]));
  else printf("Fehler!\n");

  FreeMem(myinfo,sizeof(struct FileInfoBlock));
}
```

Listing 2.12: Bestimmung der Boot-Diskette

Erzeugung einer Warteschleife

Mit der AmigaDOS-Funktion Delay können Sie Ihr Programm für einen bestimmten Zeitraum anhalten. Hier der Aufruf:

```
int zeitraum;

Delay(zeitraum);
```

"zeitraum" wird in fünfzigstel Sekunden angegeben; ein Aufruf der Form "Delay(150);" wartet also drei Sekunden, bevor Ihr Programm weiterläuft.

Herausfinden, welcher Task welches Gerät verwendet

Bei einigen AmigaDOS-Funktionen müssen komplexe Datenstrukturen übergeben werden. Daher ist es wichtig zu wissen, ob das Gerät (z.B. ein Laufwerk), das Sie ansprechen wollen, gerade von einem anderen Programm benutzt wird.

Dieser Abschnitt bildet mit dem folgenden das Ende des Überblicks über die AmigaDOS-Funktionen. Die Übergabe von Datenstrukturen und von "Messages" (Nachrichten) wird im nächsten Kapitel genauer behandelt.

Die Funktion DeviceProc stellt fest, welcher Task oder Prozeß gerade mit einem bestimmten Gerät arbeitet.

Aufgerufen wird die Funktion wie folgt:

```
prozess = DeviceProc(name_des_geraetes);
```

"name_des_geraetes" ist ein Zeiger auf eine Zeichenkette, die den Namen des Gerätes enthält, das überprüft werden soll, also z.B. "df0:" oder "df1:". Gültig ist auch die Angabe des Leerstrings (""), wobei als "Gerät" dann das aktuelle Verzeichnis verwendet wird.

Ändern des Diskettennamens

Es gibt zwei Möglichkeiten, um den Namen einer Diskette von einem Programm aus zu ändern. Zum einen können Sie die Funktion Execute verwenden, um den CLI-Befehl "Relabel" ausführen zu lassen.

Die zweite Möglichkeit ist, ein "Message Packet" ("Nachrichtenpaket") an das Laufwerk zu senden, das die betreffende Diskette enthält. Dieses "Message Packet" ist in diesem Fall vom Typ "ACTION_RENAME_DISK". Um den Prozeß zu finden, der gerade mit dem benötigten Laufwerk arbeitet, können Sie die zuvor beschriebene Funktion DeviceProc verwenden.

Das folgende Beispielprogramm benennt die Diskette mit Hilfe der Funktion Execute um:

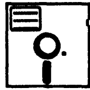
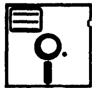
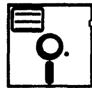
```
main()
{
  int erfolg;

  erfolg = Execute("RELABEL <alter_name> <neuer_name>",0,0);

  if(!erfolg)  printf("Diskette wurde nicht umbenannt!\n");
}
```

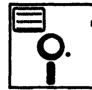
In diesem Kapitel haben Sie fast alles über die Funktionen von AmigaDOS erfahren. Sie haben gelernt, wie man Fenster öffnet, mit Dateien arbeitet und wie man Ein-/Ausgabeoperationen durchführt.

Der Schwerpunkt dieses Kapitels lag darin, Ihnen zu zeigen, welche Funktion man wofür benötigt. Auf den Inhalt und die Verwaltung von Datenstrukturen durch AmigaDOS wurde nicht eingegangen, um die Sache nicht zu erschweren.

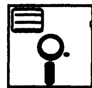
Weitere Informationen hierzu finden Sie im "AmigaDOS Technical Reference Manual" sowie in den Include-Dateien "dos.h" und "dosextens.h", die sich auf Ihrer Compilerdiskette befinden. Eine Auflistung aller Include-Dateien ist im "Amiga ROM Kernel Manual" enthalten.

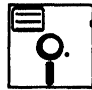
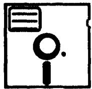
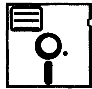
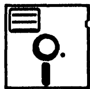
Wenn Sie die Informationen dieses Buches dazu verwenden wollen, um andere Programme auf den Amiga anzupassen, dann sollten Sie sich noch genauer mit der Ein-/Ausgabe und den virtuellen Geräten des Amiga befassen. Das folgende Kapitel behandelt diese Themen allgemein; genauere Informationen finden Sie in Kapitel 6.



3



Kapitel 3

Exec

In diesem Kapitel finden Sie die notwendigen Informationen, die zur Arbeit mit Exec erforderlich sind. Die folgenden Punkte werden behandelt:

- die Struktur von Exec
- die wichtigsten Exec-Routinen
- Exec-Hilfsfunktionen

Wie Sie aus der Auflistung ersehen können, werden nicht alle Funktionen behandelt, die Exec zur Verfügung stellt. Die genaue Erklärung von einigen Funktionen würde den Rahmen dieses Buches sprengen; daher befassen wir uns mit den Grundlagen, die die Basis für weiterführende Anwendungen bilden.

Die Struktur von Exec

Exec bildet die Hauptkomponente des Software-Systems des Amiga. Alle Systemfunktionen werden von Exec mit Hilfe von "Listen" verwaltet, auf die später in diesem Kapitel eingegangen wird.

Weiterhin ermöglicht Exec das Multitasking auf dem Amiga. Multitasking bedeutet, daß mehrere Programme zur gleichen Zeit im Rechner abgearbeitet werden können. Genau genommen laufen diese Programme (auch "Tasks"

oder "Prozesse" genannt) hintereinander ab; durch die hohe Arbeitsgeschwindigkeit des Amiga hat der Anwender jedoch den Eindruck, daß die Programme zeitgleich abgearbeitet werden.

Ohne Exec würde der Amiga diese Fähigkeit nicht besitzen. Exec übernimmt das "Umschalten" zwischen den einzelnen Tasks, d.h. die Verteilung der Rechenleistung der Prozessoren und die Zuteilung von Speicherplatz an die Tasks wird von Exec überwacht.

Die von Exec übernommenen Arbeiten finden Sie nachstehend aufgelistet:

- Exec führt eine Liste des freien Speicherplatzes und teilt diesen den Tasks zu.

- Weiterhin existieren Listen, in denen alle laufenden Programme aufgeführt sind. Exec entscheidet anhand dieser Listen, welcher Task als nächstes abgearbeitet wird und welche temporär "eingefroren" werden.

- Exec verwaltet weiterhin eine Liste der virtuellen Kommunikationsleitungen, mit deren Hilfe Daten und "Messages" (Nachrichten) zwischen den Tasks ausgetauscht werden können. Innerhalb einer solchen Liste vermerkt Exec, ob Nachrichten für eine virtuelle Kommunikationsleitung vorliegen und ob sie schon ausgelesen wurden.

- Verwaltet wird weiterhin eine Liste der vorhandenen Funktionsbibliotheken, mit deren Hilfe Programme auf die gleichen Bausteine der Systemsoftware zugreifen können. Eine Liste der verfügbaren virtuellen Geräte (Devices) ist ebenfalls vorhanden. Über diese Geräte können Programme Ein-/Ausgabeoperationen durchführen.

- Da auf dem Amiga verschiedene Arten von Interrupts verwaltet werden müssen (Interrupts der Hard- oder Software, der Coprozessoren und der Systemsoftware), führt Exec auch hierüber eine Liste.

Warum Listen so wichtig sind

Wie bereits erwähnt, verwendet Exec zur Verwaltung des Systems verknüpfte Listen. Der Vorteil dieser Art der Überwachung liegt darin, daß das System dynamisch den jeweiligen Anforderungen der Programme angepaßt werden

kann, ohne irgendwelchen Eingrenzungen von Seiten der Systemsoftware zu unterliegen. Das bedeutet, daß nach jedem Einschalten des Rechners das gesamte System von Grund auf initialisiert und organisiert wird.

Einige Komponenten der Systemsoftware benötigen Speicherplatz, mit dem intern gearbeitet werden kann. Verfügt Ihr Amiga über mehr als 512 KB Speicher, dann teilt Exec diesen Bausteinen des Software-Systems primär den zusätzlichen Speicher zu.

Der Grund hierfür liegt darin, daß die Coprozessoren nur auf die untersten 512 KB des Systemspeichers zugreifen können. Würde dieser Speicherbereich auch von oben erwähnten Modulen belegt, dann hätten Sie für Ihre Programme weniger Speicher für Grafik und Sound zur Verfügung. Diese Art Daten müssen, um von den Coprozessoren verwendet werden zu können, in den untersten 512 KB Speicher liegen.

Einige Funktionen von Exec und ihre Bezeichnungen

In den folgenden Abschnitten finden Sie Informationen über einige Funktionen von Exec. Hier eine Auflistung der behandelten Punkte:

- Tasks und Prozesse

- Speicherreservierung

- Listen

- Signale

- "Message Ports" (virtuelle Kommunikationsleitungen)

- "Messages" (Nachrichten)

- Bibliotheken

- Devices (virtuelle Geräte)

Diejenigen unter Ihnen, die primär an der Grafikprogrammierung interessiert sind, können direkt zum folgenden Kapitel übergehen, das sich mit diesem Thema befaßt. Vergessen Sie jedoch nicht, daß einige der in diesem Kapitel behandelten Funktionen später bei der Grafikprogrammierung verwendet werden; das Durcharbeiten dieses Kapitels kann also für Sie nur von Vorteil sein.

Weiterhin werden in diesem Kapitel Message Ports, Messages, Signale usw. behandelt, deren Verständnis die Grundlage für ein effektives Arbeiten mit den Ein-/Ausgabefunktionen darstellt. Die Ein-/Ausgabe mit virtuellen Geräten wird ebenfalls in diesem Kapitel behandelt; Informationen zu bestimmten virtuellen Geräten finden Sie in Kapitel 6.

Tasks und Prozesse

Wie bereits zu Beginn des Kapitels erwähnt, verwaltet Exec eine Liste, in der alle Programme (Tasks) aufgeführt sind, die momentan im System ablaufen. Da die Programme hintereinander ausgeführt werden müssen, überwacht Exec anhand dieser Liste die Zuteilung der Rechenleistung des 68000-Prozessors und seiner Coprozessoren.

Erwartet ein Task eine Eingabe, dann kann er sich mit Hilfe von Funktionen des Betriebssystems "einfrieren", so daß andere Programme während dieser Zeit abgearbeitet werden können. Wird ein Task "eingefroren", dann sagt man auch: "Der Task schläft."

Neben dieser Aufteilung der Rechenleistung überwacht Exec auch alle Arten von Interrupts (Tastatureingaben, Bildaufbau usw.). Jedesmal, nachdem ein solcher Interrupt aufgetreten ist, "sieht" Exec in seiner Liste nach, welches der vorhandenen Programme die höchste Priorität hat und bringt dieses dann zur Ausführung – sofern es nicht "schläft", also auf eine Eingabe wartet. Aus diesem Grund kann es vorkommen, daß der Task, der vor dem Auftreten eines Interrupts ausgeführt wurde, temporär gestoppt wird, wenn ein Programm mit höherer Priorität "startbereit" ist.

Exec verwaltet weiterhin für jeden Task einen sogenannten "Task Control Block". Es handelt sich hierbei um eine Datenstruktur, in der die Inhalte aller Systemregister gespeichert werden. Dies ist erforderlich, da jeder Task die Prozessoren auf eine ganz bestimmte Art nutzt.

Würde Exec die Inhalte der Register nicht für jeden Task einzeln zwischenspeichern, dann würde, nach dem Umschalten auf einen anderen Task, dieser mit den Registerinhalten des zuvor abgearbeiteten ausgeführt. Wenn also ein Task "schläft", dann sind die Inhalte der Register zum Zeitpunkt des "Einfrierens" im Task Control Block des Tasks gespeichert.

Diesen Vorgang (Zwischenspeichern der Registerinhalte, Reinitialisierung der Register und Starten eines neuen Tasks) bezeichnet man als "Task Switching", also "Umschalten zwischen Tasks". Dies alles geschieht so schnell, daß der Anwender den Eindruck hat, mehrere Programme laufen zeitgleich ab. In Wirklichkeit jedoch muß Exec schwere Arbeit leisten, um diesen Eindruck entstehen zu lassen.

Speicherreservierung

Der obige Abschnitt beinhaltet nur einen Teil dessen, was Exec tatsächlich leistet. Exec kümmert sich weiterhin – wieder anhand einer Liste – um den freien Systemspeicher. Laufende Tasks können während ihrer Ausführung Systemspeicher anfordern, z.B. um eine Grafik darzustellen. Wenn Ihr Programm Speicherplatz anfordert bzw. reserviert, dann sollten Sie ihn – sofern er später nicht weiterverwendet wird – wieder freigeben, damit andere Programme darauf zugreifen können.

Dies ist wichtig, da Exec nur den freien Speicher in seiner Liste verwaltet; sobald Sie Speicherplatz für sich in Anspruch nehmen, "kennt" Exec diesen Speicher nicht mehr. Sollten Sie Ihr Programm beenden, ohne belegten Speicherplatz wieder freizugeben, dann können andere Tasks nicht mehr darauf zugreifen – bis zum nächsten Neustart des Systems.

Eine einfache Speicherreservierung

Es gibt verschiedene Möglichkeiten, von Exec Speicherplatz anzufordern. Wir besprechen hier nur die am meisten verwendete Methode, um Speicher zu reservieren: durch einen Aufruf der Funktion AllocMem. Ein Aufruf dieser Funktion sieht wie folgt aus:

```
adresse = AllocMem(groesse, speicherart);
```

Der Parameter "groesse" gibt an, wieviel Speicher (in Bytes) Sie reservieren wollen. "speicherart" sagt Exec, welche Art von Speicher Sie verwenden wollen (siehe unten), und auf welche Art er vor der Reservierung durch das System initialisiert werden soll.

Der Returnwert der Funktion "adresse" ist ein Zeiger auf die Startadresse des reservierten Speicherbereiches. Kann das System keinen Speicherbereich der gewünschten Größe reservieren, dann liefert die Funktion AllocMem den Wert Null. Bevor Sie also den angeforderten Speicherbereich verwenden, sollten Sie prüfen, ob er vom System überhaupt bereitgestellt wurde. Nachfolgend finden Sie eine Auflistung der gültigen Angaben bei der Übergabe des Parameters "speicherart":

MEMF_CHIP — Bei dieser Speicherart handelt es sich um die untersten 512 KB des Systemspeichers – dem Bereich also, in dem die Daten (BitMaps usw.) für die Coprozessoren liegen müssen.

MEMF_FAST — Diese Art Speicher können Sie nur anfordern, wenn Ihr Amiga über mehr als 512 KB RAM verfügt. Daten in diesem Speicherbereich können von den Coprozessoren nicht verwendet werden.

MEMF_CLEAR — füllt den angeforderten Speicherbereich mit Nullbytes.

MEMF_PUBLIC — Der Speicherbereich soll auch von anderen Tasks genutzt werden können.

Wie bereits oben erwähnt, müssen Daten für die Coprozessoren (Custom Chips) in den untersten 512 KB des Systemspeichers liegen. Bei diesen Daten handelt es sich meistens um Grafik-, Sound- oder Spritedaten. Sollte Ihr Programm z.B. Grafik verwenden, dann müssen Sie Speicherbereiche vom Typ "MEMF_CHIP" reservieren. Die untersten 512 KB des Systemspeichers werden auch als "ChipMem" bezeichnet; der Grund hierfür liegt in der Tatsache, daß hier hauptsächlich Daten für die Coprozessoren ("Chips") abgelegt werden.

Speicherbereiche vom Typ "MEMF_FAST" liegen immer außerhalb des oben erwähnten Bereiches. Exec versucht immer, Programme und Daten zuerst in diesem Speicher abzulegen, damit möglichst viel ChipMem für die Coprozessoren freibleibt.

Speicherbereiche des Typs "MEMF_FAST" werden auch als "FastMem" bezeichnet. Die Erklärung hierfür ist, daß der 68000-Prozessor unter gewissen Umständen von den Coprozessoren "abgebremst" wird; nämlich dann, wenn z.B. große Speicherbereiche kopiert oder verschoben werden. Da dies jedoch nur beim ChipMem der Fall ist, laufen Programme im "FastMem" mit unverminderter Geschwindigkeit – manchmal sogar schneller – ab, weil dieser Bereich nur vom 68000-Prozessor angesprochen werden kann.

Zusätzlich zur Speicherart können Sie noch den Parameter "MEMF_CLEAR" angeben, der den angeforderten Speicherbereich mit Nullbytes füllt. Sie können hierdurch einige Programmierarbeit einsparen.

"MEMF_PUBLIC" kann zwar angegeben werden, wird in der gegenwärtigen Version des Betriebssystems jedoch ignoriert. Dieser Parameter soll dazu dienen, Speicherbereiche dort anzulegen, wo alle Tasks auf sie zugreifen können – es wird also "öffentlicher" Speicher reserviert. Auf diese Art und Weise wird sich die Kommunikation zwischen Tasks einmal sehr vereinfachen lassen, da die Daten einfach übernommen werden können.

Die Freigabe von reserviertem Speicher

Vor dem Beenden Ihres Programms sollten Sie reservierten Speicher wieder freigeben, damit andere Tasks wieder Zugriff darauf haben können. Exec verwaltet nur eine Liste des freien Speichers; sollten Sie vergessen, reservierten Speicher wieder freizugeben, dann ist er für andere Tasks nicht nutzbar.

Freigegeben wird Speicher durch die Funktion FreeMem, deren Aufruf wie folgt lautet:

```
FreeMem(adresse,groesse);
```

"adresse" ist der Returnwert der Funktion AllocMem, "groesse" gibt in Bytes an, wieviel Speicher freigegeben werden soll. Bei beiden Funktionen – AllocMem und FreeMem – rundet das System automatisch bis zum nächsten Vielfachen der Konstante "MEM_BLOCKSIZE". Der Wert dieser Konstante ist in der Include-Datei "exec/memory.h", die sich auf Ihrer Compilerdiskette befindet, definiert.

Ein Programm zur Speicherreservierung

Nachfolgend finden Sie ein Beispielprogramm, das die Funktionen AllocMem und FreeMem verwendet. Beachten Sie bitte, daß Sie immer die gleiche Menge Speicher freigeben, die Sie auch reserviert haben.

Hier das Programm:

```
#include "exec/memory.h"

main()
{
  char *adresse;

  /* 300 Bytes im ChipMem reservieren und mit Nullbytes füllen */
  adresse = AllocMem(300,MEMF_CHIP|MEMF_CLEAR);

  if(adresse) printf("300 Bytes im ChipMem reserviert.\n");
  else printf("Speicherreservierung #1 nicht durchgeführt!\n");

  /* Speicher wieder freigeben */
  FreeMem(adresse,300);
  printf("300 Bytes im ChipMem freigegeben.\n");

  /* 120 Bytes im FastMem reservieren und mit Nullbytes füllen */
  adresse = AllocMem(120,MEMF_FAST|MEMF_CLEAR|MEMF_PUBLIC);

  if(adresse) printf("120 Bytes im FastMem reserviert.\n");
  else printf("Speicherreservierung #2 nicht durchgeführt!\n");

  /* Speicher wieder freigeben */
  FreeMem(adresse,120);
  printf("120 Bytes im FastMem freigegeben.\n");

  /* 12345 Bytes im Speicher reservieren */
  adresse = AllocMem(12345,0);

  if(adresse) printf("12345 Bytes im Speicher reserviert.\n");
  else printf("Speicherreservierung #3 nicht durchgeführt!\n");

  /* Speicher wieder freigeben */
  FreeMem(adresse,12345);
  printf("12345 Bytes im FastMem freigegeben.\n");
}
```

Bei der Speicherreservierung #3 im Beispielprogramm wurde als zweiter Parameter der Wert Null angegeben. Das bedeutet soviel wie: "Reserviere den Speicher entweder im Chip- oder im FastMem". Das System versucht zuerst, FastMem zu belegen, um möglichst viel ChipMem für die Coprozessoren freizuhalten.

Listen

Die von Exec verwalteten Listen bestehen immer aus zwei Komponenten (Datenstrukturen): die "List Structure", die die Kopfzeile einer Liste enthält, und die "Node Structure", die einen Eintrag in der Liste repräsentiert.

Die "Kopfzeile" bildet die Basis einer Liste. Die Funktionen, die zum Manipulieren von Listen zur Verfügung stehen, verlangen beim Aufruf immer diese "Kopfzeile" als Parameter. Anders gesagt: Die Kopfzeile dient zur eindeutigen Bestimmung einer Liste.

Initialisierung der Kopfzeile einer Liste

Bevor Sie mit Listen arbeiten können, müssen Sie zunächst wissen, daß die Kopfzeile das "Herz" einer Liste darstellt. Aus diesem Grund muß die Kopfzeile vor der Verwendung korrekt initialisiert werden.

Der Amiga stellt Ihnen hierfür eine Funktion zur Verfügung: NewList. Mit dieser Funktion wird die Arbeit mit Listen um ein Vielfaches leichter. Der Aufruf der Funktion lautet:

```
NewList(startadresse_der_kopfzeile);
```

Der Parameter gibt die Startadresse im Speicher an, ab der die Kopfzeile abgelegt werden soll. Eine Reservierung oder Initialisierung des Speichers ist nicht notwendig; das erledigt NewList alles für Sie.

Signifikanz der "List Nodes"

Ein "List Node" ist nichts weiter als ein Eintrag in einer Liste. Ein Node ist immer eine Datenstruktur, die Informationen über den betreffenden Eintrag enthält.

Die Arbeit mit diesen Nodes gestaltet sich recht einfach, da sie fast keine Initialisierung benötigen, um sie durch die Listen-Funktionen des Systems ansprechen zu können. Sie können z.B. die Funktion AddHead oder AddTail verwenden (um einen Eintrag am Anfang bzw. am Ende einer Liste vorzunehmen), ohne zu wissen, was intern mit diesen Einträgen geschieht. Die Abb. 3.1 veranschaulicht den Aufbau einer Liste mit Kopfzeile, Einträgen und den Komponenten eines Eintrags.

Beachten Sie bitte, daß die Pfeile den Weg der Verknüpfung aller Komponenten zu einer Liste darstellen. Exec verwaltet sowohl einen Zeiger, der auf den nächsten Eintrag einer Liste zeigt, als auch einen, der auf den vorhergehenden Eintrag zeigt. Das Durchsuchen einer Liste nach einem bestimmten Eintrag wird somit sehr viel einfacher.

Die Datenstruktur eines Nodes enthält neben den Komponenten, die in der Abbildung aufgeführt sind, noch zwei andere Felder: den Namen und die Priorität des Eintrags.

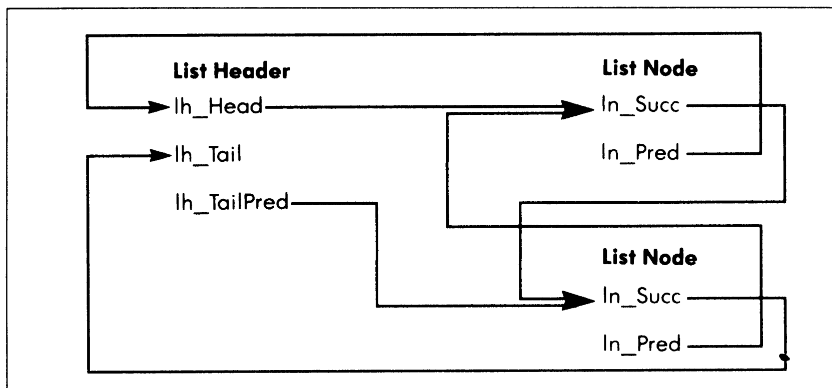


Abb.3.1: Eine verknüpfte Liste

Das erste Feld (der Name) kann einen Zeiger auf eine Zeichenkette beinhalten, die den Namen des Nodes enthält. Einige der Systemroutinen bieten die Möglichkeit, Dinge nach ihrem Namen suchen zu lassen, etwa FindTask oder FindPort.

Das zweite Feld (die Priorität) kann dazu verwendet werden, bestimmten Systemroutinen zu sagen, wo ein Eintrag in einer Liste erfolgen soll. Dies ist möglich, da Nodes mit höherer Priorität in einer Liste immer über Nodes mit kleinerer Priorität stehen. Funktionen, die das Prioritätsfeld verwenden, sind z.B. AddTask und AddPort.

Der gültige Bereich für die Angabe der Priorität liegt von –128 bis +127. Meistens werden Sie jedoch eine Priorität von 0 verwenden.

Routinen, die Listen manipulieren

In Tabelle 3.1 sind alle Funktionen aufgelistet, mit denen Sie Listen direkt manipulieren können. Die notwendigen Parameter für diese Funktionen sind:

- node Ein Zeiger auf einen Node
- list Ein Zeiger auf die Kopfzeile einer Liste
- listnode Ein Zeiger auf einen Node
- name Ein Zeiger auf eine Zeichenkette
- start Ein Zeiger auf die Kopfzeile einer Liste oder einen List Node

Beim Durchsuchen einer Liste beginnt die Suche immer bei dem Eintrag, der nach "start" in der Liste steht, damit der aktuelle Eintrag nicht mitverwendet wird. Um die komplette Liste zu durchsuchen, sollten Sie "start" immer auf die Kopfzeile einer Liste zeigen lassen.

Mit Hilfe der unten aufgeführten Funktionen ist es leicht, eigene Listen zu erzeugen und diese dann zu verwalten oder zu manipulieren. Sie müssen nur eine Kopfzeile mit der Funktion NewList erzeugen und können dann in dieser neuen Liste Einträge vornehmen oder löschen.

Funktion	Zweck
AddHead(list,node);	Nimmt einen Eintrag am Listenanfang vor
AddTail(list,node);	Nimmt einen Eintrag am Listenende vor
Enqueue(list,node);	Nimmt einen Eintrag gemäß der Priorität vor
Insert(list,node,listnode);	Fügt einen Node in eine Liste über einem bereits vorhandenen Node ein
Remove(node);	Löscht einen Eintrag aus einer Liste
node = RemHead(list);	Löscht einen Eintrag und liefert als Returnwert die Adresse des ersten Eintrags einer Liste bzw. 0, falls die Liste leer ist
node = RemTail(list);	Löscht den letzten Eintrag und liefert als Returnwert die Adresse des letzten Eintrags in einer Liste bzw. 0, falls die Liste leer ist
node = FindName(start,name);	Durchsucht eine Liste ab <start> nach einem Node mit Namen <name>

Tab. 3.1: Funktionen zum Manipulieren von Listen

Beispielprogramm: Anwendung von Listen-Funktionen

Das Beispielprogramm in Listing 3.1 zeigt eine Anwendung von einigen der oben besprochenen Funktionen. Es nimmt Einträge in einer Liste nach dem "LIFO"-Verfahren (Last In – First Out) vor.

Beachten Sie bitte, daß es egal ist, wo sich der List Node in Ihrer Datenstruktur befindet, solange seine Startadresse korrekt an Funktionen übergeben wird.

Bei den meisten Datenstrukturen, die vom System verwendet werden, ist der List Node immer der erste Eintrag in einer solchen Struktur (z.B. bei "struct MsgPort", "struct Message" oder "struct Task").

Dies gilt jedoch nicht für die Strukturen von Prozessen ("struct Process"), Bibliotheken ("struct Library") und virtuellen Geräten ("struct Device"). Hier befindet sich der List Node irgendwo innerhalb der betreffenden Datenstruktur (jedoch niemals am Ende).

Der Grund hierfür liegt darin, daß diese Strukturen vom System her in einer ganz bestimmten Art verwaltet werden: Datensätze bzw. ihre Position können nämlich relativ zur Position des List Nodes angegeben werden. Welche Vorteile dies bringt, sehen Sie später in diesem Kapitel.

```
/* list.example.c */

#include "exec/types.h"
#include "exec/lists.h"

struct MyListItem
{
  struct Node n;
  char *data;
};

main()
{
  struct MyListItem mli[3],*mynode;
  struct List MyListHead;
  int i;

  NewList(&MyListHead);    /* Kopfzeile der Liste initialisieren */

  mli[0].data = "Erster Eintrag";
  mli[1].data = "Zweiter Eintrag";
  mli[2].data = "Dritter Eintrag";

  for(i=0;i<3;AddTail(&MyListHead,&mli[i++]));

  for(i=0;i<3;i++)
        {
          mynode = (struct MyListItem *)RemTail(&MyListHead);
          printf("Eintrag %d <%s> gelöscht!\n",i+1,mynode->data);
        }
}   /* Ende des Beispielprogramms */
```

Listing 3.1: Beispiel zur Anwendung der Listen-Funktionen

Signale

Anhand von "Signalen" entscheidet Exec, was mit dem momentan laufenden Task geschehen soll. Zuvor müssen Sie Exec jedoch mitteilen, auf welche Signale Ihr Task reagieren soll und auf welche nicht.

Wird ein Signal erzeugt, dann wird ein Bit in einer Variablen vom Typ LONG (32 Bit) gesetzt. Jeder Task beinhaltet eine solche Variable in seinem "Task Control Block". Um zu entscheiden, welche Art Signal erzeugt wurde, können Sie natürlich den Control Block überprüfen, um festzustellen, welches Bit gesetzt wurde.

Diese Methode ist jedoch sehr umständlich und schwer zu handhaben. Exec liefert daher für jedes Signal einen bestimmten Returnwert, dessen Überprüfung sich ungleich einfacher gestaltet, wenn Sie herausfinden wollen, welche Art Signal generiert wurde.

Was geschehen kann, wenn ein Signal erzeugt wurde

Wenn ein Signal generiert wurde, dann sind verschiedene Verhaltensweisen des Task denkbar:

- Es geschieht nichts, wenn ein Task nicht darauf vorbereitet wurde, Signale zu erkennen und entsprechend der Art des Signals weiterzuarbeiten.

- "Schläft" ein Task, d.h. wartet er auf eine Eingabe (ein Signal), dann wird er von Exec "aufgeweckt", falls das Signal von der Art ist, die der Task erwartet.

- Ein Task kann beim Auftreten eines Signals von sich aus eine Unterscheidung der Signalart treffen. Dieser "Exception Mode" wird in diesem Buch nicht behandelt.

Signifikanz von Signalen beim Multitasking

Signale werden meist dann an einen Task gesendet, wenn auf einer virtuellen Kommunikationsleitung (Message Port) des Tasks eine Nachricht (Message) ankommt. Die oben erwähnte Variable vom Typ LONG, die Bestandteil des Task Control Blocks ist, weist dem Anwender und Exec je 16 Bits für Signale zu.

Da Tasks meistens mehr als eine virtuelle Kommunikationsleitung verwalten, kann man diese Variable überprüfen, um festzustellen, an welchem Message Port ein Signal angekommen ist.

Sollten Sie für Ihren Task mehr als 16 Message Ports benötigen, dann können Sie die Bits der Signalvariablen mit anderen Message Ports teilen. Allerdings ist es dann etwas schwieriger, die Quelle eines Signals ausfindig zu machen, da Sie alle vorhandenen Ports daraufhin überprüfen müssen, ob an einem von ihnen ein Signal angekommen ist.

Reservierung eines Signal-Bits

Mit der Funktion AllocSignal kann Ihr Task ein Signal-Bit für sich reservieren lassen. Der Aufruf der Funktion lautet:

```
bitnummer = AllocSignal(nummer);
```

Der Parameter der Funktion ist entweder ein Wert von 16 bis 31, falls Sie ein bestimmtes Bit reservieren wollen, oder –1, wenn die Bitposition egal ist. Der Returnwert "bitnummer" enthält die Nummer des reservierten Bits bzw. –1, falls eine Reservierung nicht vorgenommen werden konnte.

Sie müssen daher zuerst überprüfen, ob AllocSignal einen gültigen Wert geliefert hat, bevor Sie das Bit verwenden. Der Returnwert kann z.B. bei der Einrichtung von Message Ports verwendet werden. Sie legen so fest, welches Bit ("mp_SigBit") gesetzt werden soll, wenn am Message Port eine Nachricht ("Message") anliegt.

Nachfolgend finden Sie noch ein Beispiel zur Verwendung der Funktion AllocSignal.

```
int signalbit;
signalbit = AllocSignal(-1);  /* Irgendein verfügbares Bit reservieren */
if(signalbit == -1) fehler(); /* Prüfen des Fehlers falls erforderlich */
```

Verwendung von Signal-Bits beim Multitasking

Damit das Multitasking auf dem Amiga optimal genutzt wird, vermeidet Exec zeitintensive Warteschleifen. Wenn Sie z.B eine Funktion langsamer abarbeiten wollen, dann sollten Sie einen der Timer verwenden und Ihren Task so lange "schlafen" lassen, bis ein Signal vom Timer generiert wird. Wenn Sie eine Eingabe erwarten, dann sollten Sie es vermeiden, in einer Schleife die Signale oder Nachrichten abzuwarten, die für Ihren Task von Interesse sind.

Diese Art der Programmierung kann alle anderen Tasks unnötig verlangsamen, da Schleifen dieser Art viel Rechenzeit in Anspruch nehmen. Sie sollten vielmehr die Funktion Wait verwenden, wenn Ihr Task auf ein bestimmtes Signal oder eine Nachricht wartet. Sie übergeben der Funktion einfach die Signalbits, auf die Ihr Task warten soll. Exec "merkt" sich diese Bitmaske und "weckt" Ihren Task auf, wenn ein Signal auftaucht, auf das Ihr Programm wartet. Auf diese Art und Weise können andere Tasks ungehindert weiterarbeiten, da keine zeitintensive Schleife verwendet wird.

Damit Sie wissen, welche Signal-Bits gesetzt wurden, liefert Wait diese Bitmaske als Returnwert, wenn ein oder mehrere Bits gesetzt wurden. Durch die Analyse dieser Bitmaske können Sie entscheiden, was Ihr Task als nächstes tun soll. Ein Aufruf der Funktion Wait hat die Form:

```
gesetzte_bits = Wait(bitmaske);
```

Der Parameter der Funktion "bitmaske" ist eine logische ODER-Verknüpfung aller Signale, auf die gewartet werden soll. Wird mindestens eins der von Ihnen benötigten Signale generiert, dann wird die Wait-Funktion beendet, und Sie können anhand des Returnwertes feststellen, welche Signale erzeugt wurden.

Beachten Sie bitte, daß Sie alle Bits, die Ihnen als gesetzt angezeigt werden, nach dem Beenden der Wait-Funktion "quittieren" müssen. D.h. Sie teilen Exec auf diese Art und Weise mit, daß Ihr Task die gesetzten Bits zur Kenntnis genommen hat.

Dies ist notwendig, da nur zu dem Zeitpunkt, an dem die Wait-Funktion verlassen wird, die gesetzten Bits an Ihr Programm übergeben werden. Der Returnwert wird danach sofort wieder gelöscht. Warten Sie also z.B. auf zwei Signale, dann kann es vorkommen, daß beide zur gleichen Zeit gesetzt werden. Aus diesem Grund sollten Sie die ODER-Verknüpfung in der Wait-Funktion verwenden, da Sie so wirklich jedes gesetzte Bit quittieren können.

Verwenden Sie hingegen zwei Wait-Funktionen hintereinander, dann kann sich Ihr Task "aufhängen", wenn das zweite Signal im Zeitraum nach dem Verlassen der ersten Wait-Funktion gesetzt wird, da ja, wie oben erwähnt, die Bitmaske nach dem Verlassen der Funktion gelöscht wird; u.U. wartet Ihr Task dann ewig auf ein Signal-Bit.

Direktes Setzen eines Signal-Bits

Exec stellt Ihnen eine Funktion zur Verfügung, mit der Sie Signal-Bits "manuell" setzen können: SetSignal.

Normalerweise werden Signal-Bits automatisch vom System gesetzt, wenn z.B. eine Nachricht (Message) an einem "Message Port" anliegt. Wie dem auch sei, Sie können die Funktion dazu verwenden, um einem anderen Task mitzuteilen, daß Ihr Programm z.B. mit der Initialisierung eines Speicherbereiches fertig ist und daß in diesem jetzt Daten für ihn bereit stehen. Der Aufruf der Funktion lautet:

```
SetSignal(task,bitmaske);
```

Der Parameter "task" ist ein Zeiger auf die Struktur des Task Control Blocks des Tasks, der angesprochen werden soll. "bitmaske" enthält die Signal-Bits, die gesetzt werden sollen.

Verwendung von mehreren Signal-Bits

Das Listing 3.2 zeigt ein Beispielprogramm, in dem das Arbeiten mit Signal-Bits veranschaulicht wird. Beachten Sie bitte, daß die Funktion AllocSignal einen Integerwert liefert.

Um festzustellen, welches Bit gesetzt wurde, muß dieser Wert konvertiert werden; dies geschieht, indem man eine Verschiebung der Bits nach links vornimmt. Wollen Sie auf verschiedene Signal-Bits warten, dann werden diese einfach in einer 32-Bit-Maske durch ein logisches ODER verknüpft.

Durch eine UND-Verknüpfung können Sie feststellen, ob Signal-Bits bereits gesetzt wurden. Diese Verknüpfung bezieht sich auf die von Ihnen definierte

```
#define TASTATUR_SIGNAL    (1<<sigtaste)
#define SERIELLES_SIGNAL   (1<<sigser)

int sigtaste,sigser,bitmaske;

sigtaste = AllocSignal(-1);
if(sigtaste == -1) Fehler();

sigser = AllocSignal(-1);
if(sigser == -1) Fehler();

/* An dieser Stelle sollten Sie die Module zur
   Initialisierung der benötigten Datenstrukturen
   einsetzen (Schnittstelle, Tastatur, Message Ports
   usw.). */

/* Warten, daß ein Signal-Bit der gewünschten Art gesetzt
   wird */

bitmaske = Wait(TASTATUR_SIGNAL | SERIELLES_SIGNAL);

if(bitmaske & TASTATUR_SIGNAL)  /* Taste wurde gedrückt */
   {
     /* Weitere Module */
   }

if(bitmaske & SERIELLES_SIGNAL)
   {
     /* Weitere Module */
   }
```

Abb. 3.2: Abfrage mehrerer Signal-Bits

Bitmaske in der Wait-Funktion. Im folgenden Beispielprogramm wird auf zwei Signale gewartet; es wird jeweils ein Signal-Bit gesetzt, wenn Sie entweder auf der Tastatur eine Taste drücken oder ein Zeichen über die serielle Schnittstelle empfangen wird.

Das Beispielprogramm stellt ein Fragment dar – es fehlen jeweils die Initialisierungen für die Ports und weiterführende Programmteile, die nach dem Setzen eines Signal-Bits den weiteren Programmablauf übernehmen. Wenn Sie hier eigene Module einbinden, dann müssen Sie sicherstellen, daß Ihr Task auf jedes gesetzte Bit entsprechend reagiert.

Weiterhin dürfen Sie nicht vergessen, jedes gesetzte Bit zu "quittieren", d.h. Sie müssen Exec mitteilen, daß Ihr Programm ein gesetztes Bit erkannt hat. Dies ist erforderlich, da Sie sonst ein Signal-Bit "übersehen" könnten; Ihr Task wartet dann im ungünstigsten Fall bis zum nächsten Warmstart des Rechners.

Bevor Ihr Programm weiterläuft, sollten Sie sicherstellen, daß alle Signal-Bits quittiert wurden bzw. alle "Message Ports" ausgelesen wurden.

Virtuelle Kommunikationsleitungen (Message Ports)

Ein "Message Port" ("MsgPort") ist eine Datenstruktur, an die ein Task eine Nachricht ("Message") schicken kann. Eine solche Nachricht ist ebenfalls eine Datenstruktur (innerhalb des Speicherbereiches des Tasks), die Informationen beinhaltet und von anderen Tasks ausgelesen werden kann.

Im allgemeinen gehört ein Message Port immer zu einem bestimmten Task, der beim Auftauchen einer Nachricht an diesem Port dann entsprechend reagieren kann. Die möglichen Nachrichtenarten, die an einem Message Port ankommen können, sind in der Datenstruktur des Ports ("MsgPort") im Feld "mp_Flags" definiert:

PA_IGNORE Ankommende Nachrichten werden ignoriert.

PA_SIGNAL Eine ankommende Nachricht setzt ein Signal-Bit, auf das der Task, für den der Message Port generiert wurde, entsprechend reagieren kann.

PA_SOFTINT Liegt eine Nachricht vor, dann wird für den betreffenden Task ein Software-Interrupt ausgeführt.

Wird eine Nachricht ignoriert, dann fügt Exec sie an die zum Port gehörende Liste an. Gelöscht werden kann eine solche Nachricht durch die Funktion GetMsg(msgport).

Werden beim Ankommen einer Nachricht die Signal-Bits beeinflußt, dann wird der Task, wenn er auf ein bestimmtes Signal wartet, von Exec "aufgeweckt" und abgearbeitet.

Wird ein Software-Interrupt ausgeführt, dann stoppt der Task für eine kurze Zeitspanne seine momentane Arbeit und führt die System-Interruptroutinen aus. Diese Art der Nachrichtenbehandlung ist z.B. erforderlich, wenn in einem Terminalprogramm die Zeichen von der seriellen Schnittstelle ausgelesen werden müssen, obwohl der Anwender noch Zeichen über die Tastatur eingibt. Würde man hier ohne Interrupt arbeiten, dann könnten Zeichen von der Schnittstelle "verloren" gehen.

Erstellen und Löschen einer Kommunikationsleitung

Um eine virtuelle Kommunikationsleitung (Message Port) für einen Task einrichten zu können, stellt der Amiga die Funktion CreatePort zur Verfügung. Ein Aufruf dieser Funktion reserviert den nötigen Speicherplatz und initialisiert Teile der MsgPort-Datenstruktur.

Die Datenstruktur eines Message Ports beinhaltet eine Kopfzeile (List Header), an die Nachrichten (Messages) angehängt werden können, z.B. bei einem Aufruf der Funktion PutMsg. CreatePort verwendet zur Initialisierung dieser Kopfzeile die bereits behandelte Funktion NewList.

CreatePort reserviert beim Aufruf Speicherplatz und ein Signal-Bit. Kann aus irgendwelchen Gründen kein Speicher reserviert werden oder ist kein Signal-Bit mehr verfügbar, dann liefert CreatePort Null als Returnwert. Bevor Sie also mit dem Message Port arbeiten, sollten Sie zunächst feststellen, ob er erstellt werden konnte. Zum Löschen eines Message Ports (Freigabe des Speichers und des Signal-Bits) wird die Funktion DeletePort verwendet.

Das "mp_Flags"-Feld in der Datenstruktur des Message Ports wird beim Aufruf von CreatePort auf "PA_SIGNAL" (siehe oben) gesetzt, und der Task, der einen Message Port anfordert, wird zum "Eigentümer" des Ports, indem Exec den Wert "mp_Task" mit dem Zeiger auf den entsprechenden Task initialisiert. Kommt eine Nachricht an diesem Message Port an, dann werden die Signal-Bits des Tasks, der diesen Port generiert hat, beeinflußt; der Task kann dann entsprechend reagieren.

Der Aufruf der Funktion CreatePort lautet:

```
MeinPort = CreatePort(name,prioritaet);
```

Der Parameter "name" ist ein Zeiger auf eine Zeichenkette, die den Namen des Message Ports enthält. "prioritaet" ist der Wert, der im Prioritätsfeld des Message Ports eingesetzt wird.

Ist "name" kein Leerstring, dann wird der neu generierte Port in die vom System verwaltete Message-Port-Liste eingetragen (durch die Funktion AddPort). Andere Tasks können diese Liste mit der Funktion FindPort nach "name" durchsuchen und so ebenfalls diesen Port verwenden.

Verwaltet Ihr Task nur einen Message Port, dann können Sie als Namen einen Leerstring ("") angeben. AddPort wird dann nicht ausgeführt, wodurch Sie exklusiven Zugriff auf den Port haben.

Der Parameter "prioritaet" dient dazu, die Message Ports innerhalb der Liste zu ordnen. Ports mit niedriger Priorität stehen am Ende der Liste; solche mit hoher am Anfang. Existieren zwei gleichnamige Ports, dann liefert ein Aufruf der Funktion FindPort immer den Message Port mit der höheren Priorität.

Gelöscht wird ein Message Port mit der Funktion DeletePort, die folgendermaßen aufgerufen wird:

```
DeletePort(MeinPort);
```

Der Parameter "MeinPort" ist der Zeiger, den die Funktion CreatePort nach ihrem Aufruf liefert.

Da die Kopfzeile der Port-Datenstruktur eine Liste aller registrierten Nachrichten enthält, sollten Sie sicher gehen, daß Sie vor dem Löschen eines Ports diese Liste mit der Funktion ReplyMsg komplett ausgelesen haben. Tun Sie dies nicht, dann wird z.B. ein anderer Task gestoppt, weil er darauf wartet, daß Sie auf eine Nachricht "antworten".

Hinzufügung und Wegnahme einer Kommunikationsleitung

Wenn Sie die Datenstruktur eines Message Ports initialisiert haben, dann können Sie diesen Port mit der Funktion AddPort in die Liste aller dem System bekannten Message Ports einfügen. Wollen Sie einen Port aus der Liste entfernen, dann verwenden Sie die Funktion RemPort.

Hier der Aufruf der Funktionen:

```
AddPort(MeinPort);

RemPort(MeinPort);
```

In beiden Fällen ist "MeinPort" ein Zeiger auf eine initialisierte Message Port-Datenstruktur. Das wichtigste Feld, das initialisiert werden muß, ist das Prioritätsfeld, damit Exec weiß, an welche Stelle in der Liste der Message Port eingefügt werden soll.

Weiterhin sollten Sie jedem Message Port einen Namen geben, damit Sie ihn jederzeit mit der Funktion FindPort auffinden können. Dient der Port zum Empfang von Nachrichten, dann muß die Kopfzeile der Datenstruktur ebenfalls korrekt initialisiert werden.

Im allgemeinen werden die beiden oben genannten Funktionen selten verwendet, da CreatePort und DeletePort einfacher zu handhaben sind. Der Grund hierfür ist, daß CreatePort die Initialisierung aller notwendigen Datenfelder übernimmt; gleiches gilt für DeletePort. Sie brauchen sich also keine Gedanken über Datenstrukturen und deren Initialisierung zu machen, wenn Sie CreatePort oder DeletePort verwenden.

Das Auffinden einer Kommunikationsleitung

Um einen Message Port in der Liste aller dem System bekannten Message Ports zu finden, können Sie die Funktion FindPort verwenden, die wie folgt aufgerufen wird:

```
Port = FindPort(name);
```

Der Parameter "name" ist ein Zeiger auf eine Zeichenkette, die den Namen des Ports enthält, der gesucht werden soll. Wird als Returnwert Null geliefert, dann wurde kein Port mit Namen "name" in der Liste gefunden.

Existiert mehr als ein Message Port mit Namen "name", dann liefert FindPort als Returnwert einen Zeiger auf den Message Port, der von diesen Ports die höchste Priorität hat. Die Priorität wird im Variablenfeld "ln_Pri", das zum Feld "mp_Node" gehört, der "MsgPort"-Datenstruktur abgelegt.

Wenn Sie wissen, daß mehrere Ports mit gleichem Namen existieren, dann können Sie die Funktion FindName verwenden, damit Sie Ports mit niederer Priorität ansprechen können. Ein Aufruf sieht dann z.B. so aus:

```
struct MsgPort    *mp,*mp2;

mp = FindPort("MeinPort");   /* Ersten Port mit Namen "MeinPort" suchen */

if(mp)    /* Nächsten Port mit Namen "MeinPort" suchen */
   {
     mp2 = FindName(mp,"MeinPort");
     if(mp2) printf("Noch'n Port gefunden!\n");
   }
```

Beispiele zur Verwendung einer Kommunikationsleitung

Listing 3.3 zeigt, wie man ein Message Port generiert, Signal-Bits verwendet und einen Message Port löscht.

```
struct MsgPort    *mp,*gefundener_port; /* Zeiger auf Message Ports */

int bitmaske,wartemaske;

mp = CreatePort("MeinPort",0); /* "MeinPort" mit Priorität 0
                                  einrichten */

bitmaske = (1<<mp->mp_SigBit); /* Bitmaske formen */

wartemaske = Wait(bitmaske);   /* Auf Nachricht warten */

/* Die folgende Zeile zeigt, wie ein anderer Task den Message Port
   finden kann, um diesem Task Nachrichten zu senden. In unserem Fall
   muß der Wert <gefundener_port> gleich dem Wert von <mp> sein,
   ansonsten wurde die Funktion nicht korrekt ausgeführt. */

gefundener_port = FindPort("MeinPort");

if(!gefundener_port)

printf("Port 'MeinPort' nicht gefunden!\n");
DeletePort(mp);
```

Listing 3.3: Programmfragmente zur Arbeit mit Message Ports

Erkennen von Signalen auf einer Kommunikationsleitung

Wartet ein Task darauf, daß ein Signal-Bit gesetzt wird (durch eine Nachricht an einem Message Port), dann kann es passieren, daß beim wiederholten Auftauchen einer Nachricht gleicher Art versucht wird, dieses schon gesetzte Bit nochmals zu setzen. Dies ist dann der Fall, wenn der Task ein gesetztes Signal-Bit nicht direkt quittiert.

Die neue Nachricht geht nicht verloren; sie wird in die zum Message Port gehörende Liste eingetragen. Es ist vom System her nicht möglich, auf eine Nachricht zu warten und diese dann abzuarbeiten, um danach auf die nächste Nachricht zu warten. Der Grund hierfür ist, daß pro Message Port nur ein Signal-Bit zur Verfügung steht, aber mehrere Nachrichten gleichzeitig an diesem Port ankommen können.

Daher muß nach dem Setzen eines Signal-Bits jede angekommene Nachricht ausgelesen werden, bevor erneut die Funktion Wait (o.ä.) aufgerufen wird, da bereits vorhandene Nachrichten das Signal-Bit wieder löschen.

Im ungünstigsten Fall "hängt sich der Task auf", d.h. er wartet auf eine Nachricht, die bereits angekommen ist. Also: Nachrichten werden zwar in der Liste vermerkt, müssen aber nach dem Setzen eines Signal-Bits komplett ausgelesen werden.

Nachrichten (Messages)

Eine "Nachricht" ist eigentlich nichts weiter als ein Speicherbereich, der dazu verwendet wird, Daten von einem Task zum anderen zu transferieren. Der Speicherplatz, der hierfür verwendet wird, gehört zu dem Task, der eine Nachricht "verschickt".

Wird eine Nachricht an einen Message Port geschickt, dann wird sie in die vom Message Port verwaltete Nachrichten-Liste eingetragen. Exec kopiert die Nachricht nicht; es werden lediglich einige Vektoren verwendet, die dem Task, der die Nachricht bekommt, sagen, wo der Speicherbereich liegt, in dem die Nachricht abgelegt wurde.

Wozu benötigt man Nachrichten?

Das Versenden von Nachrichten an andere Tasks ist eine der Grundfunktionen bei Ein-/Ausgabeoperationen auf dem Amiga. So existieren z.B. Tasks, die die Tastatur, den Mausport, die Schnittstellen usw. verwalten.

Damit andere Tasks Ein-/Ausgabeoperationen durchführen können, muß es eine Möglichkeit geben, mit den Tasks zu kommunizieren, die eine bestimmte Hardware-Einheit (z.B. die Tastatur) verwalten. Diese Kommunikation spielt sich über die Message Ports und die Nachrichten ab.

Wie bereits erwähnt, ist eine Nachricht ein Speicherbereich, in dem Daten für einen anderen Task abgelegt werden können. Bei der Ein-/Ausgabe wird ein solcher Speicherbereich initialisiert; er enthält danach eine Nachricht, die von anderen Tasks ausgelesen werden kann bzw. die für den Task bestimmt ist, der z.B. die serielle Schnittstelle verwaltet.

"Schläft" ein Task, d.h. wartet er auf eine bestimmte Nachricht, dann können während dieser Zeit andere Tasks vom System abgearbeitet werden. In dem Moment, wo die Nachricht ankommt, auf die der Task wartet, wird er von Exec "aufgeweckt" und abgearbeitet.

Verwendet Ihr Task Ein-/Ausgaberoutinen, dann ist es meistens so, daß Exec ihn "einfriert", solange vom angesprochenen Ein-/Ausgabe-Task keine Nachricht verschickt wird. Ist die Ein-/Ausgabe beendet, dann wird Ihr Task aufgeweckt – durch den Empfang einer Nachricht vom Ein-/Ausgabe-Task. Diese Nachricht enthält entweder die Mitteilung, daß Daten bereitstehen oder daß die Ein-/Ausgabeoperation nicht durchgeführt werden konnte. In beiden Fällen müssen Sie jedoch die Nachricht quittieren, damit der Ein-/Ausgabe-Task korrekt weiterarbeiten kann, der ja seinerseits auch auf eine Nachricht wartet – nämlich auf die "Antwort" Ihres Tasks.

Der Inhalt einer Nachricht

Die Datenstruktur einer Nachricht sieht wie folgt aus:

```
struct Message
{
  struct Node mn_Node;            /* Ein List Node */
  struct MsgPort *mn_ReplyPort;   /* Zeiger auf einen Message Port */
  int mn_Length;                  /* Länge der Nachricht */
};
```

Hier die Erklärung der Strukturkomponenten:

"mn_Node" ist ein "List Node", der dazu verwendet wird, Nachrichten innerhalb einer Nachrichten-Liste zu verknüpfen. Der Typ dieses Nodes ("mn_Node.ln_Type") wird durch die symbolische Konstante "NT_MESSAGE" definiert, was etwa bedeutet: "Dieser Node ist vom Typ "Message"." Die zweite Komponente ("mn_ReplyPort") ist ein Zeiger auf den Message Port des Tasks, der die Nachricht verschickt.

Quittiert ein Task eine Nachricht mit der Funktion ReplyMsg, dann wird diese Nachricht an den "Absender" zurückgeschickt. Die Variable "mn_ Node. ln_Type" wird vom System dann auf den Wert "NT_REPLYMSG" gesetzt. Der sendende Task verwendet seinen Message Port meistens dazu, verschickte und retournierte Nachrichten zu empfangen. Der sendende Task kann auf diese Weise so lange "schlafen", bis Exec ihm mitteilt, daß die Nachricht (die z.B. mit PutMsg verschickt wurde) retourniert wurde.

Ein Task, der auf eine Nachricht wartet (GetMsg), kann seinerseits "schlafen", bis die Nachricht eintrifft. Nach dem "Aufwachen" und dem Auslesen der Nachricht kann er sie dann mit der Funktion ReplyMsg an den Absender zurückschicken.

Die Länge der Nachricht, die in der Variablen "mn_Length" abgelegt wird, wird durch Kopie vom System nicht überprüft. Daher können Sie hier Ihren eigenen Wert eintragen, um die Nachricht zu kennzeichnen.

Signifikanz von Nachrichten

Nachrichten werden durch Verweise auf Datenstrukturen verschickt, nicht durch Kopieren von Speicherblöcken. Wenn Ihr Task eine Nachricht an einen anderen Task verschickt, dann wird u.a. die Anfangsadresse des Speicherbereiches übergeben, in dem die initialisierte Datenstruktur der Nachricht abgelegt ist. Da sich die Datenstruktur immer in dem Speicherbereich befindet, der Ihrem Task zugeteilt wurde, "erlauben" Sie dem Task, der Ihre Nachricht bekommt, temporär den Zugriff auf diesen Speicherbereich.

Der Task, der die Nachricht empfängt, kann den Inhalt der Nachricht in seinen eigenen Speicherbereich kopieren (siehe Kapitel 5) oder seinerseits in Ihrem

Speicherbereich Daten ablegen usw. Sobald ein Task den Inhalt einer Nachricht ausgelesen hat, muß er sie an den Absender mittels ReplyMsg zurücksenden.

Sie sollten Nachrichten, die Ihr Task empfängt, so früh wie möglich zurücksenden. Andernfalls wird der Task, der auf die retournierte Nachricht wartet, unnötig lange gestoppt. Weiterhin belegt Ihr Task beim Auslesen einer Nachricht meistens Speicher des anderen Tasks.

Exec verwaltet nur eine Liste des freien Speicherplatzes; es ist daher Ihre Aufgabe, belegten Speicherplatz vor dem Beenden Ihres Programms wieder freizugeben. Wenn Sie jedoch Nachrichten nicht an den Absender zurücksenden, dann kann der Task, der auf Ihre Antwort (die retournierte Nachricht) wartet, nicht weiterlaufen und folglich auch belegten Speicherplatz nicht freigeben. Das Quittieren von Nachrichten mittels ReplyMsg ist daher ein Muß, wenn das System korrekt arbeiten soll.

Eigene Nachrichten (Custom Messages)

Einige der Datenstrukturen, die vom System benutzt werden, sind modifizierte Versionen der "Message"-Struktur. Sie werden dazu verwendet, Daten zwischen einem Task und speziellen Ein-/Ausgabe-Tasks zu transferieren.

Die Datenstrukturen "IORequest" und "IOStdReq" sind zwei Beispiele für solche speziellen Versionen der "Message"-Struktur. Eigene Nachrichten werden wie folgt strukturiert:

```
struct MeineEigeneNachricht
{
  struct Message  meine_message;
  int komponente1,komponente2,komponente3;
};
```

Dies ist ein Beispiel einer Standardnachricht mit drei Komponenten vom Typ "INT". Eigene Nachrichten können wie "normale" Nachrichten mit den besprochenen Funktionen verwendet werden.

Funktionen zum Arbeiten mit Nachrichten und Kommunikationsleitungen

Die Tabelle 3.2 enthält die Funktionen, die beim Arbeiten mit Messages (Nachrichten) und Message Ports (virtuelle Kommunikationsleitungen) notwendig sind. Die verwendeten Parameter bedeuten:

msgport	Ein Zeiger auf einen Message Port
msg	Ein Zeiger auf eine Message
priorität	Ein Wert im Bereich von –128 bis +127
name	Ein Zeiger auf eine Zeichenkette

Beispielprogramm: Arbeiten mit Messages und Message Ports

Das Listing 3.4 enthält ein Beispielprogramm, in dem der Austausch von Nachrichten zwischen Message Ports veranschaulicht wird. Ein solcher Austausch vollzieht sich normalerweise zwischen einzelnen Tasks; wir beschränken uns hier jedoch auf einen Task, der mehrere Ports verwaltet. In Kapitel 9 finden Sie weitere Informationen zum Datenaustausch zwischen Tasks.

Beachten Sie bitte, daß das Feld "mn_Node.ln_Name" in diesem Beispiel den Inhalt der Nachricht enthält. Üblicherweise werden größere Datenmengen von einem Task zum anderen geschickt; es geht hier jedoch nur darum, die Anwendung der Funktionen zu veranschaulichen. Die Daten, die mittels Nachrichten verschickt werden, sind normalerweise am Anfang oder Ende der Datenstruktur der Message zu finden.

Bibliotheken

Eine Bibliothek ist ein Zusammenschluß von Funktionen, die in irgendeiner Weise miteinander in Beziehung stehen. Wird eine Funktionsbibliothek in den Rechner geladen, dann können alle Tasks auf die Funktionen zugreifen. Auf

Funktion	**Zweck**
AddPort(msgport);	fügt einen zuvor initialisierten Message Port in die Message Port-Liste des Systems ein.
RemPort(msgport);	löscht einen Eintrag in der Message Port-Liste.
FindPort(name);	sucht den ersten Port mit Namen "name" in der Message Port-Liste.
msg = WaitPort(msgport);	wartet auf eine Nachricht am Message Port. Der zugehörige Task "schläft" bis zum Eintreffen der Nachricht. Im Gegensatz zur Funktion Wait(1<<msgport->mp_SigBit) werden ankommende Nachrichten nicht gelöscht. WaitPort zeigt immer auf die erste Message in der Liste; gelöscht werden Messages mit der Funktion GetMsg.
PutMsg(msgport,msg);	sendet eine Nachricht an einen Message Port.
msg = GetMsg(msgport);	Liegt eine Nachricht an einem Message Port vor, dann liefert diese Funktion die Startadresse der Nachricht und löscht sie aus der Liste des Message Ports. Ist keine Nachricht vorhanden, dann liefert die Funktion Null als Returnwert.
ReplyMsg(msg);	schickt die Nachricht mittels PutMsg an den Port zurück, der im Feld "ReplyPort" der Nachricht angegeben ist.
msgport = CreatePort	reserviert Speicher für einen Message Port und liefert als Returnwert einen Zeiger auf den Port. Ist "name" ungleich Null, dann wird er mittels AddPort in der Message Port-Liste eingetragen. "priorität" bestimmt, wo der Eintrag vorgenommen wird.
DeletePort(msgport);	löscht einen Message Port. Hat der Port einen Namen, dann wird er aus der Message Port-Liste des Systems gestrichen.

Tab. 3.2: Funktionen zum Arbeiten mit Messages und Message Ports

```
#include "exec/types.h"
#include "exec/ports.h"

main()
{
  struct Message m;    /* Die von uns übergebene Nachricht */
  struct Message *msg; /* Zeiger auf die Nachricht, die wir empfangen */
  struct MsgPort *mp;  /* Zeiger auf einen MessagePort */
  struct Message *rp;  /* Zeiger auf einen ReplyPort */

  mp = CreatePort(0,0);    /* Es ist nicht notwendig, einem Port einen
                              Namen zu geben, falls er später nicht mit
                              FindPort gesucht werden soll. */

  if(!mp) exit(20);  /* Port konnte nicht kreiert werden */

  rp = CreatePort("reply",0);    /* Name: reply, Priotität: 0 */

  if(!rp)
     {
       DeletePort(mp);
       exit(0);
      }

 m.mn_Node.ln_Name = "Moin!\n";
 m.mn_ReplyPort = rp;    /* ReplyPort definieren */
 m.mn_Length = 0;   /* Länge in diesem Beispiel nicht erforderlich */

  PutMsg(mp,&m);    /* Nachricht versenden */
  WaitPort(mp);    /* Und auf die Bestätigung warten */

  /* Liegt bereits eine Nachricht am Port an, dann geht unser Task nicht
     "schlafen". Wir wissen jedoch, daß tatsächlich eine Nachricht
     vorhanden ist... */

  while(msg = GetMsg(mp))
  {
    /* MessagePort "entleeren" */

    printf("Der Inhalt der Nachricht lautet: %s\n",msg->mn_Node.ln_Name);
    ReplyMsg(msg);    /* Nachricht bestätigen */
   }

  WaitPort(rp);    /* Auf eine Nachricht am ReplyPort warten */

  while(msg = GetMsg(rp))
      printf("Nachricht vom ReplyPort: %s\n", msg->mn_Node.ln_Name);
  DeletePort(mp);
  DeletePort(rp);
}
```

Listing 3.4: Verschicken von Nachrichten zwischen Message Ports

diese Weise können Programme kleiner gehalten werden, da die Bibliotheken bereits viele Funktionen zur Verfügung stellen, die man sonst zusätzlich programmieren müßte. Die Exec-Bibliothek z.B. beinhaltet Exec-spezifische Funktionen, die zum Arbeiten mit Listen, Tasks, Ein-/Ausgabe usw. gebraucht werden. Die DOS-Bibliothek beinhaltet die Funktionen, die in Kapitel 2 behandelt wurden, die Grafik-Bibliothek enthält Funktionen, die die Erstellung und Darstellung von Grafik ermöglichen.

Die Struktur einer Bibliothek

Eine Bibliothek ist eigentlich nichts weiter als eine Datenstruktur, die in die Bibliotheksliste des Systems eingebunden werden kann. Diese Datenstruktur besteht aus einem List Node, einigen Funktionsvektoren (einer Sprungtabelle) und einem Datenfeld, das Informationen über eine Bibliothek enthält. Die Struktur einer Bibliothek ist in Abb. 3.2 dargestellt.

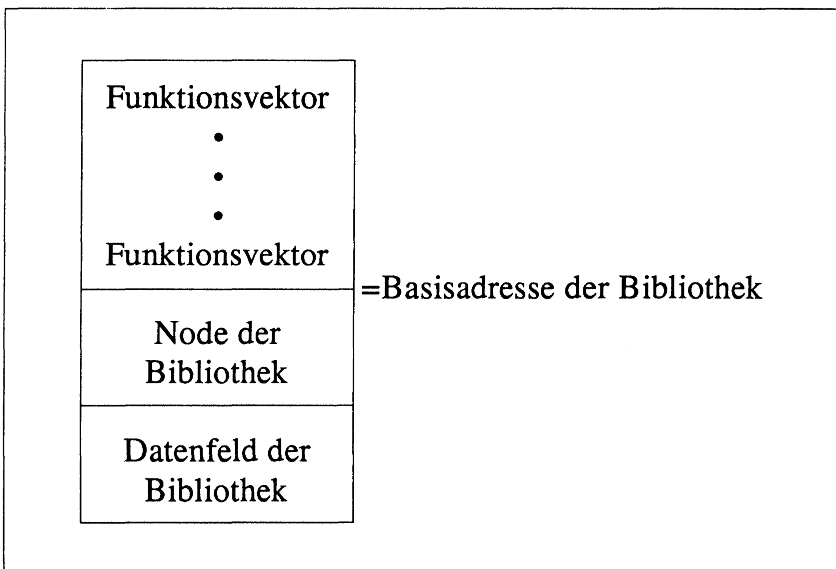


Abb. 3.2: Die Struktur einer Bibliothek

Der "Library Node" beinhaltet Informationen über die Bibliothek, z.B. ihre Größe und Komponenten, die das Arbeiten mit einer Bibliothek ermöglichen. Nachfolgend finden Sie den Inhalt der Library Node-Datenstruktur:

```
struct Library
{
  struct Node  lib_Node;
  UBYTE lib_Flags;
  UBYTE lib_Pad;
  UWORD lib_NegSize;
  UWORD lib_PosSize;
  UWORD lib_Version;
  UWORD lib_Revision;
  APTR lib_ldString;
  ULONG lib_Sum;
  UWORD lib_OpenCnt;
};
```

Die einzelnen Strukturkomponenten sollen im folgenden erklärt werden.

Das "lib_Flags"-Feld

Anhand dieses Feldes prüft Exec, was gerade mit einer Bibliothek passiert. Exec stellt fest, ob ein Sprungvektor geändert wurde, ob gerade die Prüfsumme für die Bibliothek gebildet wird oder ob die Bibliothek nach ihrer Verwendung gelöscht werden soll (falls der Speicherplatz knapp ist).

Dieses Feld ist eigentlich nur für Programmierer interessant, die in 68000-Assembler arbeiten, da die Überprüfung der Bits dieser Feldes oftmals für einen korrekten Programmablauf wichtig ist – besonders, wenn mit den Interrupt-Routinen des Prozessors gearbeitet wird. Arbeiten Sie mit einer Hochsprache, dann erübrigt sich die Prüfung dieser Strukturkomponente.

Nachfolgend finden Sie ein Programmfragment, in dem das Überprüfen des Feldes veranschaulicht wird. Das Fragment könnte z.B. zu einem Task gehören, der gerade von Exec abgearbeitet wird und der den Zugriff auf eine Funktion einer Bibliothek haben muß. Mit diesem Fragment kann er überprüfen, ob die Funktion für einen Aufruf bereit ist.

```
struct Library   *lib;  /* Wird üblicherweise am
                           Programmanfang definiert */

if(lib->lib_Flags & LIB_CHANGED)
   {
     /* Sprungtabelle der Bibliothek wurde geändert */
   }

if(lib->lib_Flags & LIB_SUMMING)
   {
     /* Bibliothek kann noch nicht verwendet werden, da die
        Prüfsumme noch nicht gebildet wurde */
   }

if(lib->lib_Flags & LIB_DELEXP)
   {
     /* Aus Speicherplatzgründen wird die Bibliothek
        geschlossen, sobald der letzte Task beendet wird */
   }

if(lib->lib_Flags & LIB_SUMUSED)
   {
     /* Ein Task hat dieses Flag gesetzt, damit Exec die
        Bibliotheksprüfsumme nicht bilden und/oder prüfen kann */
    }
```

Das "lib_Pad"-Feld

Dieses Feld wird von Exec nicht verwendet. Seine Initialisierung stellt nur sicher, daß nachfolgende Datenfelder immer an geraden Speicheradressen beginnen.

Das "lib_NegSize"-Feld

In diesem Feld wird die Anzahl Bytes abgelegt, die dem "Library Node" vorausgehen. Der Inhalt des Feldes wird normalerweise von Funktionsvektoren gebildet, wobei jeder Vektor aus 6 Bytes zusammengesetzt wird. Die ersten beiden Bytes bilden den Sprungbefehl, die letzten vier das Sprungziel, d.h. die Anfangsadresse einer Funktion im Speicher.

Das "lib_PosSize"-Feld

Dieses Feld gibt Auskunft darüber, wie viele Bytes das Datenfeld enthält, das dem "Library Node" folgt. In diesem Datenfeld werden globale Variablen abgelegt, die von Funktionen der Bibliothek verwendet werden, oder es enthält schon kleinere Funktionen, die über die Sprungtabelle der Bibliothek angesprochen werden können.

Die Felder "lib_Version" und "lib_Revision"

Diese Felder enthalten die Versions- und Revisionsnummer einer Bibliothek. Durch die Verwendung dieser Felder können Sie eigene Funktionsbibliotheken verwenden, die Sie z.B. im Verzeichnis "libs" auf Ihrer Diskette haben.

Beim Einschalten des Computers werden einige Bibliotheken bereits in das System eingebunden. Durch die Verwendung anderer Versionsnummern in einer Bibliothek sagen Sie dem System, daß Ihre eigenen Bibliotheken verwendet werden sollen.

Das "lib_IdString"-Feld

Der Name einer Bibliothek wird in diesem Feld abgelegt, damit sie z.B. beim Arbeiten mit einem "Debugger" (z.B. WACK) identifiziert werden kann. Bibliotheken haben üblicherweise zwei Namen: einen, mit dem sie beim Aufruf der Funktion OpenLibrary angesprochen werden (z.B. "graphics.library" oder "intuition.library"), und einen längeren, der eine eindeutige Identifikation durch das System ermöglicht. Die maximale Größe des "lib_IdString"-Feldes beträgt 255 Bytes.

Das "lib_Sum"-Feld

Exec verwendet dieses Feld zum Ablegen der Prüfsumme der Funktionsvektoren einer Bibliothek. Sobald eine Bibliothek in das System eingebunden wird, überprüft Exec anhand dieses Feldes die Vollständigkeit der Bibliothek. Wird ein Vektor der Sprungtabelle geändert, dann wird automatisch eine neue Prüfsumme gebildet. Dieses Feld wird ausschließlich von Exec genutzt.

Das "lib_OpenCnt"-Feld

Hier verwaltet Exec einen Zähler, der Auskunft darüber gibt, wie oft eine Bibliothek geöffnet wurde. Ist der zur Verfügung stehende Speicherplatz des Systems knapp, dann wird eine Bibliothek automatisch aus dem System genommen ("getilgt"), wenn dieses Feld den Wert Null enthält.

Bei jedem Öffnen einer Bibliothek durch die Funktion OpenLibrary wird der Zähler um eins erhöht; bei jedem Schließen wird er um eins vermindert. Durch das "Tilgen" einer Bibliothek bei Speicherknappheit wird freier Speicherplatz geschaffen, der dann wieder von den Tasks verwendet werden kann.

Öffnen einer Bibliothek

Bevor Sie die Funktionen einer Bibliothek verwenden können, müssen Sie zunächst die betreffende Bibliothek öffnen. Dies geschieht durch einen Aufruf der Funktion OpenLibrary :

```
basisadresse_der_bibliothek = OpenLibrary(bibliotheks_name,version);
```

Der Parameter "version" enthält die Versionsnummer der Bibliothek, die verwendet werden soll. Benötigen Sie eine bestimmte Bibliotheksversion, dann geben Sie hier die Nummer an; Versionsnummer 31 ist z.B. für Bibliotheken der Betriebssystemversion 1.1 reserviert. Ist die Versionsnummer für Sie nicht interessant, dann geben Sie als Wert einfach Null an.

Der Parameter "bibliotheks_name" ist ein Zeiger auf eine Zeichenkette, die den Namen der zu öffnenden Bibliothek enthält. Der Name kann auch direkt angegeben werden, wobei er in Anführungszeichen eingeschlossen wird, z.B. "icon.library".

Der Returnwert der Funktion – "basisadresse_der_bibliothek" – ist ein Zeiger auf die Startadresse der Datenstruktur der Bibliothek. Er zeigt auf das erste Byte des "Library Nodes"; es können also auch Datenblöcke davor oder dahinter abgelegt worden sein. Kann eine Bibliothek nicht geöffnet werden, dann liefert OpenLibrary Null als Returnwert.

Beachten Sie bitte, daß Sie die Variable "basisadresse_der_bibliothek" nicht beliebig benennen dürfen; der Amiga erwartet hier vordefinierte Namen. In Tab. 3.3 finden Sie die Namen aller Bibliotheken, die Namen der Basisadressen und die enthaltenen Funktionen.

Wenn Sie eine Bibliothek öffnen, so sucht der Amiga zuerst im ROM/RAM nach ihr. Ist sie dort nicht vorhanden, so wird im Verzeichnis "libs" der aktuellen Diskette gesucht.

Die Namen und Basisadressen der Bibliotheken

Wie Sie bereits wissen, muß die Basisadresse einer Bibliothek einen bestimmten Namen haben, wenn Sie OpenLibrary aufrufen. Der Grund hierfür ist, daß der Amiga spezielle Programmfragmente beim Kompilieren in Ihr Programm einbindet, die die Verwendung von Funktionen einer Bibliothek ermöglichen.

Die Hauptaufgabe dieser Fragmente ist es, die absolute Basisadresse einer Funktion im Speicher festzustellen. Sie wird aus dem Inhalt der Variablen gebildet, die die Basisadresse der Bibliothek repräsentiert. Das System findet auf diese Weise die Sprungtabelle einer Bibliothek, in der die Startadressen aller Funktionen zu finden sind. Andere Aufgaben dieser Programmteile sind das Zwischenspeichern von Registerinhalten auf dem Stapelspeicher, die Initialisierung der Register mit Ihren Werten und der Aufruf der eigentlichen Funktion.

Wie dies alles genau vom System gemacht wird, ist ausführlich im "Amiga ROM Kernel Manual" beschrieben. Wichtig ist, daß jede Bibliothek eine Basisadresse hat, die einen definierten Namen haben muß, und daß diese Variable einen gültigen Zeigerwert beinhalten muß (ungleich Null), bevor Sie Funktionen aus dieser Bibliothek verwenden.

Das Betriebssystem des Amiga ist dynamisch aufgebaut. Beim Einschalten des Rechners werden einige Bibliotheken im Speicher abgelegt; der Bereich, in dem sie liegen, kann von Fall zu Fall unterschiedlich sein. Gleiches gilt für Bibliotheken, die auf Diskette abgelegt sind und mit OpenLibrary in die Bibliotheksliste des Systems eingebunden werden; der Speicherbereich, in dem sie abgelegt werden, ist immer verschieden.

Bibliotheksname	Name der Basisadresse	Inhalt der Bibliothek
clist.library	ClistBase	Funktionen zum Arbeiten mit Zeichenketten
diskfont.library	DiskfontBase	Funktionen zum Arbeiten mit Zeichensätzen, die sich auf Diskette befinden
exec.library	ExecBase	Alle Exec-Funktionen
dos.library	DosBase	DOS-Funktionen
graphics.library	GfxBase	Grafik-Funktionen
icon.library	IconBase	Funktionen zum Arbeiten mit Workbench-Objekten
intuition.library	IntuitionBase	Funktionen der grafischen Benutzeroberfläche
layers.library	LayersBase	Funktionen zum Arbeiten mit Fenstern und Grafikbereichen
mathffp.library	MathBase	Mathematische Funktionen
mathtrans.library	MathTransBase	Funktionen der höheren Mathematik
mathieeedoubbas.library	MathIeeeDoubBasBase	Funktionen zum Rechnen mit doppelter Genauigkeit (IEEE)
timer.library	TimerBase	Funktionen zum Arbeiten mit den Hardware-Zeitgebern (Timer)
translator.library	TranslatorBase	Funktionen zum Arbeiten mit der Sprachausgabe

Tab. 3.3: Namen, Basisadressen und Inhalte der System-Bibliotheken

Wird eine solche Bibliothek von Diskette geladen, dann ist sie für alle laufenden Tasks verfügbar. Meistens steht kein genügend großer Speicherbereich zur Verfügung, um die Bibliothek ablegen zu können; in einem solchen Fall wird sie an mehreren Stellen im Speicher abgelegt. Exec korrigiert dann die Sprungtabelle der Bibliothek, damit die Funktionen gefunden werden können.

Im Gegensatz zu den Sprungvektoren ändert sich die Basisadresse einer Bibliothek nicht, sobald sie geöffnet wird. Der Zeiger behält seinen Wert bis zum Schließen der Bibliothek.

Sprungvektoren dagegen können nach Belieben geändert werden. Man sollte immer die Sprungtabelle einer Bibliothek verwenden, um eine Funktion aufzurufen, denn wenn man die Werte der Sprungvektoren zwischenspeichert, um so die Zugriffszeit auf Funktionen zu verkürzen, kann es passieren, daß Ihr Task "abstürzt", wenn ein anderer Task einen Vektor der Sprungtabelle geändert hat. Das Betriebssystem enthält eine Funktion, um die Vektoren zu verändern. Beim Verwenden der Sprungtabelle, die ja keine absoluten Speicheradressen enthält, werden sich beim Programmablauf keinerlei Probleme ergeben.

Verwendung der Funktionen einer Bibliothek

Sobald eine Bibliothek erfolgreich geöffnet wurde, können Sie jede in ihr enthaltene Funktion verwenden. Wenn Sie in C programmieren, ist ein Aufruf recht einfach; Sie geben einfach den Funktionsnamen und die notwendigen Parameter an:

```
returnwert = Funktion(parameter1,parameter2);
```

Alles Weitere wird für Sie vom Betriebssystem des Amiga übernommen. Arbeiten Sie in 68000-Assembler, dann müssen Sie vor dem Funktionsaufruf noch einige andere Dinge erledigen. Diese Arbeit wird jedoch durch Makros vereinfacht, die in der Datei "exec/libraries.i" auf Ihrer Diskette zu finden sind.

Vor dem Funktionsaufruf müssen Sie alle Registerinhalte im Stapelspeicher zwischenspeichern, dann initialisieren Sie die notwendigen Register mit den Parametern, die die aufzurufende Funktion benötigt. Im Register A6 muß die Basisadresse der Bibliothek abgelegt worden sein; erst dann kann mit dem Makro "CALL_LIB" die Funktion aufgerufen werden:

```
CALL_LIB _LVOFunktion
```

Sollte der Wert in Register A6 nicht korrekt sein, dann verwenden Sie zum Funktionsaufruf das Makro "LINK_LIB":

```
LINK_LIB _LibraryBase,_LVOFunktion
```

"_LibraryBase" enthält die Basisadresse der Bibliothek, in der der Sprungvektor für die gewünschte Funktion zu finden ist. "_LVOFunktion" enthält einen negativen Wert, der zur Basisadresse addiert wird, um so die absolute Adresse des Sprungvektors für die Funktion zu finden.

Schließen einer Bibliothek

Wenn Sie eine Bibliothek nicht mehr benötigen, dann sollten Sie sie schließen, um dem System so mitzuteilen, daß Ihre Arbeit mit der Bibliothek beendet ist. Existiert kein weiterer Task, der diese Bibliothek verwendet, dann entfernt Exec sie aus dem System, wenn der Speicherplatz knapp wird. Geschlossen wird eine Bibliothek mit der Funktion CloseLibrary, die wie folgt aufgerufen wird:

```
CloseLibrary(basisadresse_der_bibliothek);
```

Der Parameter der Funktion ist der Returnwert, den Sie nach dem Aufruf der Funktion OpenLibrary erhalten.

Programm zum Öffnen, Benutzen und Schließen einer Bibliothek

In Listing 3.5 finden Sie ein Beispielprogramm, das eine Bibliothek öffnet, eine Funktion aus ihr verwendet und sie danach wieder schließt.

Beachten Sie bitte, daß die Bibliotheken "exec.library" und "dos.library" immer geöffnet sind; es ist also nicht notwendig, eine Basisadresse usw. zu verwenden, um eine Funktion aus einer der Bibliotheken zu benutzen. Das Öffnen dieser beiden Bibliotheken erfolgt bereits beim Compilieren eines Programms, ganz egal, welchen Compiler Sie verwenden.

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"

struct IntuitionBase   *IntuitionBase, *OpenLibrary();

main()
{
  if(!(IntuitionBase=OpenLibrary("intuition.library",0)))
       printf("Intuition läßt sich nicht öffnen!\n");

  else
       {
         printf("Intuition geöffnet!\n");
         DisplayBeep(0); /* Eine Funktion der Bibliothek verwenden */
         CloseLibrary(IntuitionBase);  /* Bibliothek schließen */
        }
}
```

Listing 3.5: Öffnen, Verwenden und Schließen einer Bibliothek

Virtuelle Geräte (Devices)

Ein "Device" ist auf dem Amiga eine Datenstruktur, mit deren Hilfe die Komponenten der Hardware (serielle und parallele Schnittstelle, Laufwerke usw.) angesprochen werden können, um Ein-/Ausgabeoperationen durchzuführen. Devices und Bibliotheken stehen in einer engen Beziehung zueinander; viele der Standard-Devices sind aus Funktionen einer Bibliothek aufgebaut. Auf diese Weise kann man sich auch eigene Devices "herstellen", man muß nur auf Funktionen der Bibliotheken zurückgreifen. Genau wie für Bibliotheken, so existieren auch Routinen zum Öffnen, Schließen und "Tilgen" von Devices.

Devices und Bibliotheken haben aber noch mehr gemeinsam. Devices werden auch über Namen angesprochen (trackdisk.device, timer.device, console.device usw.), und sie können sowohl speicherresident als auch auf Diskette vorhanden sein. Soll mit der Funktion OpenDevice ein virtuelles Gerät geöffnet werden, dann sucht Exec zunächst in der Device-Liste des Systems nach ihm. Wird es hier nicht gefunden, dann wird versucht, es aus dem Verzeichnis "devs" der aktuellen Diskette nachzuladen. Ist dies geschehen, dann vermerkt Exec dieses neue "Gerät" in seiner Liste. So können auch andere Tasks darauf zugreifen.

Ein Device besteht weiterhin aus "Einheiten", die in ihrer Gesamtheit ein Device ergeben. Die Einheiten eines Device sind meistens die physikalischen Komponenten des Computers, also die Hardware. Das "trackdisk.device" besteht aus Einheiten, die jeweils ein Laufwerk verwalten. Das "timer.device" hat zwei Einheiten: die beiden Hardware-Zeitgeber des Amiga. Im Abschnitt "Öffnen eines virtuellen Gerätes" finden Sie weitere Informationen zu den Einheiten eines Devices.

Wie bereits erwähnt, haben Bibliotheken vier Standard-Vektoren, die für alle Bibliotheken gleich sind: "OPEN", "CLOSE", "EXPUNGE" und "RESERVED". Devices haben ebenfalls diese vier Vektoren, verfügen jedoch noch über zwei weitere Funktionsvektoren, über die ein Device direkt angesprochen werden kann: "BEGINIO" und "ABORTIO".

Obwohl diese "Einstiegspunkte" für ein Device von den Systemroutinen BeginIO(iorequest) und AbortIO(iorequest) verwendet werden, werden sie in diesem Buch nicht näher behandelt, da ihre Anwendung tiefgreifende Kenntnisse des Systems erfordert. Eine weiterführende Erläuterung dieser Funktionen ist im "Amiga ROM Kernel Manual" enthalten.

Vorbereitung und Durchführung einer Ein-/Ausgabeoperation

Die Kommunikation mit einem Device erfolgt durch die Initialisierung einer Datenstruktur (IORequest Block), die an den Message Port eines Device geschickt wird. Normalerweise verwaltet das System einen eigenen Prozeß für jedes Device, d.h. solange keine Nachrichten am Message Port eines Device anliegen, "schläft" es. Kommt eine Nachricht am Message Port an, dann "wacht" das Device auf und analysiert den Inhalt der Nachricht, der dem Device sagt, welche Art der Ein-/Ausgabe erfolgen soll und wie diese Operation durchzuführen ist.

Der "IORequest Block" enthält die bereits besprochene Datenstruktur einer Nachricht (Message). Durch das Auslesen der Datenfelder weiß das Device, woher die Nachricht kam und wohin sie zurückgeschickt werden muß. Andere Datenfelder beinhalten den Befehl, der ausgeführt werden soll, Angaben zur Durchführung des Befehls und oft die Startadresse des Datenblocks, der transferiert werden soll.

Befehle eines virtuellen Gerätes

Damit ein Device Ein-/Ausgabeoperationen durchführen kann, muß ihm mitgeteilt werden, welche Art der Ein-/Ausgabe gewünscht wird. Tab. 3.4 enthält alle Befehle, die an ein Device gesendet werden können, sowie die zugehörigen Befehlsnummern (in Form symbolischer Konstanten), die in der Datei "io.h", die sich auf Ihrer Compilerdiskette befindet, definiert sind.

Die Befehlsnummer wird im Feld "io_Command" des IORequest Blocks abgelegt, um dem Device mitzuteilen, welcher Befehl ausgeführt werden soll.

Befehl	Befehlsnummer	Zweck
Reset	CMD_RESET	Re-Initialisierung eines Devices; alle Variablen erhalten die Werte, die beim Öffnen des Devices angegeben wurden.
Read	CMD_READ	liest Bytes aus einen Speicherbereich.
Write	CMD_WRITE	schreibt Bytes in einen Speicherbereich.
Update	CMD_UPDATE	schreibt den Inhalt aller internen Puffer auf Diskette (o.ä.), wobei Änderungen der Daten während des Schreibvorgangs direkt übernommen werden.
Clear	CMD_CLEAR	löscht alle internen Puffer unwiderruflich.
Stop	CMD_STOP	stoppt ein Device. Auf diese Weise können Befehle übergeben werden, die zwischengespeichert, aber bis zum Neustart des Devices nicht ausgeführt werden.
Start	CMD_START	Neustart eines gestoppten Devices
Flush	CMD_FLUSH	bricht alle Ein-/Ausgabeoperationen unverzüglich ab; als Returnwert wird eine Fehlernummer geliefert.

Tab. 3.4: Befehle und Befehlsnummern für ein Device

Das Öffnen eines virtuellen Gerätes

Die Kommunikation mit einem Device erfolgt, wie bereits erwähnt, durch die Übergabe eines IORequest Blocks. Der Amiga stellt Ihnen für diese Übergabe einige Funktionen zur Verfügung, die auf jedes Device angewendet werden können, das Ein-/Ausgabeoperationen durchführt. Die Funktionen DoIO, SendIO, WaitIO und CheckIO dienen zum Verschicken des IORequest Blocks an ein Device. Damit die Funktionen wissen, welches Device angesprochen werden soll, überprüfen sie die Felder "io_Device" und "io_Unit" innerhalb der "Message"-Datenstruktur. Diese Felder werden beim Aufruf der Funktion OpenDevice initialisiert, die wie folgt ausgeführt wird:

```
fehler = OpenDevice(DEVICENAME,unit,ioRequest,Flags);
```

Der Returnwert "fehler" hat den Wert Null, wenn das Device korrekt geöffnet wurde. Trat ein Fehler auf, liefert OpenDevice einen Wert ungleich Null. Der Parameter "DEVICENAME" ist der Name des Device, mit dem kommuniziert werden soll. Der nächste Abschnitt enthält eine Tabelle aller Device-Namen. Der zweite Parameter der Funktion – "unit" – enthält Informationen darüber, mit welcher Einheit eines Device gearbeitet werden soll.

Arbeiten Sie z.B. mit den Zeitgebern (timer.device), dann geben Sie hier an, welcher der beiden Zeitgeber ("UNIT_VBLANK" oder "UNIT_MICROHZ") verwendet werden soll. Jedes Device verwendet das Datenfeld "unit" anders. Für viele Devices (Tastatur, Schnittstellen usw.) können Sie dieses Feld mit dem Wert Null initialisieren.

Der Parameter "ioRequest" ist ein Zeiger auf eine IORequest-Datenstruktur, deren Größe je nach Device verschieden ist. Exec verwendet beim Aufruf von OpenDevice Datenfelder Ihres IORequest Blocks, um weitere Initialisierungen von Datenfeldern durchführen zu können, auf die Sie dann später zugreifen können.

Vereinfacht gesagt übernimmt OpenDevice die Initialisierung eines Nachrichten-Blocks (Message Block), der u.a. die Startadresse des Device und die verwendeten Einheiten enthält. Bei der Arbeit mit den Zeitgebern verwenden Sie "timerrequest", beim Arbeiten mit der seriellen Schnittstelle hingegen kommt IOExtSer zum Einsatz.

Es ist wichtig, daß Ihr IORequest Block die richtige Größe für das Device hat, da auch das Device Datenfelder dieser Struktur initialisiert. Ist der IORequest

Block zu klein, dann wird u.U. ein Speicherbereich überschrieben, der z.B. von einem anderen Device verwendet wird, was meistens zu einem Systemabsturz führt.

Bei einigen Devices können Sie bereits beim Aufruf der Funktion OpenDevice angeben, welche speziellen Eigenschaften eines Device verwendet und bereitgestellt werden sollen. Verwenden Sie hierzu den Parameter "flags" beim Aufruf. Nachfolgend finden Sie ein Programmfragment, das den Zeitgeber (timer.device) öffnet:

```
struct timerequest   tr;
long fehler;

fehler = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);

if(fehler) printf("timer.device nicht geöffnet. Fehlernummer: %ld\n",fehler);
```

Die Bezeichnungen der verfügbaren Devices

In der Tab. 3.5 finden Sie die Namen aller verfügbaren Devices sowie eine kurze Beschreibung ihres Verwendungszwecks. Die Devices sind entweder speicherresident oder werden bei Bedarf von Diskette nachgeladen. In der Spalte "Devicename" finden Sie die Namen aufgeführt, die beim Aufruf von OpenDevice als Parameter "DEVICENAME" angegeben werden (eingeschlossen in Anführungszeichen).

Die Struktur eines typischen "IORequest"-Blocks

Nachfolgend die Datenstruktur eines "Standard IO Request" (IOStdReq):

```
struct IOStdReq
{
  struct Message io_Message;
  struct Device *io_Device;
  struct Unit *io_Unit;
  UWORD io_Command;
  UBYTE io_Flags;
  BYTE io_Error;
  ULONG io_Actual;
  ULONG io_Length;
  APTR io_Data;
  ULONG io_Offset;
};
```

Devicename	Verwendungszweck
audio.device	kontrolliert die vier Tonkanäle des Amiga.
clipboard.device	verwaltet einen Speicherbereich, in dem verschiedene Tasks Daten ablegen/abrufen können. Das Kopieren von Daten zwischen Tasks wird dadurch vereinfacht.
console.device	ermöglicht die Kommunikation mit dem Anwender über ein Fenster auf der Basis eines Terminals.
gameport.device	kontrolliert die Spieleports; es werden sowohl digitale als auch analoge Daten verarbeitet.
keyboard.device	verwaltet die Tastatur des Computers.
input.device	faßt die Datenströme der Tastatur und der Spieleports zu einem synchronisierten Datenstrom zusammen, der dann von Intuition oder einer "Console" weiterverarbeitet werden kann.
timer.device	verwaltet die beiden Zeitgeber des Amiga.
trackdisk.device	steuert die Datenübergabe vom/zum Laufwerk.
narrator.device	verwaltet das Sprachinterface des Amiga.
serial.device	verwaltet die serielle Schnittstelle.
parallel.device	verwaltet die parallele Schnittstelle.
printer.device	übernimmt die Ausgabe zum Drucker, der mit dem Programm "Preferences" selektiert wurde.

Tab. 3.5: Die Namen aller verfügbaren Devices

Ein IORequest Block muß verschiedene Voraussetzungen erfüllen. Zum einen muß er in die Liste laufender Ein-/Ausgabeoperationen übernommen werden können ("io_Message"), zum andern muß das Device, das angesprochen werden soll, durch ihn eindeutig identifiziert werden können ("io_Device", "io_Unit"). Beide Datenfelder werden von Exec beim Aufruf von OpenDevice verwendet und sollten nicht vom Programmierer geändert werden.

Weiterhin müssen Sie eine Möglichkeit haben, dem Device einen Befehl zu übergeben. Die Struktur stellt hierfür das Feld "io_Command" zur Verfügung. Device-spezifische Befehle sind in den Include-Dateien eines jeden Device definiert; allgemeingültige Befehle finden Sie in der Datei "exec/io.h" auf Ihrer Compilerdiskette.

Bei Ein-/Ausgabeoperationen werden meistens Daten zum Device gesendet bzw. vom Device abgerufen. Die Datenstruktur des IOStdReq stellt hierfür eine Anzahl Variablen zur Verfügung:

io_Flags	selektiert spezielle Eigenschaften eines Devices.
io_Data	Anfangsadresse eines Datenblocks im Speicher
io_Length	Größe des Datenblocks, der gesendet/empfangen werden soll
io_Offset	Anfangsadresse des Datenblocks, falls es sich um ein blockstrukturiertes Device (z.B. das Laufwerk) handelt
io_Message.mn_ReplyPort	sagt dem Device, wohin der IORequest Block nach seiner Verwendung geschickt werden soll.

Einige Datenfelder des IORequest Blocks werden vom Device verwendet:

io_Actual	Anzahl transferierter Bytes aus dem Block
io_Error	Dieses Feld enthält die Fehlernummer, falls bei der Ein-/Ausgabe ein Fehler aufgetreten ist.

Normalerweise stimmen die Werte von "io_Actual" und "io_Length" überein. Ist dies jedoch nicht der Fall, dann ist bei der Ein-/Ausgabeoperation ein Fehler aufgetreten, der durch das Auslesen von "io_Error" genauer spezifiziert werden kann.

"IOStdReq" stellt eine universelle Datenstruktur zur Übergabe von Daten an ein Device (von einem Device) dar. Trotzdem verwenden die meisten Devices des Amiga ihre eigenen Datenstrukturen; es handelt sich hierbei um abgewandelte Formen der "IOStdReq"-Datenstruktur. Eine Erläuterung zu diesen speziellen Datenstrukturen finden Sie später in diesem Kapitel.

Das Mindestmaß an Initialisierung für eine Ein-/Ausgabe

Obwohl die Initialisierung für eine Ein-/Ausgabeoperation immer von dem Device abhängt, das Sie verwenden wollen, finden Sie nachfolgend das Mindestmaß an Vorbereitung, das Sie treffen müssen, bevor Sie mit einem Device arbeiten können.

- Öffnen Sie das Device. Dabei werden die Felder "io_Device" und "io_Unit" initialisiert.

- Initialisieren Sie die folgenden Datenfelder des IORequest Blocks:

```
block.io_Message.mn_Node.ln_Type = NT_MESSAGE;
/* Der Node des Blocks ist vom Typ "Message" (Nachricht) */

block.io_Message.mn_Node.ln_Pri = 0;  /* Priorität der Nachricht */

block.io_Message.mn_Node.ln_Name = NULL;  /* Kein Name nötig */

block.io_Message.mn_ReplyPort = NULL;  /* Hier kann entweder NULL
                                          oder die Adresse eines
                                          Message Ports stehen */
```

  (Dies ist ein Beispiel der Initialisierung eines "IOStdReq". Wenn Sie Ihren eigenen Request Block verwenden, gehen Sie entsprechend vor.)

- Geben Sie den Befehl an, den das Device ausführen soll:

```
block.io_Command = <befehl>;
```

Senden eines Befehls an ein virtuelles Gerät

Nachdem Sie ein Device erfolgreich geöffnet haben, übergeben Sie ihm Ihren Request Block, der u.a. den Befehl enthält, der ausgeführt werden soll. Die beiden am häufigsten verwendeten Methoden zur Übergabe eines Request Blocks sind:

```
SendIO(request);        /* Ein-/Ausgabe durchführen, aber
                           nicht auf die Beendigung warten */

DoIO(request);          /* Ein-/Ausgabe durchführen und den
                           Task so lange anhalten, bis sie
                           durchgeführt wurde */
```

Einige Bemerkungen zum "Request Block"

Wenn Sie einem Device Ihren Request Block (oder eine andere Form einer Nachricht) übergeben, dann sollten Sie den Speicherbereich, in dem der Request Block abgelegt ist, nicht verwenden, solange das Device den Request Block nicht zurückgesendet hat. Dieser Speicherbereich wird dem Device für die Dauer einer Ein-/Ausgabeoperation zur Verfügung gestellt, so daß Änderungen in diesem Bereich während der Operation böse Folgen haben können.

Wurde die Ein-/Ausgabe beendet, dann sendet das Device Ihren Request Block an den "Reply Port" Ihres Tasks zurück, d.h. diese Message (Nachricht) wird in der vom Message Port verwalteten Liste eingefügt. Wollen Sie den Request Block nochmals verwenden oder die Message aus der Liste löschen, dann rufen Sie die Funktion GetMsg(ReplyPort) auf.

SendIO

Beim Aufruf dieser Funktion übergeben Sie einen Zeiger auf die Datenstruktur Ihres Request Blocks. SendIO vermerkt Ihren "Wunsch" nach einer Ein-/Ausgabeoperation in der "Message List" des betreffenden Device. Der Vorteil dieser Funktion liegt darin, daß ein so angesprochenes Device von Exec als eigener Task gehandhabt wird, d.h. Ihr eigener Task kann weiterarbeiten und muß nicht warten, bis das Device Ihren Request Block zurücksendet.

Ein-/Ausgabeoperationen werden vom Device nach dem "FIFO"-Verfahren (First-In-First-Out) durchgeführt, d.h. Ein-/Ausgabeoperationen werden in der Reihenfolge ihres Eintreffens durchgeführt. Sollte beim Eintreffen Ihres Blocks gerade eine andere Ein-/Ausgabe durchgeführt werden, dann wird Ihre Nachricht in der Message List an letzter Stelle vermerkt.

Weil Ihr Task nicht auf die Beendigung der Ein-/Ausgabe warten muß, sondern später überprüfen kann, ob sie erfolgreich durchgeführt wurde, spricht man auch von einer asynchronen Ein-/Ausgabe. Wurde Ihre Ein-/Ausgabe durchgeführt – erfolgreich oder nicht –, dann sendet das Device Ihren Request Block (Message) an den Reply Port Ihres Tasks.

Bevor Sie den Request Block nochmals verwenden können, müssen Sie diese Message erst aus der Liste aller Nachrichten, die Ihr Reply Port verwaltet, löschen. Setzen Sie bei der Initialisierung des Request Blocks das Datenfeld des Reply Ports auf den Wert Null, dann wird kein Reply Port verwendet. In einem solchen Fall sollten Sie die Funktion CheckIO verwenden, um zu prüfen, ob Ihre Ein-/Ausgabeoperation bereits beendet wurde.

DoIO

Dieser Funktion wird beim Aufruf ebenfalls ein Zeiger auf Ihren Request Block übergeben. Im Gegensatz zu SendIO wartet Ihr Task bei der Übergabe des Blocks mittels DoIO so lange, bis er von der Funktion zum Reply Port Ihres Tasks zurückgeschickt wird.

Da Ihr Task bis zum Eintreffen der Nachricht "schläft", spricht man hier von einer synchronen Ein-/Ausgabe, da ja auf die Beendigung der Operation gewartet wird, egal, ob sie erfolgreich war oder nicht. Bevor Ihr Task wieder "aufgeweckt" wird, löscht DoIO automatisch den entsprechenden Eintrag aus der Liste des Reply Ports (natürlich nur, wenn Sie einen Reply Port eingerichtet haben).

Es gibt noch eine dritte Möglichkeit, einen Request Block an ein Device zu übergeben: BeginIO. Die Verwendung dieser Funktion setzt jedoch genaue Kenntnisse über Devices voraus und wird daher in diesem Buch nicht behandelt. Die meisten Beispielprogramme in diesem Buch verwenden die Funktionen SendIO oder DoIO.

Weitere Funktionen zur Ein-/Ausgabe

Um Ein-/Ausgabeoperationen durchführen zu können, brauchen Sie noch Informationen über die anderen Funktionen, die hierfür zur Verfügung stehen: WaitIO, AbortIO und CheckIO.

Ein Aufruf der Funktion WaitIO sieht wie folgt aus:

```
WaitIO(request);
```

Diese Funktion wird verwendet, wenn Sie Ein-/Ausgaben mit SendIO durchführen und Ihr Programm an eine Stelle kommt, an welcher die Operation abgeschlossen sein muß. WaitIO hält Ihren Task so lange an, bis der Request Block am Reply Port Ihres Task ankommt, wo er dann automatisch von WaitIO aus der Liste entfernt wird.

Mit einem Aufruf der Form

```
AbortIO(request);
```

wird die Ein-/Ausgabe, die in "request" spezifiziert ist, abgebrochen. Dabei ist es egal, ob sie bereits in der Ausführung ist oder nicht.

Die Funktion CheckIO liefert den Wert "TRUE" (1) als Ergebnis, wenn die aktuelle Ein-/Ausgabeoperation beendet wurde. Sie können auf diese Art z.B. prüfen, ob Sie AbortIO aufrufen sollen oder nicht. CheckIO wird wie folgt aufgerufen:

```
returnwert = CheckIO(request);
```

Beispielprogramm:
Anwendung der Ein-/Ausgabefunktionen

Das Listing 3.6 enthält ein Beispielprogramm, in dem die oben besprochenen Funktionen zur Ein-/Ausgabe verwendet werden. Es handelt sich um ein Programmfragment, das beliebig ergänzt werden kann. In Kapitel 6 finden Sie weitere Programme zur Ein-/Ausgabe.

Die Fragmente in Listing 3.6 verwenden den Zeitgeber (timer.device) als Device sowie folgende Datenstrukturen:

```
struct timerequest *msg;          /* Zeiger auf die Nachricht
                                     eines Message Ports */

struct MsgPort MeinReplyPort;     /* Der Reply Port, an den der
                                     Request Block nach
                                     Beendigung der Ein-/Ausgabe
                                     zurückgeschickt wird */

struct timerequest MeinTimeReq;   /* Der Message Block des
                                     Timers */
```

Die Fragmente gehen weiterhin davon aus, daß das Device wie folgt geöffnet wurde:

```
long fehler;

fehler = OpenDevice("timer.device",UNIT_VBLANK,&MeinTimeReq,0);
```

Nachfolgend finden Sie die Initialisierung für den IORequest Block des Zeitgebers:

```
MeinZeitgeberReq.tr_node.ln_Type = NT_MESSAGE;   /* Node ist Msg */
MeinZeitgeberReq.tr_node.ln_Pri = 0;     /* Priorität der Message */
MeinZeitgeberReq.tr_node.ln_Name = NULL;  /* Kein Name notwendig */
MeinZeitgeberReq.tr_node.mn_ReplyPort = &MeinReplyPort;
                                          /* Der ReplyPort des Tasks */
```

Zuletzt muß der Zeitgeber noch mit den Zeitintervallen initialisiert werden:

```
MeinZeitgeberReq.tr_time.tv_secs = 3;          /* 3 Sekunden */
MeinZeitgeberReq.tr_time.tv_micro = 0;   /* 0 Mikrosekunden */
```

Beachten Sie bitte, daß es bei den Fragmenten 1 und 2 nicht notwendig ist, den Request Block, der vom Device zurückgeschickt wird, aus der Liste aller Nachrichten des Reply Ports zu löschen. Die verwendeten Funktionen DoIO und WaitIO erledigen dies automatisch.

```
/* Fragment 1: Auf Beendigung der Ein-/Ausgabeoperation warten */

DoIO(&myTimeReq);    /* Task wartet an dieser Stelle */
/* Weitere Programmteile... */


/* Fragment 2: Ein-/Ausgabe starten, aber nicht auf's Ende warten */

SendIO(&myTimeReq);    /* Timer starten, dann weiter im Programm */
/* Weitere Programmteile... */

WaitIO(&myTimeReq);  /* Auf Beendigung der Ein-/Ausgabe warten */
/* Weitere Programmteile... */


/* Fragment 3: Prüfen, ob die Ein-/Ausgabe bereits beendet wurde */

SendIO(&myTimeReq);
hier_gehts_weiter:
   /* Weitere Programmteile... */

ergebnis = CheckIO(myTimeReq);
if(!ergebnis) goto hier_gehts_weiter;

/* ReplyPort "entleeren" */

message = GetMsg(&myReplyPort);
/* Weitere Programmteile... */
/* Fragment 4: Ein-/Ausgabe starten, Programm weiterführen.
   Dann prüfen, ob die Operation abgeschlossen wurde. Wenn nicht,
   Operation abbrechen */

SendIO(&myTimeReq);
/* Weitere Programmteile... */

ergebnis = CheckIO(myTimeReq);
if(!ergebnis) AbortIO(myTimeReq);  /* Ein-/Ausgabe abbrechen */
else printf("Ein-/Ausgabe wurde beendet!\n");

/* ReplyPort "entleeren" */

message = GetMsg(&myReplyPort);
/* message muß "myTimeReq" entsprechen */
/* Weitere Programmteile... */
```

Listing 3.6: Beispiel zur Durchführung von Ein-/Ausgabeoperationen

Warum verwendet man "Reply Ports"?

Wie Sie bereits gesehen haben, ist es nicht erforderlich, einen Reply Port zu definieren, wenn eine Ein-/Ausgabeoperation mit einem Device durchgeführt werden soll. Erinnern wir uns: wird ein Reply Port für einen Task eingerichtet, dann sendet das Device, mit dem zuletzt gearbeitet wurde, den übermittelten IORequest Block an diesen Port zurück. Initialisieren Sie hingegen das Datenfeld des Reply Ports im IORequest Block mit Null, dann wird kein Reply Port eingerichtet.

Warum also verwendet man dann Reply Ports? In den Abschnitten über die Signal-Bits haben Sie erfahren, daß ein Task auf eine oder mehrere Aktionen warten kann, bevor er von Exec weiter abgearbeitet wird. Tritt ein bestimmtes Ereignis ein (d.h. es wird ein Signal-Bit gesetzt), dann schickt Exec eine Message (Nachricht) an den Message Port des betreffenden Tasks, der daraufhin "aufwacht". Dies ist jedoch auch der Fall, wenn eine Nachricht an einem Reply Port (der ja auch ein Message Port ist) anliegt. Reply Ports sind daher ein nützliches Werkzeug beim Multitasking, da ein Task ja "schläft", solange keine Nachricht an seinem Reply Port anliegt, d.h. andere Tasks können während dieser Zeit ungehindert weiterarbeiten.

Stapelverarbeitung von mehreren Ein-/Ausgabeoperationen

Wenn Sie Ein-/Ausgabeoperationen durchführen, jedoch nicht auf deren Beendigung warten wollen, dann können Sie die Funktion SendIO verwenden. Mit ihr laufen Operationen asynchron ab, d.h. Ihr Task kann weiterlaufen, während die Ein-/Ausgabe von einem eigenen Task des Devices durchgeführt wird.

Für jede Ein-/Ausgabeoperation benötigen Sie einen eigenen IORequest Block. Da seine Initialisierung beim Aufruf der Funktion OpenDevice durchgeführt wird, können Sie ihn einfach kopieren, um so einen neuen IORequest Block zu erhalten.

Die relevanten Datenfelder hierfür sind "io_Device" und "io_Unit", die einfach für jeden weiteren Block individuell angepaßt werden. Alle IORequest Blocks können so lange verwendet werden, wie das Device, für das sie verwendet werden sollen, geöffnet ist.

Nachfolgend finden Sie ein Programmfragment, das eine Kopie des "originalen" IORequest Blocks erstellt.

```
struct timerequest  tr,tr2;
long fehler;
fehler = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);

if(!fehler)
   {
     tr2.io_Device = tr.io_Device;
     tr2.io_Unit = tr.io_Unit;
   }

/* Es stehen Ihnen jetzt zwei IORequest Blocks für
   Ein-/Ausgabe zur Verfügung */
```

Zugriff auf eine Bibliotheks-Funktion für ein virtuelles Gerät

Die Struktur eines Device und die einer Bibliothek sind fast gleich; die Funktionsvektoren eines Device werden auf die gleiche Art und Weise verwaltet und initialisiert, wie dies auch bei einer Bibliothek geschieht. Zwei Devices – TimerDevice und ConsoleDevice – haben eigene Funktionen, die von C aus genauso verwendet werden können, als wären sie Bestandteil einer Bibliothek.

Aus diesem Grund existieren für beide Geräte auch feststehende Variablennamen für die Basisadressen, nämlich "TimerBase" für den Zeitgeber und "ConsoleBase" für das ConsoleDevice. Um die Funktionen der Device-Bibliothek verwenden zu können, muß man – genau wie bei den Bibliotheken – zunächst einmal die Basisadresse definieren. Dies geschieht, indem man das Device öffnet und den Wert der Variablen "io_Device" aus dem IORequest Block verwendet. Nachfolgend ein Beispiel hierzu:

```
/* Programmfragment zur Initialisierung der Basisadresse des
   Zeigebers. Nach der Definition können Device-spezifische
   Funktionen verwendet werden, z.B. AddTime, CmpTime und SubTime */

extern struct Library *TimerBase;
struct timerequest tr;
long fehler;

fehler = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);

if(!fehler) TimerBase = tr.tr_node.io_Device;
else printf("Timer nicht geöffnet. Fehlernummer: %ld\n",IoErr());
```

Sobald die Basisadresse erfolgreich definiert wurde, können Sie die Funktionen des Zeitgebers (AddTime, CmpTime und SubTime) verwenden, die im "Amiga ROM Kernel Manual" genauer beschrieben sind.

Das ConsoleDevice stellt nur eine Funktion zur Verfügung: RawKeyConvert. Die Definition der Basisadresse und genaue Erklärungen finden Sie erst im Kapitel 5, da die Verwendung dieses Gerätes Kenntnisse über Fenster erfordert – eine Sache, die bisher noch nicht in diesem Buch behandelt wurde.

Das Schließen eines virtuellen Gerätes

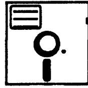
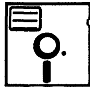
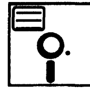
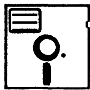
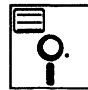
Wenn Sie ein Device nicht mehr zur Ein-/Ausgabe benötigen, dann müssen Sie es schließen, um Ihrem Task den Zugriff auf dieses Gerät zu entziehen. Dies ist notwendig, da viele Devices (insbesondere die Laufwerke) bei ihrer Initialisierung und Nutzung Speicherplatz belegen. Manche Devices benötigen zusätzlich noch Rechenzeit des Prozessors.

Wenn Sie ein Device nach dem Beenden Ihres Programms geöffnet lassen, dann wird der vom Device reservierte Speicherplatz nicht freigegeben und der Prozessor u.U. unnötig verlangsamt. Geschlossen wird ein Device mit der Funktion CloseDevice, die wie folgt aufgerufen wird:

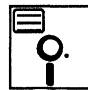
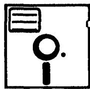
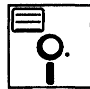
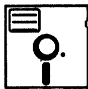
```
CloseDevice(request);
```

Der Parameter "request" ist der Zeiger auf Ihren IORequest Block, den die Funktion OpenDevice als Returnwert liefert. Haben Sie den IORequest Block kopiert, dann verwenden Sie zum Schließen nur einen IORequest Block. Anders gesagt: das Device darf nur so oft geschlossen werden, wie es auch von Ihrem Task aus geöffnet wurde. Weiterhin sollten Sie sicherstellen, daß das Device alle von Ihnen "verschickten" IORequest Blocks zurückgesendet hat. Wissen Sie nicht mehr, wie viele Blocks bereits abgearbeitet wurden, dann sollten Sie zur Sicherheit vor dem Programmende die Funktion AbortIO aufrufen, die die Message-Liste des Device komplett löscht. Erst dann sollten Sie die Funktion CloseDevice aufrufen.

Devices können von mehreren Tasks gleichzeitig verwendet werden. Jedes Device verwaltet einen Zähler, der bei jedem Öffnen um Eins erhöht wird. Jedes Schließen vermindert den Wert um Eins. Hat der Zähler den Wert Null, dann kann Exec das Device selbsttätig aus seiner Liste der aktuell verwendeten Devices streichen, da es ja von keinem Task mehr benötigt wird.

4

Kapitel 4

Grafik

Damit Sie lernen, die hervorragenden Grafikeigenschaften des Amiga zu nutzen, finden Sie in diesem Kapitel eine Vielzahl von Beispielprogrammen und Programmfragmenten, die Sie direkt in eigene Programme einbinden können. Alle Programmbeispiele sind kompatibel mit Intuition, der grafischen Benutzeroberfläche des Amiga.

Wie Sie bereits wissen, bietet das Softwaresystem des Amiga dem Programmierer eine Vielzahl von Einstiegspunkten, um ein bestimmtes Ziel zu erreichen. Einige der Systemroutinen (aus den Bausteinen "layers" und "graphics") werden in diesem Kapitel nicht behandelt, damit die Kompatibilität der Programme untereinander bzw. mit Intuition gewährleistet bleibt.

In diesem Kapitel werden Sie lernen, wie man Grafikbereiche selektiert und initialisiert, wie Farben erzeugt und angewendet werden, wie Bereiche mit Farben oder Mustern gefüllt werden und wie man Linien, Kreise und Rechtecke zeichnet.

Weiterhin finden Sie Beispiele zum Arbeiten mit dem "Workbench-Screen" und eigenen "Screens". Beim Arbeiten mit dem Workbench-Screen erstellen wir ein Programm, das ein Balkendiagramm zeichnet; bei den "Custom-Screens" werden wir den Amiga eine Landkarte zeichnen lassen. Zusätzlich erfahren Sie alles über den Einsatz von Text und über die verschiedenen Zeichensätze des Amiga.

Zum Ende dieses Kapitels finden Sie ein Programm, das ein Bild malt.

Das Öffnen eines Fensters auf dem "Workbench-Screen"

Um ein neues Fenster auf dem Workbench-Screen öffnen zu können, müssen zuvor einige Vorbereitungen getroffen werden:

- Die linke obere Ecke des Fensters muß festgelegt werden.
- Die Größe (Höhe und Breite) des Fensters muß definiert werden.
- Der Name des Fensters muß angegeben werden.
- Festlegung der Farben, die in diesem Fenster verwendet werden sollen.
- Weiterhin müssen Sie angeben, welche Art von Fenster erzeugt werden soll und welche Gadgets (siehe Kapitel 5) verwendet werden.
- Läßt sich die Größe des Fensters ändern, dann geben Sie die Minimal- und Maximalwerte der Fenstergröße an.
- Angegeben werden muß auch, auf welchem "Screen" das Fenster geöffnet werden soll (in unserem Fall ist es der Workbench-Screen).

Definition und Initialisierung eines neuen Fensters

Um ein neues Fenster erzeugen zu können, muß eine Datenstruktur namens "NewWindow" definiert werden, die alle notwendigen Parameter des Fensters enthält. Das Listing 4.1 enthält eine typische Initialisierung dieser Datenstruktur; wir verwenden sie später für das Fenster, in dem ein Balkendiagramm gezeichnet werden soll.

Das "IDCMP Flags"-Datenfeld

Intuition kann mit Ihrem Task kommunizieren, wenn der Anwender bestimmte Ereignisse hervorruft. Diese Ereignisse kommen z.B. dann zustande, wenn der Anwender Ihr Fenster vergrößert, verkleinert oder schließt.

```
/* window1.h */

struct NewWindow MeinFenster =
{
  30,
  30,   /* Linke obere Ecke des Fensters relativ zur linken oberen
           Ecke des Screens (Angaben in  Bildpunkten) */
  500,150, /* Breite und Höhe des Fensters (Angaben in Bildpunkten) */
  -1,   /* Farbregister zum Zeichnen des Rahmens des Fensters
           (DetailPen) */
  -1,   /* Farbregister zum Zeichnen der Gadgets des Fensters
           (BlockPen) */
        /* Die Werte -1 verwenden jeweils die voreingestellten Farben */
  CLOSEWINDOW|
  NEWSIZE|
  REFRESHWINDOW,     /* IDCMP Flags */
  SIMPLE_REFRESH|
  NORMALFLAGS|
  GIMMEZEROZERO,     /* Window Flags */
  NULL,      /* Erstes Gadget des Fensters (siehe Kapitel 5) */
  NULL,      /* "CheckMark" (siehe Kapitel 5) */
  "Balkendiagramm",    /* Name des Fensters */
  NULL, /* Zeiger auf den zugehörigen Screen, falls nicht Workbench */
  NULL, /* Zeiger auf eigene BitMap (falls "SuperBitMapWindow") */
  10,10,    /* Minimale Höhe und Breite des Fensters */
  640,200,  /* Maximale Höhe und Breite des Fensters */
  WBENCHSCREEN /* Art des Screens, auf dem das Fenster geöffnet wird */
};
```

Listing 4.1: Initialisierung einer NewWindow-Datenstruktur

Für die Kommunikation verwendet Intuition den "Direct Communications Message Port" (etwa: virtuelle Kommunikationsleitung zwischen Intuition und einem Task), kurz "IDCMP" genannt.

In unserem kleinen Beispielprogramm finden Sie ein Modul, das auf Nachrichten wartet, die am IDCMP ankommen. Die Art der Nachricht entscheidet dann darüber, auf welche Weise das Programm weitergeführt wird. Die im Listing 4.1 verwendeten symbolischen Konstanten zur Definition des "IDCMP"-Datenfeldes bedeuten im einzelnen:

CLOSEWINDOW — Diese Nachrichtenart wird von Intuition an Ihr Programm geschickt, wenn der Anwender das Fenster schließt.

NEWSIZE — Wird Ihr Fenster vergrößert oder verkleinert, dann wird diese Art Nachricht generiert. Sie können so entscheiden, ob z.B. der Fensterinhalt bzw. Teile von ihm neu gezeichnet werden sollen.

REFRESHWINDOW — Wird das Fenster in den Hintergrund geschoben oder wieder nach vorne geholt, dann erhalten Sie von Intuition diese Art Nachricht. Auch hier können Sie wieder entscheiden, ob der Inhalt neu gezeichnet werden soll.

Das "Window Flags"-Datenfeld

In dem Listing 4.1 werden drei symbolische Konstanten verwendet, um das Window Flags-Datenfeld zu füllen: "SIMPLE_REFRESH", "NORMALFLAGS" und "GIMMEZEROZERO".

"SIMPLE_REFRESH" bedeutet, daß Bereiche des Fensters, die überdeckt werden, nicht vom System her neu gezeichnet werden, wenn sie wieder sichtbar sind. Es wird also keine Kopie des Fensterinhaltes im Speicher abgelegt; der Inhalt des Fensters geht verloren, wenn es von einem anderen Fenster (o.ä.) überdeckt wird. Sie müssen also von Ihrem Programm aus Schritte zur Wiederherstellung des Inhalts vornehmen.

"NORMALFLAGS" ist keine Systemkonstante, sondern wurde zur Vereinfachung von mir eingeführt. Normalerweise enthalten Fenster immer alle Systemgadgets (kleine symbolische Grafiken, die mit der Maus "angeklickt" werden, um so z.B. das Fenster zu schließen). Die verfügbaren Gadgets und ihre symbolischen Konstanten sind:

WINDOWDRAG — erlaubt das Verschieben des Fensters.

WINDOWSIZING — erlaubt das Vergrößern/Verkleinern des Fensters.

WINDOWCLOSE — dient zum Schließen des Fensters.

WINDOWDEPTH — Mit diesem Gadget kann das Fenster in den Hintergrund bzw. Vordergrund "geklickt" werden.

Aus allen diesen Konstanten wird "NORMALFLAGS" wie folgt gebildet:

```
#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP|WDR)
```

Beim Öffnen eines Fensters wird auch eine Datenstruktur mit Namen "Window" definiert und initialisiert. Eine Komponente dieser Struktur ist eine weitere Datenstruktur: "RastPort".

Der RastPort eines Fensters verwaltet die Darstellung von Grafik in einem Fenster und bestimmt, auf welche Art und Weise mit den Grafikroutinen gezeichnet wird.

Die Konstante "GIMMEZEROZERO" legt den Ursprung des Grafikbereiches eines Fensters innerhalb der Fensterumrandung fest. Normalerweise beziehen sich alle Koordinatenangaben auf die linke obere Ecke eines Fensters. Ein Teil des Grafikbereiches wird dadurch verdeckt, da ja die Umrandung des Fensters auch innerhalb dieses Bereiches liegt. Verwenden Sie ein "GIMMEZEROZERO"-Fenster, dann werden zusätzlich alle Grafikteile, die nicht im Grafikbereich des Fensters liegen, "abgeschnitten", d.h. es kann nur innerhalb der Fensterumrandung gezeichnet werden.

Öffnen des Fensters

Nachfolgend finden Sie ein Programmfragment, mit dem das zuvor definierte Fenster "MeinFenster" (Listing 4.1) geöffnet werden kann:

```
struct Window *w;

if(!(w = OpenWindow(&MeinFenster))) printf("Fenster nicht geöffnet!\n");
```

Sie übergeben der Funktion OpenWindow einen Zeiger auf die "NewWindow"-Datenstruktur ("&MeinFenster"). Als Returnwert erhalten Sie einen Zeiger auf die "Window"-Datenstruktur, die entsprechend Ihren Angaben initialisiert wurde.

Erhalten Sie den Wert Null von der Funktion, dann konnte das Fenster nicht geöffnet werden, z.B. wenn nicht genügend Speicherplatz zur Verfügung steht. Ist der Returnwert hingegen ungleich Null, dann wurde OpenWindow erfolgreich durchgeführt, und Sie haben nun einen Zeiger auf die "Window"-Datenstruktur; mit anderen Worten: das von Ihnen definierte Fenster wurde auf dem Workbench-Screen geöffnet.

Verarbeitung der Ereignisse, die Intuition sendet

Das "IDCMP"-Datenfeld in der "NewWindow"-Datenstruktur legt fest, daß unser Beispielprogramm auf drei Ereignisse warten soll, die in Form von Nachrichten (Messages) von Intuition gesendet werden. Wir benötigen daher eine Routine, die ankommende Nachrichten am IDCMP entsprechend verarbeiten kann. Die von uns verwendete Routine trägt den Namen HandleEvent.

Diese Routine wird vom Hauptprogramm (main) aus aufgerufen, das später noch in diesem Kapitel behandelt wird. Entsprechend der Nachrichtenart liefert HandleEvent einen Returnwert von Null oder Eins. Schließt der Anwender mit Hilfe des Schließ-Gadgets das Fenster (d.h. er beendet das Programm), dann wird der Wert Null von HandleEvent geliefert. Dem Hauptprogramm wird auf diese Weise mitgeteilt, daß das Fenster geschlossen wurde, das Programm also beendet werden soll.

Ist die Nachrichtenart jedoch vom Typ "NEWSIZE" (das Fenster wurde in seiner Größe verändert) oder "REFRESHWINDOW" (das Fenster wurde in den Hintergrund "geklickt" und dann wieder in den Vordergrund geholt), dann ruft HandleEvent seinerseits die Funktion redraw auf, die die Wiederherstellung des Fensterinhalts erledigt. Dies ist notwendig, da es sich um ein "SIMPLE_REFRESH"-Fenster handelt, bei dem keine Kopie des Fensterinhalts im Speicher angelegt wird. Die Routine HandleEvent finden Sie im Listing 4.2.

Auffinden des "RastPort"

Als nächsten Schritt müssen wir den "RastPort" des neuen Fensters finden. Dies ist jedoch recht einfach, da es sich hierbei um eine Strukturkomponente der "Window"-Datenstruktur handelt, für die wir bereits einen Zeiger besitzen.

```
/* event1.c */

HandleEvent(art)
long art;    /* Dieser Parameter wird von main() übergeben */
{
  switch(art)
              {
                case CLOSEWINDOW: return(0);

                case NEWSIZE:
                case REFRESHWINDOW: redraw();

                default: return(1);
              }

}
```

Listing 4.2: Die "HandleEvent"-Routine

Der RastPort eines Fensters ist wiederum eine Datenstruktur, durch die die Darstellung von Grafik in einem Fenster ermöglicht wird. Für die meisten Anwendungen ist eine genaue Kenntnis des Inhalts der RastPort-Datenstruktur nicht notwendig. Es genügt, einen Zeiger auf die Startadresse des RastPort zu haben, der den Systemroutinen beim Aufruf als Parameter übergeben wird, die dann intern mit dem Inhalt der Datenstruktur weiterarbeiten.

Zum Inhalt der Datenstruktur gehört u.a. die aktuelle Position des "Zeichenstiftes", die aktuelle Zeichenfarbe sowie die Höhe und Breite des verwendeten Zeichensatzes. Eine genaue Erklärung der einzelnen Strukturkomponenten eines RastPort finden Sie im Kapitel 2 des "Amiga Programmer's Handbook, Teil 1" von Eugene P. Mortimore (SYBEX 1987). Eine solche Erläuterung würde den Rahmen dieses Buches sprengen; Sie finden lediglich Erklärungen, wenn Strukturkomponenten erstmals verwendet werden.

Der RastPort eines Fensters wird wie folgt aufgefunden:

```
struct RastPort *rp;     /* Zeiger auf einen RastPort */
struct Window *w;        /* Zeiger auf eine Window-Struktur */

rp = w->RPort;           /* Der RastPort ist eine Komponente
                            der Window-Struktur, daher
                            kann er durch einen einfachen
                            Strukturverweis gefunden werden. */
```

Ausgabe von Grafik in einem Fenster

Zu diesem Zeitpunkt haben wir Programmfragmente erstellt, die ein Fenster öffnen und den dazugehörigen RastPort auffinden. Weiterhin haben wir eine Möglichkeit geschaffen, mit Intuition kommunizieren zu können. Der Zeiger auf diesen RastPort kann nun einfach den Systemroutinen übergeben werden, um Grafik in diesem Fenster auszugeben.

Diese Routinen befinden sich in der Bibliothek "graphics.library". Wir verwenden einige von ihnen, um ein Balkendiagramm in unserem Fenster zeichnen zu lassen. Damit das Beispiel einen möglichst großen Lerneffekt hat, werden allgemein verwendete Grafikroutinen benutzt. Wir verwenden Routinen, mit denen man

- Farben auswählen kann,
- den Zeichenmodus definiert,
- die Koordinatenachsen für unser Beispiel zeichnen kann,
- die Achsen mit Bezeichnungen versieht sowie
- Rechtecke und punktierte Linien zeichnet.

Bei der Darstellung eines Graphen zeichnet man üblicherweise zuerst die Koordinatenachsen und dann erst die Funktion. Vereinfacht werden kann das Ganze, wenn man die Punkte des Graphen relativ zu seinem Koordinatensystem angeben kann, anstatt sich auf die Koordinaten des Fensters zu beziehen. Wir benötigen daher Funktionen, mit denen eine solche Vorgehensweise möglich gemacht wird.

Die im Beispielprogramm verwendeten Funktionen greifen alle auf eine gemeinsame Datenstruktur zurück, die es erlaubt, die Koordinaten des Graphen relativ zum Ursprung des Koordinatensystems anzugeben.

Diese Datenstruktur, die Sie im Listing 4.3 finden, beinhaltet vier Komponenten, mit denen die Umwandlung der Werte, die sich ja alle auf den Ursprung des Fensters beziehen (die linke obere Ecke), vorgenommen wird.

```
/* xybase.h */

struct XYBase
{
  WORD xaxis;   /* Bestimmt, wo die x-Achse im Fenster positioniert
                   wird */
  WORD yaxis;   /* Bestimmt, wo die y-Achse im Fenster positioniert
                   wird */
  WORD xlength;   /* Länge der x-Achse */
  WORD ylength;   /* Länge der y-Achse */
};
```

Listing 4.3: Definition der "XYBase"-Datenstruktur

Auswahl der Farben

Wenn Sie ein Fenster auf dem Workbench-Screen öffnen, dann stehen Ihnen vier Farben ("Zeichenstifte") zur Verfügung, mit denen Sie zeichnen können. Der Amiga verwaltet diese "Stifte" anhand von Zahlen, die die einzelnen Farbregister repräsentieren. Auf der Workbench stehen Ihnen also die Farbregister 0 bis 3 zur Verfügung.

Die Farbgebung nehmen Sie für jedes der vier Register mit dem Programm "Preferences" vor, das sich auf Ihrer Workbench-Diskette befindet. Beim Starten des Programms finden Sie unten links vier Quadrate in den Farben des jeweiligen Farbregisters. Beachten Sie bitte, daß Farbregister 0 die Hintergrundfarbe enthält.

Der Amiga stellt Ihnen zum Zeichnen drei verschiedene Arten von "Zeichenstiften" zur Verfügung: "APen", "BPen" und "OPen".

Der "APen"-Zeichenstift

Die Auswahl dieses "Stiftes" bedeutet, daß zum Zeichnen das Farbregister 1 verwendet wird. Man spricht auch von der Vordergrundfarbe, da der Amiga z.B. bei Fensterumrandungen (die den Vordergrund darstellen) immer diesen Stift verwendet. Wollen Sie für den APen ein anderes als das Farbregister 1 verwenden, dann verwenden Sie hierzu die Routine SetAPen, mit der diesem Stift ein anderes Farbregister zugeordnet werden kann. Der Aufruf der Funktion lautet:

```
SetAPen(rp,farbregister);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, für den APen geändert werden soll, "farbregister" ist die Nummer des neuen Farbregisters. Der Wert, der hier angegeben wird, muß natürlich im Bereich der verfügbaren Anzahl von Registern liegen; in unserem Beispiel darf er also nicht größer als drei sein.

Wenn Sie herausfinden wollen, welches Farbregister momentan für APen verwendet wird, dann brauchen Sie nur das zugehörige Datenfeld in der "RastPort"-Datenstruktur auszulesen. Der Name dieser Variablen ist "FgPen" (Foreground Pen, Vordergrundfarbe). Ausgelesen wird diese Variable durch einen Strukturverweis:

```
AktuellerAPen = rp->FgPen;
```

Der "BPen"-Zeichenstift

Wird der Inhalt dieser "RastPort"-Strukturkomponente nicht geändert, dann hat diese Variable den Wert 0, d.h. zum Zeichnen wird die Hintergrundfarbe (Register 0) verwendet.

Durch die Kombination von APen und BPen in einem Bitmuster (0 und 1) können so auf einfache Art Muster erzeugt werden, wobei die Bits mit dem Wert Eins den APen verwenden, Bits mit dem Wert Null hingegen BPen. Diese Art Muster kann verwendet werden, wenn man für einen RastPort den Zeichenmodus "JAM2" angibt (eine Erklärung der Zeichenmodi finden Sie später in diesem Kapitel).

Wollen Sie ein anderes als das Farbregister Null für den BPen verwenden, dann benutzen Sie die Funktion SetBPen, die wie folgt aufgerufen wird:

```
SetBPen(rp,farbregister);
```

Um festzustellen, welches Farbregister aktuell von BPen verwendet wird, können Sie die Strukturkomponente "BgPen" (Background Pen, Hintergrundfarbe) des RastPort mit Hilfe eines Strukturverweises auslesen:

```
AktuellerBPen = rp->BgPen;
```

Der "OPen"-Zeichenstift

Mit diesem Zeichenstift können Sie Objekte umranden lassen. Wenn Sie ein ausgemaltes Vieleck erstellen und OPen einen gültigen Wert enthält, dann wird dieses Vieleck vom System automatisch mit einer Linie umrandet, die die Farbe des Farbregisters hat, das OPen zugeordnet wurde. Festgelegt wird dieses Farbregister wie folgt:

```
SetOPen(rp,farbregister);
```

Den aktuellen Wert von "farbregister" können Sie ebenfalls durch einen Strukturverweis innerhalb des RastPorts feststellen:

```
AktuellerOPen = rp->AOIPen;
```

Sie können diesen Wert auch für den APen verwenden, wenn Sie Flächen ausmalen ("Flood Fill"). Nähere Informationen zum Füllen von Flächen finden Sie später in diesem Kapitel, wenn wir uns mit "Custom-Screens" beschäftigen.

Auswahl des Zeichenmodus

Wenn Sie einfache Linien zeichnen wollen, dann sollten Sie "JAM1" als Zeichenmodus verwenden. Das bedeutet soviel wie "verwende zum Zeichnen nur eine Farbe". Die Koordinatenachsen in unserem Beispielprogramm werden in diesem Modus gezeichnet. Die Auswahl eines Zeichenmodus erfolgt mit der Funktion SetDrMd:

```
SetDrMd(rp,JAM1);
```

Wollen Sie zweifarbig zeichnen, dann verwenden Sie den "JAM2"-Modus. Hierbei werden gesetzte Bits (d.h. mit dem Wert Eins) in der Farbe gemalt, die aktuell von APen verwendet wird. Bits mit dem Wert Null werden mit BPen gezeichnet. Auf diese Weise können Sie sehr einfach Muster – z.B. in Form von punktierten Linien – erstellen.

Ein weiterer Modus ist "COMPLEMENT". In diesem Modus werden alle Bits, die den Wert Null besitzen, auf Eins gesetzt und umgekehrt. Ein Beispiel: Setzt sich eine Farbe aus dem 4-Bit-Wert "0101" zusammen, dann wird beim Einsatz von "COMPLEMENT" die Farbe "1010" erzeugt.

Dieser Zeichenmodus ist dann sinnvoll, wenn Linien gezeichnet und anschließend gelöscht werden sollen, um auf diese Weise Bewegung zu erzeugen. Beim ersten Zeichnen einer Linie im "COMPLEMENT"-Modus ist sie sichtbar. Beim wiederholten Zeichnen der Linie verschwindet sie, da alle Bits wieder ihre Ausgangswerte erhalten.

Der "INVERSVID"-Zeichenmodus ist eine Kombination aus den Modi "JAM1" und "JAM2". Erklärungen zu diesem Modus finden Sie später in diesem Kapitel, wenn wir uns mit der Ausgabe von Text in einem Fenster beschäftigen.

Zeichnen der Achsen

Bevor man eine Linie zeichnen kann, muß zunächst ihr Startpunkt angegeben werden, von dem aus dann bis zum Ende der Linie gezeichnet werden soll. Die Funktionen Move(rp,x,y) und Draw(rp,x,y) stehen hierfür zur Verfügung.

Die Funktion Move setzt den aktuellen Zeichenstift an eine neue Position, ohne jedoch dabei zu zeichnen. Dies ist so, als würde man einen echten Bleistift von einem Blatt Papier anheben und ihn an eine andere Stelle setzen, um dort weiterzuzeichnen. Draw zeichnet eine Linie im gewählten Zeichenmodus von der aktuellen Position des Zeichenstiftes bis zu dem Punkt, der ihr in Form einer xy-Koordinate übergeben wurde.

Die meisten Zeichenfunktionen verändern die Position des Zeichenstiftes innerhalb des aktuellen RastPort. Wenn Sie feststellen wollen, wo sich der Zeichenstift gerade befindet, dann können Sie die Variablen "cp_x" und "cp_y" aus der Datenstruktur des RastPort auslesen:

```
AktuelleXPosition = rp->cp_x;
AktuelleYPosition = rp->cp_y;
```

Das Listing 4.4 enthält die Funktion zum Zeichnen der Koordinatenachsen, wobei die Werte der zuvor initialisierten "XYBase"-Datenstruktur verwendet werden.

Als Parameter wird der Funktion DrawAxes ein Zeiger auf diese Struktur übergeben sowie ein Zeiger auf den verwendeten RastPort, die Zeichenfarbe und Position und Länge der Achsen.

Um die Funktion einfach zu halten, wurde keine Fehlerprüfung implementiert. Ist z.B. der Wert von "yaxis" minus "ylength" negativ, dann liegt ein Fehler vor, der jedoch nicht abgefangen wird. Beachten Sie außerdem, daß DrawAxes eine von uns definierte Funktion ist und nicht etwa vom Betriebssystem zur Verfügung gestellt wird.

```
/* drawaxes.c */

DrawAxes(rp,xyb,xaxis,yaxis,xlength,ylength,color)
struct RastPort *rp;
struct XYBase *xyb;
long xaxis,yaxis,xlength,ylength,color;
{
  SetAPen(rp,color);

  /* Zeichnen der x-Achse */

  Move(rp,xaxis,yaxis);
  Draw(rp,xaxis+xlength,yaxis);

  /* Zeichnen der y-Achse */

  Move(rp,xaxis,yaxis);
  Draw(rp,xaxis,yaxis-ylength);

  /* Setzen der XYBase-Strukturkomponenten, damit sie
     von anderen Routinen verwendet werden können */

  xyb->xaxis = xaxis;
  xyb->yaxis = yaxis;
  xyb->xlength = xlength;
  xyb->ylength = ylength;
}
```

Listing 4.4: Die Routine zum Zeichnen der Koordinatenachsen

Bezeichnen der Achsen

Um die Achsen mit Bezeichnungen versehen zu können, müssen Sie wissen, wie die Textausgabe auf dem Amiga gehandhabt wird. Zur Textausgabe muß der RastPort angegeben werden, in dem der Text dargestellt werden soll. Weiterhin wird ein Zeiger auf eine Zeichenkette benötigt, die den Text enthält sowie die Angabe der Textlänge.

Um den Text zu positionieren, wird auch hier die Move-Funktion verwendet, die den Zeichenstift an eine neue Stelle im RastPort setzt. Wichtig hierbei ist, daß sich diese Position immer auf die "Grundlinie" des Textes bezieht. Für den Standardzeichensatz des Amiga, in dem Buchstaben in einer 8*8 großen Matrix dargestellt werden, ist die Grundlinie die siebte Linie von oben, bezogen auf das 8*8 große Quadrat. Um die Grundlinie des aktuellen Zeichensatzes festzustellen, kann man die Variable "TxBaseline" aus der "RastPort"-Datenstruktur auslesen:

```
AktuelleGrundlinie = rp->TxBaseline;
```

Wurde ein Zeichen ausgegeben, dann wird der Zeichenstift automatisch neu gesetzt, so daß der nächste Buchstabe rechts neben dem vorherigen ausgegeben wird.

Wenn Sie den Zeichenstift positioniert und den Zeichenmodus und die Zeichenfarbe gewählt haben, dann können Sie mit der Funktion Text Ihre Zeichenkette, die den Text enthält, ausgeben lassen. Der Aufruf der Funktion lautet:

```
Text(rp,t_zeiger,laenge);
```

Der Parameter "rp" ist wiederum ein Zeiger auf den RastPort, in dem der Text ausgegeben werden soll. "t_zeiger" ist ein Zeiger auf die Zeichenkette, die den Text enthält, und "laenge" ist die Anzahl der Zeichen, die ausgegeben werden sollen. Der auszugebene Text kann auch direkt angegeben werden, wobei er in Anführungszeichen eingeschlossen sein muß.

Um die Koordinatenachsen zu bezeichnen, muß man den Text relativ zur Länge der zugehörigen Achse positionieren. Soll der Text zentriert werden, dann ist es nützlich, wenn man weiß, wie viele Bildpunkte (Pixel) er in Anspruch nimmt. Diesen Wert liefert die Funktion TextLength, die wie folgt aufgerufen wird:

```
laenge = TextLength(rp,t_zeiger,laenge);
```

Die Parameter, die der Funktion übergeben werden, sind dieselben, die auch bei der Text-Funktion verwendet werden.

Das Programmfragment im Listing 4.5 versieht sowohl die x-, als auch die y-Achse mit Bezeichnungen. Das Feld "labels[]" enthält dabei alle verwendeten Zeichenketten. Zur Vereinfachung und aus Gründen der Geschwindigkeit wird in der Routine mit Integerwerten gearbeitet.

```
/* labelaxes.c */

LabelHorizontal(rp,xyb,labels,howmany)
struct RastPort *rp;
struct XYBase *xyb;
char *labels[];   /* In diesem Feld werden die Zeichenketten abgelegt */
long howmany;     /* Die Anzahl verwendeter Zeichenketten */
{
  int i,labelwidth,segmentwidth,currentx,actualy,actualx;

  segmentwidth = (xyb->xlength)/(howmany-1);
  currentx = xyb->xaxis;
  actualy = xyb->yaxis+1+(rp->TxBaseline);  /* Ausgabe unter der Achse */

  for(i=0;i<howmany;i++)
        {
          labelwidth = TextLength(rp,labels[i],strlen(labels[i]));
          labelwidth/=2;    /* Text zentrieren */
          actualx = currentx+(segmentwidth*i)-labelwidth;
          Move(rp,actualx,actualy);
          Text(rp,labels[i],strlen(labels[i]));
        }
}  /* Ende der Funktion 'LabelHorizontal' */

LabelVertical(rp,xyb,labels,howmany)
struct RastPort *rp;
struct XYBase *xyb;
char *labels[];  /* In diesem Feld werden die Zeichenketten abgelegt */
long howmany;    /* Die Anzahl verwendeter Zeichenketten */
{
  int i,labelwidth,segmentheight,currentx,actualy,actualx,currenty;

  segmentheight = (xyb->ylength)/(howmany-1);
  currentx = xyb->xaxis-2;  /* 2 Pixel neben der Achse */

  /* Zum vertikalen Zentrieren des Textes verwenden wir die Höhe
     der Zeichen, die in der RastPort-Datenstruktur abgelegt ist. */

  currenty = xyb->yaxis+((rp->TxHeight)/2);
  for(i=0;i<howmany;i++)
        {
          labelwidth = TextLength(rp,labels[i],strlen(labels[i]));
          actualx = currentx-labelwidth;
          actualy = currenty-(segmentheight*i);
          Move(rp,actualx,actualy);
          Text(rp,labels[i],strlen(labels[i]));
        }
}  /* Ende der Funktion 'LabelVertical' */
```

Listing 4.5: Die Routine zum Bezeichnen der Koordinatenachsen

Es gibt zwei Gründe, warum man Zeichenketten immer komplett und nicht zeichenweise ausgeben lassen sollte. Zum einen wird beim wiederholten Aufrufen einer Funktion Rechenzeit und Speicherplatz des Stapelspeichers (Stack) beansprucht.

Zum anderen nimmt das System bei Zeichenketten automatisch eine Zentrierung der einzelnen Zeichen vor, so daß die Abstände der Zeichen zueinander immer gleich sind. Geben Sie eine Zeichenkette jedoch Zeichen für Zeichen aus, dann sind die Abstände zwischen den Zeichen – abhängig von der Breite eines Zeichens – unregelmäßig; das Schriftbild sieht in einem solchen Fall "unsauber" aus. Dies gilt besonders dann, wenn Sie spezielle Darstellungsmodi für die Textausgabe verwenden, z.B. Fett- oder Kursivschrift.

Zeichnen von Rechtecken

Balkendiagramme werden aus Rechtecken (den "Balken") gebildet, die entweder nur aus Linien bestehen oder mit einer Farbe oder einem Muster gefüllt sind, damit man sie voneinander unterscheiden kann. Im folgenden lernen Sie Funktionen kennen, mit denen man derartige Rechtecke erstellen kann.

Die Routine zum Zeichnen der Rechtecke verwendet ebenfalls die Komponenten der "XYBase"-Datenstruktur, so daß ihre Koordinaten relativ zum Ursprung des Koordinatensystems angegeben werden können. Ein Anwender kann so Balkendiagramme erstellen, als würde er sie auf Papier zeichnen.

Rechtecke, die nur aus Linien zusammengesetzt sind

Diese Art Rechtecke kann auf einfache Weise erstellt werden, indem man einmal die Funktion Move und viermal die Funktion Draw verwendet. Da die Arbeitsweise dieser beiden Funktionen bereits an früherer Stelle erklärt wurde, wird hier auf ein Beispiel verzichtet.

Mit Farben oder Mustern gefüllte Rechtecke

Das Betriebssystem des Amiga stellt eine Funktion zur Verfügung, mit der gefüllte Rechtecke auf einfache Art und Weise erzeugt werden können: RectFill. Der Aufruf der Funktion lautet:

```
RectFill(rp,xmin,ymin,xmax,ymax);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, in dem gezeichnet werden soll. "xmin" und "ymin" bilden zusammen die linke obere Ecke des Rechtecks; "xmax" und "ymax" bezeichnen die rechte untere Ecke des Rechtecks.

Auf welche Art und Weise das Rechteck gezeichnet wird, hängt vom Inhalt der relevanten Komponenten der "RastPort"-Datenstruktur ab, also vom Zeichenmodus, Zeichenmuster und den Farben der APen-, BPen- und OPen-Zeichenstifte. Für ein einfarbig ausgemaltes Rechteck können Sie folgendes Fragment verwenden, bevor Sie RectFill aufrufen:

```
SetAPen(rp,farbe); /* Auswahl der Zeichenfarbe */
SetDrMd(rp,JAM1);  /* JAM1, da nur eine Farbe verwendet wird */
```

Wollen Sie das Rechteck zusätzlich mit einer Linie umranden lassen, dann fügen Sie noch die folgende Zeile vor dem Aufruf von RectFill ein:

```
SetOPen(rp,farbe_der_umrandung);    /* Farbauswahl */
```

Da das System Objekte automatisch umrahmt, wenn OPen einen gültigen Wert hat, muß es eine Möglichkeit geben, diese Option zu unterdrücken, da eine Umrandung ja nicht immer erwünscht ist. Umrandet wird sowohl bei der Funktion RectFill als auch bei Funktionen, die beliebige Grafikbereiche mit einer Farbe oder einem Muster füllen (nähere Informationen hierzu finden Sie später in diesem Kapitel bei den "Custom-Screens").

Zum Abschalten der automatischen Umrahmung wird das System-Makro "BNDRY_OFF" verwendet:

```
    BNDRY_OFF(rp);
```

Rechtecke mit zweifarbigen Mustern

Für diese Art Rechteck benötigen Sie den APen- und den BPen-Zeichenstift, den Zeichenmodus "JAM2" sowie ein Muster, mit dem das Rechteck gefüllt werden soll.

Ein solches Muster wird aus 16-Bit-Worten zusammengesetzt, die dann von den Füll-Funktionen verwendet werden. Die Komponenten dieses Datenfeldes werden von oben nach unten verwendet, d.h. das erste 16-Bit-Wort bildet die erste Linie des Musters. Ein solches Füllmuster kann beliebig groß sein, die

Gesamtanzahl von 16-Bit-Worten muß jedoch immer ein Ergebnis der Funktion x^2 sein (z.B. 2,4,8,16...).

In Listing 4.6 finden Sie ein Beispiel für ein selbsterstelltes Muster. Bei seiner Verwendung werden Objekte mit einem Schachbrettmuster gefüllt. Wird beim Füllen der "JAM2" Zeichenmodus verwendet, dann wird an jeder Stelle des Musters, an der ein Bit den Wert Eins hat, die Zeichenfarbe vom APen-Zeichenstift verwendet. Hat ein Bit den Wert Null, kommt an dieser Stelle die Farbe vom BPen zum Einsatz. Wird als Zeichenmodus "JAM1" verwendet, dann wird das Muster von den Zeichenfunktionen ignoriert; es wird nur mit dem APen-Zeichenstift gemalt.

Damit das System weiß, welches Muster verwendet werden soll, wird die Funktion SetAfPt aufgerufen. Die Funktion benötigt als Parameter einen Zeiger auf den RastPort, in dem gezeichnet werden soll, sowie einen Zeiger auf das Muster und die Größe des Datenfeldes, in dem das Muster enthalten ist. Als Größe wird der Exponent einer Exponentialfunktion zur Basis 2 angegeben. Das Muster im Listing 4.6 enthält acht Felder. Als Größe wird daher der Wert 3 angegeben (2^3).

Verwenden Sie folgendes Fragment, um ein Rechteck zu erstellen, das mit einem Muster gefüllt ist:

```
SetAPen(rp,zeichenfarbel);
SetBPen(rp,zeichenfarbe2);
SetAfPt(rp,mein_muster,groesse_des_musters);  /* Auswahl des Musters */
SetDrMd(rp,JAM2);      /* Zeichenmodus selektieren */
RectFill(rp,xmin,ymin,xmax,ymax);     /* Rechteck zeichnen */
```

```
/* mypattern.h */

UWORD mypattern[] =
{
  0xf0f0,0xf0f0,0xf0f0,0xf0f0,    /* Jeweils 4 Bits mit
                                     dem Wert 0 bzw. 1 */
  0x0f0f,0x0f0f,0x0f0f,0x0f0f
};
```

Listing 4.6: Beispiel für ein selbsterstelltes Muster

Rechtecke mit mehrfarbigen Mustern

Sie können auf dem Amiga Rechtecke erstellen, die mit mehrfarbigen Mustern gefüllt sind. Dabei hängt die Anzahl Farben, die verwendet werden können, von den verfügbaren Farbregistern ab. In unserem Beispiel können wir maximal vier Farben gleichzeitig zum Füllen verwenden, da unser Fenster auf dem Workbench-Screen geöffnet wird, der ja nur vier Farben zur Verfügung stellt.

Sie können jedoch alle 4096 Farbtöne zum Füllen benutzen, wenn Sie im sogenannten "Hold-And-Modify (HAM)"-Modus arbeiten. Im Anhang B des "Amiga Programmer's Handbook, Teil 1" (SYBEX 1987) finden Sie alle Darstellungsmodi des Amiga erklärt. Die Farben, die auf dem Bildschirm des Amiga zu sehen sind, werden von einer Anzahl Bits geformt, die intern im Speicher verwaltet werden: den "Bitplanes".

Je nachdem, wie diese Bits miteinander kombiniert werden, wird eine andere Farbe zum Zeichnen eines Bildpunktes (Pixel) verwendet. Für ein mehrfarbiges Muster (d.h. mehr als zwei Farben) wird eine mehrlagige Bitplane verwendet. "Mehrlagig" ist bildlich zu sehen, so, als lägen die einzelnen Bitplanes übereinander. Die Summe aller Bits, die in den verschiedenen Bitplanes dieselbe Position haben, entscheidet darüber, welches Farbregister verwendet wird. Für eine "zweilagige" Bitplane sieht das Ganze so aus:

Bit aus Bitplane 1		Bit aus Bitplane 2	Verwendetes Farbregister
0	+	0	0
0	+	1	1
1	+	0	2
1	+	1	3

Hier ein Beispiel für ein farbiges Streifenmuster:

Bitposition im Datenfeld des Musters

```
76543210

01020301
01020301
10203010
10203010
02030102
02030102
20301020
20301020
```

```
/* multipat.h */

UWORD mymulti[] =
{
  0x3033,0x3033,0xc0cc,0xc0cc, /* Bitplane 1 des Musters */
  0x0330,0x0330,0x0cc0,0x0cc0,

  0x0330,0x0330,0x0cc0,0x0cc0, /* Bitplane 2 des Musters */
  0x3003,0x3003,0xc00c,0xc00c
}
```

Listing 4.7: Datenfeld für ein mehrfarbiges Muster

Im Listing 4.7 finden Sie ein Programmfragment, das ein mehrfarbiges Muster definiert. Damit das System weiß, daß Sie ein mehrfarbiges Muster verwenden wollen, können Sie folgendes Programmfragment benutzen:

```
SetAPen(rp,255);     /* Zeichnen in alle verfügbaren Bitplanes */
SetBPen(rp,0);       /* Muß 0 sein */
SetDrMd(rp,JAM2);    /* Zeichenmodus festlegen */
SetAfPt(rp,meine_farben,-3);    /* -3 repräsentiert die Anzahl
                                   der 16-Bit-Worte, die pro
                                   Bitplane verwendet werden */
```

Wichtig ist, daß die Anzahl Bitplanes (geformt aus 16-Bit-Worten) des Musters mit der verfügbaren Anzahl Bitplanes des Screens übereinstimmt. Die Workbench stellt vier Farben zur Verfügung; es müssen also zwei Bitplanes verwendet werden. Der Wert, der als letzter Parameter beim Aufruf der SetAfPt-Funktion übergeben wird, ist –3, da pro Bitplane des Musters 8 (2^3) 16-Bit-Worte verwendet werden.

Allgemeines über Muster

Obwohl die Möglichkeit des Amiga, Flächen mit Mustern zu füllen, recht einfach zu handhaben ist und eindrucksvolle Ergebnisse liefert, gibt es auch hier einen kleinen Nachteil: Flächen werden immer von dem Punkt an gefüllt, der der linken oberen Ecke des zugehörigen RastPorts am nächsten liegt. Aus die-

sem Grund sieht es manchmal so aus, als wären Flächen, die mit einem Muster gefüllt sind, aus dem Hintergrund "ausgeschnitten" worden, wenn dieser ein anderes Muster hat. Dieser Nachteil wiegt besonders schwer, wenn mit Hilfe der Füllfunktionen bewegte Grafik erzeugt werden soll, indem man Objekte in schneller Folge mit wechselnden Mustern füllt, die sich vom Hintergrund abheben.

Ein Beispiel hierzu: Sie lassen zwei Spielkarten zeichnen, die beide das gleiche Muster auf der Rückseite haben. Jetzt wollen Sie den Eindruck erwecken, als ob eine über die andere geschoben wird. In diesem Fall müssen Sie eine große Anzahl verschiedener Muster definieren, die verwendet werden, sobald eine Karte die andere überdeckt. Weiterhin brauchen Sie zwei verschiedene Füllroutinen, solange sich die Karten nicht komplett überdecken.

Einfacher geht es, wenn Sie einen RastPort definieren, der außerhalb des verwendeten Screens liegt. Wenn Sie hier eine der Karten zeichnen, dann können Sie diese einfach in den "sichtbaren" RastPort hineinkopieren, ohne daß Sie das Muster ändern oder die neue Position der Karte wissen müssen.

Das Zeichnen der Balken relativ zum Ursprung des Koordinatensystems

Die Positionen der in unserem Beispiel verwendeten Balken werden relativ zum Ursprung des Koordinatensystems angegeben, nicht zur linken oberen Ecke des zugehörigen Fensters.

Im Listing 4.8 finden Sie ein weiteres Modul aus dem Beispielprogramm. Es nimmt diese Umwandlung der Positionswerte vor und "übersetzt" auch die übergebenen Parameter in solche Werte, die von der Funktion RectFill "verstanden" werden.

Vor dem Aufruf der Funktion DrawBar müssen APen, BPen, der Zeichenmodus und das Füllmuster bereits definiert worden sein. Sie können das Beispielprogramm natürlich beliebig erweitern, indem Sie z.B. die Länge eines Balkens prozentual zum größten y-Wert angeben.

```
/* drawbar.c */

DrawBar(rp,xyb,x,width,height)
struct RastPort *rp;
struct XYBase *xyb;
long x,width,height; /* Position und Größe eines Balkens */
{
  int xmin,ymin,xmax,ymax;

  /* Die Basis eines jeden Balkens ist die x-Achse
     APen,BPen,Zeichenmodus und Füllmuster müssen definiert
      sein */

  xmin = x+xyb->xaxis; /* Position auf der x-Achse */
  xmax = xmin+width-1;

  ymax = xyb->yaxis-1; /* x-Achse ist die Basis des Balkens */
  ymin = ymax-height+1;

  RectFill(rp,xmin,ymin,xmax,ymax);
}
```

Listing 4.8: Die Routine zum Zeichnen eines Balkens

Punktierte Linien

Die Koordinatenachsen in unserem Beispiel bestehen aus durchgezogenen Linien. Der Amiga bietet jedoch die Möglichkeit, auch Linien in einem bestimmten Muster zu zeichnen, so wie man auch Flächen mit einem bestimmten Muster füllen kann. Zum Zeichnen von Linien unter Verwendung eines bestimmten Musters dient die Funktion SetDrPt. Anders als bei den Flächen wird bei Linien das Muster aus nur einem 16-Bit-Wort geformt. Für eine punktierte Linie sieht ein solches 16-Bit-Wort so aus (binär):

1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0

Wird beim Zeichnen der Linie der Zeichenmodus "JAM1" verwendet, dann kommt an allen Stellen, wo ein Bit des Musters den Wert Eins hat, die Farbe von APen zum Einsatz. An den Stellen, wo ein Bit des Musters den Wert Null

hat, bleibt der Hintergrund, vor dem die Linie gezeichnet wird, unverändert. Beim Modus "JAM2" führen alle gesetzten Bits des Musters zur Verwendung von APen, alle Bits mit dem Wert Null haben den Einsatz von BPen zur Folge. Der hexadezimale Wert des oben gezeigten Beispiels lautet 0xcccc. Wird kein spezielles Muster zum Zeichnen einer Linie angegeben, dann verwendet das System den voreingestellten Wert 0xffff, mit dem durchgezogene Linien gezeichnet werden, da ja alle Bits gesetzt sind.

Zeichnen von mehreren Linien mit einem einzigen Funktionsaufruf

Im Hauptprogramm, das im Anschluß an diesen Abschnitt folgt, werden punktierte Linien verwendet, um zwei benachbarte Balken miteinander zu verbinden. Eine durchgezogene Linie bildet eine weitere Verbindung. Das Zeichnen dieser Linien wird mit einem Aufruf der Funktion PolyDraw durchgeführt, mit der mehrere Linien, die miteinander verbunden sind, gezeichnet werden können. Aufgerufen wird die Funktion PolyDraw wie folgt:

```
PolyDraw(rp,xy_tabelle,anzahl_koordinaten);
```

Der Parameter "rp" ist wiederum ein Zeiger auf den RastPort, in dem gezeichnet werden soll, "xy_tabelle" ist ein Datenfeld, in dem die Koordinatenpaare enthalten sind, und "anzahl_koordinaten" ist die Anzahl der in diesem Feld vorhandenen Koordinatenpaare.

Das Hauptprogramm

Im Listing 4.9 finden Sie das letzte Programmfragment, das zusammen mit allen besprochenen Modulen in diesem Kapitel das komplette Beispielprogramm zum Zeichnen eines Balkendiagramms ergibt. Beachten Sie bitte, daß alle vorhergegangenen Programmfragmente durch den Befehl "include" eingebunden werden; dadurch wird dieses Fragment um ein Vielfaches kürzer gehalten.

```
/* Zeichnen von Balkendiagrammen --- Hauptprogramm */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/gfxbase.h"

#define NORMALFLAGS (WINDOWSIZING|WINDOWDRAG|WINDOWCLOSE|WINDOWDEPTH)

#include "mypattern.h"   /* Einbinden der einzelnen Module */
#include "multipat.h"
#include "xybase.h"
#include "window1.h"
#include "event1.c"
#include "drawaxes.c"
#include "drawbar.c"
#include "labelaxes.c"

struct IntuitionBase   *IntuitionBase;
struct GfxBase   *GfxBase;
struct Window   *w;
struct RastPort   *rport;
struct IntuiMessage   *msg;

main()
{
  long result;

  if(!(GfxBase = (struct GfxBase *)
     OpenLibrary("graphics.library",0)))
       {
         printf("graphics.library nicht geöffnet!\n");
         exit(10);
        }

  if(!(IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library",0)))
       {
         printf("intuition.library nicht geöffnet!\n");
         exit(15);
        }
  if(!(w = OpenWindow(&MeinFenster)))
       {
         printf("Fenster nicht geöffnet!\n");
         CloseLibrary(IntuitionBase);
         CloseLibrary(GfxBase);
         exit(20);
        }
```

Listing 4.9: Beispielprogramm zum Zeichnen eines Balkendiagramms (Teil 1)

```
  rport = w->RPort;
  redraw();    /* Zeichnen des Balkendiagramms */

  /* Nach dem Zeichnen wird auf eine Nachricht von Intuition
     gewartet. Die Funktion "Wait" führt dazu, daß unser Task
     schläft, solange keine Nachricht am IDCMP anliegt. */

  WaitPort(w->UserPort);    /* Auf Nachricht warten */

  while(1)     /* "Endlose" while-Schleife */
  {
    msg = (struct IntuiMessage *)GetMsg(w->UserPort);

    handle_it:
     result = HandleEvent(msg->Class);  /* Art der Nachricht
                                           feststellen */

     if(!result) break;    /* Fenster wurde geschlossen -
                              Schleife verlassen */

     /* Da mehrere Nachrichten zugleich von Intuition kommen
        können, muß der MessagePort vor jeder neuen Überprüfung
        komplett ausgelesen werden. Die folgende Zeile erledigt
        dies. */

     msg = (struct IntuiMessage *)GetMsg(w->UserPort);

     if(msg) goto handle_it;   /* msg ist 0, wenn keine
                                  Nachrichten da sind */
    }

  CloseWindow(w);
  CloseLibrary(GfxBase);
  CloseLibrary(IntuitionBase);
}   /* Ende von main() */

char *hlabels[] = { " ","84","85","86","87" };
char *vlabels[] = { "0","10","20","30","40" };

struct XYBase myxyb;

redraw()
{
  int i;

  DrawAxes(rport,&myxyb,35,120,350,100,1);
```

Listing 4.9: Beispielprogramm zum Zeichnen eines Balkendiagramms (Teil 2)

```
  LabelHorizontal(rport,&myxyb,hlabels,5);
  LabelVertical(rport,&myxyb,vlabels,5);

  SetAPen(rport,1);
  SetDrMd(rport,JAM1);

  for(i=1;i<5;i++) DrawBar(rport,&myxyb,-31+i*348/4,20,i*12);

  SetBPen(rport,2);
  SetAfPt(rport,mypattern,3);   /* Zweifarbiges Muster */
  SetOPen(rport,3);   /* Mit Farbe aus Register 3 umranden */
  SetDrMd(rport,JAM2);

  for(i=1;i<5;i++) DrawBar(rport,&myxyb,-10+i*348/4,20,i*18);

  SetAPen(rport,255);
  SetBPen(rport,0);
  SetAfPt(rport,mymulti,-3);   /* Zweifarbiges Muster */
  SetOPen(rport,1);   /* Mit Farbe aus Register 1 umranden */

  for(i=1;i<5;i++) DrawBar(rport,&myxyb,11+i*348/4,20,i*22);
}   /* Ende der Funktion redraw() */
```

Listing 4.9: Beispielprogramm zum Zeichnen eines Balkendiagramms (Schluß)

Verhindern, daß der Fensterinhalt neu gezeichnet wird

In unserem Beispielprogramm verwenden wir ein Fenster vom Typ "SIMPLE_REFRESH". Bei dieser Art Fenster wird von seinem Inhalt keine Kopie im Speicher gehalten, d.h. bei jedem Verkleinern des Fensters gehen Teile des Inhalts verloren, wenn es anschließend wieder vergrößert wird. Gleiches gilt, wenn andere Fenster es teilweise überdeckt haben, oder wenn es in den Hintergrund "geklickt" und dann wieder in den Vordergrund geholt wurde. In allen diesen Fällen muß die Funktion redraw, die im Beispielprogramm definiert wird, aufgerufen werden, um den Fensterinhalt wiederherzustellen.

Es ist jedoch umständlich, sich jedesmal eine eigene Funktion zu definieren, die den Fensterinhalt wiederherstellt. Der Amiga stellt daher noch zwei weitere Fenstertypen zur Verfügung: "SMART_REFRESH" und "SUPER_BITMAP".

Fenster vom Typ "SMART_REFRESH"

Wählen Sie diesen Typ für Ihr Fenster, dann werden verdeckte Teile des Fensters vom System automatisch wiederhergestellt, sobald sie wieder sichtbar werden. Mit diesem Typ Fenster sind allerdings auch einige Bedingungen verbunden:

- Da im Extremfall eine komplette Kopie des Fensterinhalts im Speicher gehalten werden muß (nämlich dann, wenn das Fenster komplett verdeckt wird), muß genügend Speicherplatz zur Verfügung stehen.

- Beim Zeichnen in einem Smart-Refresh-Fenster wird auch in den Teilen des Fensters gezeichnet, die verdeckt sind. Werden die verdeckten Teile wieder sichtbar, dann brauchen sie nicht wiederhergestellt zu werden. Diese Vorgehensweise des Amiga hat Vor- und Nachteile.

  Ein Nachteil ist, daß der Vorgang des Zeichnens in den Teilen, die von anderen Objekten verdeckt sind, länger dauert. Da der Amiga jedoch ein "schneller" Computer ist, werden Sie keinerlei Unterschied zwischen einem Smart-Refresh- und Simple-Refresh-Fenster bemerken, wenn es darum geht, den Fensterinhalt erneut zu zeichnen.

- Wenn Sie ein Smart-Refresh-Fenster verkleinern, dann werden nur die Teile des Inhalts zwischengespeichert, die vor der Verkleinerung sichtbar waren. Wird das Fenster wieder vergrößert, dann kann es passieren, daß "leerer Raum" im Fenster eingefügt wird; nämlich dann, wenn Sie das Fenster größer machen, als es ursprünglich war.

  Sie sehen also: Auf eine Nachricht von Intuition des Typs "REFRESHWINDOW" müssen Sie nicht reagieren, wenn Sie ein Smart-Refresh-Fenster verwenden. Kommt jedoch eine Nachricht vom Typ "NEWSIZE" am IDCMP an, dann sollten Sie prüfen, ob die neuen Dimensionen des Fensters größer als die alten sind, um eventuell eine eigene Routine zur Ausführung zu bringen, die den Fensterinhalt neu zeichnet.

Um den Fenstertyp in unserem Beispielprogramm zu ändern, brauchen Sie nur die symbolische Konstante "SIMPLE_REFRESH" in der NewWindow-Datenstruktur durch "SMART_REFRESH" zu ersetzen. Damit wird dann auch "REFRESHWINDOW" im "IDCMP"-Datenfeld überflüssig.

Wollen Sie weiterhin verhindern, daß der Fensterinhalt bei einer Nachricht vom Typ "NEWSIZE" neu gezeichnet wird, löschen Sie einfach die symbolische Konstante "WINDOWSIZING" aus der Definition der Konstanten "NORMALFLAGS".

Ihr Fenster erhält vom System dann kein Größen-Gadget, kann also nicht verkleinert oder vergrößert werden. Die Variablen der "NewWindow"-Datenstruktur, die die minimale bzw. maximale Größe des Fensters bestimmen, werden in diesem Fall vom System ignoriert.

Fenster vom Typ "SUPER_BITMAP"

Wollen Sie jedoch weiterhin Ihr Fenster vergrößern oder verkleinern können, ohne auf Nachrichten der Art "REFRESHWINDOW" oder "NEWSIZE" achten zu müssen, dann können Sie ein Super-BitMap-Fenster verwenden.

Bei Fenstern dieser Art können Sie Ihren eigenen Grafikbereich definieren, der dann vom System anstatt des "normalen" RastPort benutzt wird. Alle Grafikoperationen beziehen sich bei einem solchen Fenster ausschließlich auf Ihren Grafikbereich (BitMap).

Werden Teile Ihres Fensters verdeckt oder wird es in den Hintergrund "geklickt", dann müssen vom System keinerlei Maßnahmen zum Neuzeichnen des Fensterinhalts getroffen werden, da Ihre BitMap ja komplett im Speicher vorhanden ist. Aus diesem Grund können Sie Nachrichten vom Typ "NEWSIZE" oder "REFRESHWINDOW" ignorieren.

Bei diesem Fenstertyp müssen Sie natürlich auch wieder einige Dinge beachten:

- Ein Super-BitMap-Fenster beansprucht – genau wie ein Smart-Refresh-Fenster – viel mehr Speicherplatz als ein Fenster vom Typ Simple-Refresh. Weiterhin müssen Sie wissen, wie eigene BitMaps initialisiert und in das Grafiksystem eingebunden werden.

- Wie beim Smart-Refresh-Fenster wird auch bei einem Super-BitMap-Fenster mehr Zeit zum Zeichnen benötigt, wenn Teile des Fensters von anderen Objekten überdeckt werden.

Ein Vorteil dieses Fenstertyps ist, daß der Grafikbereich 1024*1024 Bildpunkte groß sein kann. Durch das Fenster sieht man also immer nur einen Ausschnitt des Grafikbereichs, der sich jedoch – abhängig davon, wohin man das Fenster schiebt – ändert. Beim ersten Öffnen des Fensters stimmt die linke obere Ecke Ihrer BitMap mit der linken oberen Ecke des Fensters überein.

Ein Super-BitMap-Fenster muß immer vom Typ "GIMMEZEROZERO" sein, da Intuition sonst die Umrandung des Fensters und die Gadgets in Ihrer BitMap ablegt. Wenn Sie das Fenster vergrößern, verkleinern oder verschieben, dann bleiben diese Teile des Fensters immer am selben Platz.

Im Listing 4.10 finden Sie alle notwendigen Datenstrukturen und Initialisierungen, die aus dem Fenster in unserem Beispielprogramm ein Super-BitMap-Fenster machen. Fügen Sie einfach diese Fragmente vor dem Aufruf der Funktion OpenWindow ein. Die ersten drei Zeilen des Listings 4.10 fügen Sie nach den Variablendefinitionen im Modul main() ein.

In der NewWindow-Datenstruktur wird weiterhin die symbolische Konstante "SIMPLE_REFRESH" durch "SUPER_BITMAP" ersetzt. Im "IDCMP"-Datenfeld können die Konstanten "NEWSIZE" und "REFRESHWINDOW" gelöscht werden.

Nach dem Kompilieren des Programms wird ein kleines Fenster geöffnet, das einen Ausschnitt des Balkendiagramms zeigt. Beim Vergrößern des Fensters wird ein größerer Ausschnitt sichtbar, ohne daß das System die neuen Teile wiederholt zeichnen müßte.

Wie bereits erwähnt, stimmen die linken oberen Ecken des Fensters und der BitMap beim Öffnen des Fensters überein. Listing 4.11 zeigt ein weiteres Fragment, mit dessen Hilfe die BitMap kontinuierlich "unter" dem Fenster verschoben (gescrollt) wird.

Die Funktion "ScrollWindow" aus Listing 4.11 verwendet eine Funktion aus der Layers-Funktionsbibliothek. Aus diesem Grund muß diese Bibliothek zuvor geöffnet werden, bevor die Funktion verwendet werden kann. Fügen Sie daher am Anfang des Programms folgende Zeile ein:

```
struct LayersBase   *LayersBase;   /* Definition der Basisadresse */
```

```
/* Deklarationen für ein Super-BitMap Fenster (Fragmente) */

struct BitMap meineBitMap;
extern PLANEPTR AllocRaster();
ULONG *m;

MeinFenster.Width = 120;     /* Neue Fenstergröße */
MeinFenster.Height = 40;

/* Initialisierung der BitMap mit der Größe des Workbench-Screens */

InitBitMap(&meineBitMap,2,640,200);
                                /* Anzahl BitPlanes, Breite, Höhe */

/* Speicher für die BitMap reservieren */

m = AllocRaster(640,200);
if(!m)
   {
     printf("Nicht genügend Speicher für SuperBitmap vorhanden!\n");
     exit(30);
   }

meineBitMap.Planes[0] = m;

m = AllocRaster(640,200);
if(!m)
   {
     printf("Nicht genügend Speicher für SuperBitmap vorhanden!\n");
     FreeRaster(meineBitMap.Planes[0]);
     exit(30);
   }


meineBitMap.Planes[1] = m;   /* Ende der Speicherreservierung */


/* Nach dem erfolgreichen Initialisieren der BitMap wird der
   NewWindow-Datenstruktur ein Zeiger auf die BitMap übergeben */

MeinFenster = &meineBitMap;
```

Listing 4.10: Programmfragmente für ein Super-BitMap-Fenster

```
/* scrollwindow.c */

ScrollWindow(wi,dx,dy)
struct Window *wi;
SHORT dx,dy;
{
  struct RastPort *ra;
  struct LayerInfo *li;
  struct Layer *l;

  if(ra = wi->RPort)  /* RastPort des Fensters "ausfindig" machen */
     {
       if(l = ra->Layer)     /* Zeiger auf einen Layer */
          {
            if(li = l->LayerInfo)  /* Zeiger auf LayerInfo */
               ScrollLayer(li,l,dx,dy);
           }
        }
}    /* Ende der Funktion */
```

Listing 4.11: Die ScrollWindow-Routine

Nachdem die Bibliotheken "intuition.library" und "graphics.library" im Hauptprogramm geöffnet wurden, wird nun die "layers.library" ebenfalls geöffnet:

```
if(!(LayersBase = (struct LayersBase *)
   OpenLibrary("layers.library",0)))
     {
       printf("layers.library nicht geöffnet!\n");
       CloseLibrary(IntuitionBase);
       CloseLibrary(GfxBase);
       exit(40);
     }
```

Fügen Sie dieses Fragment an der entsprechenden Stelle im Hauptprogramm ein. Am Ende des Programms muß die Bibliothek natürlich wieder geschlossen werden, um den reservierten Speicherplatz wieder freizugeben. Die folgende Zeile wird am Ende des Hauptprogramms eingefügt, da, wo auch die "intuition.library" und die "graphics.library" geschlossen werden:

```
CloseLibrary(LayersBase);
```

Die Funktion ScrollLayer verschiebt zwei Grafikbereiche gegeneinander. Werden der Funktion Werte übergeben, die den maximalen Bereich, der zum

Verschieben zur Verfügung steht, übersteigen, dann ignoriert das System diese Werte und verwendet das tatsächlich erlaubte Maximum. Sie sollten einmal das folgende Programmfragment im Hauptprogramm einfügen, um das Fenster zu verschieben, so daß immer ein anderer Teil des Fensterinhalts sichtbar wird:

```
for(i=0;i<40;i++)
        {
          ScrollWindow(w,i,i);  /* Fenster diagonal verschieben */
          Delay(5);     /* 1/10 Sekunde warten */
        }

for(i=39;i>0;i--)
        {
          ScrollWindow(w,i,i);  /* Fenster zur Ausgangsposition
                                   zurück */
          Delay(5);     /* 1/10 Sekunde warten */
        }
```

Die Funktionen aus der Layers-Bibliothek werden häufig von Intuition verwendet, um Fenster zu erstellen, zu verschieben, zu vergrößern oder zu verkleinern. Dieses Buch geht nicht näher auf die Funktionen der Layers-Bibliothek ein. Wollen Sie mehr darüber erfahren, dann finden Sie im Kapitel 5 des "Amiga Programmer's Handbook, Teil 1" von Eugene P. Mortimore (SYBEX 1987) weiterführende Informationen zu diesem Thema.

Im Kapitel 5 dieses Buches befassen wir uns mit den "Proportional Gadgets" (den "Rollbalken"). Wenn Sie ein solches Gadget in unserem Beispielprogramm verwenden, dann können Sie mit der Maus bestimmen, welcher Teil des Grafikbereiches sichtbar sein soll.

Erstellen und Öffnen eines eigenen Bildschirms (Custom Screen)

In den folgenden Abschnitten werden Sie lernen, wie Sie einen eigenen "virtuellen Bildschirm" öffnen können und wie Sie mit ihm arbeiten. Der Amiga kann mehrere solcher virtuellen Bildschirme (Screens) verwalten. Einen Typ Screen haben Sie bereits kennengelernt: den Workbench-Screen. Bei der Arbeit mit der Workbench stehen Ihnen für die Ausgabe von Grafik nur vier Farben zur Verfügung. Verwenden Sie jedoch einen "Custom Screen", dann können Sie bis zu 32 Farben gleichzeitig verwenden.

In den folgenden Abschnitten werden wir wiederum ein Beispielprogramm erstellen (zum Zeichnen einer Landkarte), dessen Module einzeln besprochen werden. Um einen Custom Screen öffnen zu können, müssen dem System eine Anzahl Parameter übergeben werden:

- Das System muß wissen, wo der Screen geöffnet werden soll (wo seine linke obere Ecke ist).
- Sie können bestimmen, in welcher Auflösung (Gesamtanzahl der Bildpunkte) der Screen dargestellt werden soll.
- Jeder Screen kann einen Namen haben (in seiner "Titelzeile").
- Weiterhin können Sie bestimmen, in welchen Farben die Gadgets und die Titelleiste gezeichnet werden sollen.
- Sie können einen beliebigen Zeichensatz verwenden.
- Die Anzahl verwendeter Farben (bis zu 32) muß angegeben werden.
- Der Darstellungsmodus des Screens kann frei gewählt werden.

Definition des Custom Screens

Die oben genannten Parameter werden in einer Datenstruktur mit Namen "NewScreen" zusammengefaßt. Das Listing 4.12 enthält die Initialisierung der "NewScreen"-Struktur, die von uns später im Beispielprogramm verwendet wird.

Öffnen des Custom Screens

Um einen Custom Screen zu öffnen, wird die Funktion OpenScreen verwendet. Der Aufruf der Funktion lautet:

```
struct Screen *s;

if(!(s = OpenScreen(&MeinScreen)))
   {
    printf("Screen nicht geöffnet!\n");
    exit(10);
   }
```

```
/* myscreen1.h

Wir verwenden den Standard-Zeichensatz des Amiga
(topaz.font), der 80 bzw. 40 Zeichen (LO-RES) pro Zeile
darstellt. */

struct NewScreen MeinScreen =
{
  0,0,       /* Linke obere Ecke des Screens */
  320,200,   /* Breite und Höhe des Screens (320*200 = LO-RES) */
  5,         /* Anzahl der Bitplanes (5 = max. 32 Farben gleichzeitig) */
  1,0,       /* DetailPen, BlockPen (Vorder- und Hintergrundfarbe) */
  0,         /* Darstellungsmodus (0 = LO-RES) */
  CUSTOMSCREEN,          /* Art des Screens */
  NULL,      /* Voreingestellten Zeichensatz (topaz.font) verwenden */
  "32-farbiger Screen",   /* Name des Screens */
  NULL,      /* Dieses Feld wird vom System ignoriert */
  NULL       /* Zeiger auf eigene BitMap (wird in unserem Beispiel nicht
                verwendet, daher NULL */
};
```

Listing 4.12: Initialisierung der NewScreen-Datenstruktur

Erhalten Sie von der Funktion OpenScreen den Returnwert Null, dann wurde der Screen nicht geöffnet. Dieser Fehler tritt meistens dann auf, wenn nicht mehr genügend Speicherplatz zur Verfügung steht, um den Screen gemäß den Parametern der "NewScreen"-Datenstruktur öffnen zu können. Versuchen Sie in diesem Fall, alle "unnötigen" Tasks zu beenden, um wieder mehr Speicherplatz zu erhalten.

Öffnen eines Fensters auf dem Custom Screen

Listing 4.13 enthält eine modifizierte Version der "NewWindow"-Datenstruktur, die wir beim Zeichnen des Balkendiagramms verwendet haben.

Das nachfolgende Programmfragment paßt die nötigen Variablen der Datenstruktur unseren Bedürfnissen an. Auf diese Weise sehen Sie auf einen Blick, was in der "NewWindow"-Datenstruktur geändert werden muß, damit das Fenster auf dem Custom Screen geöffnet wird. Beachten Sie bitte, daß bei einem Fenster, das auf der Workbench geöffnet wird, der Zeiger auf den Screen

```
/* window2.h */

struct NewWindow MeinFenster =
{
  0,
  15,    /* Linke obere Ecke des Fensters relativ zur linken
            oberen Ecke des Screens (Angaben in Bildpunkten) */
  280,150,  /* Breite und Höhe des Fensters (Angaben in
               Bildpunkten) */
  0,     /* Farbregister zum Zeichnen des Rahmens des Fensters
            (DetailPen) */
  1,     /* Farbregister zum Zeichnen der Gadgets des Fensters
            (BlockPen) */
         /* Die Werte -1 verwenden jeweils die voreingestellten
            Farben */
  CLOSEWINDOW|
  NEWSIZE|
  REFRESHWINDOW,     /* IDCMP Flags */
  SIMPLE_REFRESH|
  NORMALFLAGS|
  GIMMEZEROZERO,     /* Window Flags */
  NULL,      /* Erstes Gadget des Fenster (siehe Kapitel 5) */
  NULL,      /* "CheckMark" (siehe Kapitel 5) */
  "Testfenster",   /* Name des Fensters */
  NULL,      /* Zeiger auf den zugehörigen Screen, falls nicht
                Workbench */
  NULL,      /* Zeiger auf eigene BitMap (falls
                "SuperBitMapWindow") */
  10,10,     /* Minimale Höhe und Breite des Fensters */
  320,200,   /* Maximale Höhe und Breite des Fensters */
  WBENCHSCREEN  /* Art des Screens, auf dem das Fenster
                   geöffnet wird */
};
```

Listing 4.13: Initialisierung einer "NewWindow"-Datenstruktur

den Wert Null hat. Da wir jedoch nun einen eigenen Screen verwenden, müssen wir dem System einen Zeiger übergeben, damit es weiß, auf welchem Screen das Fenster geöffnet werden soll.

```
MeinFenster.Screen = s;            /* Zeiger auf den Custom Screen */
MeinFenster.Type = CUSTOMSCREEN    /* Das Fenster soll auf einem
                                      Custom Screen geöffnet werden */

/* Hier folgt nun der Programmteil, der das Fenster öffnet und evtl. eine
Fehlernummer ausgibt, wenn es nicht geöffnet werden kann. Verwenden Sie
hierzu die bereits besprochenen Module. */
```

Tritt während der Programmausführung kein Fehler auf, dann haben Sie nun einen eigenen Screen geöffnet, auf dem dann ein Fenster gemäß Listing 4.13 geöffnet wird.

Auswahl der Farben

Nachdem Sie nun den Screen geöffnet haben, können Sie darangehen, die Farben zu definieren, die verwendet werden sollen. Anstatt die voreingestellten Farben der Workbench zu verwenden, können Sie nun Ihre eigene Farbpalette mit bis zu 32 Farben erstellen.

Die NewScreen-Datenstruktur im Listing 4.12 legt fest, daß 32 Farben verwendet werden können. Um ein einzelnes Farbregister zu initialisieren, stellt das System die Funktion SetRGB4 zur Verfügung. Wollen Sie eine komplette Farbpalette mit einem Mal in die Register laden, dann verwenden Sie hierzu die Funktion LoadRGB4. Jede dieser Funktionen verlangt als ersten Parameter einen Zeiger auf die "ViewPort"-Datenstruktur, die Teil der Screen-Datenstruktur ist. Die "ViewPort"-Datenstruktur enthält u.a. Datenfelder, die die aktuellen Werte der Farbregister beinhalten.

Um diesen Zeiger zu definieren, können Sie nachfolgendes Programmfragment benutzen:

```
struct ViewPort *vp;

vp = s->ViewPort;  /* Den ViewPort des Screens "ausfindig" machen */
```

Dieser Zeiger wird den Funktionen SetRGB4 und LoadRGB4 beim Aufruf als erster Parameter übergeben. Ein Aufruf von SetRGB4 hat die Form:

```
SetRGB4(vp,farbregister,rot_wert,gruen_wert,blau_wert);
```

Der Parameter "vp" ist der oben erwähnte Zeiger auf den ViewPort des Screens, "farbregister" ist die Nummer des Farbregisters (0 bis 31), das geändert werden soll. Die drei restlichen Parameter enthalten jeweils einen Wert von 0 bis 15, der den Rot-, Grün- und Blauanteil der Farbe festlegt. Hier nun drei typische Anwendungen der Funktion SetRGB4:

```
/* Farbregister 0 (Hintergrund) schwarz (r=g=b=0) */
SetRGB4(vp,0,0,0,0);

/* Farbregister 1 auf Rot setzen (r=15) */
SetRGB4(vp,1,15,0,0);

/* Die Farbe Weiß ins Farbregister 2 laden  (r=g=b=15)*/
SetRGB4(vp,2,15,15,15);
```

Durch das kontinuierliche Ändern der Werte eines Farbregisters kann eine Art von Bewegung hervorgerufen werden. Das Setzen der einzelnen Farbregister ist jedoch recht mühsam, besonders dann, wenn alle 32 Register neue Werte erhalten sollen.

Mit der Funktion LoadRGB4 kann eine komplette Farbpalette mit einem Funktionsaufruf in die Register geladen werden. Aufgerufen wird LoadRGB4 wie folgt:

```
LoadRGB4(vp,farbpalette,anzahl_farben);
```

Der Parameter "vp" ist wiederum der Zeiger auf den ViewPort des Screens, "farbpalette" ist ein Zeiger auf das Datenfeld, das die Farbwerte für die einzelnen Register enthält, und "anzahl_farben" ist die Gesamtanzahl aller Farbwerte in diesem Datenfeld.

LoadRGB4 beginnt beim Laden einer Farbpalette immer mit dem Farbregister Null und weist den nächsten Wert aus dem Datenfeld dann jeweils dem nächsthöheren Register zu. Jeder Farbwert für ein Register wird aus einem 16-Bit-Wort geformt, in dem immer 4 Bits den Farbanteil für Rot, Grün und Blau definieren.

Die Bits sind wie folgt angeordnet:

0000 RRRR GGGG BBBB

Die ersten vier Bits des Wortes werden vom System ignoriert. Danach folgen jeweils vier Bits für den Rot-, Grün- und Blauanteil einer Farbe.

Mit LoadRGB4 können maximal 32 Farben in den ViewPort eines Screens geladen werden. Die folgende Farbpalette enthält die Werte für 16 Farben, wobei der Farbton, den jeder Wert erzeugt, in der Kommentarzeile zu finden ist.

```
UWORD meinefarbpalette[] =
{
  0x0000,0x0e30,0x0fff,0x0b40,0x0fb0,0x0bf0,
  0x05d0,0x0ed0,0x07df,0x069f,0x0c0e,
  0x0f2e,0x0feb,0x0c98,0x0bbb,0x07df
};

/* Schwarz,Rot,Weiß,Feuerrot,Orange,Gelb, */
/* Hellgrün,Grün,Marineblau,Dunkelblau,Purpur */
/* Violett,Hellbraun,Braun,Grau,Himmelblau */
```

Der Funktionsaufruf zum Laden dieser Farbpalette lautet:

```
LoadRGB4(vp,&meinefarbpalette[0],16);
```

Dieser Aufruf wird auch später im Beispielprogramm verwendet.

Bestimmung der aktuell verwendeten Farben

Für jeden geöffneten Screen verwaltet das System eine Farbpalette der oben besprochenen Form. Wenn Sie für Ihren Screen nicht alle verwendeten Farbregister initialisieren oder mit LoadRGB4 keine vollständige Palette laden, dann verwendet das System für alle "fehlenden" Register die Farben aus einer voreingestellten Palette.

Wurde ein Screen von Intuition geöffnet (z.B. durch einen Aufruf der Funktion OpenScreen), dann können Sie ein Datenfeld aus der "ViewPort"-Datenstruktur auslesen, um den Inhalt eines Farbregisters festzustellen. Die Funktion, die hierbei verwendet wird, heißt GetRGB4 und wird folgendermaßen aufgerufen:

```
inhalt_des_farbregisters = GetRGB4(vp,register);
```

Der Parameter "vp" ist ein Zeiger auf den ViewPort des betreffenden Screens, "register" ist die Nummer des Farbregisters (0 bis 31), dessen Inhalt Sie auslesen wollen.

Erhalten Sie als Returnwert von der Funktion den Wert –1, dann enthält das angesprochene Farbregister keine gültigen Farbwerte. Im Falle des Erfolges erhalten Sie ein 16-Bit-Wort als Returnwert, bei dem die ersten vier Bits den Wert Null beinhalten. Die anderen drei 4-Bit-Paare enthalten jeweils den Rot-, Grün- und Blauanteil der im Register abgelegten Farbe.

Anwendung der Füllroutine Flood

Der Amiga stellt eine Funktion zur Verfügung, mit der beliebige Flächen mit einer Farbe gefüllt werden können, wobei der Startpunkt zum Füllen irgendwo innerhalb der betreffenden Fläche liegen kann.

Zwei Modi werden beim Füllen von Flächen unterstützt. Modus 0 füllt eine Fläche ab der Position, auf der der Zeichenstift momentan steht. Gefüllt wird so lange, bis ein Pixel dieselbe Farbe hat wie der OPen (der zum Umranden von Flächen oder Objekten dient).

Modus 1 beginnt mit dem Füllen ebenfalls an der Stelle, an der der Zeichenstift gerade steht. Von diesem Füllmodus sind alle Pixel (Bildpunkte) betroffen, die die gleiche Farbe haben wie der, bei dem mit dem Füllen begonnen wurde.

Interessant an dieser Füll-Funktion ist, daß sie beim Füllen von Flächen auch den jeweiligen Zeichenmodus ("JAM1" oder "JAM2") und ein eventuell vorhandenes Muster berücksichtigt.

Auf diese Weise können Flächen recht einfach mit beliebigen Mustern gefüllt werden, die auch mehrfarbig sein können. Bei der Anwendung der Funktion müssen Sie sicherstellen, daß die Fläche, die gefüllt werden soll, lückenlos umrandet ist. Ist die Umrandung auch nur an einer Stelle unterbrochen, dann wird die Füllfunktion außerhalb der Fläche fortgesetzt.

Ein Aufruf der Funktion Flood hat die Form:

```
Flood(rp,modus,x,y);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, in dem gezeichnet werden soll, "modus" ist der Füllmodus (0 oder 1), und "x" und "y" ist die Koordinate, an der mit dem Füllen begonnen werden soll.

Im Listing 4.14 finden Sie eine Routine, mit der "Diamanten" gezeichnet werden können.

```
/* drawdiamond.c */

DrawDiamond(rport,xcenter,ycenter,xsize,ysize)
struct RastPort *rport;
int xcenter,ycenter,xsize,ysize;
{
  BYTE oldAPen;
  int xoff,yoff;

  oldAPen = rport->FgPen;     /* Farbe von APen zwischenspeichern */

  SetAPen(rport,rport->AOlPen);  /* APen = OPen */

  xoff = xsize/2;             /* Offset relativ zum Zentrum */
  yoff = ysize/2;

  /* Zeichnen und Füllen des Diamanten */

  Move(rport,xcenter-xoff,ycenter);
  Draw(rport,xcenter,ycenter+yoff);
  Draw(rport,xcenter+xoff,ycenter);
  Draw(rport,xcenter,ycenter-yoff);
  Draw(rport,xcenter-xoff,ycenter);

  /* Füllen der Fläche */

  Flood(rport,0,xcenter,ycenter);  /* Bis zur Begrenzung der Fläche
                                      füllen */
  SetAPen(rport,oldAPen);     /* Ursprüngliche Farbe von APen
                                 herstellen */
}
```

Listing 4.14: Routine zum Zeichnen eines Diamanten

Füllen von Bereichen mit eigenen Mustern

Bei der Verwendung der Funktion Flood muß immer eine Fläche vorhanden sein, die von einer nicht unterbrochenen Linie begrenzt wird. Der Amiga stellt Ihnen jedoch einige Funktionen zur Verfügung, mit denen eine Fläche definiert und dann gefüllt werden kann, ohne daß eine Umrandung der Fläche vorhanden ist.

Der Vorteil dieser Funktionen liegt darin, daß beliebige Flächen geformt werden können, ohne daß man darauf achten muß, daß die Begrenzung dieser Flä-

che von einer Linie gebildet wird, die nicht unterbrochen sein darf. Andernfalls wird nämlich die Füllfunktion – bei der Funktion Flood – außerhalb der Fläche fortgeführt, so daß im Extremfall der komplette Bildschirm gefüllt wird. Hier nun die Funktionen, mit denen solche Bereiche erstellt und gefüllt werden können:

```
fehler = AreaMove(rp,x,y);
fehler = AreaDraw(rp,x,y);
AreaEnd(rp);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, in dem gezeichnet werden soll, "x" und "y" ist der Punkt, an dem mit dem Zeichnen begonnen bzw. zu dem gezeichnet werden soll.

Liefert AreaMove oder AreaDraw den Returnwert –1, so ist im zugehörigen Pufferspeicher kein Platz mehr zur späteren Ausführung der Funktion vorhanden. Wird der Returnwert Null geliefert, so ist kein Fehler beim Aufruf aufgetreten.

Die Funktion AreaMove verhält sich genauso wie die Funktion Move, d.h. der Zeichenstift wird "angehoben" und an einer neuen Stelle positioniert. Die Funktion AreaDraw liefert das gleiche Resultat wie die Funktion Draw, d.h. es wird eine Linie von der aktuellen Position des Zeichenstiftes zur angegebenen Endkoordinate gezeichnet.

Die Funktionen AreaMove und AreaDraw verwenden ihren "eigenen" Zeichenstift; die Position des Stiftes, der von den Funktionen Move oder Draw benutzt wird, wird nicht verändert.

Sie können mehrere Flächen gleichzeitig erstellen lassen, indem Sie mehrere Funktionsaufrufe durchführen. Beim Aufruf der Funktion AreaEnd werden alle diese Flächen gezeichnet; d.h. der eigentliche Zeichenvorgang setzt erst mit dem Aufruf dieser Funktion ein.

Rufen Sie die Funktion AreaMove auf, dann wird eine Fläche, die zuvor erstellt, jedoch nicht geschlossen wurde, automatisch geschlossen. Es wird also quasi die Funktion AreaDraw vom System aufgerufen, um eine direkte Verbindung zum Ausgangspunkt der betreffenden Fläche zu ziehen. Wenn Sie z.B. ein gefülltes Quadrat erstellen wollen, dann genügt ein Aufruf der Funktion AreaMove, um den Ausgangspunkt zu setzen, und drei Aufrufe der Funktion AreaDraw, um drei Seiten des Quadrats zeichnen zu lassen. Die vierte Seite der Fläche wird vom System automatisch gezeichnet, wenn Sie anschließend AreaMove oder AreaEnd aufrufen.

Vorbereitungen zur Arbeit mit AreaMove und AreaDraw

Da mit AreaMove und AreaDraw erstellte Flächen erst nach dem Aufruf von AreaEnd gezeichnet werden, verwaltet das System eine Liste, in der alle Bewegungen des Zeichenstiftes vermerkt sind. Nach dem Aufruf von AreaEnd wird diese Liste dann vom System abgearbeitet, d.h. die Flächen werden gezeichnet und gefüllt. Bevor aber die Funktionen verwendet werden können, müssen Sie noch einige Datenstrukturen initialisieren, damit das System mit Ihrem RastPort arbeiten kann. Initialisiert werden muß:

- Eine "AreaInfo"-Datenstruktur für den betreffenden RastPort sowie Speicherplatz für die zugehörige "AreaInfo"-Struktur.

- Eine "TmpRas"-Datenstruktur für den betreffenden RastPort sowie ein genügend großer Arbeitsspeicher für diese Struktur.

Die "AreaInfo"-Datenstruktur beinhaltet Variablen, die vom System bei der Ausführung Ihrer AreaMove- und AreaDraw-Angaben verwendet werden. Weiterhin enthält die Struktur einen Zeiger auf einen Speicherbereich, in dem weitere Funktionsaufrufe zwischengespeichert werden.

Denken Sie daran, daß erst nach dem Aufruf von AreaEnd die Flächen gezeichnet und gefüllt werden. Aus diesem Grund muß der Speicherbereich, in dem die Liste aller Positionsangaben vom System verwaltet wird, groß genug sein, um alle Einträge fassen zu können.

Dieses Datenfeld besteht aus 16-Bit-Worten. Das Feld muß fünfmal so groß sein wie die Gesamtanzahl der Aufrufe von AreaMove und AreaDraw, die ausgeführt werden sollen. Im folgenden Beispiel gehen wir davon aus, daß nicht mehr als 20 Funktionsaufrufe erfolgen, um eine Fläche zu zeichnen; das Datenfeld hat also eine Größe von 100 (20 * 5).

Initialisiert wird eine "AreaInfo"-Datenstruktur durch einen Aufruf der Funktion InitArea. Übergeben werden der Funktion ein Zeiger auf die "AreaInfo"-Datenstruktur, ein Zeiger auf das Datenfeld, in dem das System die Liste verwalten kann, und die maximale Anzahl von Funktionsaufrufen. Hier ein Beispielaufruf:

```
struct AreaInfo meinAreaInfo;
int area_datenfeld[100];    /* Maximal 20 (100/5) Funktionsaufrufe */

InitArea(&meinAreaInfo,&area_datenfeld[0],20);
rp->AreaInfo = &meinAreaInfo;  /* Einbinden in den RastPort */
```

Der Amiga erstellt die von Ihnen gewünschten Flächen zunächst in einem separaten Speicherbereich und kopiert die fertige Fläche dann in Ihren RastPort. Dieser separate Speicherbereich trägt den Namen "TmpRas" ("temporary raster area", Temporärer RastPort).

Die Größe dieses Speicherbereiches wird durch die größte Fläche festgelegt, die Sie erstellen wollen. Hat die größte Ihrer Flächen eine Breite von 640 und eine Höhe von 200 Bildpunkten, dann muß der Speicher für das TmpRas 16000 Bytes (640 Bits * 200 Bits) groß sein. Relevant bei Operationen in diesem Speicher ist nur seine physikalische Größe (Breite * Höhe der Fläche), d.h. wenn Sie einen Speicherbereich reservieren, in dem ein 100 mal 100 Pixel großes Rechteck gezeichnet werden kann, dann können Sie in demselben Bereich auch ein Rechteck der Größe 1000 * 10 zeichnen lassen.

Vor dem Aufruf der Funktion InitTmpRas zur Initialisierung der "TmpRas"-Datenstruktur muß zunächst der Speicherbereich reserviert werden. Am Schluß wird die "TmpRas"-Datenstruktur in den verwendeten RastPort eingebunden:

```
PLANEPTR arbeitsspeicher;
struct TmpRas meinTmpRas;

arbeitsspeicher = AllocRaster(640,200);
if(!arbeitsspeicher) printf("Nicht genügend Speicher vorhanden!\n");

else
      {
        InitTmpRas(&meinTmpRas,arbeitsspeicher,RASSIZE(640,200));
        rp->TmpRas = &meinTmpRas;     /* Einbinden in den RastPort */
      }
```

Der Aufruf des Makros "RASSIZE" teilt dem System mit, wieviel Speicherplatz beim Aufruf von AllocRaster reserviert wurde. Das System weiß dann, wie groß eine zu zeichnende Fläche maximal sein darf.

Bevor Sie mit Text arbeiten, sollten Sie für Ihren RastPort eine "TmpRas"-Datenstruktur definieren, da beim Fehlen dieser Struktur bei jedem Aufruf der Funktion Text Arbeitsspeicher reserviert und wieder freigegeben wird. Die "TmpRas"-Datenstruktur hat in diesem Fall eine höhere Arbeitsgeschwindigkeit des Systems zur Folge. Den Einsatz von Text und die notwendigen Erläuterungen finden Sie später in diesem Kapitel.

Lesen und Setzen beliebiger Bildpunkte (Pixels)

Manchmal ist es erforderlich, einen einzelnen Bildpunkt zu setzen bzw. die Farbe eines Bildpunktes festzustellen.

Der Amiga stellt hierfür zwei Funktionen zur Verfügung: WritePixel zum Setzen und ReadPixel zum Lesen eines Bildpunktes. Aufgerufen werden die Funktionen wie folgt:

```
WritePixel(rp,x,y);
farbe = ReadPixel(rp,x,y);
```

Der Parameter "rp" ist ein Zeiger auf einen RastPort, "x" und "y" sind die Koordinaten des Bildpunktes, der gelesen oder gesetzt werden soll.

Verwenden Sie die Funktion WritePixel, dann wird das Pixel in der Farbe gezeichnet, die aktuell dem APen-Zeichenstift zugeordnet ist. Der Returnwert der Funktion ReadPixel enthält einen Wert von 0 bis 255. Kann ein Bildpunkt nicht gesetzt oder gelesen werden (z.B. wenn seine Koordinaten außerhalb der RastPorts liegen), dann liefern beide Funktionen –1 als Returnwert.

Im folgenden Beispielprogramm, mit dem eine Landkarte gezeichnet werden kann, wird die Funktion WritePixel verwendet, um mit Hilfe des Zufallsgenerators des Amiga "Schneefall" zu erzeugen.

Zeichnen einer Landkarte

Mit dem Programm aus Listing 4.15 wird eine Landkarte der Staaten Utah, Colorado, Arizona und New Mexico gezeichnet. Jeder Staat wird in einer anderen Farbe dargestellt und mit einem zweistelligen Kürzel versehen.

Nachdem alle Staaten gezeichnet wurden, setzt in Arizona "Schneefall" ein. Um diesen Effekt zu starten, drücken Sie bitte zuerst die linke, dann die rechte Maustaste innerhalb des Fensters.

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/gfxbase.h"

#include "mydefines.h"
#include "myscreen1.h"
#include "window2.h"
#include "event1.c"
#include "drawdiamond.c"
#define AZCOLOR 1           /* Farbregister zum Zeichnen von Arizona */
#define WHITECOLOR  2       /* Und für den "Schnee" */
#define CORNERX 150
#define CORNERY 75          /* Gemeinsamer Startpunkt zum Zeichnen */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *w;           /* Zeiger auf ein Fenster */
struct RastPort *rp;
struct Screen *s;
struct ViewPort *vp;
struct AreaInfo myAreaInfo;
struct TmpRas myTmpRas;
PLANEPTR workspace;

int areaArray[100];         /* Maximal 20 (100/5) Funktionsaufrufe */

int utahxy[] = { 0,0,-40,0,-38,-70,-15,-70,-17,-55,0,-55,0,0 };
int coloradoxy[] = { 0,0,75,0,75,-55,0,-55 };
int arizonaxy[] = { 0,0,-40,0,-40,10,-50,10,-50,30,-60,55,-30,70,0,70 };
int newmexicoxy[] = { 0,0,0,70,8,70,68,70,68,0 };

UWORD mycolortable[] =
{
  0x0000,0x0e30,0x0fff,0x0b40,0x0fb0,0x0bf0,
  0x05d0,0x0ed0,0x07df,0x069f,0x0c0e,
  0x0f2e,0x0feb,0x0c98,0x0bbb,0x07df
};
/* Schwarz, Rot,Weiß,Feuerrot,Orange,Gelb,
   Hellgrün,Grün,Marineblau,Dunkelblau,Purpur,
   Violet,Hellbraun,Braun,Grau,Himmelblau */

main()
{
  struct IntuiMessage *msg;
  long result;
  int rx,ry;
  PLANEPTR workspace;
```

Listing 4.15: Programm zum Zeichnen einer Landkarte (Teil 1)

```
  if(!(GfxBase = (struct GfxBase *)
     OpenLibrary("graphics.library",0)))
        {
          printf("grphics.library nicht geöffnet!\n");   exit(0);
        }
 if(!(IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library",0)))
        {
          printf("intuition.library nicht geöffnet!\n");
          CloseLibrary(GfxBase);
          exit(1);
        }

  if(!(s = OpenScreen(&myscreen1)))
     {
       printf("Screen läßt sich nicht öffnen!\n");
       CloseLibrary(GfxBase);
       CloseLibrary(IntuitionBase);
       exit(2);
      }

  myWindow.Screen = s;    /* Zeiger auf unseren Custom Screen */
  myWindow.Type = CUSTOMSCREEN;
  myWindow.Title = "Landkarte";

  if(!(w = OpenWindow(&myWindow)))
     {
       printf("Fenster läßt sich nicht öffnen!\n");
       CloseScreen(s);
       CloseLibrary(GfxBase);
       CloseLibrary(IntuitionBase);
       exit(3);
      }

  vp = s->ViewPort;
  LoadRGB4(vp,&mycolortable[0],16);   /* Farbpalette laden */

  rp = w->RPort;
  workspace = (PLANEPTR)AllocRaster(640,200);
  if(!workspace)
     {
       printf("Nicht genügend Speicher für TmpRas vorhanden!\n");
       CloseWindow(w);
       CloseScreen(s);
       CloseLibrary(GfxBase);
       CloseLibrary(IntuitionBase);
       exit(4);
      }
```

Listing 4.15: Programm zum Zeichnen einer Landkarte (Teil 2)

```
  InitTmpRas(&myTmpRas,workspace,RASSIZE(640,200));
  rp->TmpRas = &myTmpRas;     /* In den RastPort einbinden */

  InitArea(&myAreaInfo,&areaArray[0],20);
  rp->AreaInfo = &myAreaInfo;     /* In den RastPort einbinden */

  redraw();     /* Landkarte zeichnen */

  /* Auf eine Nachricht von Intuition warten */

  WaitPort(w->UserPort);

  /* MessagePort auslesen, wenn eine Nachricht angekommen ist */

  while(1)      /* "Endlose" while-Schleife */
  {
    msg = (struct IntuiMessage *)GetMsg(w->UserPort);
    handle_it:
      result = HandleEvent(msg->Class);

      if(!result) break;  /* Fenster wurde geschlossen (Programmende) */

      msg = (struct IntuiMessage *)GetMsg(w->UserPort);
      if(msg) goto handle_it;     /* msg = 0, wenn keine
                                     weiteren Nachrichten */

/* An dieser Stelle müßte eigentlich die Zeile "WaitPort(w->UserPort);"
   stehen, damit unser Programm "sauber" im Multitaskingbetrieb
   arbeitet. Da wir es jedoch in Arizona "schneien" lassen wollen,
   brauchen wir eine Schleife, die sowohl den MessagePort abfragt als
   auch den "Schnee" erzeugt. Daher fehlt die oben genannte Zeile.
   Da jeder Staat eine eigene Farbe hat, können wir mit dem
   Zufallsgenerator Zahlen erzeugen lassen, aus denen wir dann
   Koordinaten für einen Punkt formen. Hat dieser Punkt (Pixel) die
   Farbe des Staates Arizona, dann verwenden wir WritePixel, um an
   seiner Stelle eine "Schneeflocke" zu erzeugen. Auf diese Weise
   "schneit" es nur in Arizona, jedoch nicht außerhalb, da wir mit
   ReadPixel ja die Farbe des Staates feststellen können. */

   rx = CORNERX-RangeRand(60);
   ry = CORNERY+RangeRand(70);

    if(ReadPixel(rp,rx,ry) == AZCOLOR)
       {
         SetAPen(rp,WHITECOLOR);
         WritePixel(rp,rx,ry);
        }
    }

  /* Die while-Schleife wurde verlassen (Programmende) */
```

Listing 4.15: Programm zum Zeichnen einer Landkarte (Teil 3)

```
  CloseWindow(w);
  CloseScreen(s);
  FreeRaster(workspace,640,200);
  CloseLibrary(GfxBase);
  CloseLibrary(IntuitionBase);
}  /* Ende von main() */

afill(w,pairs)
int *w,pairs;   /* Zeiger auf ein 16-Bit-Wort und Anzahl Worte */
{
  int i;

  AreaMove(rp,CORNERX+w[0],CORNERY+w[1]);
  w+=2;

  for(i=1;i<pairs;i++)
        {
          AreaDraw(rp, CORNERX+w[0],CORNERY+w[1]);
          w+=2;
        }
  AreaEnd(rp);
}

redraw()
{
  SetDrMd(rp,JAM1);
  SetAPen(rp,1);
  afill(&coloradoxy[0],4);

  SetAPen(rp,5);
  afill(&utahxy[0],7);

  SetAPen(rp,3);
  afill(&newmexicoxy[0],5);

  SetAPen(rp,AZCOLOR);
  afill(&arizonaxy[0],8);

  SetOPen(rp,12);
  SetAPen(rp,7);
  DrawDiamond(rp,20,20,20,10);
  SetAPen(rp,8);
  DrawDiamond(rp,20,120,20,10);

  SetAPen(rp,6);
  SetBPen(rp,0);
  SetDrMd(rp,JAM2);
```

Listing 4.15: Programm zum Zeichnen einer Landkarte (Teil 4)

```
   /* Die Staaten mit ihren Kürzeln versehen */

   Move(rp,CORNERX-30,CONRNERY-20);
   Text(rp,"UT",2);

   Move(rp,CORNERX-30,CONRNERY+30);
   Text(rp,"AZ",2);

   Move(rp,CORNERX-35,CONRNERY-20);
   Text(rp,"CO",2);

   Move(rp,CORNERX+35,CONRNERY+30);
   Text(rp,"NM",2);
}
```

Listing 4.15: Programm zum Zeichnen einer Landkarte (Schluß)

Text

In den vorangegangenen Beispielprogrammen wurde zur Textausgabe immer der Standard-Zeichensatz des Amiga (topaz.font) mit 80 Zeichen pro Zeile verwendet.

In den folgenden Abschnitten werden Sie lernen, wie man beliebige Zeichensätze zur Textausgabe verwenden kann. Um den Cursor korrekt positionieren zu können, müssen Sie wissen, wie groß der aktuell verwendete Zeichensatz ist und wo seine Grundlinie zu finden ist.

Weiterhin müssen Sie wissen, wie man zwischen verschiedenen Zeichensätzen umschaltet und wie man auf spezielle Darstellungsmodi (Fettschrift, Kursiv, Invers und Normal) zugreift. Alle diese Punkte werden in den folgenden Abschnitten genau erklärt.

In der "RastPort"-Datenstruktur sind zwei Variablen enthalten, in denen die Werte für die Höhe und Breite des aktuell verwendeten Zeichensatzes abgelegt sind. Heißt der Zeiger auf Ihren RastPort z.B. "rp", dann werden die Variablen durch einen einfachen Strukturverweis ausgelesen. Hier ein Beispiel, um die Texthöhe des Zeichensatzes festzustellen:

```
Texthöhe = rp->TxHeight;
```

Die Variable "TxWidth" enthält die Breite des Rechtecks, in dem alle Zeichen des aktuellen Zeichsatzes im normalen Darstellungsmodus (d.h. keine Fett- oder Kursivschrift) dargestellt werden können. Ausgelesen wird diese Variable wie folgt:

```
Textbreite = rp->TxWidth;
```

Mit Hilfe dieser Variablen können Sie – egal, welcher Zeichensatz verwendet wird – immer einen Cursor der richtigen Größe erstellen.

Es ist noch eine weitere Variable in der "RastPort"-Datenstruktur enthalten, die beim Arbeiten mit Zeichensätzen wichtig sein kann: "TxBaseline". Sie enthält den Wert, der über die Grundlinie des Zeichensatzes Auskunft gibt. Ausgelesen wird die Variable wie folgt:

```
TextGrundlinie = rp->TxBaseline;
```

Die Grundlinie eines Zeichensatzes legt die Basis für alle Zeichen fest. Die meisten Zeichensätze stellen unter der Grundlinie noch eine oder zwei Zeilen zusätzlich zur Verfügung, damit Buchstaben wie "y" oder "g" richtig dargestellt werden können. Die Position der Grundlinie wird relativ zur obersten Zeile der Zeichen angegeben. Ist ein Zeichensatz z.B. 10 Zeilen hoch, so liegt seine Grundlinie auf der achten Zeile von oben.

Da bei der Textausgabe immer die Grundlinie als Startpunkt angegeben werden muß, ist es bisweilen recht mühsam, die Zeichen vertikal zu positionieren. Damit Textkoordinaten verwendet werden können, die sich auf die linke obere Ecke eines Zeichens beziehen, sollte man die Funktion Move folgendermaßen aufrufen:

```
Move(rp,x,y+rp->TxBaseline);
```

Mit dieser Form der Zeichenstiftpositionierung entfällt die lästige Suche nach der richtigen vertikalen Position des Textes.

Öffnen eines Zeichensatzes

Der Amiga kennt zwei verschiedene Arten von Zeichensätzen: solche, die im ROM (speicherresident) vorhanden sind, und andere, die bei ihrem Aufruf von Diskette geladen werden.

Aus diesem Grund stehen auch zwei verschiedene Funktionen zum Öffnen eines Zeichensatzes zur Verfügung. OpenFont öffnet einen Zeichensatz, der speicherresident ist, OpenDiskFont lädt einen Zeichensatz von Diskette. Brauchen werden Sie jedoch nur die Funktion OpenDiskFont, da mit ihr auch speicherresidente Zeichensätze geladen werden können.

OpenDiskFont verwendet beim Aufruf Informationen aus der vom System verwalteten Liste aller Zeichensätze und entscheidet dann, ob ein Zeichensatz bereits im Speicher vorhanden ist oder ob er geladen werden muß.

Wollen Sie einen anderen als den Standard-Zeichensatz verwenden (60 oder 80 Zeichen pro Zeile), dann müssen Sie den neuen Zeichensatz, der benutzt werden soll, öffnen und seine Datenstruktur in Ihren RastPort einbinden. Wird ein speicherresidenter Zeichensatz geöffnet, dann bekommen Sie von der Funktion, mit der er geöffnet wurde, als Returnwert einen Zeiger auf seine Datenstruktur, in der der Zeichensatz definiert ist.

Das Öffnen eines Zeichensatzes, der auf Diskette vorliegt, hat den gleichen Vorgang zur Folge, nur daß er zuerst noch in den Speicher geladen wird. Der angesprochene Zeichensatz muß sich im Verzeichnis "fonts" auf der aktuellen Diskette befinden. Nach dem erfolgreichen Öffnen eines Zeichensatzes steht dieser dann zur weiteren Arbeit zur Verfügung. Die Funktionen OpenFont und OpenDiskFont werden wie folgt aufgerufen:

```
zeichensatz = OpenFont(&ta);
zeichensatz = OpenDiskFont(&ta);
```

Der Returnwert "zeichensatz" ist ein Zeiger auf die "TextFont"-Datenstruktur des Zeichensatzes, falls er erfolgreich geöffnet wurde; andernfalls erhalten Sie den Wert 0 von den Funktionen. Der Parameter "ta" ist ein Zeiger auf eine "TextAttr"-Datenstruktur.

Definition der "TextAttr"-Datenstruktur

Der Name, die Größe und der Darstellungsmodus eines Zeichensatzes wird in der "TextAttr"-Datenstruktur abgelegt. Die Komponenten der Struktur bedeuten:

- "ta_name" ist ein Zeiger auf die Zeichenkette, die den Namen des Zeichensatzes enthält, der geladen werden soll. Dieser Name entspricht dem, der auch im Verzeichnis "fonts" auf der Diskette verwendet wird, z.B. "garnet.font".

– "ta_YSize" ist die Höhe des Zeichensatzes (in Zeilen). Im Verzeichnis "fonts" auf der Diskette ist für jeden Zeichensatz ein Unterverzeichnis vorhanden, dessen Dateiname die Größe des jeweiligen Zeichensatzes angibt. So ist z.B. "garnet/9" ein 9 Bildschirmzeilen hoher Zeichensatz. Die Datei "9" enthält den eigentlichen Zeichensatz.

– "ta_style" ist der Darstellungsmodus eines Zeichensatzes. Einige sind z.B. nur kursiv oder fett vorhanden; andere können kursiv oder fett dargestellt werden.

– "ta_Flags" enthält voreingestellte Werte für einen Zeichensatz. Dieses Feld kann initialisiert werden, damit das System bestimmte Bedingungen, die mit diesem Zeichensatz verknüpft sind, erfüllen kann. Wenn Sie z.B. einen Zeichensatz verwenden wollen, der zur Darstellung auf Screens mit hoher Auflösung (640*200) geschaffen wurde, dann können Sie das Flag "FPF_TALLDOT" setzen.

Sollten zwei Zeichensätze mit gleichem Namen und gleicher Höhe im Speicher vorhanden sein, von denen aber nur einer zur Darstellung in hoher Auflösung geeignet ist, dann wird vom System dieser Zeichensatz geladen. Hier ein Beispiel zur Initialisierung der "TextAttr"-Datenstruktur:

```
struct TextAttr meinAttr = { "garnet.font",9,0,0 };
```

Der zugehörige Zeichensatz wird nach dieser Initialisierung wie folgt geladen:

```
struct TextFont *mein_zeichensatz;
mein_zeichensatz = OpenDiskFont(&meinAttr);
```

Einbinden des Zeichensatzes in den RastPort

Damit Sie einen Zeichensatz verwenden können, müssen Sie ihn in Ihren RastPort einbinden. Dies geschieht mit der Funktion SetFont, die folgendermaßen aufgerufen wird:

```
SetFont(rp,tf);
```

Der Parameter "rp" ist dabei ein Zeiger auf den RastPort, in dem der neue Zeichensatz verwendet werden soll. "tf" ist der Returnwert der Funktion OpenFont bzw. OpenDiskFont und ist der Zeiger auf die "TextFont"-Datenstruktur des Zeichensatzes. Nach dem erfolgreichen Aufruf von SetFont wird fortan der neue Zeichensatz zur Textausgabe im "RastPort" verwendet.

Eintragen eines neuen Zeichensatzes in der Zeichensatzliste des Systems

Die vom System verwaltete Liste, in der alle geladenen (verfügbaren) Zeichensätze aufgeführt sind, enthält nach dem Einschalten und der Initialisierung des Computers zwei Einträge, die den Standard-Zeichensatz (topaz.font) repräsentieren. Dieser Zeichensatz steht mit 40/80 oder 32/64 Zeichen pro Zeile zur Verfügung.

Um anderen Tasks, die vielleicht neben Ihrem Programm im Rechner abgearbeitet werden, den Zugriff auf Ihren Zeichensatz zu "erlauben", sollten Sie jeden geöffneten Zeichensatz in der Zeichensatzliste des Systems eintragen. Dies geschieht durch einen Aufruf der Funktion AddFont, der folgende Form hat:

```
AddFont(tf);
```

Der Parameter "tf" ist wiederum der Zeiger auf die "TextFont"-Datenstruktur des Zeichensatzes.

Wenn zwei Tasks den gleichen Zeichensatz verwenden, dann rufen beide einzeln die Funktion OpenDiskFont auf, um den "TextFont"-Zeiger zu erhalten. Bevor jedoch ein Zeichensatz geladen wird, schaut das System zunächst in seiner Liste nach, ob der vom Task angeforderte Zeichensatz schon von einem anderen Task geladen und in der Liste eingetragen wurde.

Ist dies der Fall, dann erhöht das System den Zähler, der für jeden Zeichensatz existiert und der Auskunft über die Anzahl Tasks gibt, die den Zeichensatz verwenden, um den Wert Eins. Danach wird der bereits vorhandene Zeiger auf die "TextFont"-Datenstruktur dem Task übergeben, der den Zeichensatz angefordert hat.

Wenn Sie einen Zeichensatz laden, ihn aber nicht in der Liste eintragen, dann wird der komplette Zeichensatz noch einmal geladen, wenn er von einem weiteren Task angefordert wird, was unnötigen Speicherplatz kostet.

Bestimmen, welche Zeichensätze verfügbar sind

Wenn Sie ein Programm entwickeln, dann wissen Sie normalerweise genau, welche Zeichensätze verfügbar sind, d.h. Sie können einen ganz bestimmten Zeichensatz aufgrund seines Namens oder seiner Größe (o.ä.) vom System anfordern. Die Funktionen OpenFont und OpenDiskFont versuchen bei ihrem Aufruf zunächst, den gewünschten Zeichensatz zu laden.

Ist er nicht im Verzeichnis "fonts" auf der Diskette vorhanden, dann wird versucht, einen anderen Zeichensatz zu laden, der den Angaben im "ta_Flags"-Datenfeld am nächsten kommt. Wollen Sie z.B. den Zeichensatz "garnet.font" in der Größe 9 laden, dann wird u.U. ein anderer Zeichensatz der Größe 9 geladen, falls "garnet.font" nicht verfügbar sein sollte. Um festzustellen, welcher Zeichensatz geladen wurde, können Sie nach dem Laden des Zeichensatzes mit der Funktion AskFont seine Charakteristik ausgeben lassen.

Sie können eine eigene Liste von Zeichensätzen erstellen, die anstelle eines Zeichensatzes geladen werden sollen, falls der gewünschte Zeichensatz nicht vorhanden ist. Die hierzu notwendige Funktion heißt AvailFonts. Mit ihr können Sie z.B. eine Liste aller speicherresidenten oder kursiven Zeichensätze anfertigen. Der Aufruf der Funktion hat die Form:

```
AvailFonts(&afh,AFSIZE,art);
```

Der Parameter "afh" ist ein Zeiger auf einen Speicherbereich, in dem die "AvailFontsHeader"- und AvailFonts-Datenstrukturen abgelegt werden sollen. Die Größe dieses reservierten Speicherbereiches wird mit dem Parameter "AFSIZE" übergeben.

Bei "art" handelt es sich um ein 8-Bit (1 Byte)-Datenfeld, mit dem Sie bestimmen können, welche Art Zeichensätze in der Liste eingetragen werden sollen. Sind im System mehr Zeichensätze vorhanden, als in der Liste eingetragen werden können, dann paßt das System auf, daß der reservierte Speicherplatz für die Liste nicht überschritten wird. Hat der Parameter "art" den Wert 0xff, dann werden alle dem System bekannten Zeichensätze in der Liste eingetragen (wenn der Platz ausreicht).

In der Datei "graphics/text.h", die sich auf Ihrer Compiler-Diskette befindet, sind die Definitionen aller erlaubten Werte aufgeführt (in Form von symbolischen Konstanten). Hier eine Auswahl aus diesen Konstanten:

FPF_ROMFONT trägt alle speicherresidenten Zeichensätze in der Liste ein.

FPF_DISKFONT trägt alle Zeichensätze in der Liste ein, die auf Diskette zur Verfügung stehen.

FPF_REVPATH vermerkt alle Zeichensätze in der Liste, die von rechts nach links ausgegeben werden (z.B. Hebräisch).

FPF_WIDEDOT listet alle Zeichensätze, die zur Darstellung in niedriger Auflösung (320*200 Pixel) geschaffen wurden.

FPF_TALLDOT listet alle Zeichensätze, die zur Darstellung in hoher Auflösung (640*200 Pixel) geschaffen wurden.

Um mehrere Flags gleichzeitig verwenden zu können, werden die einzelnen Konstanten einfach durch eine ODER-Operation verknüpft. Nachdem die Funktion AvailFonts erfolgreich beendet wurde, steht Ihnen eine Liste der Zeichensätze zur Verfügung, die den von Ihnen gewünschten Voraussetzungen entsprechen.

AvailFonts ruft automatisch die Routine AddFont für jeden Zeichsatz auf, der zwar in Ihrer, aber nicht in der vom System verwalteten Liste enthalten ist. Ein Nebeneffekt dieses Verfahrens ist, daß bei einem zweiten Aufruf von AvailFonts (mit dem Parameterwert "art" = 0xff) alle Zeichensätze, die auf Diskette vorhanden sind, in Ihrer Liste doppelt aufgeführt werden. Der Grund hierfür liegt im Aufruf von AddFont durch die Funktion AvailFonts. Ein Zeichensatz ist dann nämlich sowohl in der Systemliste, als auch im Verzeichnis "fonts" der Diskette aufgeführt, wird somit also auch zweimal ausgegeben.

Da ja nur zwei speicherresidente Zeichensätze auf dem Amiga zur Verfügung stehen (topaz.font), kann das Flag "FPF_ROMFONT" beim Aufruf von AvailFonts weggelassen werden, was einen kleinen Zeitgewinn bei der Ausführung der Funktion mit sich bringt. Weiterhin erhalten Sie so eine Liste, in der wirklich nur die Zeichensätze enthalten sind, die auf Diskette vorhanden sind.

Nach dem Beenden der Funktion AvailFonts sind die "AvailFontsHeader"- und AvailFonts-Datenstrukturen vom System mit den von Ihnen gewünschten Informationen gefüllt worden. Durch das Auslesen dieser Strukturen können Sie feststellen, welche Zeichensätze zur Verfügung stehen und welche Charakteristik sie besitzen. Nachfolgend finden Sie den Aufbau der "AvailFonts-Header"-Struktur sowie der dazugehörigen AvailFonts-Datenstrukturen.

```
AvailFontsHeader    { UWORD afh_NumEntries }
AvailFonts          { UWORD af_Type;
                    /* 1 = Speicherresident, 2 = Disk */
                    struct TextAttr af_Attr; }
                    /* Attribute */
AvailFonts
      ...
AvailFonts
```

"AvailFontsHeader" enthält nur eine einzige Komponente, in der die Anzahl aller AvailFonts-Datenstrukturen abgelegt wird. Diese Strukturen bestehen aus zwei Komponenten: "af_Type" enthält den Wert Eins, wenn der Zeichensatz im ROM (speicherresident) vorhanden ist bzw. den Wert Zwei, wenn er auf Diskette verfügbar ist. Die zweite Komponente ist die schon behandelte "TextAttr"-Datenstruktur, die vom System mit Informationen über den betreffenden Zeichensatz gefüllt wird.

Nach dem Aufruf von AvailFonts können Sie einen Zeiger auf eine der dann vorhandenen AvailFonts-Datenstrukturen definieren, den Sie dann der Funktion OpenDiskFont übergeben können, um den betreffenden Zeichensatz öffnen zu lassen.

Zur Reservierung des Speicherplatzes für die Datenstrukturen können Sie folgendes Programmfragment verwenden:

```
#define MAXIMALE_GROESSE    30

struct AvailFontsHeader *aftable;
struct aft =
{
  struct AvailFontsHeader  aft_Head;
  struct AvailFonts aft_Entry[MAXIMALE_GROESSE];
};

int aftablesize;

aftablesize = sizeof(struct aft);
```

Nach dieser Initialisierung können Sie AvailFonts aufrufen:

```
AvailFonts(&aft,aftablesize,0xFE);
```

Der letzte Parameter hat den Wert 0xFE, wodurch nur die Zeichensätze in der Liste eingetragen werden, die auf Diskette vorhanden sind. Die "TextAttr"-Datenstrukturen für die speicherresidenten Zeichensätze können separat verwendet werden. Sie haben folgenden Inhalt:

```
struct TextAttr groesse32_64 = { "topaz.font",9,0,0 };
struct TextAttr groesse40_80 = { "topaz.font",8,0,0 };
```

Im Listing 4.16 finden Sie ein Programmbeispiel, das Informationen aus der AvailFonts-Liste extrahiert und weiterverwendet.

```
/* Programmfragment zum Auslesen der AvailFonts-Liste */

UWORD anzahl_eintraege;
struct TextAttr *meinTextAttrZeiger;

anzahl_eintraege = aft.aft_Head.afh_NumEntries;

/* Zeiger auf den ersten Eintrag in der Liste setzen */
meinTextAttrZeiger = (struct TextAttr)(&aft.aft_Entry);

/* Alle Einträge nacheinander verwenden */
int i;
struct TextFont *tf, OpenDiskFont();

for(i=0;i<anzahl_eintraege;i++)
      {
        tf = OpenDiskFont(meinTextAttrZeiger[i]);

        if(tf)      /* Zeichensatz wurde geöffnet */
           {
             SetFont(rp,tf);
             /* rp ist ein Zeiger auf Ihren RastPort */
             Text(rp,"Beispieltext",12);
             Delay(30);
             CloseFont(tf);    /* Zeichensatz schließen */
            }
        }
```

Listing 4.16: Anwendungsbeispiel der AvailFonts-Liste

Die Darstellungsweise des Textes

Sie können nicht nur bestimmen, welchen Zeichensatz Sie verwenden wollen, Sie können dem System auch mitteilen, auf welche Art und Weise Text dargestellt werden soll. Zur Darstellungsweise von Text gehören die Farbe und die Charakteristik des Zeichensatzes (fett, kursiv, unterstrichen oder invers).

Die Textfarbe geben Sie mit Hilfe der Funktionen SetAPen, SetBPen und SetDrMd an. Verwenden Sie als Zeichenmodus "JAM1", wird der Text in der Farbe des APen-Zeichenstiftes ausgegeben. Benutzen Sie "JAM2", wird zusätzlich das Rechteck, in dem ein Zeichen dargestellt wird, in der Farbe des BPen-Zeichenstiftes ausgefüllt.

Um Text hervorzuheben, wird dieser meistens invers dargestellt, d.h. die Farben von APen und BPen werden vertauscht. Anders gesagt: der Text wird in der Farbe von BPen, der Hintergrund in der Farbe von APen gezeichnet. Sie müssen jedoch nicht die Farbwerte der beiden Zeichenstifte vertauschen, um diesen Effekt zu erzielen; es genügt der Aufruf der Funktion SetDrMd mit dem "INVERSVID"-Flag:

```
SetDrMd(rp,JAM1+INVERSVID);
```

Mit diesem Aufruf wird der Text in der Hintergrundfarbe ausgegeben, das zugehörige Rechteck jedoch in der Farbe von APen. Mit dem folgenden Aufruf wird der so ausgegebene Text wieder invertiert:

```
SetDrMd(rp,JAM2+INVERSVID);
```

Schriftarten: Fett, Kursiv oder Unterstrichen

Um für einen Zeichensatz verschiedene Schriftarten verwenden zu können, stellt das System die Funktion SetSoftStyle zur Verfügung, mit der das Aussehen eines Zeichenssatzes geändert wird. Aufgerufen wird die Funktion wie folgt:

```
SetSoftStyle(rp,stil,maske);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, in dem der Zeichensatz verwendet wird, "stil" ist eine Bitmaske, die die gewünschte Schriftart repräsentiert, und "maske" sagt dem System, welche "Stil-Bits" geändert werden sollen. Dieser Parameter ist z.B. dann sinnvoll, wenn Sie Kursivschrift gewählt haben und zusätzlich zu dieser Schriftart Fettschrift einsetzen wollen. (Eine Auflistung aller gültigen Bits finden Sie – in Form symbolischer Konstanten – in der Datei "graphics/text.h", die sich auf Ihrer Compiler-Diskette befindet.)

Um eine weitere Schriftart zu verwenden, ohne die aktuell selektierte zu löschen, geben Sie einfach für die Parameter "stil" und "maske" den gleichen Wert an, also z.B.

```
SetSoftStyle(rp,FPF_ITALICS,FPF_ITALICS);
```

Um das Feld "maske" neu zu initialisieren, ohne auf den derzeitigen Wert Rücksicht zu nehmen, übergeben Sie einfach den Wert 0xff:

```
SetSoftStyle(rp,FPF_ITALICS,0xff);
/* Rücksetzen von <maske>, Schriftart bleibt jedoch Kursiv */
```

Besitzt ein Zeichensatz schon eine bestimmte Schriftart (Fett, Kursiv oder Unterstrichen), dann hat ein Setzen des entsprechenden Bits beim Aufruf von SetSoftStyle keine weitere "Steigerung" der Schriftart zur Folge, d.h. Fettschrift wird nicht noch "fetter", da das entsprechende Bit bereits im "ta_Flags"-Feld der "TextAttr"-Datenstruktur gesetzt ist. Um festzustellen, welche Schriftarten für einen Zeichensatz noch eingesetzt werden können, gibt es die Funktion AskSoftStyle, die wie folgt aufgerufen wird:

```
noch_verfuegbare_schriftarten = AskSoftStyle(rp);
```

Der Parameter "rp" ist ein Zeiger auf einen RastPort. Der Returnwert der Funktion "noch_verfuegbare_schriftarten" ist ein Byte, das die Bits der Stilarten enthält, die für den aktuellen Zeichensatz noch gesetzt werden können. Hier ein Programmfragment, das den Gebrauch von AskSoftStyle veranschaulichen soll:

```
UBYTE noch_verfuegbare_schriftarten;

noch_verfuegbare_schriftarten = AskSoftStyle(rp);

if( noch_verfuegbare_schriftarten & FPF_UNDERLINED)
   printf("Schriftart <Unterstrichen> ist möglich.\n");

if( noch_verfuegbare_schriftarten & FPF_BOLD)
   printf("Schriftart <Fett> ist möglich.\n");

if( noch_verfuegbare_schriftarten & FPF_ITALICS)
   printf("Schriftart <Kursiv> ist möglich.\n");
```

Noch eine Bemerkung zur Schriftart "Unterstrichen": wenn das System einen Text unterstreichen soll, dann erfolgt dies in der ersten Zeile unterhalb der Grundlinie des Zeichensatzes. Stellt ein Zeichensatz unter seiner Grundlinie keine weiteren Zeilen zur Verfügung, so kann der Text nicht unterstrichen werden, da sonst "außerhalb" des Zeichensatzes gearbeitet werden müßte. Bevor Sie also diese Schriftart auswählen, sollten Sie zunächst prüfen, ob unterhalb der Grundlinie des Zeichensatzes noch eine Linie zur Verfügung steht.

Sie brauchen zu diesem Zweck nur den Wert der Variablen "TxBaseline" vom Wert der Variablen "TxHeight" abzuziehen, die beide in der Datenstruktur Ihres RastPort vorhanden sind. Liefert diese Subtraktion das Resultat 1, dann steht keine Zeile unterhalb der Grundlinie zum Unterstreichen zur Verfügung. Hier das Fragment:

```
if(rp->TxHeight-rp->TxBaseline == 1)
   printf("Unterstreichen bei diesem Zeichensatz nicht möglich!\n");
```

Löschen und Verschieben (Scrollen) von Bildschirmbereichen

Um einen Grafikbereich (RastPort) mit einem Mal zu löschen, können Sie die Funktion SetRast verwenden. Diese Funktion füllt den angegebenen RastPort mit einer von Ihnen bestimmten Farbe. Der Aufruf der Funktion lautet:

```
SetRast(rp,farbe);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, dessen Inhalt gelöscht werden soll, "farbe" ist die Nummer des Farbregisters, das zum Füllen verwendet werden soll. Sie können diese Funktion zum Beispiel für den RastPort anwenden, den Sie von Intuition beim Öffnen eines Fensters erhalten. Allerdings werden auch die Fensterumrandung sowie alle Gadgets des Fensters gelöscht, wenn es sich nicht um ein Fenster des Typs "GIMMEZEROZERO" handelt. Ist das Fenster jedoch von diesem Typ, dann wird tatsächlich nur der Grafikbereich gelöscht, jedoch nicht die Fensterkomponenten.

Das System stellt noch zwei weitere Funktionen zum Löschen von Bildschirmbereichen zur Verfügung: ClearEOL und ClearScreen. Diese Funktionen sind Bestandteil der "graphics.library"-Bibliothek. Diese Funktionen sind textorientiert und werden z.B. vom "Console Device" verwendet. Die Arbeitsweise der Funktionen hängt – da sie textorientiert sind – vom verwendeten Zeichensatz des betreffenden RastPort ab.

Die Funktion ClearEOL löscht alles von der aktuellen Position des Zeichenstiftes bis zum rechten Rand des RastPort, wobei ein Rechteck der entsprechenden Länge mit der Hintergrundfarbe gefüllt wird. Die Höhe des Rechtecks hängt von der Höhe des aktuellen Zeichensatzes ("TxHeight") ab.

ClearScreen führt zunächst die Funktion ClearEOL aus und löscht danach den Bereich unter der gelöschten Zeile bis zum unteren Rand des RastPort. Aufgerufen werden die Funktionen wie folgt:

```
ClearEOL(rp);
ClearScreen(rp);
```

Der Parameter "rp" ist ein Zeiger auf den RastPort, in dem die Funktionen ausgeführt werden sollen.

Manche Programme – z.B. Textverarbeitung oder Terminalprogramme – machen es erforderlich, daß der sichtbare Text um eine Zeile nach oben oder unten verschoben wird, um Platz für eine neue Textzeile zu schaffen. Man kann dabei entweder immer eine komplette Textzeile "aus dem Bildschirm" schieben ("Scrollen"), oder aber den Text Rasterzeilenweise ("Softscrolling") scrollen lassen, eine Methode, die "angenehmer" aussieht.

Der Amiga stellt zum Verschieben eines Grafikbereiches die Funktion ScrollRaster zur Verfügung, die folgendermaßen aufgerufen wird:

```
ScrollRaster(rp,dx,dy,xmin,ymin,xmax,ymax);
```

Der Parameter "rp" ist ein Zeiger auf einen RastPort, "xmin" und "ymin" sind die Koordinaten der linken oberen, "xmax" und "ymax" die Koordinaten der rechten unteren Ecke des Rechtecks, das verschoben werden soll. Die beiden Parameter "dx" und "dy" enthalten die Anzahl Pixel (Bildpunkte), um die der rechteckige Ausschnitt des RastPorts verschoben werden soll (in Richtung der linken oberen Ecke des RastPorts). Die Werte dieser Variablen sind entweder positiv, negativ oder null.

Wollen Sie z.B. einen Grafikbereich um acht Rasterzeilen nach oben verschieben, um so Platz für eine neue Textzeile am unteren Rand des RastPort zu schaffen, dann hat "dx" den Wert Null, da nicht nach links oder rechts verschoben wird, und "dy" hat den Wert 8 (8 Pixel nach oben scrollen).

Bei Fenstern, die nicht vom Typ "GIMMEZEROZERO" sind, muß der zu scrollende Bereich kleiner als die Fenstergröße gewählt werden, da sonst die Fensterumrandung mit verschoben wird.

Kombination mehrerer Komponenten zu einem Bild

In den folgenden Abschnitten werden Sie lernen, wie man einen Grafikbereich erstellt, der außerhalb des "Sichtfeldes" des Anwenders liegt. Alle besprochenen Beispielprogramme haben zur Grafikausgabe den RastPort verwendet, den Intuition beim Aufruf von OpenWindow liefert.

Wir werden nun Grafikbereiche und den zugehörigen RastPort direkt definieren, in dem dann Grafik erstellt werden und anschließend in ein Fenster kopiert werden kann, wo sie dann zu sehen ist. Für dieses Verfahren müssen Sie einiges wissen:

- Initialisierung einer "BitMap" und Reservierung von Speicher für ihre Bitplanes.

- Initialisierung eines RastPort.

- Kopieren von Daten einer BitMap in eine andere.

Diese Punkte werden im folgenden erläutert.

Initialisierung einer "BitMap"

Die BitMap-Datenstruktur enthält Variablen, die die Größe eines Grafikbereiches und den Speicherbereich definieren, in dem dieser zu finden ist. Mit der Funktion InitBitMap wird diese Datenstruktur initialsiert. Nach ihrem Aufruf muß für jede Bitplane (ein "Blatt" der BitMap) Speicherplatz reserviert werden, in dem die zugehörigen Bits, die eine Grafik formen, abgelegt werden können.

Wenn Sie z.B. einen Grafikbereich definieren wollen, der eine Auflösung von 320*200 Bildpunkten hat, dann können Sie maximal vier Farben verwenden. Für vier Farben muß die BitMap eine "Tiefe" von zwei Bitplanes haben.

Hier ein Beispiel zur Initialisierung einer BitMap:

```
#define ANZAHL_BITPLANES 2
#define BREITE 320
#define HOEHE 200

struct BitMap meineBitMap;
int i;
extern PLANEPTR AllocRaster();

InitBitMap(&meineBitMap,ANZAHL_BITPLANES,BREITE,HOEHE);

for(i=0;i<ANZAHL_BITPLANES;i++)
      {
        meineBitMap.Planes[i] = AllocRaster(BREITE,HOEHE);
        if(!meineBitMap.Planes[i])
           {
             /* Fehlerbehandlung...nicht genug Speicher vorhanden */
           }
       }
```

Mehr ist zur Initialisierung der "unsichtbaren" BitMap nicht erforderlich. Wird die BitMap später nicht mehr benötigt, dann muß der von ihr beanspruchte Speicherplatz wieder freigegeben werden:

```
for(i=0;i<ANZAHL_BITPLANES;i++)
      {
        if(meineBitMap.Planes[i])
           FreeRaster(meineBitMap.Planes[i],BREITE,HOEHE);
       }
```

Initialisierung eines "RastPort"

Nachdem der Speicher für die erforderlichen Bitplanes reserviert wurde, gestaltet sich die Initialisierung des RastPort recht einfach:

```
struct RastPort meinRastPort;

InitRastPort(&meinRastPort);

/* BitMap in den RastPort einbinden */
meinRastPort.BitMap = &meineBitMap;
```

Sie haben auf diese Weise einen Zeiger auf einen "unsichtbaren" RastPort erhalten, d.h. dieser Grafikbereich gehört nicht zu einem Fenster, ist also für den Anwender nicht sichtbar.

Diesen Zeiger können Sie nun wie gewohnt verwenden, um die Zeichenfarbe und den Zeichenmodus zu setzen, um Linien zu zeichnen usw. Es stehen Ihnen also für diesen RastPort alle Grafikfunktionen zur Verfügung, die Sie auch in einem "sichtbaren" RastPort anwenden können. Mit den beiden folgenden Zeilen definieren Sie den Zeiger für den RastPort:

```
struct RastPort *Unsichtbarer_RastPort; /* Zeiger auf den RastPort */

*Unsichtbarer_RastPort = &meinRastPort;
```

Kopieren der Daten einer BitMap in eine andere

Der Amiga stellt Ihnen drei Funktionen zur Verfügung, um Daten von einer BitMap in eine andere zu kopieren: BltBitMap, ClipBlit und BltBitMapRastPort.

Die erste Funktion, BltBitMap, kopiert die Daten einer BitMap in eine andere und übergibt auch den zugehörigen RastPort. Diese Funktion führt beim Kopieren der Daten ("Blitten") keine Fehlerprüfung durch, d.h. das System überprüft nicht, ob die zu kopierende BitMap auch wirklich in den angegebenen Speicherbereich hineinpaßt.

Ist die "Original-BitMap" zu groß für den angegebenen Speicherbereich, dann werden einfach Bereiche des Speichers überschrieben, die z.B. von anderen Tasks verwendet werden. In einem solchen Fall ist meistens ein Systemabsturz die Folge.

ClipBlit kopiert die Daten eines RastPort in einen anderen. Aus dem Quell-RastPort wird ausgehend von der Koordinate XY ein rechteckiger Bereich "ausgeschnitten", der dann in einem anderen RastPort an der Koordinate XY eingefügt wird. Der Nachteil dieser Funktion liegt darin, daß nur rechteckige Bereiche kopiert werden können. Wenn ein kopiertes Objekt ein anderes im Ziel-RastPort überdeckt, dann überschneiden sich die Rechtecke, nicht die Objekte als solche. Ist z.B. das aktuell kopierte Objekt von einem Rand in der Hintergrundfarbe umgeben, dann werden u.U. Teile eines anderen Objektes im Ziel-RastPort gelöscht, wenn sich die Grafiken überschneiden.

BltBitMapRastPort ist von den drei Funktionen die schnellste, da die Daten von einer BitMap in einen RastPort kopiert werden. Es werden also keine Bereiche "ausgeschnitten", sondern es wird die komplette BitMap kopiert. Daher können sich auch im Ziel-RastPort keine Objekte überschneiden, wie dies bei ClipBlit der Fall sein kann. Ein Aufruf der Funktion ClipBlit hat die Form:

```
ClipBlit(quell_rp,quell_x,quell_y,ziel_rp,ziel_x,ziel_y,
                  gr_x,gr_y,minterm);
```

Die Parameter bedeuten im einzelnen:

quell_rp und ziel_rp	sind Zeiger auf den Quell- und Ziel-RastPort.
quell_x und quell_y	sind die Koordinaten der linken oberen Ecke des Bereiches, der kopiert werden soll.
ziel_x und ziel_y	sind die Koordinaten der linken oberen Ecke des Bereiches, in den der kopierte Bereich eingefügt werden soll.
gr_x und gr_y	sind die Breite und Höhe des rechteckigen Grafikbereiches, der kopiert werden soll.
minterm	bestimmt, auf welche Art und Weise kopiert werden soll.

Hat "minterm" den Wert 0xc0, dann werden die Daten unverändert kopiert. Ein Wert von 0x30 hat eine Invertierung der Kopie zur Folge, d.h. alle Bits, die im Quellbereich den Wert Eins haben, erhalten im Zielbereich den Wert Null und umgekehrt. Bei einem Wert von 0x50 werden die Daten des Quellbereiches ignoriert. Der Zielbereich wird einfach invertiert dargestellt.

Die Funktion BltBitMapRastPort wird folgendermaßen aufgerufen:

```
BltBitMapRastPort(quell_bm,quell_x,quell_y,ziel_rp,ziel_x,ziel_y,
                                    gr_x,gr_y,minterm);
```

Die verwendeten Parameter entsprechen denen der Funktion ClipBlit, mit Ausnahme von "quell_bm", der ein Zeiger auf die BitMap ist, die in den Ziel-RastPort kopiert werden soll.

Anwendung der Funktionen zum Kopieren von Grafikdaten

Das Beispielprogramm im Listing 4.17 öffnet ein Fenster vom Typ "SIMPLE_REFRESH" und definiert einen "unsichtbaren" RastPort. Im Fenster werden einige Linien gezeichnet, die direkt sichtbar sind. Im RastPort werden einige Rechtecke gezeichnet, die dann mit den Funktionen ClipBlit und BltBitMapRastPort in das Fenster kopiert werden.

Im nächsten Kapitel erfahren Sie mehr über Intuition. So werden Sie sehen, wie man Mausbewegungen und die Betätigung der Mausknöpfe verarbeiten kann. In unserem Beispielprogramm behelfen wir uns mit der Delay-Funktion.

Kopieren des "unsichtbaren" Hintergrundes

Wie bereits erwähnt, wird beim Einsatz der Funktionen ClipBlit und BltBitMapRastPort immer ein rechteckiger Grafikbereich kopiert, nie die Grafikobjekte als solche. Dies kann man verhindern, indem man im nicht sichtbaren Grafikbereich Objekte erstellt, deren Hintergrund "unsichtbar" ist, der also die Farbe aus dem Farbregister 0 verwendet.

Im Listing 4.18 finden Sie ein Beispielprogramm, mit dem Objekte, die über einen solchen Hintergrund verfügen, kopiert werden können. In diesem Beispiel kommt eine neue Funktion zum Einsatz: ClipBlitTransparent, sowie drei weitere Funktionen, die zusammen mit der Funktion angewendet werden müssen.

ClipBlitTransparent macht aus der Funktion ClipBlit einen BOB(Blitter Object)-Generator. Jedes Pixel des Quellbereiches, das die Farbe des Registers 0 hat, wird nach dem Kopieren transparent, d.h. "unsichtbar". Ein Aufruf von ClipBlitTransparent hat die Form:

```
ClipBlitTransparent(quell_rp,quell_x,quell_y,ziel_rp,ziel_x,ziel_y,gr_x,
                                        gr_y,schatten_rp,schatten);
```

Der Parameter "schatten_rp" ist ein Zeiger auf eine BitMap, die nur eine Bitplane enthält. Die Größe dieser BitMap hängt von der Größe des zu kopierenden Objekts ab. Verwendet wird diese BitMap von der Funktion CreateShadowRP. Überall da, wo das zu kopierende Objekt nicht die Farbe des Hintergrundes hat, ist ein Bit in der BitMap gesetzt.

```
/* offscreen.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "mydefines.h"
#include "window1.h"
#include "graphics/gfxbase.h"

#define ANZAHL_BITPLANES 2
#define BREITE 640
#define HOEHE 200

struct IntuitionBase   *IntuitionBase;
struct GfxBase    *GfxBase;
struct BitMap  meineBitMap;       /* Die "unsichtbare" BitMap */
struct RastPort   meinRast;      /* Der unsichtbare RastPort */
struct RastPort   *unsichtbarer_RastPort;   /* Und der Zeiger dazu */
struct RastPort   *rport;     /* RastPort des Fensters */
struct Window *w;     /* Zeiger auf das Fenster */

extern PLANEPTR AllocRaster();
int i,j = 10;

main()
{
  if(!(GfxBase = (struct GfxBase *)
     OpenLibrary("graphics.library",0)))
         {
           printf("graphics.library nicht geöffnet!\n");
           exit(10);
         }
  if(!(IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library",0)))
         {
           printf("intuition.library nicht geöffnet!\n");
           CloseLibrary(GfxBase);
           exit(15);
         }

  if(!(w = OpenWindow(&MeinFenster)))
         {
           printf("Fenster nicht geöffnet!\n");
           CloseLibrary(GfxBase);
           CloseLibrary(IntuitionBase);
           exit(20);
         }

  rport = w->RPort;
```

Listing 4.17: Anwendung der Funktionen zum Kopieren von Grafikdaten (Teil 1)

```
InitBitMap(&meineBitMap,ANZAHL_BITPLANES,BREITE,HOEHE);

for(i=0;i<ANZAHL_BITPLANES;i++)
    {
      meineBitMap.Planes[i] = AllocRaster(BREITE,HOEHE);
      if(!meineBitMap.Planes[i])
         {
           /* Fehlerbehandlung...nicht genug Speicher vorhanden */
         }
    }

InitRastPort(&meinRast);

meinRast.BitMap = &meineBitMap;     /* BitMap in RastPort einbinden */
unsichtbarer_RastPort = &meinRast;

/* Einige Linien ins Fenster zeichnen */
SetAPen(rport,3);
SetDrMd(rport,JAM1);

for(i=0;i<30;i++)
      {
        Move(rport,j,0);
        Draw(rport,j,100);
        j += 10;
      }
/* Einige Rechtecke in den "unsichtbaren" Grafikbereich zeichnen */

SetRast(unsichtbarer_RastPort,0);    /* RastPort löschen */

SetDrMd(unsichtbarer_RastPort,JAM1);
SetAPen(unsichtbarer_RastPort,1);
RectFill(unsichtbarer_RastPort,30,30,50,50);

SetAPen(unsichtbarer_RastPort,2);
RectFill(unsichtbarer_RastPort,40,40,60,60);

SetAPen(unsichtbarer_RastPort,3);
RectFill(unsichtbarer_RastPort,50,50,70,70);

/* Die Rechtecke werden jetzt in das Fenster kopiert, damit sie
   sichtbar werden. Die erste Kopie wird mit ClipBlit erstellt,
   die den gesamten Grafikbereich kopiert. Nach dem Beenden der
   anschließenden Zeitschleife wird nochmal kopiert, diesmal
   jedoch mit BltBitMapRastPort, die nur die reinen Grafikobjekte
   transferiert. */
```

Listing 4.17: Anwendung der Funktionen zum Kopieren von Grafikdaten (Teil 2)

```
    ClipBlit(unsichtbarer_RastPort,30,30,rport,10,30,40,40,0xc0);
    Delay(150);

    BltBitMapRastPort(&meineBitMap,30,30,rport,90,30,40,40,0xc0);
    Delay(300);

    /* Vor dem Verlassen des Programmes muß alles, was geöffnet wurde,
       auch wieder geschlossen werden... */

    if(w) CloseWindow(w);
    if(GfxBase) CloseLibrary(GfxBase);
    if(IntuitionBase) CloseLibrary(IntuitionBase);

    for(i=0;i<ANZAHL_BITPLANES;i++)
        {
          if(meineBitMap.Planes[i])
             FreeRaster(meineBitMap.Planes[i],BREITE,HOEHE);
         }

}   /* Ende von main() */
```

Listing 4.17: Anwendung der Funktionen zum Kopieren von Grafikdaten (Schluß)

Während des Kopiervorganges wird diese Bitmaske dazu verwendet, ein "Loch" in den Ziel-Grafikbereich zu "reißen", in welches das Objekt dann eingesetzt wird. Es wird also nur das Objekt kopiert; der Hintergrund bleibt unsichtbar.

Die Funktionen CreateShadowBM und CreateShadowRP werden zur Initialisierung von Datenstrukturen verwendet, die dann von ClipBlitTransparent benutzt werden. Damit die temporär verwendete BitMap erstellt wird, muß jedoch zusätzlich der Parameter "schatten" in der ClipBlitTransparent-Funktion auf den Wert "TRUE" (1) gesetzt werden. Dies ist jedoch nur beim ersten Aufruf der Funktion notwendig.

Sobald die BitMap angelegt wurde, kann "schatten" den Wert "FALSE" (0) haben. Wird ClipBlitTransparent erfolgreich beendet, dann liefert die Funktion den Returnwert Null; andernfalls ist der Wert ungleich Null.

Das folgende Programmfragment illustriert den Einsatz von ClipBlitTransparent. Listing 4.18 enthält das vollständige Beispiel.

```
struct BitMap *sbm;
struct RastPort srp=NULL;

sbm = CreateShadowBM(anzahl_bitplanes,breite,hoehe);
srp = CreateShadowRP(sbm,srp);
  /* An dieser Stelle sollte geprüft werden, ob die Funktionen den Wert 0
     liefern, also erfolgreich beendet wurden */

ClipBlitTransparent(quell_rp,quell_x,quell_y,ziel_rp,ziel_x,ziel_y, gr_x,
                                          gr_y,srp,TRUE);

DeleteShadowRP(srp);
DeleteShadowBM(sbm);
```

Die Funktion CreateShadowBM initialisiert die von ClipBlitTransparent benötigte temporäre BitMap. Die BitMap besteht aus einer einzigen Bitplane, für die entsprechend der Größe des zu kopierenden Objekts Speicherplatz reserviert wird.

Der Name "SchattenBitMap" kommt daher, daß alle BitPlanes des Quell-Grafikbereiches auf eine einzige Bitplane projektiert werden, d.h. überall da, wo in einer Bitplane des Quellbereiches ein Bit gesetzt ist, wird auch ein Bit in der "SchattenBitMap" gesetzt. Von dieser BitMap aus wird dann das Objekt mit ClipBlitTransparent in den Ziel-Grafikbereich umkopiert.

Die Parameter, die CreateShadowBM übergeben werden müssen, sind (der Reihenfolge nach geordnet):

- die Anzahl Bitplanes des Ziel-Grafikbereichs.
- die Breite des zu kopierenden Objektes in Bildpunkten (Pixel).
- die Höhe des zu kopierenden Objektes in Bildpunkten (Pixel).

Wird die Funktion erfolgreich beendet, dann erhalten Sie einen Zeiger auf eine "BitMap"-Datenstruktur. Diese Struktur wiederum enthält einen Zeiger auf eine Bitplane, deren Größe dynamisch geändert werden kann. Zusammen mit der Funktion CreateShadowRP wird in diesem Bereich der "Schatten" des

zu kopierenden Objekts abgelegt. Erhalten Sie den Wert Null von einer der beiden Funktionen, dann ist nicht genügend Speicherplatz vorhanden, um die temporäre BitMap anzulegen.

ClipBlitTransparent benötigt weiterhin einen Grafikbereich, in dem die einzelnen "Lagen" des Objektes abgelegt werden können. Die Funktion CreateShadowRP initialisiert diesen Bereich, in dem dann ein oder mehrere Objekte zwischengespeichert werden können. Die Funktion wird wie folgt aufgerufen:

```
schattenRastPort = CreateShadowRP(schattenBitMap,alter_schattenRastPort);
```

Der Parameter "schattenBitMap" ist der Returnwert der Funktion CreateShadowBM, und "alter_schattenRastPort" ist der Zeiger, den Sie bei einem früheren Aufruf von CreateShadowRP erhalten haben. Wird die Funktion zum ersten Mal aufgerufen, dann muß dieser Parameter den Wert Null haben, damit vom System eine genügend große "RastPort"-Datenstruktur angelegt wird. Dieser Parameter wird vom System anstelle einer Funktion verwendet, die einfach nur neue BitMaps in eine bereits vorhandene "RastPort"-Datenstruktur einbinden würde.

Als Returnwert erhalten Sie von CreateShadowRP einen Zeiger auf einen RastPort, in dem der "Schatten" eines Objektes gezeichnet bzw. abgelegt werden kann. Kann die Funktion aus Speicherplatzmangel nicht korrekt durchgeführt werden, dann erhalten Sie Null als Returnwert.

In diesem Kapitel haben Sie fast alle Funktionen der Grafik-Bibliothek kennengelernt. Alle Beispielprogramme und die darin verwendeten Funktionen sind mit Intuition kompatibel.

Im nächsten Kapitel beschäftigen wir uns ausführlich mit Intuition, der grafischen Benutzeroberfläche des Amiga. Sie werden lernen, wie Intuition seine Grafikkomponenten handhabt und wie es mit Ihnen kommuniziert.

Natürlich wird wieder alles anhand von Beispielprogrammen erläutert, die den prinzipiellen Umgang mit den Funktionen veranschaulichen sollen. Auf diese Weise können Sie das Gelernte auf einfache Art und Weise in eigenen Programmen anwenden.

```
/* cliptransparent.c */



#include "exec/types.h"
#include "intuition/intuition.h"
#include "exec/memory.h"

#define WINDOWFLAGS (WINDOWSIZING|WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE)

struct NewWindow nw =
{
   100, 80,                /* Startposition */
   340, 110,               /* Breite, Höhe */
   -1, -1,                 /* Schrift- und Randfarbe */
   0,                      /* IDCMP-Flags */
   WINDOWFLAGS,            /* Window-Flags */
   NULL,                   /* Zeiger auf erstes User-Gadget */
   NULL,                   /* Zeiger auf User-Checkmark */
   "ClipBlitTransparent",  /* Titelzeile */
   NULL,                   /* Zeiger auf Screen */
   NULL,                   /* Zeiger auf Superbitmap */
   60,60,640,200,          /* Maximale Größe */
   WBENCHSCREEN            /* Art des zu öffnenden Screens */
};

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

extern struct RastPort *CreateShadowRP();
extern struct BitMap *CreateShadowBM();
extern struct Window *OpenWindow();


main()
{
   SHORT i;
   int x2, y2, error;

   struct RastPort testRP;
   struct BitMap testBM;


   struct RastPort saveRP;
   struct BitMap saveBM;
```

Listing 4.18: Das Programm "cliptransparent" (Teil 1)

```
struct RastPort *rp;
struct Window *w;

struct RastPort *imageShadowRP;
struct BitMap *imageShadowBM;

SHORT x,y;

GfxBase = (struct GfxBase *)OpenLibrary(
      "graphics.library",0);
if (GfxBase == NULL)
{
   printf("Graphics-Library kann nicht geöffnet werden\n");
   exit(1000);
}
IntuitionBase = (struct IntuitionBase *)
                OpenLibrary("intuition.library",0);
if(IntuitionBase == NULL)
{
  CloseLibrary(GfxBase);
  printf("Intuition-Library kann nicht geöffnet werden\n");
  exit(1000);
}

w = OpenWindow(&nw);          /* Window öffnen */
if(w == NULL)
    goto cleanup;
rp = w->RPort;

InitBitMap(&testBM,2,300,150);
InitBitMap(&saveBM,2,300,150);

error = FALSE;
for(i=0; i<2; i++)
{
   testBM.Planes[i]= (PLANEPTR)AllocRaster(300,150);
   if(testBM.Planes[i] == 0)
       error = TRUE;
}
if(error)
    goto cleanup;
for(i=0; i<2; i++)
{
   saveBM.Planes[i]= (PLANEPTR)AllocRaster(300,150);
   if(saveBM.Planes[i] == 0)
       error = TRUE;
}
```

Listing 4.18: Das Programm "cliptransparent" (Teil 2)

```
    if(error)
       goto cleanup;
    InitRastPort(&testRP);
    testRP.BitMap = &testBM;

    InitRastPort(&saveRP);
    saveRP.BitMap = &saveBM;

    SetAPen(rp,1);
    Move(rp,0,0);
    Draw(rp,200,100);      /* ins Fenster */
    SetAPen(rp,2);
    Move(rp,6,0);
    Draw(rp,206,100);      /* ins Fenster */
    SetAPen(rp,3);
    Move(rp,12,0);
    Draw(rp,212,100);      /* ins Fenster */

    SetAPen(&testRP,2);
    RectFill(&testRP,0,0,130,55);
    SetAPen(&testRP,0); /* In die Mitte des Rechtecks
                           ein Loch schneiden */
    RectFill(&testRP,10,10,120,45);

    x = 20;
    y = 10;
    x2 = 2;
    y2 = 1;

    imageShadowBM = CreateShadowBM(2,130,55);
    if(imageShadowBM==0)
          goto cleanup;
    imageShadowRP = CreateShadowRP(imageShadowBM,NULL);
    if(imageShadowRP==0)
          goto cleanup;

    /* Initialisierung */

    /* zwischenspeichern */
    ClipBlit(rp,x,y,&saveRP,0,0, 130,55, 0xc0);

    /* Dies produziert beim allerersten Mal einen Schatten, so
       daß er später einfach zu benutzen ist. */
    ClipBlitTransparent(&testRP,0,0,
                        rp,x,y,
                        130,55,
                        imageShadowRP,TRUE);
```

Listing 4.18: Das Programm "cliptransparent" (Teil 3)

```
    ClipBlit(&saveRP,0,0, rp,x,y, 130,55, 0xc0);

    /* wiederherstellen */

    for(i=0; i<60; i++)
    {
       ClipBlit(rp,x,y,&saveRP,0,0, 130,55, 0xc0);
       /* save back */
       ClipBlitTransparent(&testRP,0,0,
                      rp,x,y,
                      130,55,
                      imageShadowRP,FALSE);
       /* Schatten benutzen */
       Delay(5);

       /* wiederherstellen */
       ClipBlit(&saveRP,0,0, rp,x,y, 130,55, 0xc0);
       x += x2;
       y += y2;
       if(y < 10 | y > 45)
          {
             y2 = -y2;
             x2 = -x2;
          }
      }
cleanup:
    for(i=0; i<2; i++)
    {
          if(testBM.Planes[i])
           FreeRaster(testBM.Planes[i],300,150);
          if(saveBM.Planes[i])
           FreeRaster(saveBM.Planes[i],300,150);
    }
    if(w) CloseWindow(w);
    DeleteShadowRP(imageShadowRP);
    DeleteShadowBM(imageShadowBM);

    if(IntuitionBase)
       CloseLibrary(IntuitionBase);
    if(GfxBase)
       CloseLibrary(GfxBase);

}       /* Ende von main() */
```

Listing 4.18: Das Programm "cliptransparent" (Teil 4)

```
ClipBlitTransparent(srp,sx,sy,drp,dx,dy,width,height,
shadowRP,makeShadow)
struct RastPort *srp, *drp, *shadowRP;
SHORT sx,sy,dx,dy;
SHORT width,height,makeShadow;
{

   /* Konstruiere eine "Schattenmaske", indem mehrere
      Plane-Masken "geodert" werden! */

   /* wird für den Schatten beim ersten Durchlauf benötigt */
   if(makeShadow)
      ClipBlit(srp,sx,sy,shadowRP,0,0,width,height,0xe0);


   /* Der Wert e0 (hex) bedeutet
          B and C + B not C + C not B,
      vereinfacht zu
          B + C (B OR C)
      entspricht
      "Setzte eine 1, wo immer eine 1 in der Quelle oder im
       Ziel vorkommt."

      Erlaubt alle Planes miteinander zu verknüpfen. */

    /* Benutze "Schatten-Maske", um ein Loch im Zielbereich
       zu erzeugen */

    ClipBlit(shadowRP,0,0,drp,dx,dy,width,height,0x20);

    /* Loch mit Source-Daten füllen */

    ClipBlit(srp,sx,sy,drp,dx,dy,width,height,0xe0);
    return(0);
}

struct BitMap
*CreateShadowBM(depth,width,height)
SHORT depth, width, height;
{

      /* "Schatten-Maske" hat eine Tiefe von einer Plane und
         ist auch nur so breit und hoch wie der Quellbereich.
         Man erhält einen Zeiger auf den betreffenden
         Speicher, wobei es aber nur so aussieht, als hätte
         man 5 oder mehr Bitplanes. */
```

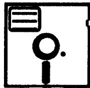
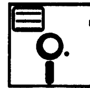
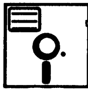
Listing 4.18: Das Programm "cliptransparent" (Teil 5)

```
        SHORT i;
        struct BitMap *shadowBM;
        if((shadowBM =  (struct BitMap *)
           AllocMem(sizeof(struct BitMap),MEMF_CHIP))
           == NULL )
           return(NULL);

        InitBitMap(shadowBM,depth, width,height);

        if((shadowBM->Planes[0] = (PLANEPTR)
           AllocMem(RASSIZE(width,height),
            MEMF_CHIP | MEMF_CLEAR))== NULL)
        {
           FreeMem(shadowBM,sizeof(struct BitMap));
           return(NULL);
        }

        for(i=1; i<depth; i++)
        {
            shadowBM->Planes[i] = shadowBM->Planes[0];
        }
        return(shadowBM);
} /* Ende von CreateShadowBM */


DeleteShadowBM(sbm)
struct BitMap *sbm;
{
        if(sbm != NULL)
        {
            if(sbm->Planes[0] != NULL)
                 FreeMem(sbm->Planes[0],
                      RASSIZE(8 *(sbm->BytesPerRow),sbm->Rows));

            FreeMem(sbm,sizeof(struct BitMap));
        }
        return(0);
}

struct RastPort
*CreateShadowRP(shadowBM, oldshadowRP)
struct BitMap *shadowBM;
struct RastPort *oldshadowRP;
{

        struct RastPort *shadowRP;
```

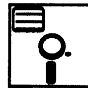
Listing 4.18: Das Programm "cliptransparent" (Teil 6)

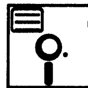
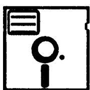
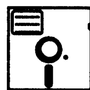
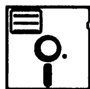
```
        /* Falls "oldshadowRP" kein Null-Pointer ist, können
           wir einfach die neue Bitmap in die breits
           existierende Datenstruktur miteinbinden. */
        if(oldshadowRP == NULL)
        {
        if((shadowRP = (struct RastPort *)
           AllocMem(sizeof(struct RastPort), MEMF_CHIP ))
            == NULL)
            return(NULL);
        InitRastPort(shadowRP);
        }
        else
        {
          /* Alten Wert nutzen, falls vorhanden */
          shadowRP = oldshadowRP;
        }
        /* Bitmap und Rastport verbinden */

        shadowRP->BitMap = shadowBM;
        return(shadowRP);
}

DeleteShadowRP(srp)
struct RastPort *srp;
{

        if(srp != NULL)
            FreeMem(srp,sizeof(struct RastPort));
        return(0);
}
```

Listing 4.18: Das Programm "cliptransparent" (Schluß)

5



Kapitel 5

Die Benutzeroberfläche "Intuition"

In diesem Kapitel werden wir uns genauer mit den Fähigkeiten von Intuition beschäftigen, als wir das in den vorangegangenen Kapiteln getan haben. So werden z.B. die Fenster und Screens, die wir in den Beispielprogrammen des Kapitels 4 verwendet haben, eingehender abgehandelt.

Bei der Arbeit mit Intuition werden Sie feststellen, daß die Variablen von Datenstrukturen meistens Werte enthalten, die ein Objekt bzw. seine Position relativ zu einer anderen Komponente des Grafiksystems darstellen. So wird z.B. die Position eines Fensters relativ zur linken oberen Ecke des zugehörigen Screens angegeben, ein Requester (spezielle Art eines Fensters, Eingabemaske) wird relativ zum zugehörigen Fenster positioniert. Text und Gadgets wiederum werden relativ zu ihrem Requester oder RastPort dargestellt, Menüpunkte und deren Untermenüs relativ zum zugehörigen Menü.

Diese Art der Angabe von Positionswerten erscheint zunächst recht umständlich. Intuition eröffnet dem Programmierer auf diese Weise jedoch ein hohes Maß an Flexibilität und trägt weiterhin dazu bei, den Programmieraufwand zu senken.

Dies wird klar, wenn Sie z.B. einen Requester neu positionieren wollen: weil alle Grafikkomponenten des Requesters relativ zu seiner linken oberen Ecke positioniert wurden, brauchen Sie tatsächlich nur den Requester zu verschieben – alle anderen Komponenten (Text, Gadgets usw.) werden von Intuition automatisch neu ausgerichtet. Wenn Sie erst einmal das Grundprinzip dieser Positionierung verstanden haben, dann gestaltet sich die Erstellung von Fenstern, Requestern, Gadgets usw. sehr viel einfacher, als wenn man absolute Koordinaten verwenden würde.

Die Kommunikation mit Intuition

Die Kommunikation zwischen Intuition und Ihrem Programm spielt sich normalerweise über den IDCMP (Intuition Direct Communication Message Port, etwa: direkte Kommunikationsleitung zwischen Intuition und einem Programm) ab. Zu diesem Zweck teilen Sie Intuition – in Form symbolischer Konstanten – mit, auf welche Ereignisse Ihr Programm reagieren soll, d.h. welche Art von Nachricht (Message) von Intuition an Ihr Programm gesendet werden soll. Tritt ein Ereignis ein, dann sendet Intuition eine Nachricht über den IDCMP an Ihr Programm, sofern Sie diese verwenden wollen. Die Nachrichten von Intuition werden "IntuiMessages" genannt. Intuition kann für die folgenden Ereignisse Nachrichten versenden:

Fenster

Wird ein von Ihrem Programm verwendetes Fenster aktiviert, deaktiviert, verkleinert, vergrößert oder geschlossen, dann kann Intuition eine entsprechende Nachricht verschicken, wenn Sie dies wünschen. Weiterhin kann Intuition Ihr Programm darüber informieren, daß der Inhalt des Fensters neu gezeichnet (refresh) werden muß, wenn es ganz oder teilweise von anderen Objekten verdeckt wurde.

Wenn Sie das Flag "SIZEVERIFY" in Ihrer "NewWindow"-Datenstruktur verwenden, dann können Sie Intuition veranlassen, ein Fenster so lange nicht zu vergrößern oder zu verkleinern, bis dies nicht problemlos geschehen kann. Der Anwender kann dann zwar das Größengadget des Fensters betätigen, aber es behält seine momentane Größe. Dieses Verfahren ist z.B. dann sinnvoll, wenn der Speicherplatz nicht ausreicht, um eine Umkopierung der Daten (denn das Verschieben oder Vergrößern/Verkleinern eines Fensters ist ja nichts anderes) durchzuführen.

Disketten

Intuition kann Ihr Programm weiterhin darüber informieren, ob der Anwender eine Diskette aus einem Laufwerk entfernt hat, die Sie evtl. später aber noch benötigen oder auf der eine Datei noch geöffnet ist. In diesem Fall fordert AmigaDOS den Anwender auf, die Diskette wieder in ein Laufwerk zu legen, damit die Datei geschlossen werden kann. Bevor dies jedoch geschieht, können Sie bereits von Ihrem Programm aus Schritte unternehmen, die Probleme dieser Art unterbinden.

Menüs

Wählt der Anwender einen Punkt aus einem Menü aus, dann sendet Intuition eine Nachricht, die die Nummer des Menüs, des Menüpunktes und ggf. die Nummer des Untermenüs enthält, aus dem ein Punkt selektiert wurde.

Gadgets

Ein Gadget ist eigentlich nichts weiter als ein von Ihnen definiertes Rechteck, das mit der Maus "angeklickt" (selektiert) werden kann, damit Ihr Programm entsprechend weiterarbeiten kann (z.B. ein Gadget, um das Programm zu beenden).

Befindet sich der Mauszeiger über einem Gadget, dann erhält Ihr Programm eine Nachricht, wenn der Anwender den linken Mausknopf drückt oder wieder losläßt. Die Nachricht enthält in diesem Fall die Nummer des selektierten Gadgets.

Tastatur

Solange Ihr Fenster aktiv ist, werden alle Tastatureingaben an dieses Fenster geschickt. Intuition kennt zwei Arten dieser Eingabeart: "RAWKEY" und "VANILLAKEY". Bei der ersten Art erhalten Sie keine darstellbaren Zeichen, sondern nur die Amiga-internen Tastaturcodes, die Sie dann selber weiterverarbeiten müssen. Die zweite Eingabeart – "VANILLAKEY" – enthält die darstellbaren Zeichen der Eingabe. Wird z.B. die Taste <1> gedrückt, dann erhält Ihr Programm den ASCII-Wert dieses Zeichens. In Kapitel 6 finden Sie genauere Informationen zur Eingabeverarbeitung der Tastatur.

Zeitgeber (Timer)

Intuition kann Ereignisse dieser Art jede zehntel Sekunde generieren. Obwohl die Timer (siehe Kapitel 6) unabhängig von anderen Dingen angesprochen und programmiert werden können, sollten Sie diese Möglichkeit von Intuition verwenden, bevor Sie mit dem Timer direkt kommunizieren.

Requester

Wird ein Requester geöffnet, dann sendet Intuition eine Nachricht der Form "REQSET" an Ihr Programm, um Sie darüber zu informieren. Wird der Requester beendet, dann

erhalten Sie die Nachricht "REQCLEAR". Wollen Sie auf Nachrichten der Art "REQVERIFY" reagieren, dann erhalten Sie diese Nachricht von Intuition, bevor ein Requester geöffnet wird.

Sie können so z.B. den Fensterinhalt, der durch die Darstellung des Requesters überdeckt wird, bei einem Fenster des Typs "SIMPLE_REFRESH" zwischenspeichern, damit er später, nachdem der Requester gelöscht wurde, neu gezeichnet werden kann. Der Requester wird erst geöffnet, nachdem Sie Intuition den Empfang der Nachricht bestätigt haben.

Maus

In der IntuitionBase-Datenstruktur ist die Position des "heißen Punktes" des Mauszeigers abgelegt. Dieser Punkt muß sich über einem Objekt befinden, bevor es mit der linken Maustaste selektiert werden kann. Weiterhin werden hier die Koordinaten des Mauszeigers relativ zu allen offenen Fenstern eingetragen. Sie können in speziellen Variablen innerhalb des Systems abgelegt werden. Die Position des Mauszeigers kann jedoch auch relativ zu Objekten von Intuition erfragt werden.

Wenn Sie z.B. in einer Gadgetstruktur das Flag "FOLLOWMOUSE" setzen, dann erhalten Sie – solange das entsprechende Gadget selektiert ist – Nachrichten über jede Bewegung, die mit der Maus gemacht wird. Sie können so entscheiden, ob sich der Mauszeiger überhaupt noch über dem selektierten Gadget befindet, wenn die linke Maustaste losgelassen wird.

In der NewWindow-Datenstruktur können Sie ebenfalls festlegen, ob Ihr Programm Nachrichten empfangen soll, wenn die Maus bewegt wird. Die Flags hierfür heißen "REPORTMOUSE" und "DELTAMOVE". Das erste Flag sagt Intuition, daß die Koordinaten des Mauszeigers relativ zur linken oberen Ecke Ihres Fensters angegeben werden soll. Das zweite Flag gibt die Mausposition relativ zum letzten "Aufenthaltsort" des Mauszeigers an.

Die meisten der oben aufgeführten Nachrichtenarten werden später in diesem Kapitel genauer besprochen.

Nachrichten (Messages) von Intuition

Wann immer Sie eine Nachricht von Intuition erhalten, sollten Sie versuchen, den Empfang der Nachricht so schnell wie möglich zu bestätigen. Taucht ein Ereignis auf, auf das Ihr Programm reagieren soll, dann schaut Intuition zunächst einmal nach, ob am IDCMP Nachrichten anliegen, die wiederverwendet werden können. Ist dies der Fall, dann füllt Intuition die IntuiMessage-Datenstruktur der Nachricht mit den neuen Werten und sendet diese Nachricht dann an Ihr Programm.

Sind jedoch keine Nachrichten vorhanden, dann legt Intuition einen neuen leeren Message-Block für diese Nachricht an und verschickt sie dann. Wenn Ihr Programm lange braucht, um eine Nachricht zu bestätigen (unter Verwendung eines ReplyPorts), dann belegt Intuition für jede weitere Nachricht Speicherplatz in dem für Ihr Programm reservierten Bereich. Im Extremfall stürzt Ihr Programm dann nach einiger Zeit ab, da kein freier Speicher mehr zur Verfügung steht. Laufen mehrere Programme zugleich auf dem System, dann wird u.U. die Arbeitsgeschwindigkeit der Programme langsamer. In einem solchen Fall dauert es dann natürlich noch länger, bis der Empfang bestätigt werden kann. Denken Sie daran: sobald Intuition Speicher für eine Nachricht reserviert, dann ist dieser bis zum Programmende oder Neustart des Systems nicht mehr verfügbar.

Es ist also wichtig, so schnell wie möglich Nachrichten zu bestätigen, damit der Speicherplatz nicht unnötig verschwendet wird. Denn durch die Bestätigung kann Intuition den Message-Block dieser "alten" Nachricht wiederverwenden, wenn eine neue Nachricht versendet werden muß; es wird in diesem Fall also kein Speicher reserviert.

Die wohl schnellste und beste Methode der Bestätigung ist es, den Inhalt der IntuiMessage in eine eigene Datenstruktur zu kopieren und dann direkt den Empfang zu bestätigen, noch bevor die Nachricht bzw. ihr Inhalt ausgelesen und analysiert wird. Diese Art der Nachrichtenverarbeitung verwenden wir später in diesem Kapitel.

Der Inhalt einer "IntuiMessage"

Die IntuiMessage-Datenstruktur besteht aus einzelnen Datenfeldern. Das wohl wichtigste dieser Felder ist das "Class"-Feld, in dem die Art der Nachricht, die verschickt wurde, definiert ist.

Nachfolgend finden Sie den Inhalt einer typischen "IntuiMessage"-Datenstruktur, wie sie auch in der Include-Datei "intuition/intuition.h", die sich auf Ihrer Compilerdiskette befindet, enthalten ist.

```
struct IntuiMessage
{
  ULONG Class;
  USHORT Code;
  USHORT Qualifier;
  APTR IAddress;
  SHORT MouseX,MouseY;
  ULONG Seconds,Micros;
  struct Window *IDCMPWindow;
  struct IntuiMessage *SpecialLink;
                              /* Nur vom System verwendet */
};
```

Das "Class"-Datenfeld

In diesem Feld ist die Art der Nachricht, die Intuition gesendet hat, enthalten. Die möglichen Werte dieses Feldes sind mit denen identisch, die Sie in der "NewWindow"-Datenstruktur im "IDCMP"-Feld angeben können. Hier geben Sie die Nachrichten, auf die Ihr Programm reagieren soll, in Form symbolischer Konstanten an – Sie können also auch beim Auslesen des "Class"-Feldes die Namen dieser Konstanten verwenden.

Das "Code"-Datenfeld

Dieses Feld enthält Werte, die mit Menüs in Verbindung stehen. Hat der Anwender einen Menüpunkt ausgewählt, dann erhalten Sie von Intuition eine Nachricht der Art "MENUPICK" im Feld "Class". Das Feld "Code" enthält in diesem Fall die Nummer des ausgewählten Menüpunktes. Enthält "Code" die Nachrichtenart "MENUVERIFY", dann erfahren Sie durch Auslesen des "Code"-Feldes, auf welche Art Bestätigung Intuition von Ihrem Programm wartet.

Das "Qualifier"-Datenfeld

Bei diesem Datenfeld handelt es sich um eine Kopie des "Qualifier"-Feldes der Datenstruktur des aktuellen Eingabeereignisses. (Eine IntuiMessage ist genau genommen immer eine Kopie einer anderen Systemnachricht. Es werden le-

diglich von Intuition einige Modifikationen vorgenommen, z.B. werden einige neue Komponenten hinzugefügt.)

Dieses Feld wird von Intuition initialisiert, wenn ein Ereignis (eine Eingabe) durch die Maus, die Tastatur oder den Zeitgeber (Timer) hervorgerufen wurde. Die möglichen Werte, die dieses Feld enthalten kann, finden Sie in der Datei "devices/inputevent.h", die sich auf Ihrer Compilerdiskette befindet.

Einige dieser Werte sind "shift-key-down", "left-alt-key-down" und "ctrl-key-down", die immer dann gesetzt werden, wenn eine Taste zusammen mit der <SHIFT>-, <CTRL>- oder <ALT>-Taste gedrückt wurde.

So können Sie z.B. entscheiden, wie in einer Textverarbeitung der Cursor positioniert werden soll: die Taste <→> alleine bewegt den Cursor um ein Zeichen nach rechts; <SHIFT> <→> hingegen könnte den Cursor ans Ende der aktuellen Zeile bewegen. Durch das Auslesen des Qualifier-Feldes gestaltet sich die Entscheidung, ob Sondertasten gedrückt wurden, recht einfach. Es ist jedoch meistens nur bei Tastatureingaben erforderlich, dieses Datenfeld auszulesen.

Das "IAddress"-Datenfeld

Intuition initialisiert dieses Feld, wenn eine Gadget-spezifische Nachrichtenart generiert wurde. In einem solchen Fall enthält "IAddress" die Startadresse der "Gadget"-Datenstruktur des selektierten Gadgets. Unter Verwendung dieser Adresse können Zeiger erstellt werden, mit deren Hilfe jede Komponente der "Gadget"-Datenstruktur angesprochen werden kann (z.B. "GadgetID", um die Nummer des selektierten Gadgets zu erfahren und so entsprechend reagieren zu können).

Die "MouseX"- und "MouseY"-Datenfelder

Diese Datenfelder enthalten – unabhängig von der Nachrichtenart – die Koordinaten des Mauszeigers relativ zur linken oberen Ecke des Screens, in der sich der Zeiger gerade befindet. Die aktuelle Mausposition wird von Intuition in einer "IntuitionBase"-Datenstruktur verwaltet, wobei als Bildschirmauflösung grundsätzlich 640 mal 400 Pixel (Bildpunkte) verwendet werden. Aus diesem Grund kann der Zeiger nur an 320 möglichen horizontalen Punkten erschei-

nen, da die interne Verwaltung immer mit der doppelten Auflösung des sichtbaren Bereiches durchgeführt wird. Dies ist mit Rücksicht auf spätere Systemerweiterungen geschehen. Weiterhin wird so das Zeichnen in hoher Auflösung ermöglicht.

Enthält das Feld "Class" eine Nachricht der Art "DELTAMOVE", dann geben "MouseX" und "MouseY" die Länge und die Richtung der letzten Zeigerbewegung an. Sind diese Werte positiv, dann wurde der Mauszeiger nach rechts oder nach oben bewegt; sind sie negativ, dann ist der Zeiger nach links oder nach unten bewegt worden. Intuition hört selbst dann nicht auf, die Datenfelder zu füllen, wenn sich der Zeiger außerhalb des aktuellen Screens befindet; die Werte sind dann allerdings nicht mehr relevant.

Erhalten Sie eine Nachricht der Form "MOUSEMOVE", dann enthalten "MouseX" und "MouseY" die Koordinaten, an dem sich der Mauszeiger bei der Formulierung der Nachricht befand. Beachten Sie bitte, daß die Werte von "MouseX" und "MouseY" immer gültig sind, egal, welche Art Nachricht Sie empfangen. Die Koordinaten des Mauszeigers können also auch bei einer Nachricht der Art "MENUPICK", "GADGETDOWN" oder "INTUITICKS" bestimmt werden. Sie sehen also, daß Nachrichtenarten und Zeigerposition immer miteinander verknüpft sind.

Wenn Sie z.B. ein Gadget erstellt haben, dann bekommen Sie von Intuition die Nachricht "GADGETDOWN", wenn der Anwender es selektiert. Wenn Sie zusätzlich noch die Werte von "MouseX" und "MouseY" auslesen, dann können Sie feststellen, an welcher Stelle das Gadget "angeklickt" wurde. Sie brauchen hierzu nur die Werte mit dem Inhalt der Variablen "LeftEdge", "TopEdge", "Width" und "Height" aus der "Gadget"-Datenstruktur zu vergleichen. Im Listing 5.1 finden Sie ein Programmfragment für eine solche Anwendung. Vor dem Aufruf der Routine muß allerdings schon von Intuition eine Nachricht der Art "GADGETDOWN" empfangen worden sein, die dann an diese Routine weitergeleitet wird.

Die "Seconds"- und "Micros"-Datenfelder

Jede "IntuiMessage"-Datenstruktur enthält diese beiden Datenfelder, die den Zeitpunkt angeben, an dem die Nachricht verschickt wurde. Es treten niemals zwei IntuiMessages auf, bei denen die Werte von "Seconds" und "Micros" gleich sind. So kann eine Nachricht von Ihrem Programm aus eindeutig identifiziert werden, falls mehrere des gleichen Typs vorliegen.

```
WhereInGadget(im)
struct IntuiMessage   *im;
{
  struct Gadget  *g;

  g = (struct Gadget *)im->IAddress;  /* Welches Gadget wurde selektiert? */

  if((im->MouseX - g->LeftEdge)<4)
      printf("Gadget wurde in der Nähe der linken Ecke selektiert.\n");

  else   if(( g->LeftEdge  + g->Width-1 - im->MouseX)<4)
      printf("Gadget wurde in der Nähe der rechten Ecke selektiert.\n");

  else printf("Gadget wurde in der Nähe der Mitte (x-Achse) selektiert.\n");

  if((im->MouseY - g->TopEdge)<4)
      printf("Gadget wurde in der Nähe der oberen Ecke selektiert.\n");

  else   if(( g->LeftEdge  + g->Width-1 - im->MouseY)<4)
      printf("Gadget wurde in der Nähe der unteren Ecke selektiert.\n");

  else printf("Gadget wurde in der Nähe der Mitte (y-Achse) selektiert.\n");
}
```

Listing 5.1: Bestimmen, an welcher Stelle ein Gadget selektiert wurde

Das "IDCMPWindow"-Datenfeld

Dieses Feld enthält einen Zeiger auf das Fenster, das für die Versendung einer IntuiMessage "verantwortlich" ist. Dieses Datenfeld ist erforderlich, da mehrere Nachrichten, die von verschiedenen Fenstern angeregt wurden, an einem einzigen Message-Port ankommen können, der vielleicht von allen Fenstern zugleich verwendet wird. Durch das Auslesen dieses Feldes kann Intuition dann feststellen, welches Fenster ein Ereignis hervorgerufen hat.

Beispielprogramm: Anwendung einer IDCMP-Routine

Im Listing 5.2 finden Sie ein Programm, das eine Erweiterung des "HandleEvent.c"-Programms darstellt, welches bereits in Kapitel 4 besprochen wurde. Die folgende Routine verarbeitet alle möglichen Nachrichten, die am IDCMP anliegen können. Weiterhin wird der Datenblock der ankommenden Nachricht direkt umkopiert, so daß noch vor der Weiterverarbeitung die Nachricht be-

stätigt werden kann. Sie werden sicher nicht alle Nachrichtenarten verwenden, daher können Sie die ensprechenden "case"-Abfragen aus dem Programm entfernen. Das Programm verarbeitet jedoch alle Nachrichtenarten, um Sie damit bekannt zu machen.

Einige Nachrichten, die Intuition sendet, sind "Mutual Exclusive", d.h. der Empfang einer bestimmten Nachrichtenart schließt den einer anderen aus, auch dann, wenn auf Nachrichten beider Art gewartet werden soll. Hier eine Auflistung dieser Nachrichtenarten:

- Wollen Sie Nachrichten vom Typ "DELTAMOVE" empfangen, dann können Sie nicht zugleich Nachrichten vom Typ "MOUSEMOVE" abfragen. ("DELTAMOVE" berechnet die Position des Mauszeigers auf eine andere Art.)

- Bei Nachrichten vom Typ "VANILLAKEY" schließen Sie solche des Typs "RAWKEY" aus. ("VANILLAKEY" wandelt die Tastaturcodes direkt in ASCII-Werte um.)

- Ist das Flag "MOUSEBUTTONS" gesetzt, dann werden Nachrichten vom Typ "GADGETDOWN" oder "GADGETUP" unterdrückt, die von Intuition dann verschickt werden, wenn ein Gadget eines Fensters selektiert wurde. Bei Requestern hingegen erhalten Sie auch weiterhin "GADGETDOWN" oder "GADGETUP"-Nachrichten. Die linke Maustaste dient normalerweise zur Selektion eines Gadgets. Es wird beim Drücken oder Loslassen dieses Knopfes immer nur eine Nachricht verschickt. Nachrichten vom Typ "MOUSEBUTTONS" haben gegenüber "GADGETDOWN" oder "GADGETUP" Vorrang.

Im Listing 5.3 finden Sie eine Routine, die die Nachrichtenarten aus der Routine im Listing 5.2 filtert, die im Malprogramm, das wir nun entwickeln wollen, nicht benötigt werden.

Entwicklung eines Malprogramms

Die Erstellung und Programmierung eines Malprogrammes ist recht schwierig, da viele Dinge verwaltet, dargestellt und berechnet werden müssen. Das kleine Malprogramm, das wir in diesem Kapitel erstellen wollen, ist recht einfach gehalten. In der Hauptsache soll immer dann, wenn der Anwender den

```
/* event2.c */

extern struct Window *w;

GadgetDown(m) struct IntuiMessage *m; { return(1); }

DoCloseWindow(m)
struct IntuiMessage *m;
{
  return(FALSE);    /* Das Fenster wird in main() geschlossen */
}

HandleEvent(ms)
struct IntuiMessage *ms;
{
  struct IntuiMessage *MeineIntuiMessage;
  int i;
  UBYTE *s,*d;
  int result;

  /* Die Variable <result> bestimmt, ob das Programm beendet werden
     soll oder nicht. Hat sie den Wert 0, dann wird das Programm
     beendet, ist sie ungleich 0, dann wird das Programm normal
     weitergeführt. */

  s = (UBYTE *)ms;    /* Die "echte" IntuiMessage */
  d = (UBYTE *)&MeineIntuiMessage;  /* ...und die Kopie */

  /* IntuiMessage kopieren */
  for(i=0;i<sizeof(struct IntuiMessage);i++)  *d++ = *s++;

  ReplyMsg(ms);    /* Und Empfang direkt bestätigen */

  /* Wir verwenden jetzt die Kopie der IntuiMessage zur Bestimmung
     der Nachrichtenart. Die Funktionen, die abhängig von der Art der
     Nachricht aufgerufen werden, erhalten die Startadresse der
     Intui-Message als Parameter, so daß Komponenten der Struktur
     ausgelesen werden können. Handelt es sich um ein zeitkritsches
     Programm, dann kann das Kopieren der IntuiMessage wegfallen; in
     diesem Fall wird die Startadresse der "echten" IntuiMessage den
     Funktionen als Parameter übergeben. */

  switch(MeineIntuiMessage.Class)
            {
              case SIZEVERIFY:
                       result = SizeVerify(&MeineIntuiMessage);
                       break;
```

Listing 5.2: Routine zur Verarbeitung von IDCMP-Nachrichten (Teil 1)

```
        case NEWSIZE:
                result = NewSize(&MeineIntuiMessage);
                break;

        case REFRESHWINDOW:
                result = RefreshWindow(&MeineIntuiMessage);
                break;

        case MOUSEBUTTONS:
                result = MouseButtons(&MeineIntuiMessage);
                break;

        case MOUSEMOVE:
                result = MouseMove(&MeineIntuiMessage);
                break;

        case GADGETDOWN:
                result = GadgetDown(&MeineIntuiMessage);
                break;

        case GADGETUP:
                result = GadgetUp(&MeineIntuiMessage);
                break;

        case REQSET:
                result = ReqSet(&MeineIntuiMessage);
                break;

        case MENUPICK:
                result = MenuPick(&MeineIntuiMessage);
                break;

        case CLOSEWINDOW:
                result = DoCloseWindow(&MeineIntuiMessage);
                break;

        case RAWKEY:
                result = RawKey(&MeineIntuiMessage);
                break;

        case REQVERIFY:
                result = ReqVerify(&MeineIntuiMessage);
                break;

        case REQCLEAR:
                result = ReqClear(&MeineIntuiMessage);
                break;
```

Listing 5.2: Routine zur Verarbeitung von IDCMP-Nachrichten (Teil 2)

```
                case MENUVERIFY:
                        result = MenuVerify(&MeineIntuiMessage);
                        break;

                case NEWPREFS:
                        result = NewPrefs(&MeineIntuiMessage);
                        break;

                case DISKINSERTED:
                        result = DiskInserted(&MeineIntuiMessage);
                        break;

                case DISKREMOVED:
                        result = DiskRemoved(&MeineIntuiMessage);
                        break;

                case WBENCHMESSAGE:
                        result = WBenchMessage(&MeineIntuiMessage);
                        break;

                case ACTIVEWINDOW:
                        result = ActiveWindow(&MeineIntuiMessage);
                        break;

                case INACTIVEWINDOW:
                        result = InactiveWindow(&MeineIntuiMessage);
                        break;

                case DELTAMOVE:
                        result = DeltaMove(&MeineIntuiMessage);
                        break;

                case VANILLAKEY:
                        result = VanillaKey(&MeineIntuiMessage);
                        break;

                case INTUITICKS:
                        result = Intuiticks(&MeineIntuiMessage);
                        break;

                 default:  result = 1;
                                break;
                 };

    return(result);

    /* Anhand des Wertes von <result> wird entschieden, ob das Programm
     beendet werden soll oder nicht. */

}
```

Listing 5.2: Routine zur Verarbeitung von IDCMP-Nachrichten (Schluß)

```
/* stubs1.c */

#define IM     struct IntuiMessage     *msg; {return(1); }

int SizeVerify(msg)              IM
int NewSize(msg)                 IM
int RefreshWindow(msg)           IM
int MouseMove(msg)               IM
int ReqSet(msg)                  IM
int RawKey(msg)                  IM
int VanillaKey(msg)              IM
int ReqVerify (msg)              IM
int ReqClear(msg)                IM
int MenuVerify(msg)              IM
int NewPrefs(msg)                IM
int DiskInserted(msg)            IM
int DiskRemoved(msg)             IM
int WBenchMessage(msg)           IM
int ActiveWindow(msg)            IM
int InactiveWindow(msg)          IM
int DeltaMove(msg)               IM
int DoCloseWindow(msg)           IM
```

Listing 5.3: Routine zur Filterung von IDCMP-Nachrichten

linken Mausknopf drückt, ein farbiger Punkt gesetzt werden. Wird die Maus bewegt, ohne daß der Knopf gedrückt wird, dann passiert nichts. Weiterhin stellt das Programm einen Requester zum Aufruf zur Verfügung, mit dem farbiger Text an jede beliebige Stelle des Grafikbereiches gesetzt werden kann.

Bilder können mit diesem Programm nicht geladen oder abgespeichert werden. Vielleicht haben Sie Lust, das Programm um diese Möglichkeiten zu erweitern? Beendet wird das Programm durch das "Anklicken" des Schließ-Gadgets des Fensters. Um ein Programm dieser Art erstellen zu können, müssen Sie u.a. folgendes wissen:

- Wie man einen Screen auswählt

- Wie die aktuelle Position des Mauszeigers bestimmt wird

- Wie der Status der Maustasten abgefragt wird

```
/* myscreen2.c */

struct TextAttr TestFont = { "topaz.font",8,0,0 };

struct NewScreen ns = {
0,0,              /* Linke obere Ecke */
320,200,4,        /* Breite, Höhe, Anzahl Bitplanes */
0,1,    /*  APen, BPen */
0,   /* Darstellungsmodus */
CUSTOMSCREEN,   /* Art des Screens */
&TestFont,    /* Verwendeter Zeichensatz */
"Custom Screen",    /* Name des Screens */
NULL      /* Zeiger auf Gadgets */
};


struct NewWindow nw =
{
  0,10,   /* Linke obere Ecke */
  WWIDTH,WHEIGHT,    /* Breite, Höhe */
  0,1,  /* APen, BPen */
  INTUITICKS|GADGETUP|GADGETDOWN|
  MOUSEBUTTONS|MENUPICK|CLOSEWINDOW,    /* IDCMP-Flags */
  WINDOWCLOSE|ACTIVATE,  /* Flags */
  NULL,  /* Zeiger auf erstes Gadget */
  NULL,  /* Zeiger auf CheckMark */
  "Magix Malprogramm",    /* Name des Fensters */
  NULL,  /* Zeiger auf zugehörige Screen */
  NULL,  /* Zeiger auf eigene BitMap */
  0,0,0,0,  /* Ignoriert, da sich das Fenster nicht vergrößern läßt */
  CUSTOMSCREEN    /* Art des Screens */
};
```

Listing 5.4: Datenstrukturen für Zeichensatz, Screen und Fenster

- Wie man den Timer abfragt, um den Zeichenvorgang zu steuern
- Wie man Menüs erstellt und verwendet
- Wie man eine Farbe auswählt
- Wie man Gadgets erstellt und verwendet

Die Auswahl des Bildschirms (Screen)

Da wir im Malprogramm Farben verwenden wollen, müssen wir uns entscheiden, auf welcher Art Screen wir unser Fenster öffnen lassen. Die Anzahl verfügbarer Farben hängt von der "Tiefe" des Screens ab, d.h. aus wieviel Bitplanes sie besteht. Auf der Workbench stehen uns nur 4 Farben zur Verfügung, da sie aus nur einer Bitplane besteht.

Wir öffnen aus diesem Grund einen eigenen (Custom-) Screen mit 4 Bitplanes, d.h. uns stehen 16 mögliche Farben zur Verfügung. In der Abb. 5.4 finden Sie die Datenstrukturen eines Custom-Screens sowie die des verwendeten Fensters und Zeichensatzes.

Die "NewScreen"-Datenstruktur

Im Kapitel 4 haben wir bereits eigene (Custom-) Screens verwendet, sind jedoch nicht näher auf die Zusammenhänge eingegangen. Da Screens bei der Programmierung von Intuition eine große Rolle spielen, wollen wir nun die "NewScreen"-Datenstruktur genauer untersuchen. Diese Datenstruktur ist wie folgt definiert:

```
struct NewScreen
{
  SHORT LeftEdge, TopEdge, Depth, Width, Height;
  UBYTE DetailPen, BlockPen;
  USHORT ViewModes;
  USHORT Type;
  struct TextAttr *Font;
  UBYTE *DefaultTitle;
  struct Gadget *Gadgets;
  struct BitMap *CustomBitMap;
};
```

Die Komponenten "LeftEdge" und "TopEdge" haben meistens den Wert Null. Das bedeutet, daß die linke obere Ecke des Screens beim Öffnen immer mit der linken oberen Ecke des momentan sichtbaren Bereiches übereinstimmt.

Die Höhe (Height) des Screens beträgt 200 bzw. 256 Rasterzeilen im "NonInterlace"-Modus und 400 bzw. 512 im Interlace-Modus.

Es ist jedoch nicht zwingend, die linke obere Ecke eines Screens in die Koordinate 0/0 zu legen. Sie können z.B. einen "geteilten" Bildschirm erstellen, der aus zwei Screens besteht. Einen Screen öffnen Sie dann z.B. im Punkt 0/0, den anderen im Punkt 0/100. Sie können so einen Grafikbereich mit zwei unterschiedlichen Auflösungen schaffen und können zusätzlich in jedem Screen eine andere Farbpalette verwenden. Die X-Koordinate ("LeftEdge") kann zwar Werte ungleich Null haben, wird vom Amiga jedoch immer so behandelt, als würde sie den Wert Null beinhalten, da der Amiga in seiner momentanen Hardwarekonfiguration Screens nur vertikal, jedoch nicht horizontal verschieben kann.

Das Datenfeld "Depth" bestimmt, aus wie vielen Bitplanes der Screen bestehen soll, d.h. wieviel Farben zur Verfügung stehen. Die gültigen Werte für diese Variablen sind 1, 2, 3, 4 oder 5, mit denen dann jeweils 2, 4, 8, 16 oder 32 Farben gleichzeitig verfügbar sind.

"DetailPen" ist die Nummer des Farbregisters, mit dessen Farbe der Text in der Titelleiste des Screens geschrieben werden soll. Die Farbe des "BlockPen" wird – zusammen mit "DetailPen" – zum Zeichnen der Tiefen-Gadgets des Screens verwendet.

Mit "ViewModes" legen Sie den Darstellungsmodus des Screens fest. Die Beispielprogramme in diesem Buch verwenden Screens der Arten "LORES" (niedrige Auflösung, "ViewModes" hat in diesem Fall den Wert Null), "HIRES" (hohe Auflösung) oder "LACE" (Interlace Modus).

Ein Screen niedriger Auflösung kann bis zu 354 Pixel breit sein, im "HIRES"-Modus sind es 704 Pixel. In niedriger und hoher Auflösung beträgt die Höhe eines Screens normalerweise 200 Pixel; in der PAL-Version des Amiga sind es 256 Pixel. Geben Sie im Datenfeld "ViewModes" den Darstellungsmodus "LACE" an, dann beträgt die vertikale Auflösung eines Screens 400 (bzw. 512 bei PAL) Pixel, wobei sie aber immer noch als Ganzes sichtbar bleibt. Diese Art Auflösung erlaubt detailreiche Grafiken, jedoch wird die Darstellung bisweilen durch ein unangenehmes Bildschirmflackern getrübt, das gerade bei kontrastreichen Grafiken oft auftritt.

Die Variable "Type" legt fest, um welchen Screentyp – "WBENCHSCREEN" oder "CUSTOMSCREEN" – es sich handelt. Normalerweise wird nur ein Workbench-Screen vom System verwaltet; neue Screens sind also immer vom Typ "CUSTOMSCREEN". Beachten Sie bitte, daß das Datenfeld "Type" in der "NewWindow"-Datenstruktur einen Einfluß auf die Verwendung bestimmter anderer Datenfelder hat.

Das Datenfeld "Font" legt fest, welcher Zeichensatz in diesem Screen zur Ausgabe von Text (Name des Screens usw.) verwendet werden soll. Der Name eines Screens wird im Feld "DefaultTitle" abgelegt. Hierbei handelt es sich um einen Zeiger auf die Zeichenkette, die den Namen enthält. Der Name kann jedoch auch direkt – eingeschlossen in Anführungszeichen – angegeben werden. Sichtbar wird er, wenn die Titelleiste des Screens zu sehen ist.

Das Feld "Gadgets" ist für spätere Erweiterungen der Systemsoftware vorgesehen und wird momentan nicht verwendet. Wollen Sie Ihre eigene BitMap für den Screen verwenden, dann übergeben Sie der Datenstruktur im Feld "CustomBitMap" den Zeiger auf die Anfangsadresse des Speicherbereiches, in dem die BitMap liegt.

Handelt es sich um einen Null-Zeiger, d.h. Sie verwenden keine eigene BitMap, dann reserviert das System vor dem Öffnen des Screens automatisch den erforderlichen Speicherplatz. Beim Schließen eines Screens wird der von ihm belegte Speicherplatz wiederum automatisch freigegeben.

Die Beispiele in diesem Buch verwenden keine eigenen BitMaps; "CustomBitMap" hat immer den Wert Null.

Die "NewWindow"-Datenstruktur

Bei der Arbeit mit Intuition benötigen Sie eine weitere Datenstruktur: "NewWindow". Der Strukturaufbau ist nachfolgend aufgeführt.

```
struct NewWindow
{
  SHORT LeftEdge, TopEdge;
  SHORT Width, Height;
  UBYTE DetailPen, BlockPen;
  ULONG IDCMPFlags;
  USHORT Flags;
  struct Gadget *FirstGadget;
  struct Image *CheckMark;
  UBYTE *Title;
  struct Screen *Screen;
  struct BitMap *BitMap;
  SHORT MinWidth, MinHeight;
  SHORT MaxWidth, MaxHeight;
  USHORT Type;
};
```

"LeftEdge" und "TopEdge" geben an, an welcher Stelle im Screen – relativ zu seiner linken oberen Ecke – das Fenster geöffnet werden soll. "Width" und "Height" legen die Breite und Höhe des Fensters fest (inklusive Titelleiste und Umrandung). Der Name des Fensters wird an die Variable "Title" übergeben. Genau wie bei Screens geben auch bei Fenstern "DetailPen" und "BlockPen" an, mit welchen Farben die Umrandung, Text und die Gadgets gezeichnet werden sollen. Die gültigen Werte für das "IDCMPFlags"-Datenfeld finden Sie im Kapitel 4. Mit dem Datenfeld "Flags" legen Sie die Charakteristik eines Fensters fest, z.B. welche System-Gadgets es erhalten soll, ob man es vergrößern oder verkleinern kann usw. Nachfolgend finden Sie alle gültigen Angaben und die entsprechenden Erläuterungen.

WINDOWSIZING	Wenn Sie dieses Flag setzen, dann kann der Anwender das Fenster vergrößern bzw. verkleinern. Die Grenzen, in denen dies geschehen kann, geben Sie in der Datenstruktur durch die Variablen "MinWidth", "MinHeight" (für die kleinste erlaubte Größe des Fensters), "MaxWidth" und "MaxHeight" (für die maximale Größe des Fensters) an. Beachten Sie bitte, daß der Wert von "Width" zwischen denen von "MinWidth" und "MaxWidth" liegen muß; "Height" muß zwischen "MinHeight" und "MaxHeight" liegen.
WINDOWDRAG	Wollen Sie das Fenster verschieben können, dann setzen Sie dieses Flag.
WINDOWCLOSE	Durch die Angabe dieses Flags erhält das Fenster vom System das Schließ-Gadget, welches sich in der linken oberen Ecke des Fensters befindet. Durch das Betätigen dieses Gadgets wird das Fensters vom System jedoch nicht automatisch geschlossen; Sie müssen dieses Ereignis von Ihrem Programm aus über den IDCMP abfragen. Zu diesem Zweck müssen Sie außerdem das Flag "CLOSEWINDOW" im "IDCMPFlags"-Datenfeld setzen.
WINDOWDEPTH	Ist dieses Flag gesetzt, dann erhält das Fenster vom System das Tiefen-Gadget, mit dem es in den Hinter- bzw. Vordergrund "geklickt" werden kann. Ist das Fenster vom Typ "BACKDROP" (s.u.), dann bleibt die Betätigung des Gadgets ohne Wirkung.

BACKDROP

Ein Fenster, das dieses Flag gesetzt hat, wird immer hinter allen anderen Fenstern geöffnet, kann also nie vor einem anderen Fenster erscheinen. Auf diese Weise wird Speicherplatz gespart, da ein Backdrop-Fenster die BitMap des zugehörigen Screens verwenden kann. Wird allerdings ein Fenster vor einem Backdrop-Fenster geöffnet, dann wird zusätzlich Speicherplatz reserviert.

REPORTMOUSE

Wenn Sie dieses Flag setzen, dann schickt Intuition bei jeder Bewegung des Mauszeigers eine entsprechende Nachricht an Ihr Programm. Das können u.U. eine Menge von Ereignissen sein, die jedoch nicht alle für Sie interessant sind.

GIMMEZEROZERO

Wenn ein Fenster geöffnet wird, dann beziehen sich Koordinatenangaben immer auf das Fenster als Ganzes, d.h. die Umrandung des Fensters wird mit einbezogen. Der Grafikbereich des Fensters liegt jedoch innerhalb der Umrandung. Aus diesem Grunde verwaltet Intuition zwei Arten von Mauszeiger-Koordinaten. "MouseX" und "MouseY" beziehen sich immer auf die linke obere Ecke des Fensters inklusive Umrandung; "GZZMouseX" und "GZZMouseY" beziehen sich auf die linke obere Ecke des Grafikbereiches, der innerhalb der Fensterumrandung liegt.

Dieser zweite Koordinatensatz ist jedoch nur verfügbar, wenn das "GIMMEZEROZERO"-Flag gesetzt ist. Alle Koordinatenangaben (Textausgabe, Grafikfunktionen usw.) beziehen sich bei einem solchen Fenster nicht auf die linke obere Ecke der Umrandung, sondern auf die des Grafikbereiches des Fensters.

BORDERLESS

Wenn Sie dieses Flag setzen, dann wird Ihr Fenster ohne die zugehörige Umrandung dargestellt. Haben Sie allerdings das "WINDOWDRAG"-Flag gesetzt, dann wird trotzdem die Umrandung gezeichnet, da das System zum Zeichnen der Systemgadgets die Umrandung automatisch darstellt.

ACTIVATE — Mit diesem Flag wird Ihr Fenster beim Öffnen automatisch das aktive Fenster des Systems. Manchmal ist es jedoch nicht erforderlich, ein Fenster automatisch aktivieren zu lassen, z.B. wenn es sich um ein Fenster handelt, in dem nur Daten aus-, jedoch nicht eingegeben werden.

RMBTRAP — Mit diesem Flag können Sie feststellen, ob die rechte Maustaste betätigt wurde; Intuition teilt Ihnen dieses Ereignis über den IDCMP mit. Ist das Flag nicht gesetzt, dann werden Ereignisse dieser Art direkt an Intuition weitergeleitet, damit Menüs (o.ä.) gesteuert werden können.

Die folgenden Flags sind "Mutual Exclusive", d.h. es kann immer nur eins gesetzt werden, niemals zwei oder mehr, da durch das Setzen eines der Flags alle anderen unwirksam werden.

SMART_REFRESH — Bei einem Fenster dieser Art wird der Fensterinhalt zwischengespeichert, wenn es komplett oder teilweise überdeckt wird. Werden überdeckte Teile wieder sichtbar, dann stellt Intuition den Fensterinhalt automatisch wieder her.

SIMPLE_REFRESH — Hier geht der Fensterinhalt teilweise oder ganz verloren, je nachdem, wieviel vom Fenster durch andere Objekte verdeckt wird. Das Programm, das dieses Fenster öffnet, ist für die Wiederherstellung des Fensterinhaltes zuständig. Damit Sie wissen, ob der Inhalt neu gezeichnet werden muß, sollten Sie das "REFRESHWINDOW"-Flag im "IDCMPFlags"-Datenfeld setzen, damit Intuition eine entsprechende Nachricht an Ihr Programm sendet.

SUPER_BITMAP — Fenster dieses Typs verwenden ihre eigene BitMap. SuperBitMap-Fenster finden Sie im Kapitel 4 erklärt.

Im Datenfeld "FirstGadget" übergeben Sie einen Zeiger auf das erste Gadget einer verknüpften Liste aller Gadgets, die Sie in diesem Fenster verwenden wollen. Damit Sie wissen, welches Gadget vom Anwender selektiert wurde,

müssen Sie im "IDCMPFlags"-Datenfeld die Flags "GADGETDOWN" bzw. "GADGETUP" setzen. Nähere Informationen zu Gadgets finden Sie später in diesem Kapitel.

Der Parameter "CheckMark" ist ein Zeiger auf eine "Image"-Datenstruktur, in der Sie Ihre eigene Checkmark ("√") definieren können. Die Checkmark wird von Intuition immer dann verwendet, wenn ein Gadget oder ein Menüpunkt selektiert wurde. Sie sehen so auf einen Blick, welche Optionen gesetzt sind und welche nicht. Wenn Sie mit der Funktion SetMenuStrip eine Menüleiste für Ihr Fenster einrichten, dann können Sie hier Ihre eigene Checkmark verwenden. Übergeben Sie der "NewWindow"-Datenstruktur für "CheckMark" den Wert Null, dann verwendet Intuition die voreingestellte Checkmark ("√").

Der Parameter "Type" wird vom System nur verwendet, wenn Sie Ihr Fenster auf einem Custom-Screen öffnen. Öffnen Sie es auf dem Workbench-Screen, wird dieser Wert ignoriert. Hat "Type" den Wert "CUSTOMSCREEN", so müssen Sie den Zeiger auf den zugehörigen Screen vor dem Öffnen des Fensters angeben. Nachfolgend ein Beispiel hierzu.

```
struct Screen  *MagicScreen;
struct Window *MagicWindow;

if(!(MagicScreen = OpenScreen(&MeineScreen))) exit(0);

MeinFenster.Screen = MagicScreen;   /* Zeiger auf Screen
                                       übergeben */

if(!(MagicWindow = OpenScreen(&MeinFenster))) exit(0);
```

Bestimmen der Position des Mauszeigers

Die aktuelle Position des Mauszeigers kann durch Auslesen der "MouseX" und "MouseY"-Variablen aus der "NewWindow"-Datenstruktur gefunden werden. Wenn z.B. "w" ein Zeiger auf ein Fenster ist, dann können Sie die Position mit folgendem Fragment bestimmen:

```
AktuelleXPosition = w->MouseX;
AktuelleYPosition = w->MouseY;
```

Diese Werte geben jedoch nicht jede einzelne Bewegung der Maus wieder. Wenn Sie die Werte durch einen Strukturverweis in einer "IntuiMessage"-Datenstruktur auslesen, bekommen Sie immer die tatsächliche Position.

Ist z.B. "im" ein Zeiger auf eine IntuiMessage, dann lesen Sie die Werte wie folgt aus:

```
AktuelleXPosition = im->MouseX;
AktuelleYPosition = im->MouseY;
```

In unserem Malprogramm wird jedesmal ein Punkt gesetzt, wenn die linke Maustaste gedrückt ist und eine Nachricht vom Timer ("INTUITICKS") an unser Programm geschickt wird.

Im Listing 5.5 finden Sie diese Routine. Anhand einer globalen Variablen ("selectbutton") wird entschieden, ob die Taste gedrückt (0) ist oder nicht (1). Die Variable "rp" ist ein Zeiger auf den von uns verwendeten RastPort.

```
/* ticks.c */

extern int selectbutton;   /* Linke Maustaste gedrückt = 0, sonst 1 */
int CurrentMouseX, CurrentMouseY;  /* Akt. Position des Mauszeigers */

extern struct RastPort   *rp;

IntuiTicks(im)
struct IntuiMessage *im;
{
  if(!selectbutton) return(1);  /* Linke Maustaste nicht gedrückt */

  CurrentMouseX = im->MouseX;
  CurrentMouseY = im->MouseY;

  WritePixel(rp,CurrentMouseX,CurrentMouseY);  /* Punkt setzen */
  return(1);
}
```

Listing 5.5: Routine zum Setzen eines Pixels bei gedrückter Maustaste

Auslesen des Status der linken und rechten Maustaste

Durch das Setzen des Flags "MOUSEBUTTONS" in der "NewWindow"-Datenstruktur erhalten Sie von Intuition jedesmal eine Nachricht über den IDCMP, wenn die linke Maustaste gedrückt wird. Die rechte Maustaste, die zur Auswahl von Menüs dient, wird normalerweise nur von Intuition verwaltet.

Sie haben jedoch die Möglichkeit, den Status dieser Taste (gedrückt / nicht gedrückt) zu erfahren, wenn Sie im "Flags"-Datenfeld der "NewWindow"-Datenstruktur "RMBTRAP" setzen. Auf diese Weise können Sie so beide Maustasten abfragen.

Im Listing 5.6 finden Sie die in unserem Malprogramm verwendete Routine zur Abfrage der beiden Maustasten.

```
/* mousebuttons.c */

int selectbutton;    /* Flag (TRUE / FALSE) für linke Maustaste */

MouseButtons(im)
struct IntuiMessage *im;
{
  if(im->Code == SELECTDOWN)       /* Linke Maustaste gedrückt */
     {
       selectbutton = TRUE;
       IntuiTicks(im);       /* Pixel setzen */
       return(1);
      }

  if(im->Code == SELECTUP)     /* Nicht gedrückt */
     {
       selectbutton = FALSE;
       return(1);
      }

  return(1);    /* Status der rechten Maustaste wird nicht abgefragt */
}
```

Listing 5.6: Routine zur Abfrage der linken Maustaste

Die Erstellung und Verwendung von Menüs

Damit wir in unserem Malprogramm ein Menü verwenden können, müssen wir uns zunächst einmal mit dem Aufbau von Menüs, den Menüpunkten und Untermenüs beschäftigen.

Menüs gehören immer zu einem Fenster; die Menüleiste hingegen wird beim Druck auf die rechte Maustaste immer über die Titelleiste des zugehörigen Screens geblendet. Jedes Menü besteht aus einer eigenen Datenstruktur, die es komplett und eindeutig definiert.

Sie können ein Menü durch die Angabe der horizontalen und vertikalen Position an eine beliebige Stelle der Menüleiste setzen. Jeder zu diesem Menü gehörende Menüpunkt wird relativ zu diesem Menü plaziert. Wenn Sie also das Menü neu positionieren wollen, brauchen Sie die Koordinaten der Menüpunkte und Untermenüs nicht neu zu berechnen.

Hat ein Menüpunkt ein Untermenü, dann werden dessen Koordinaten relativ zum zugehörigen Menüpunkt angegeben. Positionieren Sie also einen Menüpunkt neu, dann wird auch das Untermenü neu plaziert. Im Listing 5.7 finden Sie die Datenstrukturen des Menüs, das wir in unserem Malprogramm verwenden wollen. Die Menüleiste beinhaltet zwei Menüs. Durch die Anwahl des ersten kann der Benutzer eine Zeichenfarbe auswählen; mit dem zweiten kann Text in das Bild eingefügt werden. In diesem Fall wird ein Requester geöffnet, in dem der Text und die Startposition der Textausgabe eingegeben werden.

Die Beziehung zwischen Menüs und Menükomponenten

Damit Sie die Datenstrukturen aus Listing 5.7 besser verstehen, erläutern wir im folgenden die Beziehung zwischen Menüs und deren Menüpunkten.

- Die Einträge in der Menüleiste (d.h. die Namen der verfügbaren Menüs) bestehen aus einer verknüpften Liste von Zeichenketten, die vom Programmierer seinen Wünschen entsprechend positioniert werden. Beim Druck auf die rechte Maustaste wird diese Menüleiste über die Titelleiste des zugehörigen Screens geblendet; die Menünamen werden sichtbar und können angewählt werden.

- Die Liste der Menünamen wird durch eine weitere ergänzt, in der Zeiger auf die Menüpunkte enthalten sind. Jeder dieser Zeiger gehört zu dem Menüpunkt, der direkt unter dem Menünamen erscheint, wenn das Menü angewählt wird.

- Die einzelnen Menüpunkte sind ebenfalls in Form einer verknüpften Liste abgelegt. Hat ein Menüpunkt ein Untermenü, so existiert eine weitere Liste, die den Aufbau dieses Untermenüs definiert. Intuition erlaubt maximal ein Untermenü pro Menüpunkt.

Initialisierung von Menüs

Wenn Sie Menüs erstellen, dann sollten Sie darauf achten, daß sie sich in der Menüleiste nicht überdecken. Intuition vergleicht jedesmal, wenn Sie ein Menü auswählen, die Position des Mauszeigers mit der Breite der einzelnen Menüs, die Sie in der Struktur definiert haben.

Die Menüs werden durch die Parameter "LeftEdge", "TopEdge", "Width" und "Height" eindeutig definiert; auf diese Weise kann Intuition feststellen, ob sich der Mauszeiger über einem Menü befindet. Wenn sich jedoch zwei Menüs überdecken, dann wird von Intuition immer das Menü ausgewählt, das in Ihrer verknüpften Liste über dem anderen steht.

Ein Beispiel hierzu: Nehmen wir an, Sie erstellen zwei Menüs "Color" und "Text" und ordnen diese wie in Abb. 5.1 gezeigt an. Werden die Menüs auf folgende Art verknüpft:

```
menu[0].NextMenu = &menu[1];
menu[1].NextMenu = NULL;
```

dann können beide Menüs ausgewählt werden, wenn "Text" als "menu[0]" definiert wird. Wird "Text" hingegen als "menu[1]" deklariert, dann werden Sie es niemals auswählen können, da das erste Menü-"Color" jetzt "menu[0]" ist, also vor "Text" in der Liste steht, welches ja innerhalb von "Color" liegt, wie Abb. 5.1 zeigt. In unserem Beispiel verwenden wir daher die globale Variable "Leftside", die verhindert, daß sich zwei Menüs überdecken.

```
/* initmenu.c */

char menunames[2];

extern struct Menu   menu[2];
struct MenuItem textitem;
extern struct MenuItem coloritem[32]; /* Max. 5 Bitplanes (= 32 Farben) */
extern struct Image colorimage[32];

InitMenu()
{
  struct Menu *menuptr;

  int n, leftside = 2;

  menunames[0] = "Color";
  menunames[1] = "Text";

  menuptr = &menu[0];

  for(n=0;n<2;n++)
        {
          menuptr->LeftEdge = leftside;
          menuptr->TopEdge = 0;
          menuptr->Width = 9 * strlen(menunames[n]);
          leftside += (menuptr->Width+2);
          menuptr->Height = 10;
          menuptr->Flags = MENUENABLED;
          menuptr->MenuName = menunames[n];

          if(!n)
             {
               menuptr->FirstItem = &coloritem[0];
               menuptr->NextMenu = &menu[1];
             }

           else
                  {
                    menuptr->FirstItem = &textitem;
                    menuptr->NextMenu = NULL;
                  }

          menuptr->JazzX = 0;
          menuptr->JazzY = 0;
          menuptr->BeatX = 0;
          menuptr->BeatY = 0;
          menuptr++;
        }
```

Listing 5.7: Fragment zur Initialisierung von Menü-Datenstrukturen (Teil 1)

```
  InitTextItem();
  InitColorItems(DEPTH);
}

  /* Jedes Menü muß mindestens einen Menüpunkt haben. Nachfolgend die
     Initialisierung des Menüpunktes für das "Text"-Menü. */

  struct IntuiText textitemtext =
  { 1,0,JAM2,0,0,NULL,"Text einfügen",NULL };

  /* DetailPen, BlockPen, Zeichenmodus, Position (X,Y) relativ zur
     linken oberen Ecke des zugehörigen Menüs, Zeichensatz
     (NULL = Standardzeichensatz), Menüname, Zeiger auf die nächste
     IntuiText-Struktur (hier NULL, da wir ja nur ein Menüpunkt haben) */

InitTextItem()
{
  textitem.NextItem = NULL;   /* Keine weiteren Menüpunkte */
  textitem.ItemFill = &textitemtext;  /* Zeiger auf Menünamen */
  textitem.LeftEdge = 0;    /* Menüpunkt erscheint ganz links, */
  textitem.TopEdge = 8;    /* unterhalb des Menünamens */
  textitem.Width = 9 *  16;  /* "Text einfügen" plus Rand */
  textitem.Height = 10;
  textitem.Flags = HIGHCOMP|ITEMTEXT|COMMSEQ|ITEMENABLED;

  textitem.MutualExclude = 0;  /* Durch die Selektion dieses Menüpunktes
                                  werden keine anderen, zuvor
                                  selektierten Punkte zurückgesetzt */

  textitem.SelectFill = NULL  /* Keine Umrandung bei Selektion */
  textitem.Command = 't';
  textitem.SubItem = NULL;
  textitem.NextSelect = 0; /* Keine weiteren Menüpunkte oder Untermenüs */

  /* Durch Setzen von COMMSEQ im Flags-Datenfeld und die Angabe von "t"
     bei "textitem.Command" kann der Menüpunkt auch über die Tastatur
     selektiert werden. Hierzu drückt man die rechte Amiga-Taste (neben
     der <SPACE>-Taste) und die Taste "t" gleichzeitig. Dies hat
     denselben Effekt, als wenn man den Menüpunkt mit der Maus auswählen
     würde. */

InitColorItems(depth)
SHORT depth;     /* Maximale Anzahl Farben, die Menü gewählt werden
                    können, hängt von der "Tiefe" (Anzahl Bitplanes) des
                    Screen ab. */
```

Listing 5.7: Fragment zur Initialisierung von Menü-Datenstrukturen (Teil 2)

```
{
  SHORT n, colors;
  struct Image *colorimageptr;
  struct MenuItem *coloritemptr,*nextcoloritemptr;

  colors = palette[depth-1];

  colorimageptr = &colorimage[0];
  coloritemptr = &coloritem[0];
  nextcoloritemptr = coloritemptr;

  for(n=0;n<colors;n++)
        {
          nextcoloritemptr++;

          coloritemptr->NextItem = nextcoloritemptr;
          coloritemptr->ItemFill = colorimageptr;
          coloritemptr->LeftEdge = 2+CW * (n%4);
          coloritemptr->TopEdge = CH * (n/4);
          coloritemptr->Width = CW;
          coloritemptr->Height = CH;
          coloritemptr->Flags = ITEMSTUFF;
          coloritemptr->MutualExclude = 0;
          coloritemptr->SelectFill = NULL;
          coloritemptr->Command = 0;
          coloritemptr->SubItem = NULL;
          coloritemptr->NextSelect = 0;

          colorimageptr->LeftEdge = 1;
          colorimageptr->TopEdge = 1;    /* Farbfläche zentrieren */

          colorimageptr->Width = CW-2;  /* Und ein Stück verkleinern */
          colorimageptr->Height = CH-2;
          colorimageptr->Depth = depth;

          colorimageptr->ImageData = NULL;  /* Kein eigenes "Image" */
          colorimageptr->PlanePick = 0;
          colorimageptr->PlaneOnOff = n;

          colorimageptr++;
          coloritemptr++;
        }
  coloritemptr--;   /* Zeiger auf letzten Eintrag setzen */
  coloritemptr->NextItem = NULL;  /* Keine weiteren Menüpunkte */
}
```

Listing 5.7: Fragment zur Initialisierung von Menü-Datenstrukturen (Schluß)

Im "Flag"-Datenfeld verwenden wir "MENUENABLED", damit die Menüs ausgewählt werden können. Hat dieses Feld den Wert Null, dann erscheint das betreffende Menü in "Geisterschrift", kann also nicht selektiert werden. Der Name des Menüs, der in der Menüleiste erscheint, muß aus Text bestehen; Menüpunkte und Untermenüs können hingegen kleine Grafiken (Images) enthalten. Sie können jedoch der Menüleiste ein eigenes Aussehen verleihen, wenn Sie einen anderen als den voreingestellten Zeichensatz verwenden. In diesem Fall müssen Sie das "Font"-Datenfeld in der "NewScreen"-Datenstruktur entsprechend intitalisieren.

Initialisierung von Menükomponenten

Wie bei den Menüs, so muß man auch bei den einzelnen Menüpunkten darauf achten, ob sie sich überschneiden; denn hier kommt es bisweilen darauf an, daß sie sich überschneiden. Das mag sich widersprüchlich anhören, aber Intuition erzeugt selbsttätig den rechteckigen Bereich, in dem die Menüpunkte angeordnet werden und der bei der Auswahl eines Menüs "nach unten klappt", also die Menüpunkte sichtbar werden läßt.

Die Größe der "Menübox" wird von Intuition berechnet, damit auch der größte Menüpunkt des Textes dargestellt werden kann. Diese Berechnung bezieht sich auf die angegebenen Werte für jeden einzelnen Menüpunkt ("LeftEdge", "TopEdge", "Width" und "Height"). Sollte bei der Auswahl eines Menüpunktes dieser durch das Zeichnen eines farbigen Rechtecks (o.ä.) hervorgehoben werden, dann spart Intuition hier genug Platz zwischen zwei Menüpunkten aus.

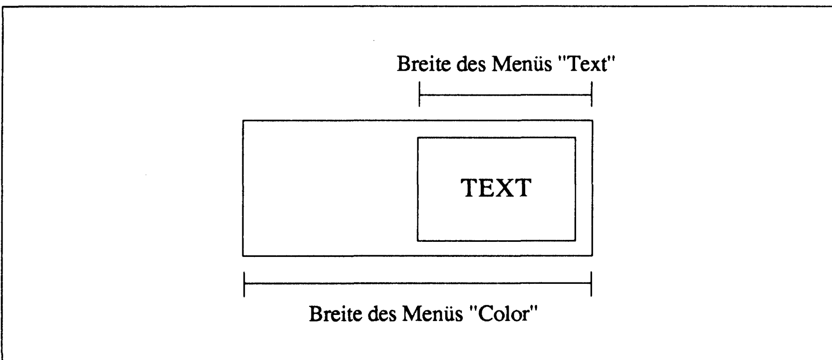


Abb 5.1: Beispiel für sich überdeckende Menüs

Wenn Sie z.B. einen Menüpunkt erstellen, der 12 Pixel breit ist, aber den Text im Standard-Zeichensatz (topaz.font, 80 Zeichen) ausgeben lassen (36 Pixel), dann erstellt Intuition eine Menübox, die groß genug ist, diesen Menüpunkt darstellen zu können, da ja andernfalls über die Hälfte des Textes nicht mehr sichtbar wäre.

Weiterhin kennt Intuition eigene Regeln, um einen Menüpunkt im zugehörigen Menü zu positionieren. Die Menübox, die Intuition zur Aufnahme der Menüpunkte erstellt, beginnt immer unterhalb der Menüleiste in Höhe der "LeftEdge" des Menünamens und reicht mindestens bis zur rechten unteren Ecke des Menünamens. Sollten Menüpunkte über diese Grenze hinausragen, dann vergrößert Intuition die Menübox entsprechend. Auf diese Art und Weise können die Menüpunkte weiter rechts als normal üblich positioniert werden. Wie Sie aber in der Abb. 5.2 sehen können, füllt Intuition einen leeren Bereich um den linken Menüpunkt selbsttätig aus.

Wenn Sie also einen Menüpunkt wie in Abb. 5.2 anordnen, dann erzeugt Intuition einen leeren Bereich an der linken Seite, damit der Menüpunkt auch weiterhin ausgewählt werden kann. Dies wäre nicht möglich, wenn der Platz leer bleiben würde, da die Grenze des Menünamens ja verlassen werden müßte, um den Menüpunkt selektieren zu können. Zur Auswahl reicht es allerdings nicht, den Mauszeiger über den leeren Bereich zu führen; er muß sich mindestens über der linken oberen Ecke des Menüpunktes befinden.

Wenn Sie ein Menü oder ein Untermenü erstellen, dann sollten Sie darauf achten, daß sich zusammengehörige Menüpunkte immer ein wenig überschneiden. So stellen Sie sicher, daß bei der Anwahl eines Menüs immer mindestens ein Menüpunkt angewählt ist.

Text oder Grafik

Wie bereits erwähnt, können Menüpunkte entweder durch Text oder Grafik dargestellt werden. Verwenden Sie Text, dann übergeben Sie Intuition jeden Menüpunkt in Form einer "IntuiText"-Datenstruktur. In diesem Fall muß das "ITEMTEXT"-Flag in der "MenuItem"-Datenstruktur gesetzt sein. Wenn dieses Flag gesetzt ist, dann zeigen sowohl "ItemFill" als auch "SelectFill" auf einen Menüpunkt, der durch Text dargestellt wird.

Lassen Sie dieses Flag hingegen weg, dann können Sie anstatt Text auch Grafik für die Menüpunkte verwenden; dies gilt sowohl für "ItemFill" als auch für

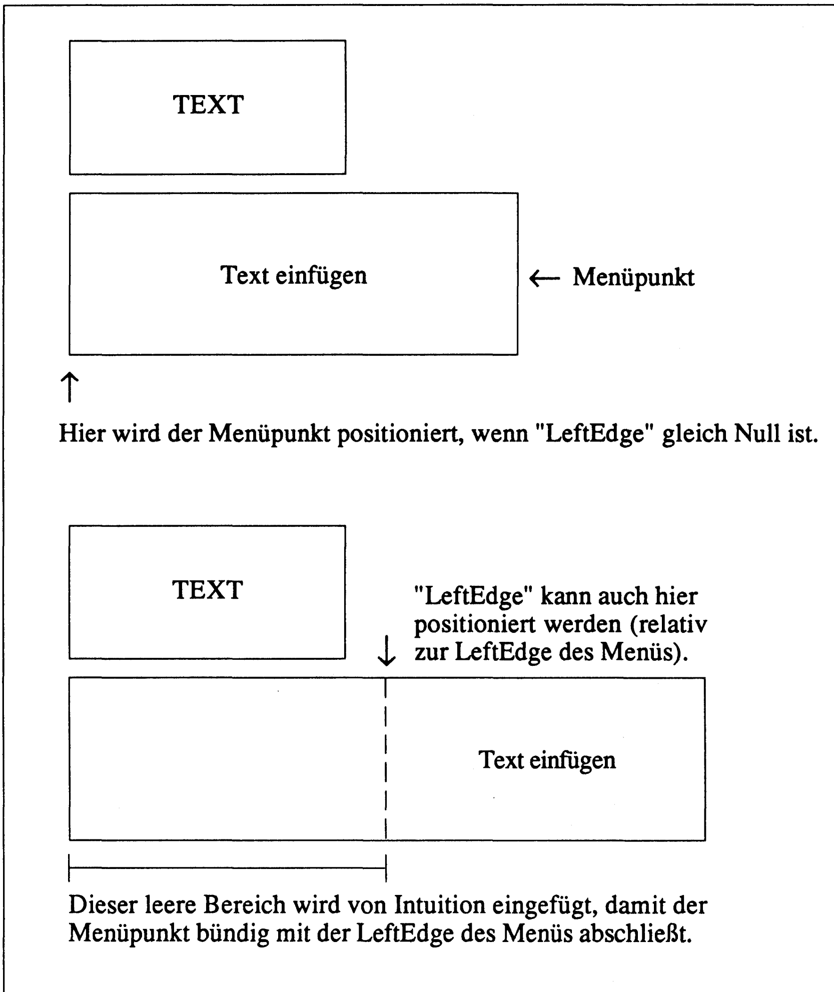


Abb. 5.2: Positionierung von Menüpunkten

"SelectFill". Auf diese Art und Weise haben Sie eine große Palette von Möglichkeiten zur Verfügung, um Menüpunkte zu erstellen. "ItemFill" ist der Menüpunkt im "Normalzustand", d.h. wenn er nicht angewählt ist; "SelectFill" repräsentiert den Menüpunkt im angewählten Zustand.

Im Programmfragment Listing 5.7 werden die "ColorItem"- und "Color-Image"-Datenstrukturen zur gleichen Zeit initialisiert, da Grafiken ("Images") nur an dieser Stelle des Malprogramms verwendet werden. Sie können anhand der "Image"-Datenstruktur festlegen, an welcher Stelle die Grafik in der Menübox positioniert werden und wie breit und hoch sie sein soll. Weiterhin kann durch sie Speicherplatz gespart werden, da durch die Parameter "PlanePick" und "PlaneOnOff" direkt angegeben werden kann, wie die Grafik dargestellt werden soll.

Sie können z.B. eine vierfarbige Grafik als Menüpunkt definieren, wenn Sie einen Screen mit 16 Farben verwenden. Wollen Sie diese Grafik mehr als einmal verwenden, dann können Sie für die nächste Grafik vier andere Farben zur Darstellung benutzen. Der folgende Abschnitt zeigt Ihnen, wie man diese Möglichkeit von Intuition nutzt.

Farbige Grafiken

Farben werden auf dem Amiga durch Binärkombinationen von Bits erzeugt, die in den verschiedenen Bitplanes eines Screens verwaltet werden. Um einen Bildpunkt in der Farbe des Registers 1 zu setzen (wobei es jetzt egal ist, welche Farbe darin enthalten ist), verwendet man normalerweise die Funktionen SetAPen(rp,1) zur Auswahl der Farbe und WritePixel(rp,x,y), um den Pixel zu setzen.

Bei diesem Vorgang werden jedesmal die verschiedenen Bitplanes beeinflußt, je nachdem, wie viele Bitplanes zur Darstellung der Grafik erforderlich sind. Werden 4 Bitplanes (16 Farben) benutzt, dann sieht die Bitkombination der einzelnen Bitplanes für einen Bildpunkt der Farbe 1 so aus:

Bitplane	3 2 1 0
Gesetzte Bits	0 0 0 1

Für ein Pixel, das die Farbe des Registers 10 hat, lautet die Kombination:

Bitplane	3 2 1 0
Gesetzte Bits	1 0 1 0

Angenommen, Sie erstellen eine vierfarbige Grafik folgender Form:

1100330011
1100330011
0110330110
0011331100
0022332200
0220330220
2200330022
2200330022

Jede Zahl repräsentiert ein Pixel der Grafik in einer der vier möglichen Farben. Da nur vier Farben verwendet werden, reichen zwei Bitplanes zur Darstellung der Grafik aus. Die Bitkombinationen der Bitplanes finden Sie in der Abb. 5.3. Wenn Sie diese Bits in die Bitplanes 0 und 1 schreiben, dann werden zur Darstellung immer die Farben der Register 0, 1, 2 und 3 verwendet.

Intuition bietet zur Grafikdarstellung noch eine weitere Möglichkeit: Sie können bestimmen, welche Bitplanes zur Darstellung benutzt werden sollen. Wählen Sie die Bitplanes 3 und 1 aus, dann werden die Bits der Lage 0 aus Ihrer Grafik der Bitplane 1 zugeordnet; die der Lage 1 der Bitplane 3. Weiterhin können Sie Intuition durch das "PlaneOnOff"-Datenfeld sagen, was mit den nicht selektierten Bitplanes geschehen soll.

Ist ein Bit einer solchen Bitplane gleich null, dann wird das zugehörige Rechteck, in dem der Menüpunkt (die Grafik) dargestellt wird, ebenfalls mit Nullen gefüllt. Hat ein Bit den Wert Eins, dann wird das Rechteck mit Einsen gefüllt. In unserem Beispiel werden die Bitlpanes 1 und 3 verwendet, d.h. die Bitplanes 0 und 2 sind unbenutzt. Die Bits im "PlaneOnOff"-Datenfeld bestimmen nun, wie diese Bitplanes verwendet werden. Sind beide gesetzt, dann sieht die Initialisierung so aus:

```
meineGrafik.PlanePick = 0x0a; /* Bitplanes 1 und 3 werden benutzt */
meineGrafik.PlaneOnOff = 0x05;  /* Bitplanes 0 und 2 "abschalten" */
```

Die Daten der Grafik werden also in den Bitplanes 1 und 3 abgelegt, Binärkombinationen mit dem Wert Eins werden – innerhalb des Rechtecks des Menüpunktes – in den Bitplanes 0 und 2 verwendet.

Plane 0 der Grafik	1100110011
	1100110011
	0110110110
	0011111100
	0000110000
	0000110000
	0000110000
Plane 1 der Grafik	0000110000
	0000110000
	0000110000
	0000110000
	0011111100
	0110110110
	1100110011
	1100110011

Abb. 5.3: Eine Bitplane-Grafik

Die verfügbaren Farben kommen in einem solchen Fall nicht mehr aus den Registern 0 bis 3, sondern man hat nun Zugriff auf alle verfügbaren Farbregister. "PlaneOnOff" setzt alle Bits der Bitplanes 0 und 2:

Bitplane	3 2 1 0
Gesetzte Bits	1 x 1 x

Die Farbregister, die zur Darstellung verwendet werden, sind:

1 0 1 0 = Farbregister 10
1 0 1 1 = Farbregister 11
1 1 1 0 = Farbregister 14
1 1 1 1 = Farbregister 15

Im Farbmenü, das wir in unserem Malprogramm verwenden, werden als Menüpunkte farbige Rechtecke verwendet, d.h. die eigentlichen Daten, die die Grafik ausmachen, sind nicht vorhanden. Die Variable "PlanePick" hat den Wert Null, die "Grafik" wird also in keine der Bitplanes geschrieben – die verwendete Farbe wird ausschließlich durch den Wert von "PlaneOnOff" bestimmt. Für jedes Rechteck, das als Menüpunkt angeboten wird, werden also verschiedene Bitkombinationen der Bitplanes verwendet, so daß Grafikdaten zur Darstellung nicht erforderlich sind.

Wollen Sie zur Darstellung von Menüpunkten Grafik verwenden, dann werden die Daten in einem Feld aus vorzeichenlosen "Worten" (UWORD, unsigned int) abgelegt. Die einzelnen Bits der Grafik werden linksbündig in einer Matrix abgelegt, wobei die Matrix groß genug sein muß, um das Objekt in der Breite und Höhe fassen zu können. Da die Bits linksbündig vom System verwendet werden, bleiben alle Bits, die außerhalb des Grafikbereichs liegen, ungenutzt. Wenn Sie z.B. eine Grafik erstellt haben, die 29 Pixel breit ist, dann benötigen Sie zur Ablage zwei Datenworte (32 Bits), um die Breite fassen zu können.

Wenn das System die Grafik dann zeichnet, werden nur die untersten 29 Bits verwendet; die restlichen drei Bits werden ignoriert. In der Abb. 5.8 finden Sie eine solche Matrix, die die bereits besprochene Grafik enthält. Die zugehörige "Image"-Datenstruktur wird dann wie folgt initialisiert:

```
struct Image meineGrafik =
{
  1,1,10,8,2,&meinegrafikmatrix[0],0x05,0x0a
/* LeftEdge, TopEdge, Breite, Höhe, Anzahl Bitplanes,
   Grafikdaten, PlanePick, PlaneOnOff */
};
```

"Checkmarks"

Sie können Menüpunkte, die ausgewählt wurden, mit einer "Checkmark" (√) versehen lassen, um so anzuzeigen, daß ein Menüpunkt bereits aktiv ist. Damit Intuition die Checkmark darstellt, muß das Flag "CHECKIT" in der MenuItem-Datenstruktur des betreffenden Menüpunktes gesetzt werden. Beachten Sie bitte, daß Sie am linken Rand des Menüpunktes genügend Platz für die Checkmark lassen, da Intuition sie ansonsten über den Text oder die Grafik legt, wenn der Menüpunkt angewählt wird.

```
/* myimageshape.h */

UWORD meinegrafikmatrix[] =
{
  0xccc0,   /* Lage 1 der Grafik */
  0xccc0,
  0x5d80,
  0x3f00,
  0x0c00,
  0x0c00,
  0x0c00,
  0x0c00,   /* Lage 2 der Grafik */
  0x0c00,
  0x0c00,
  0x0c00,
  0x3f00,
  0x5d80,
  0xccc0,
  0xccc0
};

struct Image meineGrafik =
{
  1,1,10,8,2,&meinegrafikmatrix[0],0x05,0x0a
};
```

Listing 5.8: Initialisierung einer Grafikmatrix

Intuition stellt jedoch nicht nur die Checkmark dar, sondern mit dem Setzen des "CHECKIT"-Flags sind noch andere Dinge verbunden. So können Sie z.B. den Status der "CHECKED"-Flags von verschiedenen anderen Menüpunkten abfragen, bevor Sie nach der Anwahl eines Menüpunktes mit der Programmausführung fortfahren. Sie können weiterhin das Flag "MENUTOGGLE" setzen. So wird bei der Anwahl eines Menüpunktes die Checkmark gesetzt ("CHECKED") und beim nächsten Mal wieder gelöscht.

Intuition stellt Ihnen zwar eine voreingestellte Checkmark zur Verfügung (√), Sie können jedoch eine eigene entwerfen und diese dann verwenden. In diesem Fall übergeben Sie in der "NewWindow"-Datenstruktur einen Zeiger auf die Grafikdaten dieser Checkmark (jedes Fenster kann eine eigene Menüleiste und eine eigene Checkmark haben). Geben Sie als Zeiger den Wert Null an, dann verwendet Intuition die Standard-Checkmark.

Warum aber sollte man sich eine eigene Checkmark entwerfen? Ein Grund ist vielleicht, daß die Standard-Checkmark nur 8 mal 8 Pixel groß ist und vielleicht etwas "klein" aussieht, wenn Sie einen anderen als den Standard-Zeichensatz verwenden. Andererseits kann man Menüs ein eigenes Aussehen geben, wenn man eine neue Checkmark verwendet, z.B. einen Punkt oder einen Pfeil.

Interessant ist es, eine Checkmark zu entwerfen, die nur aus einem Pixel besteht, der zudem die gleiche Farbe wie das Menü hat, also unsichtbar ist, egal, ob der Menüpunkt angewählt ist oder nicht. In diesem Fall sieht der Benutzer zwar nicht direkt, ob ein Menüpunkt bereits angewählt wurde oder nicht, Sie können dies jedoch anhand des "CHECKED"-Flags vom Programm aus feststellen. Damit können Sie – je nachdem, welchen Status das "CHECKED"-Flag hat – dem Benutzer eine Grafik im Menü präsentieren, die für beide Zustände (angewählt oder nicht) jeweils verschieden aussieht. Sie können sogar die Zeiger für die "ItemFill" und "SelectFill"-Datenstrukturen ändern und so mehr als zwei Grafiken darstellen lassen.

"Mutual Exclusion" – Automatisches Freigeben von Menüpunkten

Das "MutualExclude"-Datenfeld in der "MenuItem"-Datenstruktur besteht aus einer Variabeln des Typs "LONG" (32 Bit). Jedes Bit in dieser Variablen repräsentiert einen Menüpunkt des aktuell angewählten Menüs. Die Menüpunkte werden vom System durchnumeriert, angefangen bei der Ziffer Null. Ist in der "MutualExclude"-Variablen z.B. das Bit 4 gesetzt, wird bei der Anwahl des zugehörigen Menüpunktes der Menüpunkt mit der Nummer 4 automatisch freigegeben – natürlich nur dann, wenn er zuvor selektiert ("CHECKED" = "TRUE") wurde.

Mutual Exclusion wird immer dann verwendet, wenn durch die Anwahl eines Menüpunktes andere nicht mehr selektiert werden dürfen, da sonst vielleicht unerwünschte Effekte auftreten. Haben Sie z.B. ein Menü, in dem Sie die Laufwerke DF0: bis DF3: zur Auswahl anbieten, um eine Datei zu lesen, dann schließt die Auswahl von einem Menüpunkt alle anderen aus, da nicht von zwei Laufwerken gleichzeitig gelesen werden kann.

Durch die Verwendung einer 32-Bit-Variablen können Sie so eine Liste aller Menüpunkte definieren, die bei der Anwahl des zugehörigen Menüpunktes freigegeben werden sollen.

Hier noch ein weiteres Beispiel, um die Anwendung der Mutual Exclusion zu veranschaulichen. Nehmen wir an, Sie haben ein Menü mit folgendem Aufbau:

```
Füllmuster:
   Transparent
   Einfarbig
   Schraffiert
   Gestreift
```

Gäbe es die Möglichkeit der Mutual Exclusion nicht, dann müßten Sie jedesmal, wenn ein Menüpunkt ausgewählt wird, die "CHECKED"-Flags der restlichen drei Menüpunkte überprüfen und sie auf den Wert Null setzen, da ja auch hier nur ein Menüpunkt zur Zeit gesetzt werden kann.

Intuition nimmt Ihnen diese Arbeit mit Hilfe der "MutualExclude"-Variablen ab. Ihr Programm braucht sich also nicht darum zu kümmern, welche Menüpunkte freigegeben werden müssen und welche nicht. Jedes gesetzte Bit in der Variablen gibt bei der Anwahl des zugehörigen Menüpunktes den der Bitposition entsprechenden Menüpunkt automatisch frei. Im Listing 5.9 finden Sie ein Programmfragment, das die Mutual Exclusion für das oben gezeigte Beispielmenü übernimmt. Zur besseren Übersicht ist im Listing 5.9 allerdings nur die Mutual Exclusion gezeigt.

Das System verwendet nur die Bits, die im Bereich der Anzahl verfügbarer Menüpunkte liegen. In unserem Beispiel können wir also alle Bits der Variablen auf den Wert Eins setzen, mit Ausnahme der unteren 4 Bits. Diese 4 Bits sind die einzigen, die vom System verwendet werden, da ja nur 4 Menüpunkte vorhanden sind; alle anderen Bits werden ignoriert. Da alle Bits der Variablen gesetzt sind (d.h. alle Menüpunkte werden freigegeben), müssen wir eine Exklusiv-Oder-Verknüpfung vornehmen, die das Bit des angewählten Menüpunkts auf Null setzt, um Intuition so mitzuteilen, daß dieser Menüpunkt selektiert ist. Auf diese Weise werden bei der Anwahl eines Menüpunktes alle anderen automatisch freigegeben.

Hervorheben von Menüpunkten bei ihrer Anwahl

Wenn der Anwender ein Menü anwählt und den Mauszeiger über die Menüpunkte bewegt, kann man den Menüpunkt, über dem der Zeiger sich gerade befindet, hervorheben, damit der Anwender weiß, welcher Menüpunkt nach dem Loslassen der rechten Maustaste angewählt wird. Befindet sich der Maus-

```
/* setexclude.c */

#define EXCLUDE_ALL 0xffffffff

SetExclude(mi, anzahl)
struct MenuItem *mi;
int anzahl;
{
  int i;

  for(i=0;i<anzahl_menuepunkte;i++)
        {
          mi.MutualExclude = EXCLUDE_ALL ^ (1 << i);
          /* Nur ein Bit wird mit obiger Zeile auf den
             Wert 0 gesetzt, d.h. der der Bitposition
             entsprechende Menüpunkt wird selektiert */

           mi++;    /* MutualExclude für nächsten
                       Menüpunkt setzen */
        }
}
```

Listing 5.9: Beispiel zur Definition der MutualExclude-Variablen

zeiger über keinem der verfügbaren Menüpunkte, sondern z.B. über dem Menünamen, dann sendet Intuition nach dem Loslassen der rechten Maustaste keine Nachricht an Ihr Programm. Intuition stellt Ihnen mehrere Möglichkeiten zur Verfügung, um einen Menüpunkt bei seiner Anwahl hervorheben zu lassen.

Der Parameter "HIGHBOX", den wir auch im Malprogramm ("ITEMSTUFF" = "MENUENABLED"+"HIGHBOX") verwenden, sagt Intuition, daß bei der Anwahl eines Menüpunktes ein farbiger Kasten um den Text des Menüpunktes gezeichnet werden soll.

Wird ein Menüpunkt angewählt, dann können Sie den zugehörigen Text auch invers darstellen lassen. Dies geschieht durch die Angabe des Flags "HIGHCOMP". Die Farben, in der der Menüpunkt dargestellt ist, werden durch die entsprechenden Komplementärfarben ersetzt. Im Malprogramm verwenden wir hingegen "HIGHBOX", da ja bereits farbige Rechtecke als Menüpunkte angeboten werden. Würden diese bei Ihrer Anwahl invertiert, dann entspräche die dargestellte Farbe ja nicht mehr der, die tatsächlich verwendet wird.

"HIGHIMAGE" können Sie dann verwenden, wenn Sie das "SelectFill"-Datenfeld in der "MenuItem"-Datenstruktur initialisiert haben. Sie legen hier einen Zeiger auf eine Grafik (Image) oder einen Text ab, so daß bei der Anwahl des Menüpunktes dann eine neue Grafik oder ein anderer Text erscheint.

Das letzte verfügbare Flag ist "HIGHNONE". Bei der Anwahl eines Menüpunktes ändert sich an seiner Darstellung nichts; der Anwender weiß also nie genau, ob ein Menüpunkt tatsächlich angewählt wurde oder nicht. So ist es sehr wahrscheinlich, daß der Anwender einen Menüpunkt anwählt, obwohl er dies gar nicht beabsichtigt hat; das "HIGHNONE"-Flag sollte man also nach Möglichkeit nie verwenden.

Requester

Ein Requester gehört immer zu einem Fenster. Seine Position innerhalb des Fensters wird relativ zur linken oberen Ecke des Fensters angegeben. Ist ein Fenster kleiner als der Requester, der geöffnet werden soll, dann wird dieser trotzdem komplett sichtbar ausgegeben, auch wenn er über die Umrandung des Fenster hinausragen sollte. Der Grund hierfür ist, daß ein Requester eine besondere Art eines "Layers" ist (Grafisches Objekt, das durch die "layers.library" verwaltet wird).

Die Datenstruktur eines Requesters wird durch einen Aufruf der Funktion InitRequester initialisiert, der als Parameter ein Zeiger auf eine "leere" Requester-Datenstruktur übergeben wird. Weiterhin geben Sie an, wo die linke obere Ecke ("LeftEdge", "TopEdge") des Requesters – relativ zur linken oberen Ecke des zugehörigen Fensters – liegen und wie breit und hoch er sein soll.

Beim Öffnen eines Requesters wird von Intuition ein rechteckiger Bereich mit einer von Ihnen definierten Farbe gefüllt ("BackFill") sowie eine Anzahl von Linien gezeichnet ("ReqBorder"), deren Koordinaten ebenfalls von Ihnen angegeben werden können. Die Linien, die Sie mit der "ReqBorder"-Struktur übergeben, können auch außerhalb des Requesters liegen. Sie können auf diese Weise einen "Schatten" des Requesters zeichnen lassen, so daß der Eindruck entsteht, der Requester würde über dem Hintergrund "schweben". Diese Art der Linienanordnung nennt man "Drop-Shadow". Nachdem Intuition das Rechteck und die Linien gezeichnet hat, wird der von Ihnen gewünschte Text (IntuiText) im Requester ausgegeben. Verwenden Sie Gadgets in diesem Re-

quester (es muß mindestens einer vorhanden sein), dann übergeben Sie Intuition anhand der Requester-Datenstruktur einen Zeiger auf das erste Gadget aus der verknüpften Liste aller in diesem Requester verwendeten Gadgets. Im Listing 5.10 finden Sie ein Programmfragment zur Initialisierung eines Requesters.

```
/* inittr.c */

struct Requester textrequest;
struct TextAttr modfontattr;

InitTextRequest()
{
  BYTE *s, *t;

  InitRequester(&textrequest);

  textrequest.LeftEdge = 20;
  textrequest.TopEdge = 20;
  textrequest.Width = 280;
  textrequest.Height = 130;
  textrequest.ReqGadget = &trg[0];
  textrequest.ReqText = &textreqtext[0];
  textrequest.BackFill = 1;
  textrequest.ReqBorder = NULL;

  s = &textstring[0];
  t = defaulttext;

  while((*s++ = *t++) != '\0');     /* String kopieren */

  txfont = 80;     /* 80 Zeichen pro Zeile */
  txmode = 1;     /* JAM1 Modus */

  modfontattr.ta_Name = "topaz.font"
  modfontattr.ta_YSize = 8;
  modfontattr.ta_Style = 0;
  modfontattr.ta_Flags = 0;
}
```

Listing 5.10: Routine zur Initialisierung eines Requesters

Gadgets

Der Requester, den wir in unserem Malprogramm verwenden, enthält 3 Gadgets: zur Eingabe des Textes, zum Abbruch der Operation und zur Positionierung des Textes. Jedes der drei Gadgets ist dabei von einer anderen Art: "Boolean", "String" und "Proportional". Nähere Informationen hierzu sind in den folgenden Abschnitten enthalten. Zunächst jedoch wollen wir auf die Dinge eingehen, die alle Gadgetarten gemeinsam haben.

Die Position eines Gadgets

Gehört ein Gadget zu einem Fenster, dann wird seine Position relativ zur linken oberen Ecke des Fensters angegeben. Wird es in einem Requester verwendet, dann bezieht sich seine Position relativ zur linken oberen Ecke des Requesters. In beiden Fällen jedoch erfolgt die Positionsangabe durch die Initialisierung der Variablen "LeftEdge" und "TopEdge" in der "Gadget"-Datenstruktur.

Die Größe eines Gadgets

Durch die Übergabe von Werten an die Variablen "Width" und "Height" in der "Gadget"-Datenstruktur bestimmen Sie die Größe des Gadgets. Diese Parameter legen – genau wie bei Menüpunkten – fest, wie groß das Rechteck ist, innerhalb dessen das Gadget durch einen Druck auf die linke Maustaste selektiert werden kann.

Wie bei den Menüpunkten, so richtet sich die Priorität eines Gadgets nach seiner Position innhalb der Gadgetliste; je näher es am Anfang der Liste steht, desto höher ist sie. Überschneiden sich zwei Gadgets, dann wird vom System immer das mit der höheren Priorität selektiert.

Das Aussehen eines Gadgets

Gadgets sehen – wie auch Menüpunkte – bei ihrer Anwahl anders aus als im "Ruhezustand". Die Datenfelder "GadgetRender" und "SelectRender" der "Gadget"-Datenstruktur entsprechen den Feldern "ItemFill" und "SelectFill"

der "MenuItem"-Datenstruktur. Sie können bei den Gadgets ebenfalls durch das Setzen eines Flags angeben, auf welche Art es bei der Anwahl dargestellt werden soll.

Wie bei den Menüpunkten, so können auch Gadgets bei der Anwahl durch die Komplementierung der Farben hervorgehoben werden ("GADGHCOMP"). Das Flag "GADGHBOX" zeichnet einen farbigen Kasten um das Gadget, das gerade angewählt ist. Soll bei der Anwahl eine Grafik dargestellt werden, dann nimmt man hierzu das Flag "GADGHIMAGE". Und wenn bei der Anwahl nichts geschehen soll, dann verwendet man "GADGHNONE".

Der Zeiger von SelectRender kann – im Gegensatz zu Menüpunkten – entweder auf eine "Image"- oder eine "Border"-Datenstruktur zeigen. Damit das System weiß, daß eine Grafik verwendet werden soll, setzen Sie das Flag "GADGIMAGE". Grafiken belegen im allgemeinen mehr Speicherplatz als "Border"-Strukturen.

Die Identifikation von Gadgets

Wenn Sie Menüs einsetzen, dann wird die Nachricht, die Sie von Intuition bei der Anwahl eines Menüpunktes erhalten, durch die Datenstruktur des zugehörigen Menüs erzeugt. Die Nummer, die sie im Feld "Code" erhalten und die besagt, welcher Menüpunkt angewählt wurde, richtet sich nach der Position des Menüpunktes innerhalb der Datenstruktur; einen anderen Einfluß können Sie nicht nehmen – der Wert von "Code" hängt immer von der Position in der Struktur ab.

Wird ein Gadget angewählt, dann erhalten Sie von Intuition nicht die Positionsnummer des Gadgets, sondern die Anfangsadresse der Datenstruktur, die zum angewählten Gadget gehört. Diese Struktur beinhaltet eine Komponente namens "GadgetID", der sie einen beliebigen Wert zuordnen können (der auch bei Mutual Exclusion verwendet wird).

Auf diese Weise können Sie eine beliebige Anzahl von Gadgets freigeben lassen, wenn ein bestimmtes Gadgets angewählt wird, da Sie ja ein frei definierbares Zahlenschema zur Mutual Exclusion angeben können. Hier liegt der Unterschied zu den Menüpunkten, von denen Sie ja maximal 32 gleichzeitig freigeben können. Die Reihenfolge der Gadgets ist bei einer Anwahl nicht relevant; durch einen Strukturverweis auf die Variable "GadgetID" können Sie feststellen, welches Gadget vom Anwender angewählt wurde.

"Mutual Exclusion" – Automatisches Freigeben von Gadgets

Wie für Menüpunkte, so können Sie auch für Gadgets ein "MutualExclude"-Datenfeld angeben. Der Aufbau dieser Variablen entspricht dem, der auch bei Menüpunkten verwendet wird. Allerdings wird bei Gadgets zur Formulierung des Wertes die "GadgetID"-Variable verwendet, da Gadgets ja nicht – wie es Menüpunkte sind – positionsgebunden sind.

Gadgettypen

Ein Gadget gehört entweder zu einem Fenster oder zu einem Requester. In der Datenstruktur eines Gadgets muß daher angegeben werden, wozu das entsprechende Gadget gehört. Setzen Sie das Flag "REQGADGET" im Feld "GadgetType", dann gehört es zu einem Requester. Das Feld enthält noch andere Werte, die aber nur von Intuition verwendet werden.

Verfügbare Flags für Gadgets

Neben den Flags, die das Aussehen eines Gadgets definieren, können zusätzlich die folgenden angegeben werden:

GADGDISABLED — Wird dieses Flag gesetzt, dann kann das entsprechende Gadget nicht angewählt werden. Das System stellt solche Gadgets "verschwommen" dar, damit sie direkt als nicht selektierbar ausgemacht werden können.

SELECTED — Ein Gadget mit diesem Flag wird vom System direkt im angewählten Zustand dargestellt. Durch die Abfrage dieses Flags können Sie herausfinden, ob ein Gagdet selektiert ist oder nicht.

GRELBOTTOM — Durch das Setzen dieses Flags wird ein Gadget nicht relativ zum oberen Rand ("TopEdge") des Fensters oder Requesters positioniert, sondern zum unteren Rand ("BottomEdge"). Auf diese Weise kann man

ein Gadget in der Nähe des unteren Randes eines Requesters oder Fensters positionieren; es bleibt auch beim Vergrößern oder Verkleinern des Fensters immer an derselben Stelle und ist somit immer sichtbar.

GRELRIGHT

Dieses Flag ist mit "GRELBOTTOM" gleichzusetzen, jedoch wird hier auf den linken Rand ("LeftEdge") Einfluß genommen, d.h. die Positionsangabe bezieht sich nun auf den rechten Rand ("RightEdge").

Die Anwahl von Gadgets

Die folgenden Parameter sagen Intuition, was mit einem Gadget passieren soll, wenn der Anwender es selektiert und den linken Mausknopf gedrückt hält.

GADGHIMMEDIATE

Ist dieses Flag gesetzt, dann sendet Intuition in dem Moment, in dem das Gadget angewählt wird, eine entsprechende Nachricht an Ihr Programm. Dies geschieht jedoch nur, wenn Sie das "GADGETDOWN"-Flag im "IDCMP"-Datenfeld der "NewWindow"-Datenstruktur gesetzt haben.

RELVERIFY

Mit diesem Flag bekommen Sie von Intuition eine Nachricht, wenn ein Gadget selektiert wird (die linke Maustaste gedrückt wird) und der Anwender den Mausknopf dann losläßt. Diese Nachricht erhalten Sie allerdings nur, wenn sich der Mauszeiger beim Loslassen des Mausknopfes noch über dem Gadget befindet. Wollen Sie nur Nachrichten vom Typ "GADGETUP" oder "GADGETDOWN" empfangen, dann müssen Sie einigen zusätzlichen Programmieraufwand leisten, da Sie ja sonst nicht wissen, wo sich der Mauszeiger befindet, wenn der linke Mausknopf losgelassen wird. Beachten Sie bitte, daß Sie das Flag "GADGETUP" im "IDCMP"-Datenfeld Ihrer "NewWindow"-Datenstruktur setzen müssen.

FOLLOWMOUSE

Wird ein Gadget selektiert, dann sendet Intuition die aktuelle Position des Mauszeigers an Ihr Programm. Den Zeitpunkt der Anwahl können Sie bestimmen, indem Sie das Flag "GADGIMMEDIATE" setzen.

Auch wenn Sie Positionsangaben des Mauszeigers mit "FOLLOWMOUSE" anfordern, müssen Sie zusätzlich das "MOUSEMOVE"-Flag im "IDCMP"-Datenfeld Ihrer NewWindow-Datenstruktur setzen. Ist "FOLLOWMOUSE" gesetzt, dann können Sie jede Art mausbezogener Nachrichten analysieren und so feststellen, ob der linke Mausknopf losgelassen wurde und an welcher Position der Mauszeiger sich gerade befindet.

ENDGAGDET

Verwenden Sie Gadgets in einem Requester, dann muß mindestens eins dieses Flag gesetzt haben. Wird dieses Gadget dann angewählt, wird der Requester automatisch vom System geschlossen, so, als ob Sie die Funktion EndRequest aufgerufen hätten. In unserem Malprogramm verwenden wir "ENDGADGET" beim Gadget "Abbruch".

Border Flags

Es existieren noch vier andere Flags, die Sie verwenden können: "RIGHTBORDER", "TOPBORDER", "BOTTOMBORDER" und "LEFTBORDER". Wird eins der Flags gesetzt, dann wird der entsprechende Border (Teil der Umrahmung des Gadgets) dem Gadget angepaßt. Proportional-Gadgets (Rollbalken) haben meistens eins dieser Flags gesetzt.

In einem "GIMMEZEROZERO"-Fenster muß man oft darauf achten, daß Gadgets oder ihre Rollbalken nicht von anderen Objekten überdeckt werden. Durch das Setzen der entsprechenden Flags wird das System veranlaßt, genügend freien und geschützten Platz zur Darstellung eines Gadgets bereitzustellen.

In Listing 5.11 finden Sie das Programmfragment, das alle Datenstrukturen der Gadgets initialisiert, die in dem von uns im Malprogramm verwendeten Requester eingesetzt werden.

```
/* trgadgets.c */

struct Gadget trg[] = {               /* ABBRUCH */
{ &trg[1],                            /* Zeiger auf nächstes Gadget */
205,115,60,9,                         /* LeftEdge,TopEdge,Width,Height */
GADGHCOMP,                            /* Flags */
GADGIMMEDIATE|ENDGADGET,              /* Msg wenn Knopf gedrückt */
REQGADGET|BOOLGADGET,                 /* Requestergadget, Typ Boolean */
NULL,                                 /* Border */
NULL,                                 /* SelectRender */
&textreqtext[11],                     /* Text "Abbruch" */

0,                                    /* Mutual Exclusion */
NULL,                                 /* SpecialInfo */
TEXTWRITEGADGETS+1,                   /* Nummer des Gadgets */
NULL },                               /* Zeiger auf UserInfo */

{ &trg[2],                            /* DARSTELLUNGSMODUS */
190,3,40,9,
GADGHCOMP,
RELVERIFY|GADGIMMEDIATE,
REQGADGET|BOOLGADGET,
NULL,
NULL,
&textreqtext[7],
0,
NULL,
TEXTWRITEGADGETS+2,
NULL },

{ &trg[3],                            /* TEXTSTIL*/
190,13,80,9,
GADGHCOMP,
RELVERIFY|GADGIMMEDIATE,
REQGADGET|BOOLGADGET,
NULL,
NULL,
&textreqtext[9],
0,
NULL,
TEXTWRITEGADGETS+3,
NULL },

{ &trg[4],
55,60,140,10,
GADGHCOMP,
RELVERIFY|ENDGADGET,
```

Listing 5.11: Routine zur Initialisierung von Gadgets (Teil 1)

```
REQGADGET|STRGADGET,            /* StringGadget */
NULL,
NULL,
NULL,                           /* Zeiger auf IntuiText */
0,
&textstringstuff,
TEXTWRITEGADGETS,
NULL },

{ NULL,                         /* Keine weiteren Gadgets */
190,25,80,42,
GADGIMAGE|GADGHNONE,
RELVERIFY|GADGIMMEDIATE,
REQGADGET|PROPGADGET,           /* Proportional-Gadget */
&textimage,
&textimage,
NULL,
0,
&textslider,
TEXTWRITEGADGETS+4,
NULL }
};
```

Listing 5.11: Routine zur Initialisierung von Gadgets (Schluß)

Gadgets vom Typ "Boolean"

Das erste Gadget, das in Listing 5.11 initialisiert wird, ist vom Typ "Boolean". Gadgets dieses Typs haben nur zwei Zustände, die sie annehmen können: selektiert oder nicht selektiert.

Das Flag-Datenfeld des Gadgets enthält "GADGHCOMP"; die Farben des Gadgets werden also komplementiert, wenn es angewählt wird. Das "Activation"-Datenfeld beinhaltet "GADGIMMEDITE" und "ENDGADGET". Wird das Gadget angewählt, dann ist nur "ENDGADGET" für das System wichtig. "ENDGADGET" veranlaßt das System, den Requester automatisch zu schließen; "GADGIMMEDIATE" ist nur der Vollständigkeit halber hier aufgeführt.

Da ein Requester mehrere Gadgets mit "ENDGADGET" enthalten kann, ist es manchmal wichtig zu wissen, welches Gadget angewählt wurde, um den Re-

quester zu schließen. Wird "GADGIMMEDIATE" nicht gesetzt, dann sendet Intuition keine Nachricht an Ihr Programm, um mitzuteilen, daß ein Gadget angewählt und der Requester geschlossen wurde.

Das Datenfeld "GadgetType" wird von "REQGADGET" und "BOLLGADGET" gebildet, d.h. das Gadget ist vom Typ Boolean und gehört zu einem Requester. Das Gadget verfügt nicht über eine Umrahmung oder eine spezielle Grafik, die bei der Anwahl dargestellt wird. Wollen Sie eine andere Grafik bei der Anwahl ausgeben lassen, dann müssen Sie das Flag "GADGHIMAGE" setzen.

"GadgetText" ist ein Zeiger auf eine "IntuiText"-Datenstruktur, die u.a. die Zeichenkette "Abbruch" enthält. Mutual Exclusion wird nicht verwendet, "SpecialInfo" auch nicht, da dieses Feld nur bei Gadgets vom Typ "String" oder "Proportional" vom System verwendet wird. Die Erklärung dieser Gadgettypen folgt auf diesen Abschnitt.

Die Nummern der Gadgets (GadgetID) basieren alle auf einer Konstanten mit Namen "TEXTWRITEGADGETS". Wenn Sie das Programm erweitern und Gadgets hinzufügen, dann können Sie diese auf einfache Art und Weise neu numerieren.

Das zweite und dritte Gadget, das in Listing 5.11 definiert wird, dient zum Ändern des Zeichensatzes und des Darstellungsmodus des Textes. Als Darstellungsmodus kann entweder "JAM1" oder "JAM2" gewählt werden; als Zeichensätze stehen die beiden speicherresidenten Zeichensätze Topaz-60 und Topaz-80 zur Auswahl. Diese beiden Gadgets sind ebenfalls vom Typ Boolean. Sie können natürlich noch andere Zeichensätze in der Auswahl anbieten; Informationen zur Auswahl eines bestimmten Zeichensatzes finden Sie im Kapitel 4.

Der Unterschied zwischen diesen und dem ersten Gadget ist, daß Gadgets 2 und 3 keine "ENDGADGETs" sind, d.h. der Requester bleibt auch nach ihrer Anwahl geöffnet. Unser Programm ändert lediglich – je nach Auswahl – den Zeichensatz oder den Darstellungsmodus des Textes.

Gadgets vom Typ "String"

Das vierte Gadget, das wir in Abbildung 5.11 verwenden, ist ein String-Gadget. Mit einem solchen Gadget kann der Anwender Text oder Zahlen eingeben, die dann von Intuition an das Programm weitergegeben werden. In unserem

Fall ist das Eingabefeld genauso groß wie die maximale Anzahl von Zeichen, die eingegeben werden dürfen. Man kann jedoch ein kleines Eingabefeld definieren, in dem trotzdem eine sehr lange Zeichenkette eingegeben werden kann.

String-Gadgets müssen das Flag "GADGHCOMP" gesetzt haben. Dieses Gadget hat ebenfalls das Flag "ENDGADGET" gesetzt, so daß der Requester automatisch geschlossen wird, wenn die <Return>-Taste gedrückt wird. Mit der <Return>-Taste wird die Eingabe in einem String-Gadget abgeschlossen, egal, ob etwas eingegeben wurde oder nicht.

Das "GadgetType"-Datenfeld enthält "REQGADGET" und "STRGADGET". Weil "STRGADGET" gesetzt ist, müssen wir den "SpecialInfo"-Zeiger angeben, der auf eine "StringInfo"-Datenstruktur zeigt, die von Intuition zur Ablage der eingegebenen Zeichen verwendet wird. Die Initialisierung dieser Datenstruktur finden Sie in Listing 5.12.

Diese Datenstruktur beinhaltet Informationen, die über die normale Gadget-Datenstruktur hinausreichen. Sie besteht aus einem Pufferspeicher, in dem die Eingabe des Anwenders abgelegt wird, sowie einem – optionalen – "UNDO"-Pufferspeicher, mit dessen Hilfe die vorherige Eingabezeile wieder "zurückgerufen" werden kann. Während das Gadget aktiv ist, kann auf diese Weise eine zuvor gelöschte Eingabe wieder hergestellt werden. Wird mehr Text eingegeben, als im Eingabefeld dargestellt werden kann, dann wird der Text automatisch vom System verschoben, um Platz für neue Zeichen zu schaffen.

Beim Anwählen eines String-Gadgets erscheint automatisch der Cursor, so daß der Anwender weiß, welches Gadget aktiv ist. Damit das System weiß, wie ein String-Gadget beim ersten Aufruf aussehen soll, definiert die "StringInfo"-Datenstruktur weiterhin:

- die Position des Cursors innerhalb des Eingabefeldes (links, rechts oder in der Mitte), wenn das Gagdet angewählt wird. Wir verwenden den Wert Null, d.h. der Cursor steht bei der Anwahl in der Mitte.

- die maximale Anzahl von Zeichen, die eingegeben werden kann. In unserem Beispiel werden max. 10 Zeichen akzeptiert.

- die Position der Zeichen innerhalb des Eingabefeldes (linksbündig, rechtsbündig oder zentriert) während der Eingabe. Wir verwenden den Wert Null, d.h. die Zeichen werden bei der Eingabe zentriert.

```
/* stringinfostuff.c */

UBYTE textstring[10];
UBYTE textundo[10];
UBYTE *defaulttext = "Test";

struct StringInfo textstringstuff = {
  &textstring[0],       /* Pufferspeicher für Eingabezeile */
  &textundo[0],         /* Pufferspeicher für gelöschte Eingabezeile */
  0,                    /* Cursorposition */
  10,                   /* Maximale Anzahl Zeichen */
  0,                    /* Darstellungsart der Zeichen im Eingabefeld */
  0,0,0,0,0,NULL,0 };   /* Lokale Variablen (Intuition) */
```

Listing 5.12: Initialisierung einer "StringInfo"-Datenstruktur

Die lokalen Variablen, die intern von Intuition verwendet werden, haben in unserem Beispiel alle den Wert Null. Die globalen Variablen in dem Listing 5.12 sind die Pufferspeicher, in denen die Eingaben abgelegt werden. Beim ersten Aufruf des Requesters enthält das Eingabefeld des String-Gadgets die Zeichenkette "Test", die zuvor in den Pufferspeicher "textstring" kopiert wurde.

Gadgets vom Typ "Proportional"

Der im Malprogramm verwendete Requester verfügt noch über ein Proportional-Gadget, das die Form einer Pfeilspitze hat. Dieses Gadget ("Slider") kann mit der Maus horizontal und vertikal verschoben werden, um z.B. einen Ausschnitt aus einem großen Grafikbereich auswählen zu können.

Neben der eigentlichen "Gadget"-Datenstruktur müssen noch drei weitere Datenstrukturen verwendet werden, um ein Proportinal-Gadget darstellen zu können.

- Die "PropInfo"-Datenstruktur (sie heißt in Listing 5.13 "textslider"), die die Eingrenzung des Gadgets definiert und festlegt, auf welche Art und Weise das Gadget bei seinem ersten Aufruf dargestellt wird.

```
/* slider.c */

/* Die folgende Matrix enthält das Bild eines Diamanten. Wir
   verwenden jedoch nur die untersten 4 Bits eines jeden UWORDs und
   erhalten auf diese Weise eine nach links weisende Pfeilspitze */

UWORD textsliderimage = {
         0x03c0,0x0ff0,0x3ffc,
         0xffff,0x3ffc,0x0ff0,
         0x03c0
};

UWORD *chipsliderimage;     /* Zeiger auf einen Speicherbereich im
                               Chip-Memory. Die Grafikdaten werden
                               vom Programm hier hineinkopiert. */

struct Image textimage =
{
  /* Die Grafik ist 16 Bits breit, wir verwenden jedoch immer nur
     die untersten 8 Bit, um eine nach links weisende Pfeilspitze
     zu erhalten, die die Position des zu setzenden Textes angibt. */

  0,0,8,7,1,&textsliderimage[0],0x1,0
};

struct PropInfo textslider =
{
  FREEHORIZ|FREEVERT,     /* In jede Richtung verschiebbar */
  0,0,           /* HorizPot und VertPot. Anhand dieser Werte stellt
                    Intuition fest, wo sich der "Schieberegler" gerade
                    befindet. */
  0xffff,        /* HorizBody...wird von uns nicht verwendet */
  0xffff,        /* VertBody */
  0,0,0,0,0,0    /* Globale Variablen von Intuition */
};

InitChipSliderImage()
{
  int i;
  UWORD *s, *d;

  chipsliderimage = AllocMem(14, MEMF_CHIP);
  if(!chipsliderimage) return(FALSE);  /* Kein Speicher reserviert */

  s = textsliderimage;     /* Grafikdaten des "Schiebereglers" */
  d = chipsliderimage;     /* Speicherbereich im Chip-Memory */

  for(i=0;i<7;i++) *d++ = *s++;        /* Grafikdaten umkopieren */
}

DeleteChipSliderImage()
{
  FreeMem(chipsliderimage, 14);
}
```

Listing 5.13: Initialisierung eines Proportional-Gadgets

- Weiterhin wird eine "Image"-Datenstruktur benötigt ("textimage" in Listing 5.13), die die Größe des "Sliders" ("Schieberegler") festlegt und dem System sagt, wo die Grafikdaten im Speicher gefunden werden können. Diese Grafikdaten müssen in den untersten 512KB des Systemspeichers ("MEMF_CHIP") liegen, damit die Co-Prozessoren darauf zugreifen können. Wenn man nur die "textsliderimage"-Datenstruktur des Listings 5.13 verwenden würde, dann wäre der "Schieberegler" auf einem Amiga, der über mehr als 512KB Speicher verfügt, nicht sichtbar, da die Grafikdaten dann nicht mehr in den untersten 512KB abgelegt werden.

- Schließlich wird noch ein Datenfeld aus vorzeichenlosen "WORDS" (16 Bit) verwendet, in dem die eigentlichen Grafikdaten abgelegt werden. In dem Listing 5.13 ist "chipsliderimage" ein Zeiger auf einen vom Programm reservierten Speicherblock im Chip-Memory, in den die Grafikdaten dann hineinkopiert werden, so daß die Co-Prozessoren darauf zugreifen können.

Das "Flag"-Datenfeld in der "PropInfo"-Datenstruktur enthält "FREEHORIZ" und "FREEVERT". Der "Schieberegler" kann also mit der Maus sowohl senkrecht als auch waagerecht verschoben werden.

Beim ersten Aufruf des Gadgets haben "HorizPot" und "VertPot" den Wert Null, d.h. der Schieberegler wird in der linken oberen Ecke des Gadgets positioniert. Diese Werte ändern sich jedoch immer dann, wenn der Anwender den Schieberegler mit der Maus bewegt. Wir verwenden in der Datenstruktur statische Werte, d.h. jedesmal, wenn der Requester aufgerufen wird, befindet sich der Schieberegler an der Stelle, an die er beim letzten Requesteraufruf plaziert wurde.

Wird der Schieberegler bewegt, dann ändern sich die Werte von "HorizPot" und "VertPot" im Bereich von $0000 bis $FFFF (hexadezimal). Wie groß die Schrittweite zwischen zwei Werten ist, hängt von der Größe des Bereichs ab, innerhalb dessen der Schieberegler bewegt werden kann.

Ist dieser Bereich z.B. 20 Pixel hoch, dann erhalten Sie 20 mögliche vertikale Positionen, an denen der Schieberegler positioniert werden kann, d.h. es existieren 20 Werte, die "VertPot" annehmen kann. Jeder Wert unterscheidet sich vom benachbarten um den Wert, den die Division $FFFF (hex) geteilt durch $0014 (20 dezimal) liefert.

Um die Werte von "HorizPot" und "VertPot" interpretieren zu können, nimmt man folgende Formel:

```
mein_bereich = mein_maximum - mein_minimum;
/* Bereich feststellen, in dem der Regler verschoben werden kann */

mein_aktueller_wert = (mein_bereich * textslider.VertPot)/0xffff;
```

Intuition bietet für Proportional-Gadgets noch eine weitere interessante Möglichkeit, die wir allerdings hier nicht verwenden: Autoknob. Intuition generiert den Schieberegler bei dieser Option selbsttätig und paßt dessen Größe den jeweiligen Gegebenheiten an. Wenn Sie einen solchen Schieberegler oder Rollbalken in einem Fenster verwenden, dann ist der Regler genauso groß wie das zugehörige Gadget (er kann also nicht verschoben werden). Dies trifft nur zu, falls der gesamte Inhalt des Fensters komplett sichtbar dargestellt werden kann.

Kann im Fenster nur ein Teil des Inhalts ausgegeben werden, dann verkleinert Intuition den Rollbalken automatisch um den Prozentsatz, der dem des nicht sichtbaren Bereiches entspricht. Wollen Sie diese Möglichkeit nutzen, dann setzen Sie in der "PropInfo"-Datenstruktur das Flag "AUTOKNOB". Sie brauchen in diesem Fall keinen Zeiger auf Grafikdaten zu übergeben.

Im "Amiga Intuition Manual" erfahren Sie mehr über die einzelnen Komponenten der "PropInfo"-Datenstruktur. Die restlichen Parameter werden jedoch nur intern von Intuition verwendet und sind daher für uns nicht interessant.

Menüverarbeitung

Um festzustellen, welcher Menüpunkt aus welchem Menü oder Untermenü vom Anwender selektiert ("MENUPICK") wurde, können Sie die Systemmakros "MENUNUM", "ITEMNUM" und "SUBNUM" auslesen. Die hier enthaltenen Werte werden aus dem "Code"-Datenfeld der entsprechenden "IntuiMessage" gebildet.

Die gültigen Wertebereiche reichen von 0 bis 30 bei den Makros "MENUNUM" und "SUBNUM" bzw. von 0 bis 63 für "ITEMNUM". Jeder Wert repräsentiert die Nummer des Menüs, des Menüpunktes und eventuell die Nummer der Größe des angewählten Untermenüs.

Ruft ein Anwender ein Menü auf, ohne jedoch einen Menüpunkt auszuwählen, dann enthält das "Code"-Datenfeld der "IntuiMessage" den Wert "MENUNULL" und das "Class"-Datenfeld den Wert "MENUPICK".

Um festzustellen, ob ein Anwender zwar ein Menü aufgerufen, aber nichts selektiert hat, können Sie entweder diese Datenfelder direkt überprüfen oder die Werte der Makros auslesen. "MENUNUM"(code) enthält in einem solchen Fall den Wert "NOMENU", "ITEMNUM"(code) enthält "NOITEM", und das Makro "SUBNUM"(code) liefert den Wert "NOSUB". Am einfachsten ist es, auf gültige Werte hin zu prüfen und alles andere zu ignorieren. Die von Intuition gesetzten Grenzen für Menüs und deren Komponenten sind:

- maximal 31 Menüs (Menünamen) pro Menüleiste
- maximal 63 Menüpunkte (Text oder Grafik) pro Menü
- maximal 31 Menüpunkte (Text oder Grafik) pro Untermenü

Mit diesen festgesetzten Grenzen steht eine große Anzahl von Möglichkeiten zur Menügestaltung zur Verfügung. In Listing 5.14 finden Sie die Routine zur Menüverarbeitung, die im Malprogramm verwendet wird; sie wird aufgerufen, wenn eine IntuiMessage der Art "MENUPICK" vorliegt. Zur Erinnerung: das Farbmenü dient zur Auswahl einer Zeichenfarbe, mit dem Textmenü wird der Requester aufgerufen, der zur Eingabe und Positionierung von Text dient.

In der Routine werden nur gültige Werte akzeptiert; Werte außerhalb des Bereiches oder "MENUNULL" werden einfach ignoriert, um die Routine nicht unnötig schwerer zu machen.

Gadgetverarbeitung

Die Routine in Listing 5.15 ist ein wenig kompliziert, da sie drei verschiedene Gadgetarten verarbeiten soll – String-, Proportional- und Boolean-Gadgets. Mit Hilfe einer "switch"-Anweisung gestaltet sich dies aber relativ einfach.

Das Listing 5.16 beinhaltet drei Routinen, die die Darstellung des Textes kontrollieren – Darstellungsmodus, Zeichensatz und schließlich die Ausgabe des Textes. Die ersten beiden Routinen sind recht einfach zu verstehen. Die dritte Routine analysiert die Werte der Schieberegler-Position, um so festzulegen, an welcher Stelle des Bildschirmes der Text ausgegeben werden soll.

Beachten Sie bitte, daß in der Funktion textstyle das System direkt die Zeigerwerte des Textes vergleicht, anstatt die Funktion strcmp zu verwenden. Sind die beiden Werte gleich, dann weisen beide Zeiger auf die gleiche Zeichenkette. Weiterhin enthält Listing 5.16 die "IntuiText"-Datenstruktur für den verwendeten Requester, sowie die Verarbeitungsroutinen für die Gadgets.

Das Malprogramm

In Listing 5.17 finden Sie schließlich das Hauptprogramm, das alle einzelnen Programmteile zusammenfaßt und so das lauffähige Malprogramm erstellt.

```
/* menupick.c */

#define COLORMENU 0
#define TEXTMENU 1
#define FIRSTITEM 0

MenuPick(im)
struct IntuiMessage *im;
{
  USHORT code, k;

  code = im->code;

  switch(MENUNUM(code))
           {
             case COLORMENU: k = ITEMNUM(code);
                             if(k >= 0 && k <= 15) /* Im Bereich? */
                               {
                               SetAPen(rp,ITEMNUM(code));
                               SetDrMd(rp,JAM1);
                               }
                             break;

             case TEXTMENU:  /* Requester öffnen */
                             Request(&textrequest,w);
                             break;

             default: break;
             }
}
```

Listing 5.14: Die Routine zur Menüverarbeitung

```
/* gadgetup.c */

#define GADGETID ((struct Gadget *)IAddress)->GadgetID

GadgetUp(ms)
struct IntuiMessage *ms;
{
  SHORT id;
  struct Gadget *g;

  g = (struct Gadget *)(ms->IAddress);
  id = g->GadgetID;     /* Nummer des Gadgets feststellen */

  switch(id)
        {
         case TEXTWRITEGADGETS:

                                textwrite();  /* Text ausgeben */
                                break;

         case TEXTWRITEGADGETS+1:

                                break;  /* Das Gadget "Abbruch"
                                           wurde selektiert;
                                           Requester schließen */

         case TEXTWRITEGADGETS+4:

                                break;  /* Proportional-Gadget.
                                           Werte werden erst bei
                                           der Textausgabe
                                           ausgewertet. */

         case TEXTWRITEGADGETS+2:

                                textstyle();     /* Zeichensatz */
                                break;

         case TEXTWRITEGADGETS+3:

                                textmode();      /* JAM1, JAM2 */
                                break;

         default: break;
        }
}
```

Listing 5.15: Routine zur Verarbeitung von Gadgets

```
/* textstuff.c */

int txfont, txmode;      /* Globale Variablen zur Textdarstellung */

extern struct Window *w;
extern struct TextAttr TestFont,modfontattr;
extern struct PropInfo textslider;
extern struct Gadget trg[];
extern struct Requester textrequest;

struct IntuiText textreqtext[] = {
{ 0,1,JAM1,5,3,&TestFont,"Neuer Modus:",&textreqtext[1] },
{ 0,1,JAM1,5,13,&TestFont,"Neuer Stil:",&textreqtext[2] },
{ 0,1,JAM1,5,30,&TestFont,"Text positionieren:",&textreqtext[3] },
{ 0,1,JAM1,5,80,&TestFont,"Text eingeben",&textreqtext[4] },
{ 0,1,JAM1,25,90,&TestFont,"im Grafikbereich",&textreqtext[5] },
{ 0,1,JAM1,5,105,&TestFont,"<RETURN>=Textausgabe",&textreqtext[6] },
{ 0,1,JAM1,5,115,&TestFont,"Abbruch = Ende",NULL },

{ 1,0,JAM2,1,0,&TestFont,"JAM1",NULL },
{ 1,0,JAM2,1,0,&TestFont,"JAM2",NULL },
{ 1,0,JAM2,1,0,&TestFont,"Topaz-80",NULL },
{ 1,0,JAM2,1,0,&TestFont,"Topaz-60",NULL },
{ 1,0,JAM2,1,0,&TestFont,"Abbruch",NULL }
};

textstyle()
{
  if(trg[1].GadgetText == &textreqtext[7])
    {
     trg[1].GadgetText = &textreqtext[8];
     txmode = 2;
    }
  else
    {
     if(trg[1].GadgetText == &textreqtext[8])
       {
        trg[1].GadgetText = &textreqtext[7];
        txmode = 1;
       }
    }

  RefreshGadgets(&trg[1],w,&textrequest);   /* Gadgets erneuern */
}

textmode()
{
 if(trg[2].GadgetText == &textreqtext[9])
   {
    trg[2].GadgetText = &textreqtext[10];
    txfont = 60;
   }
```

Listing 5.16: Routinen zur Initialisierung von "IntuiText"-Datenstrukturen (Teil 1)

```
  else
     {
      if(trg[2].GadgetText == &textreqtext[10])
        {
         trg[2].GadgetText = &textreqtext[9];
         txfont = 80;
        }
     }

  RefreshGadgets(&trg[1],w,&textrequest);    /* Gadgets erneuern */
}

textwrite()
{
  ULONG temp1, temp2;

  struct TextFont *oldfontsave, *myfontptr;

  /* Gadget skalieren, die obersten 8 Bits werden verwendet */
  temp1 = ((textslider.HorizPot >> 8) * (WWIDTH-1)) >>8;
  temp2 = ((textslider.VertPot >> 8) * (WHEIGHT-1)) >>8;

  Move(rp,temp1,temp2);

  if(txmode == 1) SetDrMd(rp,JAM1);
  else SetDrMd(rp,JAM2);

  if(txfont == 80) modfontattr.ta_YSize = 8;
  else modfontattr.ta_YSize = 9;

  oldfontsave = rp->Font;  /* Alten Zeichensatz zwischenspeichern */

  myfontptr = (struct TextFont *)OpenFont(&modfontattr);
        /* Neuen Zeichensatz verwenden */

  if(!myfontptr)
     {
      printf("Zeichensatz läßt sich nicht öffnen!\n");
      return(0);
     }

  SetFont(rp,myfontptr);
  Text(rp,&textstring[0],strlen(textstring));

  SetFont(rp,oldfontsave);   /* Alten Zeichensatz verwenden */

  CloseFont(myfontptr);   /* Und den neuen schließen */
}
```

Listing 5.16: Routinen zur Initialisierung von "IntuiText"-Datenstrukturen (Schluß)

```
/* main.c */

#include "exec/types.h"

#define EDITLEFT 4
#define EDITRIGHT 324
#define EDITTOP 12
#define EDITBOTTOM 180
#define MAXVIEWS 9
#define BOBDEPTH 4
#define FRAMEWIDTH 80
#define FRAMEHEIGHT 42
#define getc() Read(stdin,c,2);
#define TXHEIGHT 8
struct Frame
{
  SHORT xmin,ymin;
  SHORT xmax,ymax;
  struct BitMap bitmap;
};

#define qr(r,rl,rh) ((r >= rl && r <= rh) ? 1 :0)
#define MOVEGADGETS 0x0
#define COLORGADGETS 0x10
#define TEXTCOLORGADGETS 0x30
#define TEXTWRITEGADGETS 0x50
#define DISKRWGADGETS 0x60
#define SCROLLGADGETS 0x68
#define RECONFIGGADGETS 0x70
#define HELPGADGETS 0x90
#define EXITGADGETS 0x98
#define DEPTH 4
#define WWIDTH 320
#define WHEIGHT 190

include "intuition/intuition.h"
include "exec/memory.h"
include "ram:imageedit.h"
include "ram:myscreen2.h"
include "ram:event2.c"
include "ram:stubs1.c"
include "ram:ticks.c"
```

Listing 5.17: Das Hauptprogramm (Teil 1)

```
include "ram:mousebuttons.c"
include "ram:stringinfostuff.c"
include "ram:slider.c"
include "ram:textstuff.c"
include "ram:trgadgets.c"
include "ram:inittr.c"
include "ram:initmenu.c"
include "ram:menupick.c"
include "ram:gadgetup.c"

struct Window *w;
struct RastPort *rp;
struct ViewPort *vp;
struct Screen *screen;
struct Image colorimage[32];   /* Max. Anzahl Farben */
struct MenuItem coloritem[32];  /* Nur bei DEPTH = 5 */
struct Menu menu[2];

long IntuitionBase=0;
long GfxBase=0;

#define ITEMSTUFF (ITEMENABLED|HIGHBOX)
#define CW 40
#define CH 25

SHORT palette[] = { 2,4,8,16,32,64 };

main()
{
  struct IntuiMessage *mess;
  int havevalidimage;

  GfxBase = OpenLibrary("graphics.library",0);
  if(!GfxBase)
      {
        printf("graphics.library nicht geöffnet!\n");
        exit(0);
      }

  IntuitionBase = OpenLibrary("intuition.library",0);
  if(!IntuitionBase)
      {
        printf("intuition.library nicht geöffnet!\n");
        exit(0);
      }
```

Listing 5.17: Das Hauptprogramm (Teil 2)

```
    if(!(screen = OpenScreen(&ns))) exit(0);

    nw.Screen = ns;

    w = OpenWindow(&nw);
    rp = w->RPort;
    vp = &w->WScreen->ViewPort;

    InitMenu();

    SetMenuStrip(w, menu);

    InitTextRequest();

    havevalidimage = FALSE;

    if(InitChipSliderImage())    /* Grafikdaten im Chip-Memory? */
      {
      textimage.ImageData = chipsliderimage;
      havevalidimage = TRUE;
      }

    while(1)      /* "Ewige" Schleife; warten auf CLOSEWINDOW */
         {
         /* Intuition kann mehr als nur eine Message an Ihr Programm
            senden. Daher muß der MessagePort ausgelesen werden, bevor
            auf neue Nachrichten gewartet wird. Diese Routine erledigt
            dies und stellt so sicher, daß unser Task nicht auf eine
            Message wartet, die schon längst angekommen ist */

         mess = (struct IntuiMessage *)GetMsg(w->UserPort);

         if(!mess) WaitPort(w->UserPort);
         else if(!HandleEvent(mess)) break;
         }

    /* Programmende...Speicherplatz etc. freigeben */

    if(havevalidimage) DeleteChipSliderImage();

    ClearMenuStrip(w);
    CloseWindow(w);
    CloseScreen(screen);
    CloseLibrary(GfxBase);
    CloseLibrary(IntuitionBase);

}   /* Das war's... */
```

Listing 5.17: Das Hauptprogramm (Schluß)

Zusätzliche Extras

Das in diesem Kapitel schrittweise entwickelte Malprogramm enthält viele Routinen, die Sie auch in eigenen Programmen verwenden können. Wenn Sie das Malprogramm durch weitere Optionen ergänzen wollen, dann finden Sie in den folgenden Abschnitten einige Anregungen, die vielleicht nützlich sind, um eigene Ideen zu verwirklichen.

Die Kombination von Text und Grafiken

Intuition bietet Ihnen die Möglichkeit, mit Hilfe des "GADGIMAGE"-Flags Gadgets zu erstellen, die nicht nur Text (IntuiText) enthalten, sondern die mittels "GadgetRender" und "SelectRender" eine eigene kleine Grafik darstellen.

Dies gilt auch für Menüpunkte, bei denen das "ITEMTEXT" verwendet wird, um Intuition mitzuteilen, daß der Menüpunkt aus Text und nicht aus einer Grafik besteht. "ItemFill" und "SelectFill" haben in diesem Fall den Wert Null. Was aber tun Sie, wenn Sie sowohl Grafik als auch Text in einem Gadget oder Menüpunkt verwenden wollen?

Sie brauchen hierzu nur zwei Gadgets oder Menüpunkte miteinander zu kombinieren; ein Gadget oder Menüpunkt besteht dabei aus Text (IntuiText), das andere wird durch eine Image-Datenstruktur definiert. Die Größe der Rechtecke, innerhalb derer der Menüpunkt oder das Gadget "angeklickt" werden kann, muß natürlich so gewählt werden, daß z.B. ein Gadget vom anderen komplett umrahmt wird.

Das Gadget oder der Menüpunkt, der vom anderen teilweise überdeckt wird, sollte in der verknüpften Liste der Datenstrukturen die höhere Proiorität besitzen. In der Abbildung 5.4 finden Sie ein Beispiel, wie eine Grafik (in diesem Fall ein farbiges Rechteck) in einen Text-Menüpunkt eingebettet werden kann, so daß der Eindruck entsteht, es würde sich um einen einzigen Menüpunkt handeln, der sowohl Text als auch Grafik enthält.

Der Menüpunkt 1 in diesem Beispiel hat das "ITEMTEXT"-Flag gesetzt, d.h. er enthält Text, keine Grafik. Menüpunkt 2 hat dieses Flag nicht gesetzt; es wird durch eine "Image"-Datenstruktur definiert. Da Menüpunkt 2 eine kleinere Priorität als Menüpunkt 1 hat, kann er niemals angewählt werden, da er komplett von Menüpunkt 1 umgeben wird.

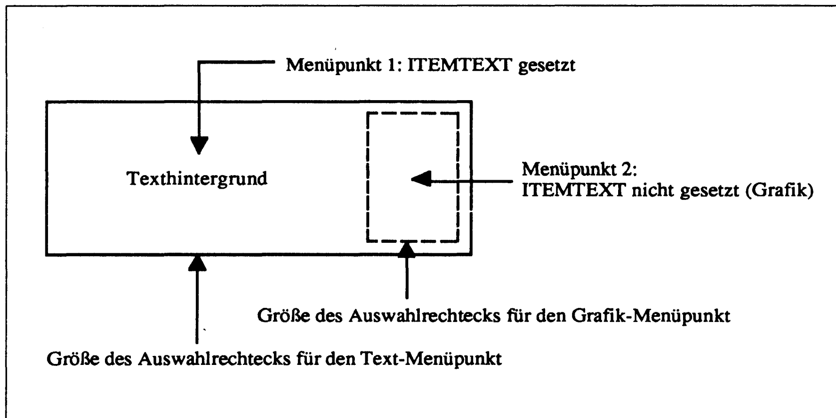


Abb. 5.4: Kombination zweier Menüpunkte (Grafik und Text)

Als Ergebnis erhält man auf diese Weise einen Menüpunkt, der Text und Grafik enthält; der Anwender wird diesen "Trick" nicht bemerken, auch dann nicht, wenn der Menüpunkt angewählt wird.

Menükomponenten

Intuition läßt dem Programmierer die Freiheit, die verknüpften Listen der Menüpunkt-Datenstrukturen beliebig einzusetzen. Sobald eine solche Liste erstellt ist, kann sie sowohl in einem Menü als auch beliebig oft als Untermenü für Menüpunkte verwendet werden. Wenn Sie z.B. für das Malprogramm ein Menü folgender Form erstellt haben:

```
Farbauswahl

    Zeichenfarbe

    Text (Vordergrund)
    Text (Hintergrund)
```

können Sie das Menü, das die farbigen Rechtecke als Menü zur Verfügung stellt, jedem der drei Menüpunkte zuweisen, so daß jeder von ihnen ein identisches Untermenü erhält, das bei der Anwahl sichtbar wird.

Eingebunden wird dieses Untermenü wie folgt (schematisch):

```
<farbauswahl>.FirstItem = <zeichenfarbe>

<zeichenfarbe>.NextItem = <text (vordergrund)>
<zeichenfarbe>.SubItem = <erster_menuepunkt_im_farbmenue>

<text (vordergrund)>.NextItem = <text (hintergrund)>
<text (vordergrund)>.SubItem = <erster_menuepunkt_im_farbmenue>

<text (hintergrund)>.NextItem = NULL;
<text (hintergrund)>.SubItem = <erster_menuepunkt_im_farbmenue>
```

Jeder der drei Menüpunkte erhält auf diese Weise ein Untermenü, aus denen der Anwender die gewünschte Farbe auswählen kann. Wenn Sie zusätzlich die Möglichkeit zur Kombination zweier Menüpunkte (Grafik und Text) verwenden wollen, um die aktuell verwendete Farbe anzuzeigen, dann müssen Sie das Flag "CHECKIT" aus der Datenstruktur entfernen, da ja von allen drei Menüpunkten das gleiche Untermenü benutzt wird. In der Abbildung 5.5 sehen Sie ein Beispiel zur Positionierung eines Untermenüs, wie es sich bei der Anwahl dem Benutzer präsentiert.

Da ein Untermenü immer relativ zum zugehörigen Menüpunkt positioniert wird, ändert sich am Aussehen des Untermenüs nichts, egal, welcher der drei Menüpunkte angewählt wird. Beachten Sie bitte, daß die Werte für "LeftEdge" und "TopEdge" bei Menüpunkten auch negativ sein dürfen. In diesem Fall werden Untermenüs etwas verschoben dargestellt, was u.U. zur Übersichtlichkeit beitragen kann. In der Abbildung 5.6 finden Sie ein Beispiel für diese Art der Positionierung.

In diesem Kapitel wurden die wichtigsten Komponenten behandelt, die zur Arbeit und Programmierung von Intuition notwendig sind: Screens, Fenster, Gadgets, Requester und Menüs. Hier noch einmal eine ganze kurze Gedankenstütze zum Abschluß:

- Mit Screens legen Sie die Auflösung und die Anzahl verfügbarer Farben für Ihren Grafikbereich fest.

- Fenster dienen zur Unterteilung eines Screens in mehrere unabhängige Bereiche.

- Mit Menüs, Requestern und Gadgets können Sie Eingaben vom Anwender an Ihr Programm machen lassen.

Im "Amiga Intuition Manual" finden Sie noch mehr Informationen zur Arbeit mit Intuition bzw. zu jeder Komponente, die in diesem Kapitel besprochen wurde. Im oben erwähnten Buch finden Sie auch die Funktionen und Datenstrukturen genauestens erläutert.

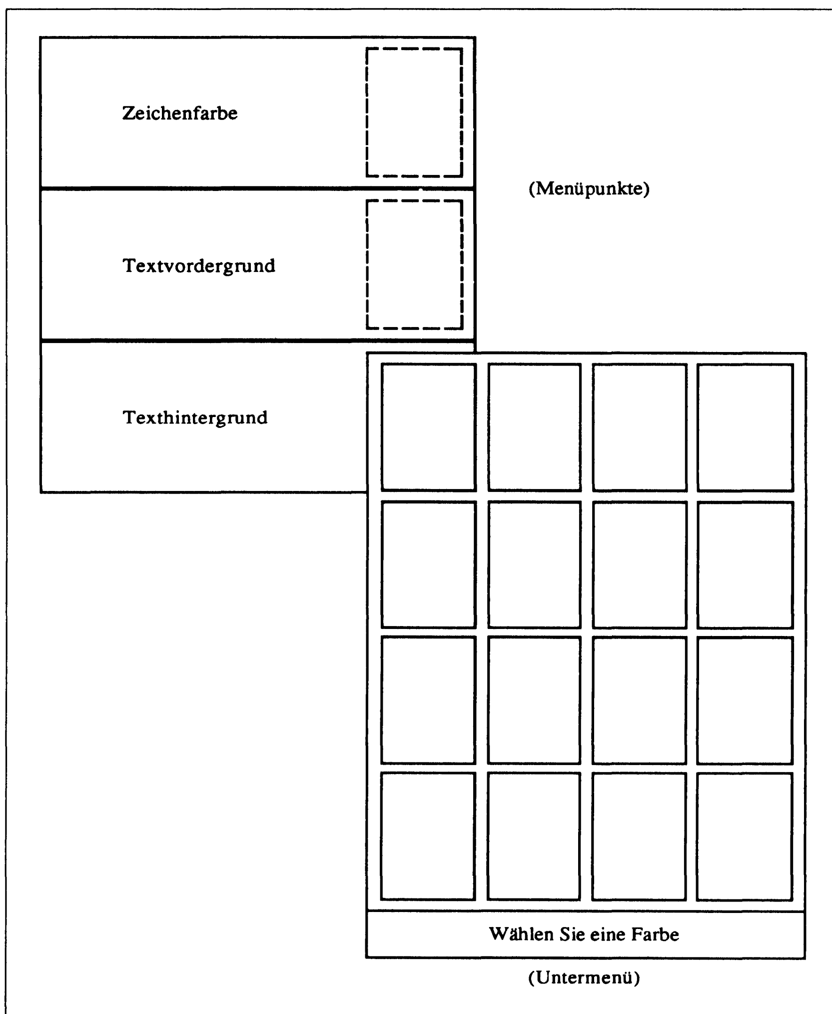


Abb. 5.5: Ein zu einem Menüpunkt gehörendes Untermenü

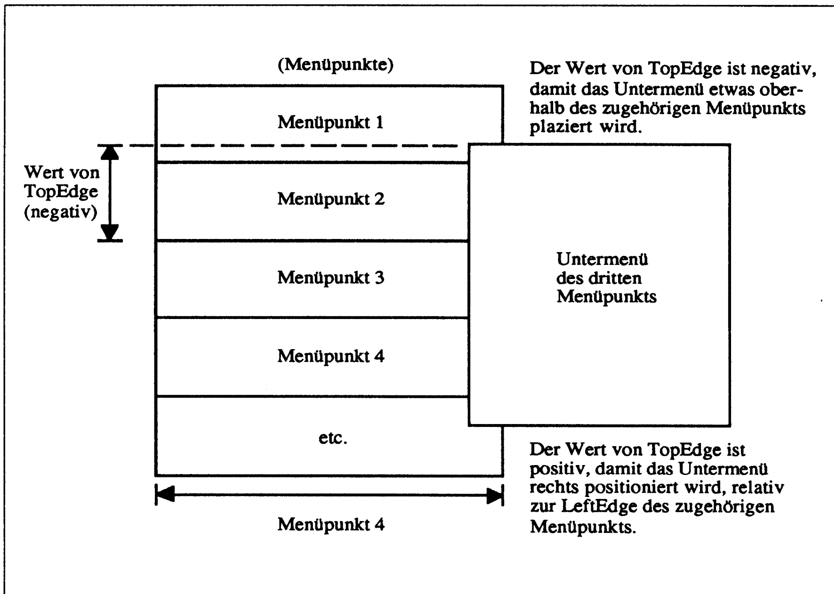
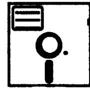
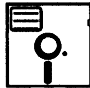
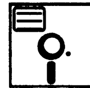
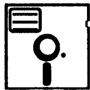
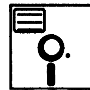
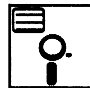
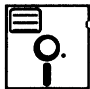
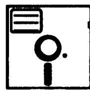
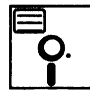
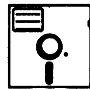


Abb. 5.6: Relative Positionierung von Untermenüs

6

Kapitel 6

Devices – Virtuelle Geräte

Wie wir bereits in Kapitel 3 gesehen haben, verwendet Exec für alle Ein-/Ausgabeoperationen ein einheitliches Protokoll. Ein-/Ausgabeoperationen werden in Form eines "IORequest"-Datenblocks vom Programm aus – in Form einer Message (Nachricht) – an den "Gerätetreiber" übergeben, der, je nach Art der Operation, ein eigenständiger Task ist. Die Aufgabe dieses Treibers ist z.B. die für eine Ein-/Ausgabe notwendigen Bits zu manipulieren, um die Operation durchführen zu können. Auf diese Weise kann man mit den Devices auf einer "höheren" Programmebene kommunizieren; man muß sich also nicht mit hardware-internen Vorgängen beschäftigen, um ein Device korrekt ansprechen zu können.

Etwas ist jedoch enorm wichtig, wenn Sie mit Devices kommunizieren: der Speicherplatz, der für Ihren "IORequest"-Datenblock verwendet wird, "gehört" bis zum Ende der Ein-/Ausgabe dem angesprochenen Gerät, welches den Speicherblock intern verwendet. Versuchen Sie daher niemals, in den Speicherbereich eines "IORequest"-Blocks zu schreiben oder Daten aus ihm zu lesen, wenn das Device ihn noch für die laufende Operation verwendet. Andernfalls ist ein Systemabsturz die Folge, da Sie vielleicht Daten ändern, die vom Device erst noch verwendet werden müssen.

In diesem Kapitel beschäftigen wir uns mit verschiedenen Devices: dem Zeitgeber (Timer), dem Console-, Input-, Tastatur- und dem Gameport-Device. Die serielle Schnittstelle und der Druckerport werden nicht behandelt. Wenn Sie Informationen hierzu benötigen, dann finden Sie diese Devices ausführlich im "Amiga ROM Kernel Manual" erläutert. Beachten Sie weiterhin, daß mit dem Betriebssystem Version 1.2 viele Parameter der seriellen Schnittstelle mit dem Programm "Preferences" geändert werden können, so daß Programme sich nicht mehr um die Initialisierung dieser Parameter kümmern müssen.

Wenn Sie das Malprogramm aus Kapitel 5 ausprobiert haben, dann waren Sie sicherlich über die langsame Geschwindigkeit der Grafikausgabe erstaunt; der Amiga ist nämlich in der Lage, um ein Vielfaches schneller zu zeichnen. Der Grund für die langsame Ausgabe ist in der Routine zu suchen, die entscheidet, wann ein Bildpunkt gesetzt werden soll. Gesetzt wird ein Pixel im Malprogramm nur dann, wenn der Anwender die linke Maustaste gedrückt hält und eine Nachricht der Art "INTUITICKS" von Intuition an das Malprogramm gesendet wird. Nachrichten der Art "INTUITICKS" werden aber nur jede zehntel Sekunde vom System erzeugt; es können also maximal 10 Bildpunkte pro Sekunde gesetzt werden.

Es gibt andere Möglichkeiten, zu entscheiden, wann ein Bildpunkt gesetzt werden kann. Sie können z.B. das Flag "FOLLOWMOUSE" in der NewWindow-Datenstruktur setzen und anhand der von Intuition gelieferten Werte die Position des Mauszeigers bestimmen und an der entsprechenden Stelle dann einen Pixel setzen lassen. Die wohl eleganteste Möglichkeit zu bestimmen, wann ein Pixel gesetzt werden soll, ist die Verwendung des Hardware-Zeitgebers (Timer), mit dem wir uns in den folgenden Abschnitten genauer beschäftigen wollen.

Der Zeitgeber (Timer Device)

Das "Timer Device" besteht aus zwei Einheiten. Eine von ihnen verwaltet einen sehr genauen Zeitgeber, der in Intervallen von 16,67 Millisekunden getaktet wird; d.h. es können bis zu 60 "Events" (Ereignisse) pro Sekunde erzeugt werden. Dieser Zeitgeber ist also um ein Vielfaches schneller als Ereignisse vom Typ "INTUITICKS", die jedoch nicht für schnelle Verarbeitungsgeschwindigkeiten geschaffen wurden. Der oben erwähnte Präzisionszeitgeber hat den Namen "VBLANK".

Die andere Einheit des Timer Device ist der "MICROHZ"-Zeitgeber, der in Schritten programmiert werden kann, die im Mikrosekundenbereich liegen können. Dieser Zeitgeber ist nicht so präzise wie "VBLANK", jedoch können mit ihm noch mehr Events pro Sekunde erzeugt werden; man verwendet ihn also hauptsächlich dann, wenn die Geschwindigkeit in einem Programm einen entscheidenden Faktor darstellt.

Mit Hilfe der Zeitgeber kann man Tasks für eine bestimmte Zeit "schlafen" lassen, bis eine Nachricht vom Timer Device am Message Port des Tasks vorliegt. Während andere Tasks weiterhin von Exec abgearbeitet werden, "schläft" ein solcher Task dann bis zum Eintreffen der Nachricht. Manchmal kann es jedoch sein, daß ein schlafender Task nicht direkt nach dem Ankommen einer Nachricht "aufgeweckt" wird.

Ein Task wartet immer mindestens für den Zeitraum, den Sie im "IORequest"-Block für den Timer angegeben haben. Ist dieser Zeitraum verstrichen, dann sendet der Timer eine entsprechende Nachricht an den Task, der dann normalerweise von Exec "aufgeweckt" wird. Laufen jedoch noch andere Tasks im System, dann kann es – je nach Auslastung des Systems – länger dauern, bis der Task wieder lauffähig gemacht wird. Bevor das Timer Device verwendet werden kann, sind zunächst noch einige Arbeiten vom Programm zu erledigen, die Sie nachfolgend aufgelistet finden.

1. Sie müssen angeben, welche Einheit des Timers ("VBLANK" oder "MICROHZ") verwendet werden soll und den Timer öffnen.

2. Zur Kommunikation mit dem Device muß eine "IORequest"-Datenstruktur initialisiert werden.

3. Ihr Programm muß einen Reply Port erhalten, damit die Nachricht des Timers nach Ablauf der angegebenen Zeit empfangen werden kann.

4. Die Startadresse dieses Reply Ports muß dann in die Datenstruktur des Message-Blocks geschrieben werden.

5. Zur Kommunikation mit dem Timer wird der "IORequest"-Block an das Device geschickt, das dann die Ein-/Ausgabeoperation durchführt.

6. Während der Ein-/Ausgabe kann Ihr Task "schlafen" und auf das Eintreffen der Nachricht vom Timer warten oder weiterlaufen und später den Reply Port auslesen.

7. Wird der Timer nicht mehr verwendet, dann muß er korrekt geschlossen werden, um den belegten Speicherplatz freizugeben.

Die meisten Devices des Amiga können immer nur von einem einzigen Task geöffnet und verwendet werden. Das Timer Device kann hingegen von mehreren Tasks gleichzeitig benutzt werden.

Wenn Sie dem Timer einen "IORequest"-Block übergeben, also eine Ein-/Ausgabeoperation durchführen wollen, dann wird dieser Datenblock vom Timer modifiziert. Weiterhin verwaltet der Timer eine Liste, in der alle laufenden Operationen eingetragen sind. Beim Eintreffen eines neuen "IORequest"-Blocks wird diese Liste neu sortiert, wobei der "IORequest"-Block als erstes in der Liste steht, der als nächstes beendet wird. Nehmen wir als Beispiel einmal an, daß drei Tasks den Timer ansprechen:

- Task 1: Der Timer soll fünf Minuten warten, dann eine Nachricht schicken.
- Task 2: Der Timer soll eine Minute warten, dann eine Nachricht schicken.
- Task 3: Der Timer soll zwei Minuten warten, dann eine Nachricht schicken.

Der Timer modifiziert die "IORequest"-Blocks; die vom Timer verwaltete Liste wird neu sortiert und nach anderen Gesichtspunkten geordnet:

- Task 2: Nachricht an diesen Task in einer Minute versenden.
- Task 3: Nachricht an diesen Task eine Minute nach der letzten verschickten Nachricht versenden.
- Task 1: Nachricht an diesen Task drei Minuten nach der letzten verschickten Nachricht versenden.

Diese neu erstellte Liste wird dann von den eigentlichen Hardware-Zeitgebern sequentiell (nacheinander) abgearbeitet. Ist der angegebene Zeitraum verstrichen, dann wird der "IORequest"-Block an die entsprechende Einheit des Timer Device zurückgegeben, die dann ihrerseits eine Nachricht an den entsprechenden Task sendet.

Jedes Device des Amiga verwendet zur Kommunikation mit Programmen eine eigene Version des "IORequest"-Blocks. Man kann jedoch die Arbeit mit dem Timer (als Beispiel) vereinfachen, indem man eine Systemroutine namens CreateStdIO benutzt, mit der sich die Initialisierung eines "IORequest"-Blocks recht einfach gestaltet. Die Datenfelder, die im "IORequest"-Block verwendet werden, den diese Funktion als Returnwert liefert, können eigene Namen erhalten, so daß man mit Hilfe von "#define"-Anweisungen "lesbare" Bezeichnungen erhält:

```
#define SEKUNDEN  io_Actual
#define MIKROSEK  io_Length
```

Sowohl "io_Actual" als auch "SEKUNDEN" sind in einer normalen "timeval"-Datenstruktur vom Typ "ULONG", so daß durch die Angabe der Namen bereits entsprechender Speicherplatz reserviert wird, wodurch die Verwendung einer speziellen Funktion (o.ä.) nicht notwendig ist.

In Listing 6.1 finden Sie eine Routine, mit der ein "IORequest"-Block für die Kommunikation mit dem Timer Device initialisiert werden kann. Beachten Sie bitte, daß in dieser Routine auch der Reply Port zum Empfang der Timer-Nachricht nach Beendigung der Operation eingerichtet wird. Die Adresse dieses Reply Ports ist einer der Parameter, der der Funktion "CreatePort" beim Aufruf übergeben wird. Diese Routine können Sie immer dann verwenden, wenn Sie einen "IORequest"-Block für einen Zeitgeber benötigen.

Die Routine InitTimer in Listing 6.2 wird zur allerersten Kommunikation mit dem Timer verwendet. In einem Programm werden Sie normalerweise nach dem Eintreffen der Timer-Nachricht entsprechende Dinge tun und den "IORequest"-Block zur weiteren Verwendung reinitialisieren. Die Variablen "timerSeconds" und "timerMicros" sind global definiert, d.h. sie können auch von anderen Routinen geändert werden.

Um die gezeigte Routine aufrufen zu können, müssen Sie ihr den Zeiger auf den "IOStdReq"-Datenblock übergeben, den Sie nach dem Aufruf der Funktion PrepareTimer als Returnwert erhalten. Wenn Sie den Zeitgeber nicht mehr benötigen, dann verwenden Sie die Routine aus Listing 6.3, um den Timer zu schließen. Stellen Sie jedoch vor dem Aufruf sicher, daß alle von Ihnen angeforderten Ein-/Ausgabeoperationen auch beendet wurden.

Das Listing 6.4 enthält ein Programmfragment, in dem der Aufruf der notwendigen Funktionen gezeigt ist, um mit dem Timer Device arbeiten zu können. Wir verwenden in diesem Beispiel die Möglichkeit von Exec, einen Task vorübergehend "schlafen" zu lassen, bis ein ihn betreffendes Ereignis an seinem Message Port ankommt. Sie können Exec mitteilen, daß Ihr Task "aufgeweckt" werden soll, wenn eine Nachricht vom Timer kommt (d.h. wenn die angegebene Zeit verstrichen ist) oder wenn eine "IntuiMessage" vorliegt. (Beachten Sie bitte, daß Sie im Listing 6.2 auch "UNIT_MICROHZ" anstatt "UNIT_VBLANK" verwenden können.)

Im Malprogramm aus Kapitel 5 verwendeten wir folgende Möglichkeit, den Task bis zum Eintreffen einer Nachricht schlafen zu lassen:

```
WaitPort(w->UserPort);
```

```
/* preparetimer.c */

struct IOStdReq *tr, CreateStdIO();
struct MsgPort *tp, *CreatePort();

PrepareTimer()
{
  tr = NULL;
  tp = CreatePort(0,0); /* Port einrichten (kein Name, Priorität 0) */

  if(!tp)   /* Port nicht eingerichtet */
     {
       /* Hier folgt die Fehlerbehandlung */

       return(tp);   /* <tp> hat hier den Wert NULL */
      }

  tr = CreateStdIO(tp);

  if(!tr)     /* Kein Speicher für IORequest-Datenblock */
     {
       /* Hier folgt die Fehlerbehandlung */
     }

  return(tp);
}
```

Listing 6.1: Die "PrepareTimer"-Routine

```
/* timerstuff.c */

int timerSeconds, timerMicros;    /* Global definiert */

InitTimer(trq)
struct IOStdReq *trq;
{
  OpenDevice(TIMERNAME,UNIT_VBLANK,trq);
  SendTimer(trq);
}

SendTimer(trq)
struct IOStdReq *trq;
{
  trq->SEKUNDEN = timerSeconds;
  trq->MIKROSEK = timerMicros;
  SendIO(trq);
}
```

Listing 6.2: Routinen zur Kommunikation mit dem Zeitgeber

```
/* deletetimer.c */

DeleteTimer(trq)
struct IOStdReq *trq;
{
  struct MsgPort *mp;

  mp = trq->ReplyPort;

  DeleteStdIO(trq);
  DeletePort(mp);
}
```

Listing 6.3: Routine zum Schließen des Timer Device

```
struct IOStdReq *meinTimerRequest;

/* Weitere Initialisierungen... */

meinTimerRequest = PrepareTimer();   /* IORequest-Block anfordern */
InitTimer(meinTimerRequest);        /* Ein-/Ausgabeoperation starten */

/* Weitere Programmteile (evtl. Warteschleife), warten auf
   Programmende durch Anwender o.ä. */

AbortIO(meinTimerRequest);     /* Vor dem Programmende sicherstellen,
                                  daß alle Operationen abgeschloßen
                                  wurden */

DeleteTimer(meinTimerRequest);      /* Device schließen, Speicher
                                       freigeben */
```

Listing 6.4: Programmfragment zur Verwendung des Timer Device

Es gibt jedoch noch eine zweite Methode, einen Task schlafen zu lassen: man wartet darauf, daß Signalbits gesetzt werden. Hat ein solches Bit den Wert Eins, dann wurde ein bestimmtes Ereignis (Event) vom System erzeugt. (Informationen zu Signal-Bits finden Sie im Kapitel 3.). Um abzufragen, ob ein Signal-Bit gesetzt wurde, können Sie entweder den IDCMP (w->UserPort) oder den Reply Port für die Timer-Nachrichten verwenden:

```
meinTimerRequest->ReplyPort
```

```
/* multiwakeup.c */

if(magicbits & IDCMPSIGNAL)
   {
     /* IDCMP UserPort auslesen, wie in "event2.c", Kapitel 5,
        gezeigt */
   }

if(magicbits & TIMERSIGNAL)
   {

     GetMsg(meinTimerRequest->ReplyPort);
     timer->SEKUNDEN = timerSeconds;
     timer->MIKROSEK = timerMicros;
     SendIO(meinTimerRequest);

     /* An dieser Stelle erfolgt der Rücksprung zur
        "Wait-Schleife", da zu diesem Zeitpunkt bereits alle
        Messages abgearbeitet wurden */
   }
```

Listing 6.5: Routine zur Abfrage der Signal-Bits an Message Ports

Jeder Port hat ein bestimmtes Signal-Bit, das ihm zugeordnet ist. Die folgenden "#define"-Anweisungen zeigen Ihnen, wie man für jede Art von Port die Signal-Bits abfragen kann:

```
#define IDCMPSIGNAL  w->UserPort->mp_SigBit
#define TIMERSIGNAL  meinTimerRequest->ReplyPort->mp_SigBit
```

Wollen Sie Ihren Task bis zum Eintreffen einer beliebigen Nachricht schlafen lassen, dann verwenden Sie die Funktion Wait anstatt WaitPort. Sie müssen hierzu noch die Kombination der Signal-Bits angeben, die die Art von Ereignissen repräsentieren, bei dessen Eintreffen der Task aufgeweckt werden soll:

```
ULONG magicbits;   /* In dieser Variablen wird der
                      Returnwert von "Wait" (die Bitmaske
                      der gesetzen Signal-Bits) abgelegt */

magicbits = Wait(IDCMPSIGNAL | TIMERSIGNAL);
```

Wenn Ihr Task dann "aufwacht", können Sie den Returnwert "magicbits" auslesen, um festzustellen, welche Signal-Bits gesetzt wurden. In dem Listing 6.5 finden Sie die entsprechende Routine hierzu.

Nach der Verwendung des Timer Device sollten Sie die Routine DeleteTimer aufrufen, damit der von PrepareTimer belegte Speicherplatz wieder freigegeben und das Device geschlossen wird.

Das "Console Device"

Manchmal ist es erforderlich, die Funktionen eines einfachen ASCII-Terminals zu emulieren. Der Amiga stellt diese Möglichkeit zur Verfügung; wenn nötig auch in mehreren Fenstern gleichzeitig. Verwendet wird hierbei das "Console Device", welches viele der ANSI-Sequenzen zur Bildschirmsteuerung beinhaltet. Die Entwickler haben nicht alle ANSI-Sequenzen implementiert; es wurden jedoch einige neue, Amiga-spezifische Sequenzen eingebunden. Im Kapitel 6 des "Amiga ROM Kernel Manual" finden Sie genaue Informationen über die ANSI-Kontrollsequenzen und wie diese vom Console Device verarbeitet werden.

Zur Arbeit mit dem Console Device benötigen Sie ein Fenster, da sich die gesamte Kommunikation zwischen Ihnen und dem Console Device hierüber abspielt. Richten Sie für ein Fenster den IDCMP ein, dann hat dieser gegenüber dem Console Device ein höhere Priorität. Geben Sie als IDCMP-Flags z.B. "VANILLAKEY" oder "RAWKEY" an, dann wird ein Console Device, das ebenfalls mit diesem Fenster arbeiten soll, niemals Tastatureingaben bekommen, d.h. die Kommunikation ist wegen des IDCMP nicht mehr möglich.

Im Listing 6.6 finden Sie einige Routinen, die Sie in eigene Programme einbinden können, um so mit dem Console Device arbeiten zu können. Diese Routinen eröffnen und löschen ein Console Device und übernehmen den Datentransfer, d.h. es werden Zeichen gelesen und versendet. In dem Listing 6.6 wird zur Kommunikation Speicher reserviert und Daten auf eine ganz bestimmte Weise verschickt.

Der Grund hierfür ist, daß Sie beim Öffnen eines Console Device vom System einen Zeiger auf den "IORequest"-Block des Device als Returnwert bekommen (sofern alles gut geht). Sie können auf diese Weise mit einem ganz speziellen Console Device arbeiten, indem Sie einfach diesen Zeiger beim Daten-

transfer verwenden. Mit der Funktion DeleteConsole wird eine Console geschlossen. Dieser Funktion wird ebenfalls der Zeiger auf die Message Blocks des Console Device übergeben.

Die Codes der einzelnen Zeichen

Das Console Device erhält seine Eingaben über verschiedene Stufen der Hard- und Software. Von der Tastatur kommen Codes in Form von "Raw Key"-Daten (jede Taste hat ihren eigenen Raw Key-Code, wobei beim Drücken und Loslassen einer Taste jeweils ein anderer Code erzeugt wird), die vom "Keyboard Device" zusammengefaßt und als Eingabestrom an das "Input Device" weitergeleitet werden. Vom Input Device werden die Daten der Tastatur und des "Gameport Device", welches u.a. die Maus verwaltet, zu einem einzigen Datenstrom komprimiert und schließlich an Intuition übergeben.

Verwaltet Intuition für ein Fenster keinen IDCMP, dann kommen die Daten ungehindert am Console Device an, wo sie dann schließlich in "lesbare" Zeichen oder Steuersequenzen umgewandelt werden.Das Console Device verwaltet zur Interpretation der ankommenden Daten eine "Landkarte" der Tastatur. Diese kann auf vielfältige Art und Weise geändert werden; die voreingestellte "Landkarte" jedoch ist recht einfach: Die Zeichen, die auf der Tastatur zu finden sind, werden beim Drücken einer Taste auch dargestellt.

Die Buchstabentasten erzeugen – je nach Status der Taste <Shift> – den ASCII-Code für große oder kleine Buchstaben. Die Ziffemtasten (auch die des Zehnerblocks) werden vom Console Device in die entsprechenden ASCII-Codes umgewandelt. Gleiches gilt für die Tasten <Return> und <Enter>, die die gleiche Funktion haben, und für <Del>, <Tab> und <Backspace>.

Die Sondertasten, also z.B. die Cursor-, Funktions- und Helptasten, erzeugen mehr als einen Code, wenn Sie gedrückt werden. Die entsprechenden Codes dieser Tasten finden Sie in der Tabelle 6.1. <CSI> ist in dieser Tabelle ein Zeichen mit dem Wert von 9B hexadezimal. Immer, wenn Sie in Ihrem Programm diesen Code erhalten, dann wissen Sie, daß der Anwender eine der Sondertasten gedrückt hat. Beachten Sie bitte, daß die beiden letzten "geshifteten" Werte in Tabelle 6.1 ein <SPACE> nach dem Zeichen <CSI> beinhalten; <SPACE> gehört in diesem Beispiel zur Kontrollsequenz. Das Setzen und Ändern der "Tastaturlandkarte" (Key Map) wird in diesem Buch nicht behandelt. Informationen hierzu finden Sie im "Amiga ROM Kernel Manual".

```
/* consolestuff.c */

struct ConIOBlocks {
struct IOStdReq *writeReq; /* Ein-/Ausgabe-Datenblock (Schreiben) */
struct IOStdReq *readReq;  /* Ein-/Ausgabe-Datenblock (Lesen) */
struct MsgPort *tpr;       /* Zeiger auf den ReplyPort der Console */
};

struct ConIOBlocks *
CreateConsole(window)
struct Window *window;
{
  struct ConIOBlocks *c;
  struct MsgPort *tpw;
  int error;

  c = (struct ConIOBlocks *)AllocMem(sizeof(struct ConIOBlocks),
        MEMF_CLEAR);

  if(!c) goto cleanup1;    /* Nicht genug Speicher */

  tpw = CreatePort(0,0);
  if(!tpw) goto cleanup2;

  c->tpr = CreatePort(0,0);
  if(!c->tpr) goto cleanup3;

  c->writeReq = CreateStdIO(tpw);
  if(!c->writeReq) goto cleanup4;

  c->readReq = CreateStdIO(c->tpr);
  if(!c->readReq) goto cleanup5;

  c->writeReq->io_Data = window;
  c->writeReq->io_Length = sizeof(struct Window);

  error = OpenDevice("console.device",0,c->writeReq,0);
->io_Dev  if(error) goto cleanup6;    /* Console nicht geöffnet */

  c->readReq->io_Device = c->writeReq->io_Device;
  c->readReq->io_Unit = c->writeReq->io_Unit;
  return(c);

cleanup6:
  DeleteStdIO(c->readReq);
cleanup5:
  DeletePort(c->tpr);
```

Listing 6.6: Routinen zum Arbeiten mit dem Console Device (Teil 1)

```
cleanup4:
  DeleteStdIO(c->writeReq);
cleanup3:
  DeletePort(tpw);
cleanup2:
  FreeMem(c,sizeof(struct ConIOBlocks));
cleanup1:
  return(NULL);
}

DeleteConsole(c);
struct ConIOBlocks *c;
{
  struct MsgPort *mp;

  AbortIO(c->readReq);  /* Laufende Leseoperation abbrechen */
  CloseDevice(c->writeReq);   /* Device schließen */

  mp = c->writeReq->io_Message.mn_ReplyPort;

  DeleteStdIO(c->writeReq);
  DeletePort(mp);

  mp = c->readReq->io_Message.mn_ReplyPort;

  DeleteStdIO(c->readReq);
  DeletePort(mp);

  FreeMem(c,sizeof(struct ConIOBlocks));
}

#define CONREAD c->readReq
#define CONWRITE c->writeReq

/* Console soll ein Zeichen einlesen (asynchron) */

EnqueueRead(c,location)
struct ConIOBlocks *c;
char *location;
{
  struct IOStdReq *conr;

  conr = CONREAD;

  conr->io_Command = CMD_READ;
  conr->io_Length = 1;
  conr->io_Data = location;
  SendIO(conr);   /* Asynchrone Ein-/Ausgabe starten */
}
```

Listing 6.6: Routinen zum Arbeiten mit dem Console Device (Teil 2)

```
/* Anzahl Zeichen vom Puffer an ein Console Device senden */

WriteConsole(c,data,length)
struct ConIOBlocks *c;
char *data;
int length;
{
  struct IOStdReq *conw;

  conw = CONWRITE;

  conw->io_Command = CMD_WRITE;
  conw->io_Length = length;
  conw->io_Data = data;
  DoIO(conw);    /* Synchrone Ein-/Ausgabe, Task wartet */
}

CGetCharacter(c,wait)
struct ConIOBlocks *c;
BOOL wait;    /* wait = TRUE: warten auf Eingabe.
                 wait = FALSE: Eingabe als Returnwert liefern.
                 Error = -1 */
{
  struct MsgPort *mp;
  struct IOStdReq *conr;
  char *dataAddr;
  int temp;

  mp = c->tpr;

  if(wait) WaitPort(mp);
  conr = (struct IOStdReq *)GetMsg(mp);

  if(!conr) return(-1);

  else
        {
          dataAddr = conr->io_Data;
          temp = *dataAddr;
          EnqueueRead(c, dataAddr);
          return(temp);
        }
}
```

Listing 6.6: Routinen zum Arbeiten mit dem Console Device (Schluß)

Taste	Erzeugter Code	
	<SHIFT> nicht gedrückt	*<SHIFT> gedrückt*
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~
↑	<CSI>A~	<CSI>T~
↓	<CSI>B~	<CSI>T~
→	<CSI>C~	<CSI> A~
←	<CSI>D~	<CSI> @~

Tabelle 6.1: Die Codes der Sondertasten für das Console Device

Eingaben vom Typ "RAWKEY"

Wenn Sie "RAWKEY" im "IDCMP"-Datenfeld Ihres Fensters setzen, dann hat dieser Port gegenüber dem Console Device eine höhere Priorität; das Console Device erhält vom System keinerlei Nachricht, wenn eine Taste gedrückt oder losgelassen wird. Ihr Programm jedoch wird von Intuition benachrichtigt, wenn die Tastatur vom Anwender benutzt wird. Mittels IDCMP werden Ihnen diese Nachrichten übermittelt, die Sie dann in jeder Weise weiterverarbeiten können.

Bitte beachten Sie, daß Ereignisse vom Typ "RAWKEY" vom System nicht in die ASCII-Äquivalente der Tasten übersetzt werden. Ihr Programm erhält eine Nachricht von Intuition, wenn eine Taste gedrückt oder losgelassen wird. In der Abbildung 6.1 finden Sie die Codes, die jede Taste beim Niederdrücken hervorruft.

ESC 45 | F1 50 | F2 51 | F3 52 | F4 53 | F5 54 | F6 55 | F7 56 | F8 57 | F9 58 | F10 59 | DEL 46
~ ` 00 | ! 1 01 | " 2 02 | § 3 03 | $ 4 04 | % 5 05 | & 6 06 | / 7 07 | ( 8 08 | ) 9 09 | = 0 0A | ? ß 0B | ` ´ 0C | | \ 0D | BACK SPACE 41
TAB 42 | Q 10 | W 11 | E 12 | R 13 | T 14 | Z 15 | U 16 | I 17 | O 18 | P 19 | Ü 1A | * + 1B | RETURN 44 | HELP 5F
CTRL 63 | CAPS LOCK 62 | A 20 | S 21 | D 22 | F 23 | G 24 | H 25 | J 26 | K 27 | L 28 | Ö 29 | Ä 2A | ^ # 2B | ↑ 4C
SHIFT 60 | > < 30 | Y 31 | X 32 | C 33 | V 34 | B 35 | N 36 | M 37 | ; , 38 | : . 39 | _ - 3A | SHIFT 61 | ← 4F | → 4E
ALT 64 | A 66 | 40 | A 67 | ALT 65 | ↓ 4D

7 3D | 8 3E | 9 3F
4 2D | 5 2E | 6 2F
1 1D | 2 1E | 3 1F
0 0F | . 3C
– 4A | ENTER 43

Abb. 6. 1: Die Scan-Codes der Amiga-Tastatur

Erhalten Sie eine IntuiMessage vom Typ RAWKEY (siehe Kapitel 5), dann finden Sie im Datenfeld "Code" der Nachricht den Wert der gedrückten Taste. Die Werte in Abbildung 6.1 gelten für gedrückte Tasten. Wird eine Taste losgelassen, dann wird zu diesen Werten vom System 80 (hex) hinzuaddiert. Das Datenfeld "Qualifier" der IDCMP-Nachricht enthält weiterhin Informationen darüber, ob zusammen mit einer Taste eine Sondertaste (Shift, die Amigatasten, CTRL usw.) gedrückt wurde. In der Include-Datei "devices/inputevent.h" finden Sie Informationen zu den "Qualifier"-Bits.

Kontrollieren des "Console Device"

Das Console Device dient nicht nur zur Ausgabe darstellbarer ASCII-Zeichen. Sie können zusätzlich bestimmte Kontrollsequenzen übermitteln, durch die das Device veranlaßt wird, den Cursor zu positionieren, den Bildschirm zu löschen, die Darstellungsart der Zeichen zu ändern usw. Weiterhin können Sie mit den Funktionen OpenFont und SetFont einen beliebigen Zeichensatz verwenden.

Wenn Sie einen neuen Zeichensatz verwenden wollen, dann reinitialisiert sich das Console Device selbsttätig. Der aktuelle Grafikbereich wird gelöscht und es werden die neuen Werte für die maximale Anzahl von Zeichen pro Zeile bzw. Zeilen pro Bildschirmseite berechnet. Bei der nächsten Textausgabe werden diese Werte dann verwendet.

Um die Ausgabe steuern zu können, müssen Sie bestimmte Kontrollsequenzen an das Console Device übergeben. Diese Sequenzen finden Sie in Tabelle 6.2 sowie eine Erläuterung zu jeder Kontrollsequenz. Sie können diese Sequenzen zusammen mit "normalem" Text übergeben. In der Tabelle sind die Sequenzen als Byte-Werte im Hexadezimalsystem angegeben.

Die Werte "N" und "M" in der Tabelle repräsentieren einen Wert, der von Ihnen angegeben werden muß. Steht dort z.B. ""N" Leerstellen einfügen", dann kann für "N" z.B. die Zahl 12 eingesetzt werden. Dem Console Device wird in diesem Fall folgende Sequenz (in Byte-Werten) übergeben:

9B 31 32 40

In ASCII-Darstellung entspricht dies der Sequenz

<CSI>12@

Der Parameter "werte" in der Tabelle für die Darstellungsart des Textes besagt, daß für ihn mehrere Werte zugleich angegeben werden können. Alle Angaben sind optional. Werden jedoch mehrere Werte zugleich übergeben, dann müssen sie durch ein Semikolon (3B hex) getrennt werden. Nachfolgend die verfügbaren Darstellungsarten von Text (hex):

00 Standard
01 Fettschrift
03 Kursiv
04 Unterstrichen
07 Invers

Weiterhin können Sie die Textfarbe (Vorder- und Hintergrund) ändern. Hierzu stehen die ersten acht Farbregister (0 bis 7) des Systems zur Verfügung. Der Textvordergrund wird durch die Angabe eines Wertes von 30 bis 37 (hex) geändert, entsprechend den Farbregistern 0 bis 7. Der Hintergrund des Textes wird auf die gleiche Weise selektiert; hier kommen jedoch die Werte 40 bis 47 (hex) zum Einsatz.

Wie erwähnt, können Sie mehrere Optionen in einer Kommandosequenz übergeben. Wenn Sie z.B. Text in kursiver Fettschrift mit Vordergrundfarbe 2 und Hintergrundfarbe 3 ausgeben wollen, dann übergeben Sie die folgende Sequenz:

9B 01 3B 03 3B 32 3B 43 3B 6D

Befehl	Übergebene Byte-Sequenz
Backspace (destruktiv)	08
Line Feed	0A
Vertikaler Tabulator	0B
Seitenvorschub (Form Feed)	0C
Return	0D
Shift In	0E
Shift Out	0F
Escape	1B
<CSI>	9B
Reset	9B 63
<N> Leerzeichen einfügen	9B <N> 40
Cursor <N> Zeilen hoch	9B <N> 41
Cursor <N> Zeilen runter	9B <N> 42
Cursor <N> Zeichen links	9B <N> 43
Cursor <N> Zeichen rechts	9B <N> 44
Cursor zur folgenden Zeile <N>	9B <N> 45
Cursor zur vorigen Zeile <N>	9B <N> 46
Cursor nach Zeile, Spalte	9B <M>3B<N>
Fenster ab Cursor löschen	9B 4A
Bis Zeilenende löschen	9B 4B
Zeile oberhalb Cursor einfügen	9B 4C
Aktuelle Zeile löschen	9B 4D
<N> Zeichen löschen	9B <N> 50
<N> Zeilen hochscrollen	9B <N> 53
<N> Zeilen runterscrollen	9B <N> 54
Modus setzen (LF = CR+LF)	9B 32 30 68
Modus rücksetzen	9B 32 30 6C
Seitenlänge setzen	9B <N> 74
Zeilenlänge setzen	9B <N> 75
Linken Rand setzen	9B <N> 78
Oberen Rand setzen	9B <N> 79
Text-Darstellungsart setzen	9B <werte> 6D
Device Statusbericht	9B 6E
Fenster Statusbericht	9B 71

Tab. 6.2: Kommandosequenzen zum Steuern des Console Device

Vom Console Device erhalten Sie dann einen "Device-Statusbericht", der Auskunft über die aktuelle Cursorposition gibt. Diese Sequenz hat folgende Form:

9B zeile 3B spalte 52

Der Parameter "zeile" ist dabei die Zahl, die die aktuelle Zeile spezifiziert, in der sich der Cursor befindet; "spalte" gibt an, in welcher Spalte des Fensters sich der Cursor befindet. Diese Werte bestehen entweder aus einer oder zwei Ziffern (30 – 39 hex). Befindet sich der Cursor z.B. in der linken oberen Ecke des Fenster, so erhalten Sie (1/1) als Koordinaten.

Der "Fenster-Statusreport" gibt Auskunft darüber, wie viele Spalten und Zeilen mit dem aktuellen Zeichensatz verwendet werden können. Sie erhalten diese Werte ebenfalls als Byte-Sequenz folgender Form:

9B 31 3B 31 3B zeilen 3B spalten 73

Die Parameter werden ebenfalls wieder durch eine oder mehrere Ziffern dargestellt. Das Beispielprogramm in Listing 6.7 verwendet die zuvor besprochenen Routinen, um einige der Möglichkeiten des Console Devices zu zeigen, z.B. Ein-/Ausgabe und die Steuerung durch Kommandosequenzen.

```
/* conmain.c */

/* Es können mehrere Console Devices verwaltet werden, indem man
   jedes über den Zeiger anspricht, der von CreateConsole als
   Returnwert geliefert wird.
   EnqueueRead beinhaltet die Position, an die Daten vom aktuellen
   Console Device gespeichert werden. */

#include "exec/types.h"
#include "exec/memory.h"
#include "intuition/intuition.h"

ULONG IntuitionBase;
```

Listing 6.7: Beispielprogramm zum Arbeiten mit dem Console Device (Teil 1)

```
struct NewWindow nw = {
10,10,300,100,-1,-1,0,GIMMEZEROZERO|ACTIVATE|
SIMPLE_REFRESH|WINDOWDRAG,0,NULL,"Console Fenster",NULL,
NULL,0,0,0,0,WBENCHSCREEN };

char homecursor[] = { 0x9b, '1', 0x3b, '1', 0x48 };
char backspace[] = { 0x08 };
char linefeed[] = { 0x0a };
char carreturn[] = { 0x0d };
char cursorfwd[] = { 0x9b, 0x43 };
char formfeed[] = { 0x0c };
char insertchar[] = { 0x9b, 0x40 };
char deletechar[] = { 0x9b, 0x50 };

#define HOMECURSOR(c) WriteConsole(c,homecursor,5);
#define BACKSPACE(c) WriteConsole(c,backspace,1);
#define LINEFEED(c) WriteConsole(c,linefeed,1);
#define CARRETURN(c) WriteConsole(c,carreturn,1);
#define CURSORFWD(c) WriteConsole(c,cursorfwd,2);
#define FORMFEED(c) WriteConsole(c,formfeed,1);
#define INSERTCHAR(c) WriteConsole(c,insertchar,2);
#define DELETECHAR(c) WriteConsole(c,deletechar,2);

#include "ram:console.c"

main()
{
  struct ConIOBlocks *cio,CreateConsole();
  struct Window *w;
  int i, myinput;
  char mybuffer[], mychar[];

  IntuitionBase = OpenLibrary("intuition.library",0);
  if(!IntuitionBase)
     {
       printf("intuition.library nicht geöffnet!\n");
       exit(0);
     }

  if(!(w = OpenWindow(&nw)))
     {
       printf("Fenster nicht geöffnet!\n");
       goto finish1;
     }
```

Listing 6.7: Beispielprogramm zum Arbeiten mit dem Console Device (Teil 2)

```
cio = CreateConsole(w);
if(!cio)
   {
     printf("Console Device nicht geöffnet!\n");
     goto finish2;
    }

EnqueueRead(cio,mybuffer);    /* Lesezugriff initialisieren */

WriteConsole(cio, "Hello Magic\n\r",13);
WriteConsole(cio,"Test: Backspace",15);

for(i=0;i<15;i++)
     {
       BACKSPACE(cio);
       Delay(25);
      }

LINEFEED(cio);
CARRETURN(cio);

WriteConsole(cio,"Test: Cursor vorwärts",20);
CARRETURN(cio);


for(i=0;i<20;i++)
     {
       CURSORFWD(cio);
       Delay(25);
     }

LINEFEED(cio);
CARRETURN(cio);

WriteConsole(cio,"Test: Zeichen einfügen",22);
CARRETURN(cio);

for(i=0;i<22;i++)
     {
       INSERTCHAR(cio);
       Delay(25);
     }

LINEFEED(cio);
CARRETURN(cio);
```

Listing 6.7: Beispielprogramm zum Arbeiten mit dem Console Device (Teil 3)

```
   WriteConsole(cio,"*********Test: Zeichen löschen",30);
   CARRETURN(cio);

   for(i=0;i<8;i++)
        {
          DELETECHAR(cio);
          Delay(25);
         }

   LINEFEED(cio);

   WriteConsole(cio,"Test: Cursor HOME",17);
   Delay(50);     /* 1 Sekunde warten */

   HOMECURSOR(cio);
   Delay(100);

   FORMFEED(cio);
   LINEFEED(cio);

   WriteConsole(cio,"LineFeed-Bildschirm gelöscht!\n\r",27);
   WriteConsole(cio,"Bitte eine Zeile eingeben...\n\r",25);
   WriteConsole(cio,"Zeichen werden geechoet bis\n\r",28);
   WriteConsole(cio,"Sie <RETURN> drücken\n\r",24);

       /* Die Funktionstasten und <HELP> erzeugen kein Echo */

   do
       {
       /* Es wird hier nur zeichenweise eingelesen. Durch einen
          zweiten Task könnten die Zeichen zwischengespeichert
          werden, die der Anwender eingibt, während das
          Console Device arbeitet. */

       myinput = CGetCharacter(cio,TRUE);    /* Auf Zeichen warten */
       mychar[0] = (myinput & 0xff);
       WriteConsole(cio,mychar,1);
       }
       while(mychar[0] != '\r');

   finish3: DeleteConsole(cio);
   finish2: CloseWindow(w);
   finish1: CloseLibrary(IntuitionBase);

   exit(0);
 }
```

Listing 6.7: Beispielprogramm zum Arbeiten mit dem Console Device (Schluß)

Das "Input Device"

Dieses Device kombiniert die Eingaben von der Tastatur und durch die Maus zu einem einzigen Datenstrom. Das Herausnehmen und Einlegen einer Diskette wird vom Input Device ebenso als Nachricht definiert wie Ereignisse der Tastatur und der Maus.

Die Tastatur

Die Daten der Tastatur werden ausschließlich an das Input Device übergeben. Unter den aktuellen Versionen von Exec, Intuition und AmigaDOS ist es nicht möglich, das Input Device abzuschalten, um so einen exklusiven Zugriff auf das "Keyboard Device" zu erhalten.

Wenn Ihr Programm ein Fenster verwaltet und Sie Tastatureingaben verarbeiten wollen, dann sollten Sie hierzu den IDCMP verwenden. Ist kein Fenster geöffnet, dann sollten Sie das Input Device von Ihrem Programm aus einbinden, um Tastatureingaben empfangen zu können.

Der Joystick-/Mausport

Dieser Port, der durch das "Gameport Device" verwaltet wird, übergibt seine Daten ebenfalls exklusiv an das Input Device. Diese Daten sind dann mit Hilfe von Intuition bzw. direkt über das Input Device verfügbar. Hierdurch haben Sie eine Reihe von Möglichkeiten. Sie können die Maus z.B. in den zweiten Port einstecken und abfragen, anstatt den ersten der beiden Ports zu verwenden. Oder Sie benutzen das Input Device, um Joysticks abzufragen.

Wie bei allen Devices, so wird auch beim Input Device ein "IORequest"-Block zur Kommunikation verwendet. Zusätzlich wird dem Device mitgeteilt, welche Art Eingabegerät am Port vorhanden ist (Maus oder Joystick) und wie mit ihm kommuniziert werden soll. Das Programm in Listing 6.8 kann für beide Eingabegeräte kompiliert werden; voreingestellt ist die Maus. Wenn Sie einen Joystick verwenden wollen, ersetzen Sie einfach die Zeile "#define MOUSE 1" durch "#define JOYSTICK 1".

```
/* joymouse.c */

/* Es kann immer nur ein Gerät (Maus oder Joystick) angegeben werden! */

#define MOUSE 1

#include "exec/types.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "devices/gameport.h"
#include "devices/inputevent.h"

#define abs(x) (x<0 ? -x : x)

main()
{
  int error,errout,delta,timeouts;
  struct GamePortTrigger trig;
  struct IOStdReq *gameMessage;    /* Ein-/Ausgabeblock */
  struct MsgPort *gameReplyPort;
  struct InputEvent gameEvent;
  struct InputEvent *ge;
  UBYTE *g;

  ge = &gameEvent;

  gameReplyPort = CreatePort(0,0);
  if(!gameReplyPort) exit(0);    /* Kein MsgPort kreiert */

  gameMessage = CreateStdIO(gameReplyPort);
  if(gameMessage)
     {
      DeletePort(gameReplyPort);
      exit(0);     /* Keine Message kreiert */
     }

  error = OpenDevice("gameport.device",1,gameMessage,0);
  /* Der Wert 0 entspricht dem linken, der Wert 1 dem rechten Mausport */
  if(error)
     {
      errout = 102;
      goto cleanup;
     }
```

Listing 6.8: Beispiel zur Abfrage des Maus-/Joystick-Ports (Teil 1)

```
  gameMessage->io_Command = GPD_SETCTYPE;
  gameMessage->io_Length = 1;
  gameMessage->io_Data = &gameEvent;

  g = &gameEvent;

#ifdef MOUSE
  *g = GPCT_MOUSE;
  delta = 5;     /* Schrittweite */
#endif MOUSE

#ifdef JOYSTICK
  *g = GPCT_ABSJOYSTICK;
  delta = 1;    /* Schrittweite */
#endif JOYSTICK

  DoIO(gameMessage);

  if(gameMessage->io_Error)
     {
       errout = 103;
       goto cleanup;    /* Fehler während der Initialisierung */
     }

  gameMessage->io_Command = GPD_SETTRIGGER;
  gameMessage->io_Length = sizeof(trig);
  gameMessage->io_Data = &trig;

  /* Message senden, wenn Feuer- oder Mausknopf gedrückt wird */
  trig.gpt_Keys = GPTF_UPKEYS + GPTF_DOWNKEYS;

  /* Timeout setzen für Eingaben aller Art */
  trig.gpd_Timeout = 10 * 60;

  /* Message senden, wenn <delta> größer oder gleich den folgenden
     Werten ist */

  trig.gpd_XDelta = delta;
  trig.gpd_YDelta = delta;

  DoIO(gameMessage);
  if(gameMessage->io_Error)
     {
       errout = 104;
       goto cleanup;
      }
```

Listing 6.8: Beispiel zur Abfrage des Maus-/Joystick-Ports (Teil 2)

```
  timeouts = 0;
  gameMessage->io_Command = GPD_READEVENT;
  gameMessage->io_Data = &gameEvent;

  do
      {
        gameMessage->io_Length = sizeof(struct InputEvent);
        DoIO(gameMessage);

        switch(ge->ie_Code)
                    {
                      case IECODE_LBUTTON:
                      printf("Linke Maustaste gedrückt!\n");
                      break;

                      case IECODE_LBUTTON | IE_UP_PREFIX:
                      printf("Linke Maustaste losgelassen!\n");
                      break;

                      case IECODE_RBUTTON:
                      printf("Rechte Maustaste gedrückt!\n");
                      break;

                      case IECODE_LBUTTON | IE_UP_PREFIX:
                      printf("Rechte Maustaste losgelassen!\n");
                      break;

                      case IECODE_NOBUTTON:
                      if(abs(gameEvent.ie_X) < delta &&
                           abs(gameEvent.ie_Y) < delta)
                      {
                        printf("Timeout...\n");
                        timeouts++;
                      }

                      else printf("Maus wurde bewegt!\n");

#ifdef MOUSE
                      printf("Mausbewegungen:\n");
#endif MOUSE

#ifdef JOYSTICK
                      printf("Joystickbewegungen:\n");
#endif JOYSTICK
```

Listing 6.8: Beispiel zur Abfrage des Maus-/Joystick-Ports (Teil 3)

```
                printf("x-delta = %ld\n",gameEvent.ie_X);
                printf("y-delta = %ld\n",gameEvent.ie_Y);

                default : break;
               }
  }

  while(timeouts<10);
  errout = 0;
  /* Wenn ein Joystick verwendet wird, dann kann dieses Programm nicht
     unterscheiden, ob ein Timeout vorliegt oder ein Knopf gedrückt
     wurde */

cleanup:
  DeleteStdIO(gameMessage);
  DeletePort(gameReplyPort);
  printf("Fertig.\n");
  exit(errout);
}
```

Listing 6.8: Beispiel zur Abfrage des Maus-/Joystick-Ports (Schluß)

Tastatur-Spezialitäten

Einige Entwickler schreiben Programme, die die Tasten der Tastatur anders belegen. Um Keyboard-Events in andere zu übertragen, können Sie alle Eingaben durch Console-Devices oder IDCMP filtern und die Sequenzen anpassen, bevor sie Ihre Applikation erreichen. Wenn Sie als Entwickler etwas erarbeiten wollen, was nicht nur mit Ihrer eigenen, sondern mit jeder Applikation arbeitet, dann müssen Sie sich intensiv mit dem System beschäftigen. Sie möchten dann möglicherweise Ihren eigenen Input-Handler schreiben, um ihn in die Kette der Input-Handler von Input-Devices einzufügen.

In dieser Kette gibt es verschiedene Prioritäten, so daß Intuition, wenn es den Event als erstes erhält, ihn bearbeiten kann und niemals an Ihren Handler weitergibt. Daher möchten Sie möglicherweise Ihren Handler in der Kette vor Intuition anordnen, so daß Sie alle eingehenden Events überprüfen können und eine neue Kette von Events erzeugen, die dann von Intuition verarbeitet werden.

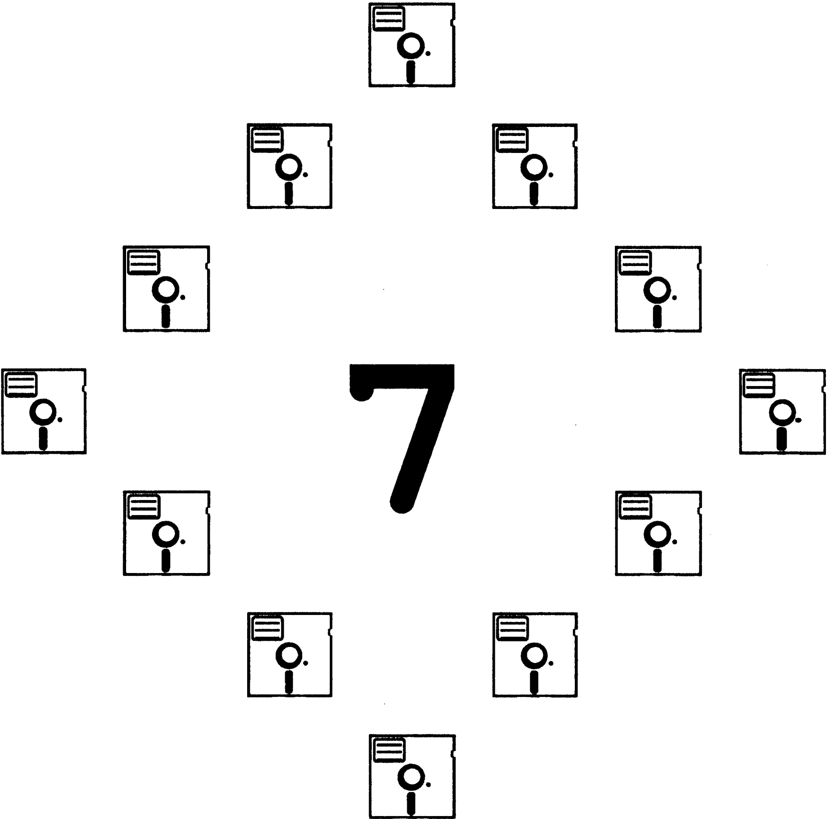
Auf diese Weise können Sie ein Programm schreiben, das sich an alle Eingaben des Anwenders erinnert. Sie können auch einen Handler erzeugen, der eine aufgenommene Serie von Mausbewegungen wiederholen kann oder Sie schreiben ein Programm, das Tastatur-Makros zu Ihrer Bequemlichkeit verwendet.

Ihre erste Überlegung sollte sein, daß jeder Input-Event, den Sie aufzeichnen, bis zu 24 Byte groß sein kann (das ist die Größe einer "InputEvent"-Datenstruktur). Keyboard-Events werden als Tastenbewegung an Ihre Applikation gemeldet (Drücken oder Loslassen der Taste). Innerhalb einer Applikation können Sie beispielsweise "InputEvent"-Speicher wiederverwenden, z.B. indem von jeder Eingabe nur ein oder zwei Events berücksichtigt werden. Für ein Beispiel mit voller Aufnahme der Events werden Sie aber alles haarklein registriert wissen wollen.

Es gibt noch eine andere interessante Überlegung in bezug auf die Verwendung von Input-Devices für die Auswertung von Tastatur-Eingaben. Was passiert, wenn ein Anwender mit mehreren Applikationen gleichzeitig kommunizieren will? Was passiert, wenn er ein oder mehrere Zeichen in eine Applikation eingibt, sich dann einer anderen zuwendet und weitertippt?

Wenn Sie einen Expander für Abkürzungen erzeugen wollen, dann muß Ihr Expander-Programm in der Lage sein, zu überprüfen, ob eine Applikation alle Zeichen des Eingabe-Stroms erhalten hat. Nehmen Sie z.B. an, daß es zwei Fenster auf dem Bildschirm gibt, die Text verarbeiten und ein Abkürzungs-Expander die Tastatur überwacht, der "mult" in "Multiplikation" übersetzt. Wenn der Anwender im ersten Fenster "mu" eingibt, dann das zweite anwählt und "lt" eingibt, werden Sie es vermeiden wollen, das Wort im zweiten Fenster erscheinen zu lassen (da das zweite Fenster die Abkürzung "mult" nicht erhalten hat). Sie werden dem Anwender mitteilen müssen, daß Abkürzungen nur dann korrekt in die gewünschten Wörter verwandelt werden können, wenn kein Mausklick diese Operation stört.

Die Verwendung der Input-Devices, bedingt nicht nur fundierte Kenntnisse in C, sondern auch in der Maschinensprache des Amigas. Bitte sehen Sie im "Amiga ROM Kernel Manual" nach, um mehr Informationen über die Erstellung von Input-Handlern zu finden. Das Manual enthält ein sehr gutes Beispiel dafür.


7

Kapitel 7

Animation – Bewegte Grafik

In diesem Kapitel beschäftigen wir uns mit den Möglichkeiten des Amiga, bewegte Grafiken zu erzeugen. Dem Programmierer bietet das System zwei Hilfsmittel: das "Simple-Sprite-System" und das "Gel-System". Das Gel-System (Graphics Element, grafische Elemente) ist aus zwei Komponenten aufgebaut: den "Virtual-Sprites" und den "Bobs" (Blitter Objects).

Bei der Animation durch das Gel-System wird das zu bewegende Objekt und seine Bewegung relativ zu anderen Objekten auf dem Bildschirm definiert. Simple-Sprites werden beim Gel-System durch "Hardware-Sprites" erzeugt; die Darstellung von Bobs übernimmt der Blitter – einer der Co-Prozessoren des Amiga, mit dem große Datenblöcke sehr schnell umkopiert werden können. Objekte werden in der von Ihnen bestimmten Abfolge und in den angegebenen Positionen dargestellt, wobei der Teil des Hintergrundes, der von einem Objekt überdeckt wird, vom System zwischengespeichert und später wiederhergestellt wird, wenn das Objekt eine neue Position einnimmt.

Das Gel-System erlaubt dem Programmierer eine große Flexibilität bei der Programmierung von bewegten Grafiken. In diesem Kapitel werden alle Möglichkeiten des Gel-Systems behandelt; einige von ihnen werden weiterhin in den Beispielprogrammen verwendet, um so die Grundlagen für eigene Programme zu legen.

Simple-Sprites

Verwenden Sie zur Animation Ihrer Grafik "Simple-Sprites", dann können Sie maximal sieben dieser Objekte gleichzeitig darstellen und bewegen lassen. Die Breite eines solchen Simple-Sprites beträgt 16 Pixel (Lo-Res), zur Farbgebung

stehen Ihnen maximal drei Farben zur Verfügung. Simple-Sprites können sehr schnell auf dem Bildschirm bewegt werden (der Mauszeiger ist z.B. ein Simple-Sprite). Simple-Sprites werden vom System durch direkte Zugriffe auf Hardware-Bausteine des Amiga erzeugt – Zugriffe auf das "Hardware-Sprite-System".

Die Darstellung von Simple-Sprites ist von der des Hintergrundes (der in diesem Zusammenhang "Playfield" heißt) unabhängig, lediglich bei der Auswahl der Farben stehen diese beiden Komponenten in Beziehung. Es gibt viele Anwendungen für Simple-Sprites. So kann z.B. der Mauszeiger beim Bewegen geändert werden, oder man verwendet Simple-Sprites innerhalb von Spielen, um kleine flexible Grafiken zu erzeugen.

Der Amiga verwaltet acht Simple-Sprites, von denen jedes mit den entsprechenden Hardware-Komponenten korrespondiert. Der Unterschied zwischen einem Simple-Sprite und einem Hardware-Sprite besteht darin, daß die System-Software pro Bildaufbau durch den Rasterstrahl nur eine Verwendung eines Hardware-Sprites durch ein Simple-Sprite zuläßt. Wenn man jedoch die Register der Sprite-Hardware direkt manipuliert, dann können Hardware-Sprites mehrfach verwendet werden. Simple-Sprites nutzen diese Möglichkeit nicht, was jedoch kein Nachteil sein muß, da man so jederzeit bestimmen kann, an welcher Stelle des Bildschirms sich ein Simple-Sprite befindet und auf welche Art und Weise es dargestellt wird.

Die "SimpleSprite"-Datenstruktur

Für jedes Sprite, das Sie in einem Programm verwenden wollen, müssen Sie eine "SimpleSprite"-Datenstruktur definieren. Diese Struktur (sie befindet sich in der Datei "graphics/sprite.h" auf Ihrer Compiler-Diskette) sieht wie folgt aus:

```
struct SimpleSprite
{
  UWORD *posctldata;
  UWORD height;
  UWORD x,y;
  UWORD num;
};
```

Die Initialisierung der einzelnen Datenfelder wird durch den Aufruf von Systemroutinen (ChangeSprite, MoveSprite) übernommen. Durch Strukturver-

weise können die Datenfelder ausgelesen werden, um so z.B. die Position eines Sprites auf dem Bildschirm zu bestimmen oder um festzustellen, welches Hardware-Sprite gerade vom System verwendet wird.

Jedes Simple-Sprite, das auf dem Bildschirm dargestellt wird, ist immer 16 Bildpunkte breit. Es spielt keine Rolle, ob der zugehörige Screen im Lo-Res-Modus (320 mal 200 Pixel) oder im Med-Res-Modus (640 mal 200) geöffnet wurde; die Breite des Sprites beträgt immer 16 Pixel im Lo-Res-Format.

Die Höhe eines Simple-Sprites wird im Datenfeld "Height" der SimpleSprite-Datenstruktur abgelegt, seine Bildschirmposition ist in den Variablen "x" und "y" der Struktur angegeben. Die Darstellungsweise eines Sprites wird durch den Speicherbereich bestimmt, auf den der Zeiger "posctldata" weist.

Reservieren eines Sprites

Bevor Sie Datenstrukturen für Sprites anlegen können, müssen Sie zunächst einmal vom System ein Sprite anfordern, welches Sie dann weiterverwenden können. Mit Hilfe der Funktion GetSprite können Sie eine solche Reservierung vornehmen. Der Aufruf der Funktion lautet:

```
spritenummer = GetSprite(ssp,nr);
```

Der Funktion werden beim Aufruf zwei Parameter übergeben. "ssp" ist ein Zeiger auf eine SimpleSprite-Datenstruktur, "nr" ist die Nummer des Sprites, das Sie reservieren wollen (0 bis 7).

Liefert die Funktion als Return-Wert –1, dann wird das Sprite bereits von einem anderen Task verwendet. In allen anderen Fällen entspricht der Return-Wert "spritenummer" dem Parameter "nr", d.h. das gewünschte Sprite wurde vom System für Ihren Task reserviert.

Ist die Nummer des Sprites für Sie nicht relevant, dann kann der Parameter "nr" auch den Wert –1 annehmen; Sie teilen dem System auf diese Weise mit, daß irgendein unbenutztes Sprite reserviert werden soll. Sind bereits alle acht Sprites belegt, dann erhalten Sie auch hier wieder –1 als Return-Wert. Nachfolgend finden Sie ein Programmfragment, das eine solche Reservierung vornimmt.

```
struct SimpleSprite *MeinSprite;
BYTE spritenummer;

Datenspritenummer = GetSprite(&MeinSprite,-1);

if(spritenummer == -1)    /* Alle Sprites bereits belegt */
   printf("Alle Sprites belegt!");
```

Nach dem erfolgreichen Aufruf der Funktion GetSprite wird vom System das Datenfeld "num" in der "SimpleSprite"-Datenstruktur initialisiert. Der Return-Wert der Funktion dient nur als Indikator für einen erfolgreichen Funktionsaufruf; später kann dieser Wert jedoch zur Farbänderung des Sprites verwendet werden.

Ändern eines Sprites

Durch einen Aufruf der Funktion ChangeSprite werden weitere Datenfelder der "SimpleSprite"-Datenstruktur initialisiert. Aufgerufen wird die Funktion wie folgt:

```
ChangeSprite(zeiger,adrsprite,adrdaten);
```

Der erste Parameter der Funktion – "zeiger" – ist ein Zeiger auf eine "ViewPort"-Datenstruktur bzw. ein Null-Zeiger. Übergeben Sie der Funktion einen Zeiger auf einen ViewPort, dann werden die Sprites relativ zu seiner linken oberen Ecke positioniert. Im "SimpleSprite"-Beispielprogramm können Sie sehen, wie die Adresse eines ViewPorts – die ja Bestandteil der "Screen"-Datenstruktur ist – aus dieser extrahiert werden kann. Wenn Sie Ihren Screen (und somit auch den ViewPort) durch Intuition initialisieren lassen, dann werden Sprites beim Verschieben eines Screens erst dann neu positioniert, wenn der Screen wieder "stillsteht". (Wenn Sie das Beispielprogramm ausprobieren, werden Sie diesen Effekt bemerken.)

Enthält der Parameter "zeiger" den Wert Null, dann positioniert das System die Sprites in bezug auf die View-Datenstruktur. Diese Struktur spiegelt die Gesamtheit des sichtbaren Bildschirmbereichs wider; es sind also Screens und Fenster zusammengefaßt. Die Positionierung von Sprites erfolgt relativ zur linken oberen Ecke des sichtbaren Bildschirmbereichs; die Sprites können sich

auf diese Weise auch außerhalb "ihres" Screens bewegen. Dies hat einen interessanten Effekt zur Folge, da die Sprites ihre Farben entsprechend des Screens ändern, über der sie sich gerade befinden.

Der zweite Funktionsparameter – "adrsprite" – ist wiederum ein Zeiger auf eine "SimpleSprite"-Datenstruktur. Diese Struktur sollte vor dem Aufruf von ChangeSprite bereits durch die Funktion GetSprite initialisiert worden sein, damit das Datenfeld "num" eine gültige Sprite-Nummer enthält.

Der letzte Parameter ist ein Zeiger auf den Speicherbereich, in dem die Daten abgelegt sind, die das Aussehen des Sprites definieren.

Die Daten eines Sprites

Nachfolgend finden Sie eine Datenstruktur, durch die das Aussehen eines Sprites festgelegt wird. Der Zeiger auf diese Struktur wird der Funktion ChangeSprite als dritter Parameter übergeben. Eine solche Datenstruktur wie die folgende hat keinen bestimmten Namen; sie steht jedoch immer in Beziehung zur "SimpleSprite"-Datenstruktur.

```
/* spritedata.c */

UWORD sprite_data[] = {
   0,0,                /* Position */
   0x0fc3, 0x0000,     /* Oberste Zeile (Nr. 1) des Sprites */
   0x3ff3, 0x0000,
   0x30c3, 0x0000,
   0x0000, 0x3c03,
   0x0000, 0x3fc3,
   0x0000, 0x03c3,
   0xc033, 0xc033,
   0xffc0, 0xffc0,
   0x3f03, 0x3f03,     /* Unterste Zeile (Nr.9) des Sprites */
   0,0                 /* Ende der Struktur */
};
```

Datenstrukturen, die das Aussehen eines Simple-Sprite definieren, haben immer die oben gezeigte Form. Den Anfang der Struktur bilden zwei WORDs (32 Bit) mit dem Wert Null, gefolgt von WORD-Paaren, die in ihrer Gesamtheit die Höhe des jeweiligen Sprites festlegen. Das Ende der Struktur wird ebenfalls wieder durch zwei WORDs mit dem Wert Null gebildet.

Die ersten beiden Werte der Datenstruktur dienen zur Positionskontrolle. Bei einem Hardware-Sprite geben diese Werte an, wo das Sprite auf dem Screen dargestellt wird. (Die Werte werden vom System noch für andere Zwecke verwendet, die hier jedoch nicht näher erläutert werden.) Die beiden letzten Werte werden ebenfalls zur Positionskontrolle verwendet, jedoch wird durch sie bestimmt, ob sich ein Sprite noch innerhalb seines Screens befindet. Die oben gezeigte Datenstruktur erzeugt ein Sprite mit folgendem Aussehen:

```
S!
```

Die Daten, die das Aussehen eines Sprites bestimmen, müssen sich im Chip-Memory befinden, d.h. sie müssen in den untersten 512 KB Speicher des Systems abgelegt werden, damit die Co-Prozessoren auf sie zugreifen können. Das Simple-Sprite-Beispielprogramm verwendet die oben gezeigte Datenstruktur für alle verwendeten Sprites. Das Programm reserviert Speicher vom Typ "MEMF_CHIP" und kopiert anschließend die Sprite-Daten in diesen Bereich.

Ein Aufruf der Funktion ChangeSprite sagt dem System, wo die Daten für jedes Sprite zu finden sind; in unserem Fall weisen alle Zeiger auf dieselbe Datenstruktur. Weiterhin muß für jedes Sprite zusätzlich ein kleiner Speicherbereich reserviert werden, da das System die Werte der Positionskontrolle in der "SimpleSprite"-Datenstruktur ändert und dem Hardware-Sprite dann mitteilt, wo diese Daten zu finden sind.

Die Farben eines Sprites

Die in der Datenstruktur verwendeten Daten-Wortpaare geben die Farbe an, die jedes Pixel innerhalb der Sprite-Zeile haben soll. Zur Veranschaulichung nehmen wir Zeile eins der Datenstruktur als Beispiel:

```
0x0fc3, 0x0000
```

Der hexadezimale Wert 0x0fc3 sieht im Binärformat wie folgt aus:

0000111111000011

Der Wert 0x0000 ergibt in binärer Schreibweise folgendes Bild:

0000000000000000

Bits werden in den jeweiligen Positionen kombiniert, um auf diese Weise die Farbe zu bestimmen, die ein Pixel an der entsprechenden Position der Sprite-Zeile erhalten soll. Die Kombinationen ergeben sich wie folgt:

Wert des ersten Datenwortes:	0 0 1 1
Wert des zweiten Datenwortes:	0 1 0 1
Resultierende Farbe:	t 1 2 3

Die Farbe "t" bedeutet, daß ein Pixel die Farbe des Hintergrundes annimmt, d.h. das Sprite ist an solchen Stellen "durchsichtig". Die Farbregister werden vom System immer für zwei Sprites vergeben: Die Sprites 0 und 1 greifen auf dieselben Register zu; gleiches gilt für die Sprites 2 und 3, 4 und 5, 6 und 7. Jedes Paar kann auf maximal drei Farbregister zugreifen, d.h. jedes Sprite kann aus bis zu vier Farben bestehen, da ja auch die Hintergrundfarbe gewählt werden kann.

In der Tab. 7.1 finden Sie die für jedes Sprite-Paar zugeteilten Nummern der Farbregister.

Sprites	Farbe Nr.	Farbregister Nr.
0 und 1	1	17
	2	18
	3	19
2 und 3	1	21
	2	22
	3	23
4 und 5	1	25
	2	26
	3	27
6 und 7	1	29
	2	30
	3	31

Tab 7.1: Die Farbregister der Simple-Sprites

Bei den Simple-Sprites stehen Ihnen vier verschiedene Farbpaletten zur Verfügung, die jedoch unter den Sprites aufgeteilt sind. Intuition verwendet das Simple-Sprite 0 zur Darstellung des Mauszeigers; eine Reservierung dieses Sprites für Ihren Task ist daher normalerweise nicht möglich. Wenn Sie die Funktion LoadRGB4 verwenden, um für einen Screen eine eigene Farbpalette zu definieren, dann werden jedoch auch die Farben des Mauszeigers beeinflußt, wenn Sie die Farbregister 17 bis 19 mit neuen Werten initialisieren.

Im Beispielprogramm zu den Simple-Sprites tritt dieser Effekt ebenfalls auf. Eine Hälfte der Sprites wird relativ zur linken oberen Ecke des ViewPorts positioniert (d.h. relativ zum Screen, in dem das Fenster geöffnet wird), die andere Hälfte hingegen relativ zur linken oberen Ecke des gesamten Darstellungsbereiches, d.h. unabhängig von Screens oder Fenstern.

Wenn Sie nun mit dem Mauszeiger den neu geöffneten Screen herunterziehen, dann werden die Sprites, die relativ zum ViewPort positioniert wurden, vom System an anderer Stelle dargestellt (nämlich relativ zur neuen Position der linken oberen Ecke des Screens). Die anderen Sprites bleiben an ihrem alten Platz, ändern jedoch ihre Farben, sobald sie sich nicht mehr über "ihrem" Screen befinden. Sie werden feststellen, daß die Farben der Sprites direkt geändert werden. Das liegt daran, daß Intuition die Inhalte der Farbregister dynamisch ändert; auf diese Weise kann jeder Screen seine eigene Farbpalette verwalten. Bewegen sich die Sprites über die Grenzen "ihres" Screens hinaus, dann nehmen sie automatisch die Farben des darunterliegenden Screens an.

Freigabe eines Sprites

Wenn Sie mit der Funktion GetSprite ein Sprite für Ihren Task reserviert haben, dann müssen Sie es durch einen Aufruf der Funktion FreeSprite wieder freigeben, wenn Sie es nicht mehr benötigen. Beenden Sie Ihr Programm, ohne ein reserviertes Sprite freizugeben, dann kann kein anderer Task auf dieses Sprite zugreifen, da das System nicht weiß, welcher Task welches Sprite reserviert hat und ob dieser Task überhaupt noch läuft.

Ein Aufruf der Funktion FreeSprite lautet wie folgt:

```
FreeSprite(nr);
```

Der Parameter "nr" ist die Nummer des Sprites, das freigegeben werden soll.

Beispielprogramm: SimpleSprite

Das Listing 7.1 enthält ein Beispielprogramm, mit dem die Anwendung von Simple-Sprites veranschaulicht werden soll. Es wird ein 32-farbiger Screen geöffnet, der ein Fenster enthält, mit dessen Schließ-Gadget das Programm beendet werden kann. Weiterhin dient dieses Fenster zum Empfang von Nachrichten der Art "INTUITICKS", die jede Zehntelsekunde von Intuition generiert werden.

Beachten Sie bitte, daß Simple-Sprites wesentlich schneller bewegt werden können, als dies im Beispielprogramm demonstriert wird. Sie können das Programm so ändern, daß Ereignisse von Intuition oder von den Zeitgebern (siehe Kapitel 6) verarbeitet werden; auf diese Weise wird die Ablaufgeschwindigkeit erhöht.

Wenn Sie das Programm starten, dann erscheinen sieben Sprites auf dem Bildschirm; sie werden von – links nach rechts – durch die Simple-Sprites 1 bis 7 repräsentiert. Die Priorität eines Sprites ist abhängig von seiner Nummer; je kleiner die Nummer, desto größer ist die Priorität gegenüber anderen Objekten.

Sprite 0 (der Mauszeiger) hat die höchste Priorität, er erscheint immer vor allen anderen Sprites, wird also niemals verdeckt dargestellt. Sprite Nummer 1 erscheint vor allen anderen Sprites, jedoch hinter dem Mauszeiger usw. Sprites werden weiterhin vom System immer vor allen anderen Dingen dargestellt (z.B. dem Screen). Es ist jedoch möglich, die Prioritäten der Sprites so zu ändern, daß sie hinter bestimmten Objekten dargestellt werden können. Im "Amiga Hardware Manual" finden Sie genauere Informationen hierzu.

Wenn Sie das Programm starten, dann können Sie mit dem Mauszeiger den Screen herunterziehen; die eintretenden Effekte veranschaulichen den Einsatz der Funktionen MoveSprite und ChangeSprite.

Beachten Sie bitte, daß im Beispielprogramm bei der Definition der "NewScreen"-Datenstruktur ein neuer Parameter verwendet wird, der bisher nicht besprochen wurde. Die Variable "Modes" der Datenstruktur enthält den Wert "SPRITES". Auf diese Weise wird Intuition mitgeteilt, daß Sprites in dem zu öffnenden Screen verwendet werden sollen; das System trifft bei der Initialisierung des Screens dann einige Sprite-spezifische Vorkehrungen.

```
/* simplesprite.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"

SHORT sprgot[7];         /* benutztes Sprite */
UWORD *sprdata[7];       /* 7 Zeiger auf Sprite-Daten */
SHORT xmove[7], ymove[7];   /* Bewegung */
struct SimpleSprite sprite[7];  /* 7 Simple-Sprites */
struct SimpleSprite *spr;       /* Zeiger auf ein Sprite */
short maxgot;           /* Maximal mögliche Sprites */

struct Window *w;         /* Zeiger auf Window    */
struct RastPort *rp;      /* Zeiger auf RastPort */
struct Screen *s;         /* Zeiger auf Screen */
struct ViewPort *vp;      /* Zeiger auf ViewPort */

struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* Die nachfolgende Datenstruktur bestimmt das Aussehen der
   Sprites. Alle 7 Sprites verwenden diese Struktur, sie
   werden jedoch an unterschiedlichen Stellen des
   Bildschirms positioniert. Beachten Sie bitte, daß
   Sprite-Daten im Chip-Memory liegen müssen, damit die Co-
   Prozessoren auf sie zugreifen können. */
/* 22 Worte Sprite-Daten = 44 Bytes Sprite-Daten */

UWORD sprite_data[] = {
                       0,0,              /* Position */
                       0x0fc3, 0x0000, /* Daten für Zeile 1*/
                       0x3ff3, 0x0000, /* Daten für Zeile 2*/
                       0x30c3, 0x0000, /* Daten für Zeile 3*/
                       0x0000, 0x3c03, /* Daten für Zeile 4*/
                       0x0000, 0x3fc3, /* Daten für Zeile 5*/
                       0x0000, 0x03c3, /* Daten für Zeile 6*/
                       0xc033, 0xc033, /* Daten für Zeile 7*/
                       0xffc0, 0xffc0, /* Daten für Zeile 8*/
                       0x3f03, 0x3f03, /* Daten für Zeile 9*/
                       0,0 /* Strukturende */
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 1)

```
   /* Strukturende ist die sogenannte "position control", die
      beim nächsten Zugriff auf die Hardware-Sprites verwendet
      werden. Das Simple-Sprite-System erlaubt nur die
      einmalige Verwendung eines Hardware-Sprites auf dem
      Bildschirm. Jede beliebige Kombination aus den Bits der
      Worte 1 und 2 pro Zeile ergibt den Farbwert des Pixels
      in dieser Zeile.
      Jedes Pixel in den Zeilen 1 bis 3, welches nicht 0 ist,
      ist die Farbnummer 1 des Sprites. Die Zeilen 4 bis 6
      entsprechen dann Nummer 2, 7-9 Nummer 3. */
                      };

/*
   Folgendes ist lediglich zur Information:
   Das Simple-Sprite-System setzt diese Bits direkt, so daß
   sich der Benutzer nicht um diese Dinge kümmern muß.
   Benutzen Sie die Funktionen ChangeSprite und MoveSprite,
   um die entsprechenden Effekte zu erzielen.

   "Position-Control":

        erstes UWORD:
             Bits 15-8, Beginn der Vertikalen, die niederen
                        8 Bit des Wertes sind hier abgelegt.
                  7-0,  Beginn der Horizontalen, die oberen 8 Bit
                        des Wertes sind hier abgelegt.

        zweites UWORD:
           Bits 15-8, Ende der Vertikalen, die niederen 8
                        Bit des Wertes sind hier abgelegt.
           Bit  7 = Attach-Bit (Wird benutzt um zusätzliche
                    Farben für verbundene Sprites zu erhalten.
                    Anstelle von 3 sind 15 Farben möglich,
                    wobei dies von der Hardware, nicht aber vom
                    Simple-Sprite-System ermöglicht wird.
          Bits 6-3 (nicht benutzt)
          Bit  2    Beginn der Vertikalen, Bit 8 des Wertes.
          Bit  1    Ende der Vertikalen, Bit 8 des Wertes.
          Bit  0    Beginn der Horizontalen, Bit 0 des Wertes.
*/

movesprites()
{
   short i;
   short newx, newy;
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 2)

```
        /* Ist der erste Parameter für "MoveSprite" gleich
           Null, so wird das Sprite relativ zum "View"
           anstelle zum " Viewport" plaziert.  Dies
           bedeutet, daß der Screen heruntergezogen wird,
           während das Sprite bewegt wird. Das Sprite sollte
           dabei an der Stelle bleiben, an der es auch zu
           Beginn war. Wenn die Null durch "vp" ersetzt wird,
           bleibt das Sprite auf dem Screen.
        */

   spr = &sprite[0];

   for (i=0; i<maxgot; i++)
     {
        newx = xmove[i]+spr->x;
        newy = ymove[i]+spr->y;

            /* Das sieht zwar etwas eigenartig aus, zeigt aber
               Möglichkeiten einen Sprite in Abhängigkeit vom
               "Viewport" oder des gesamten Bildschirms zu
               verschieben. Beim Verschieben bleibt eine
               Hälfte der Sprites auf dem Bildschirm, die
               andere Hälfte bewegt sich mit. Wenn die Sprites
               in einen Bereich eines anderen Screens fahren,
               ändern sich ihre Farben entsprechend der
               Farbpalette in diesem Screen. */
        if( (i % 2) == 0 )
                {
                     /* Sprites mit gerader Nummer auf dem
                        Bildschirm bewegen */
                     MoveSprite(0,spr,newx,newy);
                }
        else
                {
                     /* Sprites mit ungerader Nummer auf dem
                        Screen bewegen */
                     MoveSprite(vp,spr,newx,newy);
                }

        if(spr->x >= 320 || spr->x <= 0) xmove[i]=-xmove[i];
        if(spr->y >= 190 || spr->y <= 0) ymove[i]=-ymove[i];
           spr++;  /* Zeiger für das nächste Sprite ordnen */
     }
}

#define AZCOLOR 1
#define WHITECOLOR 2
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 3)

```
/*#include "mydefines.h"*/
/* enthält die Zuweisungen der Window-Flags */
/* mydefines.h */

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP|WDR)


/*#include "myscreen1.h"  */
/* myscreen1.h */


/* myfont1 beschreibt die Eigenheiten des Default-Fonts. In
   diesem Falle wird der 80-Spalten-Font ausgewählt, der 40
   Spalten in niedriger Auflösung darstellt. */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,       /* LeftEdge, TopEdge */
        320, 200,   /* Width, Height ... Größe des Screens */
        5,          /* Tiefe von 5 Planes bedeutet, man hat
                       die Wahl von 2 aus 32 verschiedenen
                       Farben, die vom Screen bereitgestellt
                       werden. */
        1, 0,       /* DetailPen, BlockPen */
        SPRITES,    /* ViewModes ...
                       Wert 0 = niedrige Auflösung */
        CUSTOMSCREEN, /* Screen-Art */
        &myfont1,     /* Default-Font des Screens */
        "32 Color Test", /* DefaultTitle, Titelzeile */
        NULL,          /* User-Gadgets des Screens, immer Null
                          -> wird ignoriert */
        NULL };   /* Adresse der eigenen Bitmap für den Screen,
                     wird hier aber nicht benutzt. */

/*#include "window1.h"   */
/* window1.h */

struct NewWindow myWindow = {
        0,      /* LeftEdge für das Fenster (in Pixels) in der
                   aktuellen Auflösung von der linken Ecke des
                   Screens gerechnet. */
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 4)

```
      15,       /* TopEdge für das Fenster (in Pixels) von der
                   oberen Ecke des Screens gerechnet. */
 320, 150,      /* Breite und Höhe des Fensters */
       0,       /* DetailPen - Nummer des Stiftes, der zum
                   Zeichnen der Window-Ränder verwendet wird. */
       1,       /* BlockPen - Nummer des Stiftes, der zum
                   Zeichnen der systemeigenen Window-Gadgets */
                /* (Der Wert -1 bedeutet bei den zwei
                    Einträgen DetailPen and BlockPen, daß
                    der Defaultwert benutzt wird.) */
  CLOSEWINDOW | INTUITICKS, /* IDCMP-Flags */
 SIMPLE_REFRESH | NORMALFLAGS | GIMMEZEROZERO | ACTIVATE,
                /* Window-Flags */
       NULL,    /* FirstGadget */
       NULL,    /* CheckMark */
       "Close-Gadget zum Abbruch anklicken",
               /* Titel des Windows */
       NULL, /* Pointer auf Screen, falls kein
                Workbench-Screen */
       NULL, /* Zeiger auf BitMap, falls
                SUPERBITMAP-Window */
       10, 10,          /* minimale Breite und Höhe */
       320, 200,        /* maximale Breite und Höhe */
       CUSTOMSCREEN
};

#include "graphics/gfxmacros.h"

/*#include "event1.c"  */
/* holt den Event-Handler */
/* event1.c */

HandleEvent(code)
LONG code;      /* von main übergeben */
{
   switch(code)
   {
   case CLOSEWINDOW:
      return(0);
      break;
   case INTUITICKS:
      movesprites();  /* 10 Bewegungen pro Sekunde */
   default:
      break;
   }
return(1);
}
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 5)

```
UWORD mycolortable[] = {
   0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
   0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
   0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

   0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
   0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
   0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df
   };

   /* Schwarz, Rot, Weiß, Feuerrot, Orange, Gelb, Blaßgrün,
      Grün, Hellblau, Dunkelblau, Purpur, Violett, Gelbbraun,
      Grau, Himmelblau (und alles noch einmal) */

main()
{
   struct IntuiMessage *msg;
   LONG result;
   short k, j;
   UWORD *src, *dest; /* Zum Kopieren der Spritedaten
                         ins RAM */

   GfxBase = OpenLibrary("graphics.library",0);


   IntuitionBase = OpenLibrary("intuition.library",0);
   /* Um das Programm kurz zu halten wurde auf eine
      Fehlerbehandlung verzichtet */
   s = OpenScreen(&myscreen1);  /* Versuchen, den Screen zu
                                   öffnen */
   if(s == 0)
      {
         printf("Myscreen1 kann nicht geöffnet werden\n");
         exit(10);
      }
   myWindow.Screen = s;   /* Adresse des Screens mitteilen */

   w = OpenWindow(&myWindow);
   if(w == 0)
   {
      printf("Window konnte nicht geöffnet werden!\n");
      CloseScreen(s);
      exit(20);
   }
   vp = &(s->ViewPort);
   /* Farben für diesen Viewport setzen */
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 6)

```
LoadRGB4(vp, &mycolortable[0], 32);
rp = w->RPort;
   /* Nun auf Nachricht von Intuition warten
    * (Task "schläft" während dieser Wartezeit)
    */

/* ****************** */
/* Simple-Sprite-Demo  */
/* ****************** */

maxgot = 0; /* Wie viele Sprites haben wir erhalten? */

spr = &sprite[0]; /* Adresse des ersten Simple-Sprites */

for(k=0; k<7; k++)
{
   xmove[k]=1;
   ymove[k]=1;

   /* Das nächstes freie Sprite benutzen */
   sprgot[k] = GetSprite(spr,-1);

   if(sprgot[k] == -1) break;
   maxgot++;

   /* Info für Position und Größe initialisieren */
   sprite[k].x = 0;
   sprite[k].y = 0;

   /* Dem System die Höhe mitteilen   */

   sprite[k].height = 9;

   /* Jetzt CHIP-Memory für das aktuelle Sprite
      reservieren */

   sprdata[k] = (UWORD *)AllocMem(44, MEMF_CHIP);

   if(sprdata[k] == NULL)
   {
      maxgot--;
      FreeSprite(sprgot[maxgot]);
      break; /* Wenn der Speicher nicht ausreicht, wird
                die Belegung weiterer Sprites abgebrochen.
                Irgendwie geht es aber weiter! */
   }
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 7)

```
      /* Nun werden die Sprite-Daten in das CHIP-RAM kopiert */

      src = sprite_data; /* Quelle */
      dest = sprdata[k]; /* Ziel */

      for( j=0; j<22; j++)
      {
         *dest++ = *src++;
      }
      /* System-Sprite-Manager mitteilen, wo sich die Sprite-
         Daten befinden */

      ChangeSprite(vp,spr,sprdata[k]);

      /* Startpunkt den Sprites wählen */

      MoveSprite(0,spr,10 + 20*sprgot[k],30);

      spr++;  /* jetzt das nächste Sprite */
   }

   while(1)   /* "forever" */
   {
      /* Warten auf eingehende Nachricht */
      WaitPort( w->UserPort );

      /* Nachricht vom Port abholen */
      msg = (struct IntuiMessage *)GetMsg(w->UserPort);
handleit:
      result = -1;
      if(msg != 0)
      {
         /* Zur Behandlung des Ereignisses: s. CLOSEWINDOW */
         result = HandleEvent(msg->Class);

         /* ermöglicht Intuition die Wiederverwendung
            von msg */
         ReplyMsg(msg);
      }
      if(result == 0)
      {
         break;  /* CLOSEWINDOW erhalten */
      }
      /* Bevor wieder gewartet wird, muß der Port geleert
         werden */
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Teil 8)

```
        msg = (struct IntuiMessage *)GetMsg(w->UserPort);

        if(msg != 0)
           goto handleit;    /* 0, falls keine weiteren
                                Nachrichten */
    }
    /* Alles erledigt! Jetzt wird aufgeräumt! */

    /* Alle belegten Sprites freigeben */
    for(k=0; k<maxgot; k++)
    {
       FreeSprite(sprgot[k]);  /* Sprites freigeben, damit
                                  andere sie wieder benutzen
                                  können... */
       FreeMem(sprdata[k],44); /* ... und den von ihnen
                                  belegten Speicher */
    }

    CloseWindow(w);
    CloseScreen(s);
}
```

Listing 7.1: Beispielprogramm zur Anwendung von Simple-Sprites (Schluß)

"Virtual-Sprites"

Die Animation von Grafik wird durch Software-Bausteine ermöglicht, die Teil der System-Software des Amiga sind. Das Simple-Sprite-System ist direkt mit der Hardware verbunden; die Einschränkungen der Hardware gelten daher auch für das Simple-Sprite-System. Ohne Tricks und Kniffe ist es nicht möglich, mehr als acht Sprites pro Screen darzustellen, da die Hardware von sich aus nur acht Hardware-Sprites verwaltet.

Das Gel-System stellt jedoch noch eine andere Art von Sprites zur Verfügung, die diesen Einschränkungen nicht unterliegen: "Virtual-Sprites". Das Arbeiten mit dieser Art Sprites ist – verglichen mit den Simple-Sprites – schwieriger. Das Virtual-Sprite-System ist ebenfalls direkt mit der Hardware verbunden, jedoch läuft die Arbeitsweise mit der Hardware hier anders als bei den Simple-Sprites ab. Virtual-Sprites sind "virtuelle" Objekte, d.h. sie werden allein durch die Software verwaltet und werden erst dann "echte" Sprites, wenn das System sie einem Hardware-Sprite zuteilt.

Auf diese Weise können mehr als acht Sprites pro Screen dargestellt werden; je nach Größe können z.B. 16, 24, 32 oder auch über 100 Sprites gleichzeitig dargestellt werden. Hier sollten Sie ruhig ein wenig experimentieren, um ein brauchbares Ergebnis zu ermitteln.

Das Gel-System "weiß", daß acht Hardware-Sprites zur Verfügung stehen, und verwaltet diese so, daß ein Sprite innerhalb eines Screens mehrmals dargestellt werden kann.

Die Datenstruktur eines Virtual-Sprites ("VSprite") enthält Datenfelder für dessen Position, Größe, Farbe und Aussehen. Jedesmal, wenn ein Sprite vollständig vom System an einer Stelle des Bildschirms dargestellt wurde, kann es für einen gewissen Zeitraum zur Darstellung eines anderen Objektes verwendet werden.

Das System verwendet also ein Hardware-Sprite zur Darstellung einer Anzahl unterschiedlicher Virtual-Sprites, wobei die Koordinierung der Darstellungen allein von der System-Software übernommen wird.

Die Vorteile eines "Virtual-Sprites"

Der Hauptvorteil eines Virtual-Sprites liegt darin, daß man sich als Programmierer nicht um die Reservierung von Sprites kümmern muß, da dies vollständig vom System übernommen wird. Sie geben lediglich an, wie ein Sprite aussehen und wo es dargestellt werden soll. Ein weiterer Vorteil ist, daß das System ein Hardware-Sprite mehrmals pro Screen verwenden kann; es stehen daher mehr als acht Sprites zur Verfügung.

Die Nachteile eines "Virtual-Sprites"

Wenn Sie mehr als acht Virtual-Sprites in derselben Rasterzeile darstellen wollen, dann kann es passieren, daß einige Sprites verschwinden; dieser unerwünschte Nebeneffekt beruht auf Einschränkungen seitens der Hardware.

Ein weiterer Nachteil ist, daß Ihnen genau genommen gar keine acht Hardware-Sprites zur Verfügung stehen, da Sprite 0 von Intuition zur Darstellung des Mauszeigers verwendet wird. Die Hardware-Sprites 0 und 1 verwenden je-

doch die gleichen Farbregister; es ist daher ebenfalls nicht möglich, das Sprite 0 als Virtual-Sprite einzusetzen, da sich die Farben des Mauszeigers ändern würden, wenn man Sprite 1 zur Darstellung mehrerer Virtual-Sprites einsetzt.

Der dritte Nachteil der Virtual-Sprites liegt in der Vergabe von Darstellungsprioritäten. Bei den Simple-Sprites gehörte jedes Sprite zu einem Hardware-Sprite, wobei die Darstellungspriorität zunahm, je kleiner die Nummer des Simple-Sprites wurde. Bei den Virtual-Sprites hingegen übernimmt jedes Hardware Sprite die Darstellung von mehreren Virtual-Sprites – es können mehr als acht Sprites gleichzeitig dargestellt werden. Wenn sich nun aber zwei Virtual-Sprites überdecken, dann kann es vorkommen, daß sich die Darstellungsprioritäten der Sprites fortwährend ändern; das Sprite im Vordergrund liegt dann z.B. hinter dem anderen Sprite und umgekehrt. Dieser Effekt läßt sich nicht vermeiden, er ist das Ergebnis der dynamischen Zuordnungen von Virtual-Sprites an die Hardware-Sprites.

Der Programmierer muß jedoch noch eine weitere Einschränkung in Kauf nehmen, wenn er Virtual-Sprites verwenden will: die Anzahl der Farben, die zum Zeichnen von Objekten (keine Sprites) verwendet werden kann, ist ebenfalls eingeschränkt. Wenn Sie z.B. einen Screen öffnen, der alle 32 Farbregister verwaltet, dann können nur die Farbregister verwendet werden, die nicht für die Sprites reserviert sind, da sonst – durch die Zuordnung mehrerer Virtual-Sprites zu einem Hardware-Sprite – auch die Objekte ihre Farbe ändern, die mit der Farbe aus einem der Sprite-Farbregister gezeichnet wurden.

Es werden vom System nicht nur die Virtual-Sprites dynamisch den acht Hardware-Sprites zugeordnet, die Inhalte der Farbregister ändern sich ebenfalls – je nachdem, welches Virtual-Sprite gerade dargestellt werden soll. Dieser Effekt wird im "makevsprite"-Beispielprogramm in diesem Kapitel zur Veranschaulichung absichtlich hervorgerufen.

Die Initialisierung des Gel-Systems

Um Virtual Sprites oder andere Komponenten des Gel-Systems verwenden zu können, muß dieses System zunächst einmal initialisiert werden. Sie können zu diesem Zweck die Routine ReadyGels aus Listing 7.2 verwenden. Vor dem Aufruf muß jedoch eine "GelsInfo"-Datenstruktur definiert worden sein, in der Daten von Virtual-Sprites oder Bobs für das Gel-System abgelegt werden. Weiterhin muß vor dem Aufruf von ReadyGels der korrespondierende Screen bereits geöffnet worden sein.

"SpriteHead" und "SpriteTail"

Im Beispielprogramm des Listings 7.2 finden Sie die Variablen "SpriteHead" und "SpriteTail". Das System verwaltet für Gels (Sprites, Bobs) eine Liste, in der sie in ihrer Darstellungsabfolge eingetragen sind. Die Listenpriorität, d.h. der Platz in der Liste, an dem ein Gel eingetragen wird, nimmt von rechts unten nach links oben zu. "SpriteHead" und "SpriteTail" bezeichnen die beiden Enden dieser Liste.

Reservierte Sprites

Sie können dem System mitteilen, welche Hardware-Sprites zur Darstellung von Virtual-Sprites verwendet werden sollen und welche nicht. Durch das Setzen bestimmter Bits auf den Wert 1 wird ein Hardware-Sprite zur Darstellung von Virtual-Sprites vom System verwendet. In Listing 7.2 wird die Variable "sprRsrvd" so initialisiert, daß die Bitpositionen den Nummern der Hardware-Sprites entsprechen, d.h. Sprite 7 wird durch Bit 7 der Variablen gesteuert, Sprite 6 durch Bit 6 usw.

Die von uns im Beispielprogramm verwendete Bitmaske hat den Wert 0xFC, was der binären Schreibweise 11111100 entspricht. Die Bitpositionen 0 und 1 sind in dieser Maske nicht gesetzt, d.h. die Hardware Sprites 0 und 1 werden vom System nicht zur Darstellung von Virtual Sprites verwendet. Eine solche Bitmaske sollte jedoch auch dynamisch angelegt werden, da einzelne Hardware-Sprites bereits von anderen Tasks belegt sein können. Durch das Auslesen des Datenfeldes "GfxBase->SpriteReserved" können Sie feststellen, welche Sprites noch verfügbar sind; sie sind in der Bitmaske, die dieses Datenfeld liefert, durch den Wert 0 an der entsprechenden Bitposition gekennzeichnet. In unserem Beispielprogramm reicht daher eine einfache UND-Verknüpfung zum Reservieren von Hardware-Sprites aus.

Nachfolgend finden Sie ein Programmfragment, das diese Vorgehensweise veranschaulichen soll.

```
 struct GfxBase *GfxBase;

 /* Die folgende Zeile wird im Programm aus Listing 7.2
anstelle von "g->sprRsrvd = 0xfc" eingesetzt, um die
Bitmaske dynamisch anzupassen */

 g->sprRsrvd = 0xfc & (!(GfxBase->SpriteReserved));
```

```
/* readygels.c */

struct VSprite *SpriteHead = NULL;
struct VSprite *SpriteTail = NULL;

ReadyGels(g, r)
struct RastPort *r;
struct GelsInfo *g;
{
   /* Anfang und Ende der Liste belegen */
   if ((SpriteHead = (struct VSprite *)AllocMem(sizeof
         (struct VSprite), MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      return(-1);
   }

   if ((SpriteTail = (struct VSprite *)AllocMem(sizeof
         (struct VSprite), MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      FreeMem(SpriteHead, sizeof(struct VSprite));
      return(-2);
   }
   g->sprRsrvd = 0xFC; /* Sprites 0 oder 1 nicht benutzen. */
   if ((g->nextLine = (WORD *)AllocMem(sizeof(WORD) * 8,
         MEMF_PUBLIC | MEMF_CLEAR)) == NULL)
   {
      FreeMem(SpriteHead, sizeof(struct VSprite));
      FreeMem(SpriteTail, sizeof(struct VSprite));
      return(-3);
   }

  if ((g->lastColor = (WORD **)AllocMem(sizeof(LONG) * 8,
         MEMF_PUBLIC | MEMF_CLEAR)) == NULL)
   {
      FreeMem(g->nextLine,8 * sizeof(WORD));
      FreeMem(SpriteHead, sizeof(struct VSprite));
      FreeMem(SpriteTail, sizeof(struct VSprite));
      return(-4);
   }

  /* Als nächstes bereiten wir eine Tabelle von Zeigern auf
     Funktionen vor, die angesprungen werden sollen, sowie
     DoCollision eine Kollision bemerkt.  Diese Deklaration
     ist zwar nicht für ein einfaches VSprite nötig, bei
     dem keine Kollision erkannt wird, es gehört aber
     einfach zu einem vollständigen Beispiel. */
```

Listing 7.2: Routine zur Initialisierung des Gel-Systems (Teil 1)

```
    if ((g->collHandler = (struct collTable *)AllocMem(sizeof(
       struct collTable), MEMF_PUBLIC|MEMF_CLEAR))==NULL)

    {
       FreeMem(g->lastColor, 8 * sizeof(LONG));
       FreeMem(g->nextLine,8 * sizeof(WORD));
       FreeMem(SpriteHead, sizeof(struct VSprite));
       FreeMem(SpriteTail, sizeof(struct VSprite));
       return(-5);
    }

   /* Sollte irgendein Teil des Objektes diese Grenzen
      berühren oder überfahren, so wird die dafür vorgesehene
      Routine aufgerufen. Dies ist in smash[0] in der
      Collision-Handler-Table untergebracht und wird nur
      dann aufgerufen, wenn DoCollision ausgeführt wurde. */

    g->leftmost = 0;
    g->rightmost = r->BitMap->BytesPerRow * 8 - 1;
    g->topmost = 0;
    g->bottommost = r->BitMap->Rows - 1;

    r->GelsInfo = g;  /* Die beiden Strukturen verbinden */

    InitGels(SpriteHead, SpriteTail, g );

   /* Zeiger auf einen Dummy-Sprite initialisieren, das vom
      System benutzt wird, um das Animationssystem verfolgen
      zu können. */

    SetCollision(0, border_dummy, g);
    WaitTOF();
    return(0); /* Bei einem Rückgabewert von 0 ist alles
                  glatt gelaufen. Negative Werte zeigen
                  einen Fehler an. Um zu erkennen, was die
                  Routine dagegen unternimmt, sehen Sie
                  bitte im Listing nach. Alle Fehler sind
                  Ergebnis von zu wenig Speicher. */

}

void border_dummy()      /* Dummy-Collision-Routine */
{
    return;
}
```

Listing 7.2: Routine zur Initialisierung des Gel-Systems (Schluß)

Diese Art der Sprite-Reservierung ist erforderlich, da das Simple-Sprite-System unabhängig vom Gel-System entwickelt wurde. Die einzige Möglichkeit festzustellen, welche Hardware-Sprites bereits belegt sind, ist das Auslesen des oben erwähnten Datenfeldes; es stellt eine Art Schnittstelle zwischen Simple-Sprites und Virtual-Sprites dar.

Die Datenfelder "NextLines" und "LastColors"

Der Amiga verwaltet verschiedene Variablen, mit deren Hilfe das System entscheidet, welches Hardware-Sprite zur Darstellung des nächsten Virtual-Sprites verwendet werden kann. Diese Variablen sind Bestandteil der "GelsInfo"-Datenstruktur, die weiterhin die Datenfelder "NextLines" und "LastColors" beinhaltet.

Das "NextLines"-Datenfeld enthält Informationen darüber, an welchen Rasterzeilen des Bildschirms ein Hardware-Sprite zur Darstellung eines neuen Virtual-Sprites verwendet werden kann. Das Datenfeld "LastColors" enthält einen Zeiger auf die Farbwerte, die vom zuletzt dargestellten Virtual-Sprite verwendet wurden. Diese Farbwerte werden in die Farbregister des Hardware-Sprites kopiert, das vom System zur Darstellung des aktuellen Virtual-Sprites verwendet wird. "LastColors" enthält für jedes verwendete Hardware-Sprite einen Zeiger auf die von ihm zuletzt verwendeten Farben.

Wenn das System ein Hardware-Sprite zur Darstellung eines Virtual-Sprites auswählt, dann wird jedesmal das Datenfeld "LastColors" überprüft. Entsprechen die Farbwerte des neuen Virtual-Sprites denen des zuletzt dargestellten, dann erübrigt sich ein erneutes Umkopieren der Farbwerte in die Hardware-Farbregister; die "Copper List" (Co-Prozessor-Befehlsliste) braucht in diesem Fall nicht geändert zu werden. (Im "Amiga Hardware Manual" finden Sie nähere Informationen zum Copper).

Wenn jedes Virtual-Sprite andere Farben verwendet (d.h. LastColors enthält acht verschiedene Zeigerwerte), dann können maximal vier Sprites pro Rasterzeile dargestellt werden. Ordnen Sie hingegen die Farbregister paarweise zu (d.h. die Virtual-Sprites 0 und 1 haben dieselbe Farbe, ebenso 2 und 3 usw.), dann können acht Virtual-Sprites pro Rasterzeile vom System dargestellt werden.

Da das System die Farbregister der Hardware-Sprites paarweise verwaltet, kann jedem Hardware-Sprite ein Virtual-Sprite zugeordnet werden, was exakt den Möglichkeiten der Hardware entspricht. Beachten Sie bitte, daß das "Last-Color"-Datenfeld nicht für "Bobs" verwendet wird; seine Verwendung durch das System beschränkt sich ausschließlich auf Sprites.

Die MakeVSprite-Routine

In Listing 7.3 finden Sie ein Beispielprogramm, mit dessen Hilfe Virtual-Sprites definiert werden können. Beachten Sie bitte, daß die Routine zwar ein Virtual-Sprite erzeugt, dieses jedoch nicht in die vom System verwaltete Liste zur Sprite-Darstellung einträgt; das Eintragen in diese Liste ist Aufgabe des Programmteils, das die MakeVSprite-Routine aufruft. Erhalten Sie von der Routine den Return-Wert Null, dann steht nicht genug Speicher zur Definition und Ablage des Virtual-Sprites zur Verfügung. In allen anderen Fällen liefert die Routine die Startadresse des Virtual-Sprites im Speicher. Der Zeiger auf diese Adresse wird dann der Funktion AddSprite als Parameter übergeben, um das Sprite in die Systemliste eintragen zu lassen.

Die "VSprite"-Datenstruktur

Die MakeVSprite-Routine verwendet bei ihrem Aufruf Parameter und Datenfelder der "VSprite"-Datenstruktur. Diese Struktur wird zur Definition von Virtual-Sprites und von Bobs verwendet. Das System arbeitet mit dieser Struktur dynamisch und paßt die Inhalte der Datenfelder den jeweiligen Gegebenheiten an. Die Struktur enthält weiterhin Kollisionsmasken, mit deren Hilfe das System feststellt, ob ein Sprite z.B. den linken Rand des Darstellungsbereichs berührt hat (es muß zu diesem Zweck eine entsprechende Routine im Programm vorhanden sein; wir verwenden eine "leere" Funktion). Nachfolgend finden Sie eine Erläuterung der einzelnen Datenfelder der Struktur.

Das Datenfeld "Height"

Dieses Datenfeld gibt die Höhe (gemessen in Rasterzeilen) des zugehörigen Virtual-Sprites an. Die Höhe eines Virtual-Sprites (oder eines Bobs) kann be-

```
/* makevsprite.c */

struct VSprite *MakeVSprite(lineheight, image, colorset, x, y,
                                wordwidth, imagedepth, flags)
SHORT lineheight;  /* Wie groß ist dieses VSprite? */
WORD *image;  /* Wo sind die VSprite-Bilddaten?
                 Müssen doppelt so viele Worte sein, wie in
                 "lineheight" angegeben. */
WORD *colorset;  /* Wo sind die drei Worte, die die Farbe des
                    Sprites beschreiben? */
SHORT x, y;   /* "Initialisierungsposition" auf dem Screen */
SHORT wordwidth, imagedepth, flags;
{
   struct VSprite *v; /* Zeiger auf die VSprite-Struktur, die
                         von dieser Routine dynamisch
                         angefordert wird. */

   if ((v = (struct VSprite *)AllocMem(sizeof(struct VSprite),
         MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   return(0);

   v->Flags = flags;      /* Ist es ein VSprite oder etwa ein
                             Bob? */

   v->Y = y;  /* Initialierungsposition relativ zu den
                 Display-Koordinaten */

   v->Height = lineheight;  /* Der Aufrufer teilt die Höhe
                               mit. */
   v->Width = wordwidth;    /* Ein VSprite ist stets ein Wort
                               (16 Bit) breit. */

   /* Es existieren zwei Arten von "Tiefen". Zum einen ist da
      die Tiefe der Bilddaten selbst, zum anderen die Tiefe
      des Playfields in dem es gezeichnet wird. Die Tiefe der
      Bilddaten gibt an, wieviel Speicher man dafür dynamisch
      anfordern muß. Die Tiefe des Playfields gibt den
      Speicherbedarf an, der zum Abspeichern des
      Hintergrundes benötigt wird, sowie Bobs gezeichnet
      werden. Ein VSprite hat immer eine Tiefe von zwei
      Planes, wenn es aber dazu benutzt wird, die "Tiefe"
      eines Bobs zu erhöhen, dann ... */

   v->Depth = imagedepth;

   /* Annahme, daß der Aufrufer zumindest eine Routine hat,
      die Standard-Kollisionen mit den Rändern registrieren.
      Bit 1 aus dieser Maske ist für Kollision mit den
      Rändern reserviert, der von DoCollision() festgestellt
      wird. Dies ist auch die einzige Berührung über die man
      informiert wird. Aber das alles kann der Aufrufer
      später noch ändern. */
```

Listing 7.3: Die Makevsprite-Routine (Teil 1)

```
    v->MeMask = 1;
    v->HitMask = 1;


    v->ImageData = image; /* Aufrufer teilt mit, wo die Image-
                             Daten zu finden sind. */


    /* Zeigt dem System, wo die Maske für das VSprite deponiert
       ist. (Erlaubt eine rasche Erkennung bei horizontalen
       Rändern). */


    if ((v->BorderLine = (WORD *)AllocMem((sizeof(WORD)
                 *wordwidth), MEMF_PUBLIC | MEMF_CLEAR)) == 0)
    {
         FreeMem(v, sizeof(struct VSprite));
         return(0);
    }


    /* Mitteilung an das System, wo die Maske zu finden ist,
       die für jede Position im Objekt in jeder Plane ein Bit
       beinhaltet (alle Planes werden miteinander GeODERt). */


    if ((v->CollMask = (WORD *)AllocMem(sizeof(WORD)*lineheight
        *wordwidth, MEMF_CHIP | MEMF_CLEAR)) == 0)
    {
         FreeMem(v, sizeof(struct VSprite));
         FreeMem(v->BorderLine, wordwidth * sizeof(WORD));
         return(0);
    }

    /* Dies wird nicht für einen Bob, sondern nur für ein
       VSprite benötigt. Dies ist der vom Aufrufer mitgeteilte
       Ort, an dem sich die Farben befinden. */

    v->SprColors = colorset;

    /* Folgendes wird nicht für ein VSprite gebraucht, MakeBob
       belegt es aber für Bobs. */
    v->PlanePick  = 0x00;
    v->PlaneOnOff = 0x00;

   InitMasks(v); /* "collMask" und "borderLine" erstellen */

    return(v);
}
/* Ende von Makevsprite.c */
```

Listing 7.3: Die Makevsprite-Routine (Schluß)

liebig gewählt werden, jedoch muß der Wert des Datenfeldes mit der Anzahl der Datenworte in der "Image"-Struktur (das ist die Struktur, in der das Aussehen des Sprites definiert wird) übereinstimmen. Beachten Sie bitte, daß das System bei Bobs mit zunehmender Höhe längere Zeit für die Darstellung benötigt.

Das Datenfeld "Image"

Dieses Datenfeld enthält einen Zeiger auf die Datenstruktur, in der das Aussehen des Sprites definiert ist. Die Datenstruktur aus Listing 7.1 ist ein typisches Beispiel für eine solche Auflistung von Datenworten. Der Zeigerwert bezieht sich immer auf das erste Datenwort einer solchen Struktur; Sprites werden also immer von oben nach unten dargestellt.

Virtual-Sprites benötigen – anders als Simple-Sprites – keine fortwährende Präsenz der Sprite-Daten. Trotzdem müssen diese Daten im Chip-Memory – dem untersten 512KB Speicher des Systems – abgelegt werden, damit die Co-Prozessoren auf sie zugreifen können.

Das Datenfeld "Colors"

In diesem Datenfeld verwaltet das System einen Zeiger auf drei Datenworte (48 Bits), die die Farben für das zugehörige Virtual-Sprite ergeben. Jedes Datenwort korrespondiert mit einer der drei möglichen Farben (1, 2 und 3).

Wie bereits erwähnt, verwaltet das System weiterhin einen Zeiger auf die zuletzt verwendeten Farbwerte für ein Virtual-Sprite. Wenn Sie mehrere Virtual-Sprites mit identischen Farben haben, dann sind demnach auch die zugehörigen Zeigerwerte gleich. Sucht das System ein freies Hardware-Sprite zur Darstellung eines Virtual-Sprites, dann "weiß" es, daß Sprite-Paare die gleichen Farbregister verwenden. Greifen alle Virtual-Sprites auf dieselben Farbregister zu, dann ist es für das System ungleich einfacher, ein freies Hardware-Sprite zu finden.

Haben alle Virtual-Sprites verschiedene Farben, dann kann es unter Umständen vorkommen, daß das System kein freies Hardware-Sprite zur Darstellung reservieren kann; es ist also leicht möglich, daß Sprites nicht dargestellt werden. (Das System prüft 60mal in der Sekunde, ob ein Hardware-Sprite zur Darstellung freigegeben werden kann.)

Die Datenfelder "x" und "y"

Die Position des aktuellen Virtual-Sprites wird in diesen Datenfeldern abgelegt. Virtual-Sprites können aber nur relativ zum zugehörigen Screen positioniert werden. (Simple-Sprites hingegen unterliegen dieser Einschränkung nicht; sie können auch relativ zum gesamten Darstellungsbereich View bzw. ViewPort positioniert werden.)

Das Datenfeld "Width"

In diesem Datenfeld wird die Breite eines Virtual-Sprites oder Bobs abgelegt. Wenn Sie Virtual-Sprites verwenden, dann hat dieses Datenfeld immer den Wert Eins. Bobs hingegen können beliebig breit dargestellt werden, jedoch benötigt das System bei zunehmender Breite mehr Zeit zur Darstellung.

Das Datenfeld "Depth"

Dieser Parameter gibt an, aus wie vielen Bitplanes ein Virtual-Sprite oder Bob besteht. Bei Virtual-Sprites hat "Depth" immer den Wert Zwei. Wenn Sie Bobs verwenden, dann kann dieser Parameter einen Wert annehmen, der kleiner oder gleich dem "Depth"-Parameter der "NewScreen"-Datenstruktur ist.

Das Datenfeld "Flags"

Der Inhalt dieses Datenfeldes sagt dem System, welche Art von Gel durch die VSprite-Datenstruktur definiert wird. Verwenden Sie Virtual-Sprites, dann setzen Sie dieses Datenfeld auf den Wert "VSPRITE". Verwenden Sie Bobs, dann hat dieses Datenfeld den Wert Null.

Beispielprogramm: VirtualSprite

Das Listing 7.4 zeigt ein Beispielprogramm, daß das Arbeiten mit Virtual-Sprites veranschaulichen soll. Es kommen Routinen zum Einsatz, die bereits bei den Simple-Sprites verwendet wurden; einige von ihnen wurden jedoch den Gegebenheiten und Anforderungen angepaßt. Die Unterschiede zum

Simple-Sprites-Beispielprogramm liegen darin, daß nun mehr als sieben Sprites auf dem Bildschirm dargestellt werden. Weiterhin verwendet das Programm nun den Rasterstrahl des Bildschirms zur Synchronisation der Sprite-Darstellung. (Im Simple-Sprite-Beispielprogramm wurden die Sprites beim Auftreten eines Ereignisses vom Typ "INTUITICKS" bewegt.) Aus diesem Grund bewegen sich die Sprites im folgenden Beispielprogramm etwa sechsmal schneller als die Simple-Sprites.

Die Positionierung der Sprites beim Starten des Programms erfolgt zufällig. Weiterhin wurden so viele Sprites definiert, daß der Effekt des kurzzeitigen "Verschwindens" von Sprites deutlich zu sehen ist.

Dieser Effekt tritt immer dann auf, wenn kein Hardware-Sprite zur Darstellung eines Virtual-Sprites vorhanden ist. Sie können weiterhin zwei andere Effekte beobachten, auf die bereits schon eingegangen wurde: das Ändern der Darstellungspriorität von Sprites, wenn sie sich überdecken und die dynamische Änderung der Farbregisterwerte. Beeinflußt werden die Farbregister 21 bis 23, 25 bis 27 und 29 bis 31. Sie können an diesem Beispiel sehen, was das Gel-System mit den Werten dieser Farbregister macht, wenn sich die Sprites über den Bildschirm bewegen.

Noch ein Hinweis: Das Beispielprogramm aus Listing 7.4 läuft nur unter der Betriebssystemversion 1.2 (oder größer) fehlerfrei. In der Version 1.1 waren noch gravierende Mängel im Gel-System (im Hinblick auf die Darstellung von Virtual-Sprites), die jedoch in der Version 1.2 nicht mehr vorhanden sind. Bobs laufen sowohl unter Version 1.1 als auch unter 1.2 fehlerfrei.

```
/* vsprite.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"
#include "graphics/gels.h"

/* System mit der Erzeugung und Verwaltung von MAXSP VSprites
   beauftragen */
#define MAXSP 28
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 1)

```
/* Mögliche Geschwindigkeit für VSprites definieren (Angabe
   in Zahl der "vblank"-Interrupts). */

SHORT speed[] = { 1, 2, -1, -2 };

SHORT xmove[MAXSP], ymove[MAXSP];    /* Bewegung */
struct VSprite *vsprite[MAXSP]; /* MAXSP Simple-Sprites */
struct VSprite *vspr;           /* Zeiger auf Sprite */
short maxgot;                   /* Maximal mögliche Sprites */

struct GelsInfo mygelsinfo;  /* Der RastPort des Windows
                                braucht dies im Zusammenhang
                                mit VSprites. */

struct Window *w;       /* Zeiger auf Window   */
struct RastPort *rp;    /* Zeiger auf RastPort */
struct Screen *s;       /* Zeiger auf Screen */
struct ViewPort *vp;    /* Zeiger auf ViewPort */

struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* 18 Worte Sprite-Daten = 9 Zeilen Sprite-Daten */

UWORD sprite_data[] = {
                       0x0fc3, 0x0000, /* Daten für Zeile 1*/
                       0x3ff3, 0x0000, /* Daten für Zeile 2*/
                       0x30c3, 0x0000, /* Daten für Zeile 3*/
                       0x0000, 0x3c03, /* Daten für Zeile 4*/
                       0x0000, 0x3fc3, /* Daten für Zeile 5*/
                       0x0000, 0x03c3, /* Daten für Zeile 6*/
                       0xc033, 0xc033, /* Daten für Zeile 7*/
                       0xffc0, 0xffc0, /* Daten für Zeile 8*/
                       0x3f03, 0x3f03, /* Daten für Zeile 9*/
                       };
UWORD *sprdata;
movesprites()
{
   short i;

   for (i=0; i<maxgot; i++)
   {
      vspr = vsprite[i];
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 2)

```
        vspr->X = xmove[i]+vspr->X;
        vspr->Y = ymove[i]+vspr->Y;

        /* Sprites bewegen */

        if(vspr->X >= 300 || vspr->X <= 0) xmove[i]=-xmove[i];
        if(vspr->Y >= 190 || vspr->Y <= 0) ymove[i]=-ymove[i];

    }
    SortGList(rp);      /* Liste sortieren */
    DrawGList(rp, vp); /* Spritebefehle erzeugen */

    MakeScreen(s);      /* Alles vereinigen und von */
    RethinkDisplay();   /* Intuition zeigen lassen */
}


#define AZCOLOR 1
#define WHITECOLOR 2

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP)
/* Das Flag WINDOWDRAG wird nicht benutzt, da der Screen
   nicht verschoben werden soll. */

/* Erlaubt dem Benutzer die Größe des Fensters zu verändern.
   Zuerst kann es verkleinert und anschließend vergrößert
   werden, so daß der Text im Hintergrund gelöscht wird.
   Der Benutzer kann einfach erkennen, daß einige VSprites
   flackern, sobald zu viele Sprites innerhalb einer einzigen
   horizontalen Plane erscheinen, da das VSprite-System keine
   Sprites mehr zuweisen kann. */

/* myfont1 beschreibt die Eigenheiten des Default-Fonts. In
   diesem Falle wird der 80-Spalten-Font ausgewählt, der 40
   Spalten in niedriger Auflösung darstellt. */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,       /* LeftEdge, TopEdge */
        320, 200,   /* Width, Height ... Größe des Screens */
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 3)

```
        5,          /* Tiefe von 5 Planes bedeutet, man hat
                       die Wahl von 2 aus 32 verschiedenen
                       Farben, die vom Screen bereitgestellt
                       werden. */
        1, 0,       /* DetailPen, BlockPen */
        SPRITES,    /* ViewModes ...
                       Wert 0 = niedrige Auflösung */
        CUSTOMSCREEN, /* Screen-Art */
        &myfont1,     /* Default-Font des Screens */
        "32 Color Test", /* DefaultTitle, Titelzeile */
        NULL,         /* User-Gadgets des Screens, immer Null
                         -> wird ignoriert */
        NULL }; /* Adresse der eigenen Bitmap für den Screen,
                   wird hier aber nicht benutzt. */


struct NewWindow myWindow = {
   0,          /* LeftEdge für das Fenster (in Pixeln) in der
                  aktuellen Auflösung von der linken Ecke des
                  Screens gerechnet. */
   0,          /* TopEdge für das Fenster (in Pixeln) von der
                  oberen Ecke des Screens gerechnet. */
   320, 185,   /* Breite und Höhe des Fensters */
   0,          /* DetailPen - Nummer des Stiftes, der zum
                  Zeichnen der Windowränder verwendet wird. */
   1,          /* BlockPen - Nummer des Stiftes, der zum
                  Zeichnen der systemeigenen Window-Gadgets */
               /* (Der Wert -1 bedeutet bei den zwei Einträgen
                  DetailPen and BlockPen, daß der Defaultwert
                  benutzt wird.) */
   CLOSEWINDOW,   /* Simple-Sprite-Programm benutzt sowohl
                     INTUITICKS, als auch IDCMP-Flags */
   NORMALFLAGS | GIMMEZEROZERO | ACTIVATE, /* Window-Flags */
   NULL,       /* FirstGadget */
   NULL,       /* CheckMark   */
   "Close-Gadget zum Abbruch anklicken",
               /* Titel des Windows */
   NULL, /* Pointer auf Screen, falls kein
                  Workbench-Screen */
   NULL, /* Zeiger auf BitMap, falls
                  SUPERBITMAP-Window */
   10, 10,        /* minimale Breite und Höhe */
   320, 200,      /* maximale Breite und Höhe */
   CUSTOMSCREEN
};

#include "graphics/gfxmacros.h"
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 4)

```
/*#include "event1.c"  */
/* holt den Event-Handler */
/* event1.c */

HandleEvent(code)
LONG code;      /* von main übergeben */
{
   switch(code)
   {
   case CLOSEWINDOW:
      return(0);
      break;
   case INTUITICKS:
      movesprites();  /* 10 Bewegungen pro Sekunde */
      default:
      break;
   }
return(1);
}


UWORD mycolortable[] = {
   0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
   0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
   0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

   0x0000, 0x0e30, 0x0fff, 0x0b40,
   0x0fb0, 0x0bf0, 0x05d0, 0x0ed0,
   0x07df, 0x069f, 0x0c0e, 0x0f2e,
   0x0feb, 0x0c98, 0x0bbb, 0x07df
   };

   /* Schwarz, Rot, Weiß, Feuerrot, Orange, Gelb, Blaßgrün,
      Grün, Hellblau, Dunkelblau, Purpur, Violett, Gelbbraun,
      Grau, Himmelblau (und alles noch einmal) */

UWORD colorset0[] = { 0x0e30, 0xffff, 0x0b40 };
                        /* Gleiches, wie Farben 17-19 */
UWORD colorset1[] = { 0x0bf0, 0x05d0, 0x0ed0 }; /* 21-23 */
UWORD colorset2[] = { 0x069f, 0x0c0e, 0x0f2e }; /* 25-27 */
UWORD colorset3[] = { 0x0c98, 0x0bbb, 0x07df }; /* 29-31 */
UWORD *colorset[] = {
         colorset0, colorset1,
         colorset2, colorset3 };
int choice;
char *numbers[] =
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 5)

```
{
   "17","18","19",
   "20","21","22","23",
   "24","25","26","27",
   "28","29","30","31"
};


#include "ram:purgegels.c"
#include "ram:readygels.c"
#include "ram:makevsprite.c"


main()
{
   struct IntuiMessage *msg;
   LONG result;
   SHORT k, j, x, y, error;
   UWORD *src, *dest; /* Zum Kopieren der Spritedaten
                         ins RAM */


   GfxBase = OpenLibrary("graphics.library",0);

   IntuitionBase = OpenLibrary("intuition.library",0);
   /* Um das Programm kurz zu halten, wurde auf eine
      Fehlerbehandlung verzichtet */
   s = OpenScreen(&myscreen1);  /* Versuchen, den Screen zu
                                   öffnen */
   if(s == 0)
      {
         printf("Myscreen1 kann nicht geöffnet werden\n");
         exit(10);
      }
   myWindow.Screen = s;    /* Adresse des Screens mitteilen */

   ShowTitle(s, FALSE);    /* Screen darf nicht nach unten
                              gedraggt werden */
   w = OpenWindow(&myWindow);
   if(w == 0)
   {
      printf("Window konnte nicht geöffnet werden!\n");
      CloseScreen(s);
      exit(20);
   }
   vp = &(s->ViewPort);
   /* Farben für diesen Viewport setzen */
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 6)

```
LoadRGB4(vp, &mycolortable[0], 32);
rp = w->RPort;
   /* Nun auf Nachricht von Intuition warten
    * (Task "schläft" während dieser Wartezeit)
    */

/* Text mit den Spritefarben schreiben, so daß man an der
   Demo sieht, wie das System die Farben einlädt. Es
   sollte darauf geachtet werden, grundsätzlich bei
   Verwendung von VSprites auch die betreffenden
   Farbregister zu benutzen.

   Beachten Sie, daß sie Farben mit den Nummern 17-19
   nicht angetastet werden. Grund: sprResrvd=0xFC in
   ReadyGels. (Verbietet dem Virtual-Sprite-System den
   Zugriff auf die beiden Sprites 0 und 1. 0 wird als
   Mauszeiger benutzt und 1 teilt sich die Farben mit
   Sprite 0. Daher werden einfach beide reserviert. */

for(j=8; j<180; j+=50)
{
   for(k=0; k<15; k++)
   {
      Move(rp,k*20,j);
      SetAPen(rp,k+17);        /* Farben 17-31 zeigen */
      /* 16, 20, 24, 28 werden nicht von VSprites berührt,
         da sie nicht von den Hardware-Sprites benutzt
         werden. */

      Text(rp,numbers[k],2);
   }
 }
/* ****************** */
/* VSprite-Demo       */
/* ****************** */

/* Um die gültigen Spritedaten aufzunehmen wird
   CHIP-Memory belegt. Notwendig, falls es auf einem Amiga
   mit RAM-Erweiterung gestartet werden soll. */

sprdata = (UWORD *)AllocMem(36, MEMF_CHIP);

if(sprdata == NULL)
{
   /* Zu wenig Speicher für das Sprite */
   printf("Kein freier Speicher für Spritedaten!\n");
   goto finish;
}
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 7)

```
  /* Nun werden die Sprite-Daten in das CHIP-RAM kopiert */

src = sprite_data; /* Quelle */
dest = sprdata;    /* Ziel */


for( j=0; j<18; j++)
{
   *dest++ = *src++;
}


choice = 0;
maxgot = 0;


/* GELS-System vorbereiten, um mit VSprites und Bobs zu
   arbeiten */

error = ReadyGels(&mygelsinfo, rp);

for(k=0; k<MAXSP; k++)  /* Maximale Anzahl der VSprites */
{
   xmove[k]=speed[RangeRand(4)];
   ymove[k]=speed[RangeRand(4)];

   /* Position für Sprite einrichten */
   x = 10 + RangeRand(280);
   y = 10 + RangeRand(170);

   /* VSprite erzeugen */
   vsprite[k] = MakeVSprite( 9, sprdata, colorset[choice],
          x, y, 1, 2, VSPRITE);

   /* 9 Zeilen groß, benutzt Spritedaten aus MEMF_CHIP,
      mit einer Auswahl an Farben, an Position X,Y mit der
      Breite eines Wortes, Tiefe: 2 Planes (Alle VSprites
      besitzen eine Tiefe von 2 Planes) und es ist ein
      VSprite */

   if(vsprite[k] == 0)
   {
      break;  /* Kein Speicher mehr */
   }
   AddVSprite(vsprite[k], rp);
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Teil 8)

```
        maxgot++;
        choice++; /* Unterschiedliche Farbauswahl treffen */
        if(choice >= 4)
        {
           choice = 0; /* Farbauswahl rotieren lassen */
        }
     }
     while(1)   /* "forever" */
     {
        WaitTOF();
        movesprites();

        result = -1;    /* Nachsehen, ob "CLOSEWINDOW" wartet */
        msg = (struct IntuiMessage *)GetMsg(w->UserPort);
        if(msg != 0)
        {
           result = HandleEvent(msg->Class);

           /* ermöglicht Intuition die Wiederverwendung
              von msg */
           ReplyMsg(msg);
        }
        if(result == 0)
        {
           break;   /* CLOSEWINDOW erhalten */
        }
     }

        /* Bevor wieder gewartet wird, muß der Port geleert
           werden */

     /* Alle erzeugten Sprites wieder freigeben */
finish:
     for(k=0; k<maxgot; k++)
     {
          DeleteGel(vsprite[k]);
     }
     /* Alles von ReadyGels Erstellte wieder löschen */

     PurgeGels(&mygelsinfo);

     FreeMem(sprdata, 36);    /* Belegtes freigeben */

     CloseWindow(w);
     CloseScreen(s);
}
```

Listing 7.4: Beispielprogramm zur Anwendung von Virtual-Sprites (Schluß)

Blitter Objects (BOBS)

Der Amiga kennt neben Simple-Sprites und Virtual-Sprites noch eine dritte Art animierter Objekte: die "Bobs". Diese Art grafischer Objekte wird durch die "Bob"-Datenstruktur definiert. Bobs sind – genau wie Virtual-Sprites – Teile des Gel-Systems. Der Unterschied zwischen Bobs und Virtual-Sprites besteht darin, daß Virtual-Sprites unabhängig vom zugehörigen Hintergrund dargestellt werden; Bobs hingegen sind ein Teil des Hintergrundes. Das bedeutet, daß beim Bewegen eines Bobs der Hintergrund "zerstört" wird; dies wäre z.B. ein wünschenswerter Effekt in einem Malprogramm, wenn man ein Bob als "Pinsel" einsetzt. Bobs können aus mehr als vier Farben bestehen, das Maximum an verfügbaren Farben wird durch den zugehörigen Hintergrund (den Screen) festgelegt. Der Programmierer kann dem System jedoch mitteilen, daß der Teil des Hintergrunds, über dem sich ein Bob befindet, zwischengespeichert und später – nach dem Bewegen des Bobs – wiederhergestellt werden soll.

In der "Bob"-Datenstruktur ist ein entsprechendes Datenfeld vorhanden, dessen Inhalt dem System sagt, auf welche Weise ein Bob dargestellt und bewegt werden soll. Die Struktur enthält weiterhin einen Zeiger auf eine "VSprite"-Datenstruktur, in der der Rest der notwendigen Definition abgelegt ist. Durch die Verwendung der "VSprite"-Datenstruktur für Bobs gestaltet sich die Programmierung dieser Objekte recht einfach, da durch eine einzige Routine alle Arten grafischer Objekte kontrolliert werden können.

Die MakeBob-Routine

In Listing 7.5 finden Sie ein Beispielprogramm, mit dessen Hilfe Bobs definiert werden können. Die meisten Parameter der Routine entsprechen denen der MakeVSprite-Routine. Für Bobs sind jedoch einige weitere Datenfelder erforderlich, so z.B. "PlanePick" und "PlaneOnOff".

Der "Image"-Zeiger

Wie bereits erwähnt, werden die einzelnen Zeilen von Sprites durch jeweils zwei Datenworte gebildet, z.B.

```
0x0fc3, 0x0000        /* Zeile 1 des Sprites */
```

```
/* makebob.c */

struct Bob *MakeBob(bitwidth,lineheight,imagedepth,image,
      planePick,planeOnOff, x,y, flags)
SHORT bitwidth,lineheight,imagedepth,planePick,planeOnOff,x,y,flags;
WORD *image;
{
   struct Bob *b;
   struct VSprite *v;
   SHORT wordwidth;

   wordwidth = (bitwidth+15)/16;

  /* Für diesen Bob ein VSprite erzeugen; muß wieder freigegeben
     werden, wenn der Bob gelöscht wird.
     Hinweis: keine Farbauswahl bei Bobs. */

   if ((v = MakeVSprite(lineheight, image, NULL, x, y,
       wordwidth, imagedepth, flags)) == 0)
      return(0);        /* Zuwenig Speicher für VSprite */

   /* Aufrufer wählt die Bitplane, in die die Grafik gezeichnet wird. */
   v->PlanePick = planePick;

   /* Was geschieht mit der Bitplane in die nicht gezeichnet wurde? */
   v->PlaneOnOff = planeOnOff;

   if ((b = (struct Bob *)AllocMem(sizeof(struct Bob),
         MEMF_PUBLIC | MEMF_CLEAR)) == 0)
      return(0);        /* kein Speicher für den Bob */

   v->VSBob = b; /* Bob und zugehörige VSprite-Strukturen verbinden */

   b->Flags = 0; /* Kein Teil eines Animationsobjektes (BOBISCOMP).
                    Nach dem Verschieben des Bobs ist die Grafik nicht
                    mehr vorhanden (SAVEBOB) */

  /* Wo soll der Hintergrund zwischengespeichert werden? Es muß
     genügend Speicher dafür vorhanden sein, der von der Tiefe der
     Bitplanes abhängt, in der die Grafiken eingezeichnet werden.

   if ((b->SaveBuffer = (WORD *)AllocMem(sizeof(SHORT) *
       wordwidth * lineheight * imagedepth,
       MEMF_CHIP | MEMF_CLEAR)) == 0)
   {
      FreeMem(b, sizeof(struct Bob));
      return(0);
   }
```

Listing 7.5: Routine zur Definition eines Bobs (Teil 1)

```
    b->ImageShadow = v->CollMask;

   /* Die Prioritäten der Bobs untereinander werden folgendermaßen
      vergeben: Der zuerst definierte Bob besitzt die niedrigste, der
      zuletzt definierte die höchste Priorität, überlagert also alle
      anderen. */


    b->Before = NULL;     /* Der Aufrufer soll sich später um die
                             Priortäten kümmern. */
    b->After = NULL;
    b->BobVSprite = v;
    /* Initmask stellt zwar nicht "imageShadow" dar, obgleich es in der
       Regel das gleiche ist. Der Benutzer könnte ja vielleicht
       "collMask" nutzen oder seine eigene Schattenversion erzeugen
       wollen. */

    b->BobComp = NULL; /* Kein Teil eines Animationsobjektes */
    b->DBuffer = NULL; /* Nicht zweifach zwischenspeichern   */

    /* Gibt einen Zeiger auf den neuen, gerade erzeugten Bob zurück,
       entweder für weitere Aktionen des Aufrufers oder aber für die
       Routine "AddBob(b)" */

    return(b);
}
/* Ende von makebob.c */
```

Listing 7.5: Routine zur Definition eines Bobs (Schluß)

Die Datenstruktur zur Darstellung eines Bobs sieht anders aus, da ein Bob aus mehr als zwei Bitplanes bestehen kann. Aus diesem Grund sind bei einem Bob alle Daten, die zu einer Bitplane gehören, in einem gesonderten Abschnitt der Datenstruktur zusammengefaßt. Der Vorteil dieser Vorgehensweise liegt darin, daß die Bitmuster, die ein Bob formen, leichter zu erkennen sind; dies ist beim Programmieren oftmals sehr hilfreich. Die Auswahl der Farbregister unterscheidet sich ebenfalls bei der Verwendung von Bobs. Durch die Kombination der Bits der verschiedenen Bitplanes ergeben sich – je nach Anzahl der Bitplanes – mehr als nur vier mögliche Kombinationen zur Farbgebung. Zur Selektion eines Farbregisters werden die Bits mit gleichen Positionen innerhalb der Bitplanes zusammengefaßt. Der resultierende Wert liefert dann die Nummer des Farbregisters.

Zur Veranschaulichung betrachten wir ein 32 Bit breites Bob, das durch folgende Datenstruktur definiert wird:

```
/* Bitplane 0 */
0x0000, 0x1111,
0xcccc, 0xeeee,
/* Bitplane 1 */
0xffff, 0x0000,
0xaaaa, 0x7777
```

Die Farbe des linken, oberen Pixels des Bobs wird hier durch die beiden linken Bits der ersten beiden Datenworte (0x0000 und 0x1111) bestimmt. Die Priorität der Bits hängt von der Nummer der Bitplane ab, in der sie angeordnet sind. Aus diesem Beispiel ergeben sich die folgenden Kombinationen für die Bits der Bitplanes 0 und 1:

Bit der Bitplane 1:	0 0 1 1
Bit der Bitplane 0:	0 1 0 1
Gewähltes Farbregister:	0 1 2 3

Die Datenfelder "PlanePick" und "PlaneOnOff"

Die Bitkombinationen, die für die Farbauswahl verwendet werden, können anhand der Werte von "PlanePick" und "PlaneOnOff" weitergehend manipuliert werden. Mit diesen Parametern ist es möglich, einem Bob exakt vier mögliche Farben zuzuordnen – aus dem gesamten Spektrum der 32 vorhandenen Farbregister.

Der Parameter "PlanePick" ist ein einzelnes Byte, mit dessen Hilfe das System die Bitplanes auswählt, die zur Darstellung eines Bobs verwendet werden sollen. Der Programmierer hat so die Möglichkeit, jede Bitkombination zur Farbauswahl zu erzeugen. Die Priorität der Bitplanes bei der Darstellung des Bobs nimmt mit steigender Nummer der Bitplane ab, d.h. es wird immer zuerst in die Bitplane geschrieben, die die niedrigste Nummer von allen ausgewählten Bitplanes besitzt.

Hat "PlanePick" z.B. den Wert 0101 (binär), dann werden die folgenden Bitplanes zur Darstellung des Bobs verwendet:

Nummer der Bitplane:	3 2 1 0
Wert von PlanePick:	0 1 0 1
Verwendete Planes des Bobs:	x 1 x 0

Das Bob wird also nur durch die Bitplanes 0 und 2 dargestellt (Plane 0 der "Image"-Datenstruktur wird für Bitplane 0 verwendet, Plane 1 für Bitplane 2). Was aber geschieht mit den Bitplanes, die durch das Setzen von PlanePick "ausgeschaltet" wurden, in unserem Beispiel also die Bitplanes 1 und 3?

Hier kommt das zweite Datenfeld – "PlaneOnOff" – zur Anwendung. Der Inhalt dieses Feldes ist ebenfalls ein Binärwert, der dem System sagt, was mit den unbenutzten Bitplanes geschehen soll. Jede nicht verwendete Bitplane wird vom System mit Null-Bits gefüllt, wenn das entsprechende Bit im Parameter "PlaneOnOff" gesetzt ist und Null-Bits in der nicht verwendeten Bitplane vorhanden sind. Sind die Bits in der Bitplane gesetzt, dann füllt das System die entsprechenden Stellen mit gesetzten Bits aus.

Das Gel-System generiert und verwendet eine Bitmaske, in der überall dort Bits gesetzt sind, wo auch beim korrespondierenden Bob gesetzte Bits vorhanden sind. Mit Hilfe dieser Bitmaske entscheidet das System, wie die Bitplanes des Darstellungsbereiches manipuliert werden, die vom Inhalt des Parameters "PlanePick" nicht beeinflußt werden.

Hat der Parameter "PlaneOnOff" z.B. den Wert 0000 (binär), dann ergeben sich die verfügbaren Farben aus den Positionen der gesetzten Bits des Objekts:

0 x 0 x (Wert des Parameters "PlaneOnOff")

x 1 x 1 (Wert des Parameters "PlanePick")

Die Bits, die in diesem Beispiel mit einem "x" gekennzeichnet sind, spielen bei der Farbauswahl keine Rolle. Aus den oben gezeigten Inhalten der Parameter ergeben sich die folgenden Farbregisterwerte:

0000 = Farbregister 0 (Hintergrundfarbe, transparent)
0001 = Farbregister 1
0100 = Farbregister 4
0101 = Farbregister 5

Sie sehen also, wie die Inhalte der Parameter kombiniert werden, um so die Farbregister auszuwählen. Hier noch ein weiteres Beispiel zur Veranschaulichung, mit einem Wert von 1010 (binär) für den Parameter "PlaneOnOff":

1010 = Farbregister 10
1011 = Farbregister 11
1110 = Farbregister 14
1111 = Farbregister 15

In diesem Beispiel wird zur Darstellung des Bobs keine transparente Farbe verwendet. "PlanePick" und "PlaneOnOff" sind 8-Bit-Werte (UBYTE). In der gegenwärtigen Hardware-Version des Amiga werden jedoch nur 6 Bits verwendet, da die maximale Anzahl verfügbarer Farben auf 32 begrenzt ist. Sie sollten mit verschiedenen Bitkombinationen experimentieren, um sich mit der Farbauswahl vertraut zu machen.

Die PurgeGels-Routine

Da durch die Definition der Bobs Speicherplatz belegt wird, müssen Sie diesen vor dem Programmende wieder freigeben, damit ihn andere Tasks wieder verwenden können. Im folgenden Beispielprogramm kommen zwei solcher Routinen zur Speicherplatzfreigabe zur Anwendung: PurgeGels und DeleteGels. Die erste Routine, PurgeGels, gibt den Speicher frei, der durch den Aufruf der Routine ReadyGels belegt wurde. DeleteGels löscht die Einträge aus der Liste aller grafischen Objekte, die das System intern verwaltet. Diese Einträge erfolgen im Programm durch die beiden Funktionen MakeBob und MakeVSprite.

In Listing 7.6 finden Sie das Programmfragment der beiden oben genannten Routinen.

```
/* purgegels.c */

/* Benutzen Sie Folgendes, um den ganzen Gels-Kram
   loszuwerden, sobald er nicht mehr benötigt wird. Sie
   müssen aber das Gels-Info mittels "ReadyGels"-Routine
   belegt haben. */

PurgeGels(g)
struct GelsInfo *g;
{
   if (g->collHandler != NULL)
      FreeMem(g->collHandler, sizeof(struct collTable));
   if (g->lastColor != NULL)
      FreeMem(g->lastColor, sizeof(LONG) * 8);
```

Listing 7.6: Die Purgegels-Routine (Teil 1)

```
    if (g->nextLine != NULL)
       FreeMem(g->nextLine, sizeof(WORD) * 8);
    if (g->gelHead != NULL)
       FreeMem(g->gelHead, sizeof(struct VSprite));
    if (g->gelTail != NULL)
       FreeMem(g->gelTail, sizeof(struct VSprite));
}


/* Den von "Makexxx" belegten Speicher freigeben. */
/* Grafikdaten und Schatten werden irgendwo anders
   freigegeben. */


DeleteGel(v)
struct VSprite *v;
{
   if (v != NULL) {
      if (v->VSBob != NULL) {
         if (v->VSBob->SaveBuffer != NULL) {
            FreeMem(v->VSBob->SaveBuffer, sizeof(SHORT) *
               v->Width * v->Height * v->Depth);
         }
         if (v->VSBob->DBuffer != NULL) {
            if (v->VSBob->DBuffer->BufBuffer != 0) {
               FreeMem(v->VSBob->DBuffer->BufBuffer, sizeof(
                     SHORT) * v->Width * v->Height *
                     v->Depth);
            }
            FreeMem(v->VSBob->DBuffer, sizeof(struct
               DBufPacket));
         }
         FreeMem( v->VSBob, sizeof(struct Bob));
      }
      if (v->CollMask != NULL) {
         FreeMem(v->CollMask, sizeof(WORD) * v->Height *
            v->Width);
      }
      if (v->BorderLine != NULL) {
         FreeMem(v->BorderLine, sizeof(WORD) * v->Width);
      }
      FreeMem(v, sizeof(struct VSprite));
   }
}


/* Ende von purgegels.c */
```

Listing 7.6: Die Purgegels-Routine (Schluß)

Die Vorteile eines "Bobs"

Wenn Sie Bobs zur Erzeugung von animierter Grafik verwenden, dann können Sie Objekte definieren, die beliebig breit und hoch sein können. Die maximale Größe eines Bobs hängt nur vom Speicherplatz ab, der Ihnen für die Darstellung zur Verfügung steht. Hier liegt der Hauptvorteil gegenüber den Simple-Sprites oder den Virtual-Sprites, die in ihrer Breite auf maximal 16 Bit begrenzt sind.

Weiterhin unterliegen Bobs nicht den Beschränkungen in der Farbauswahl, wie dies bei Virtual-Sprites oder Simple-Sprites der Fall ist. (Hier stehen nur maximal 4 Farben gleichzeitig zur Verfügung.) Die maximale Anzahl Farben hängt bei Bobs von der Anzahl der Bitplanes der zugehörigen Screens ab; wenn Sie im "Hold-And-Modify"-Modus arbeiten, können Sie Bobs erstellen, die aus bis zu 4096 verschiedenen Farben bestehen.

Ein weiterer Vorteil der Bobs ist, daß der Programmierer die Darstellungspriorität frei wählen kann; zu diesem Zweck brauchen nur zwei Zeigerwerte in den zugehörigen Datenstrukturen geändert werden. Die Priorität eines Bobs hängt also nicht von seiner laufenden Nummer ab, wie dies bei den Simple-Sprites oder Virtual-Sprites der Fall ist.

Die Nachteile eines "Bobs"

Da Bobs immer als Teil des Hintergrundes dargestellt werden, ist die Ausgabegeschwindigkeit des Systems langsamer, als dies bei den Sprites der Fall ist. Weiterhin ist es erforderlich – wie dies auch im Beispielprogramm in Listing 7.7 der Fall ist – den vom Bob verdeckten Hintergrund doppelt zwischenzuspeichern, da dieser sonst durch die Darstellung des Bobs verlorengeht. Man arbeitet also mit zwei Screens, einem zur Darstellung der Bobs und einem zur Berechnung der neuen Positionen. Im Programmablauf wird dann kontinuierlich zwischen diesen beiden Screens umgeschaltet.

Bobs benötigen mehr Speicherplatz als Virtual-Sprites oder Simple-Sprites, da sie Teil des Hintergrundes sind und nicht unabhängig von diesem dargestellt werden können. Durch das Setzen des Flags "SAVEBACK" wird der vom Bob verdeckte Hintergrund zwischengespeichert und später, nachdem das Bob seine Position geändert hat, wiederhergestellt. Für jedes Bob, das auf dem Bild-

schirm dargestellt wird, muß ein separater Speicherbereich vorhanden sein, der groß genug sein muß, den verdeckten Teil des Hintergrundes zwischenzuspeichern.

Beispielprogramm: BOBS

Wenn Sie das Beispielprogramm aus Listing 7.7 ausprobieren, dann werden Sie feststellen, daß die Ausgabe der grafischen Objekte direkt auf den Screen erfolgt; das Fenster (bzw. sein ViewPort) ist für die Ausgabe nicht relevant. Der Grund hierfür ist, daß Bobs Teil des Hintergrundes sind und nicht unabhängig von diesem produziert und dargestellt werden können.

In diesem Kapitel haben Sie alles über die Animation von Grafik auf dem Amiga erfahren; die gezeigten Beispielprogramme können Ihnen als Basis für eigene Projekte dienen.

```
/* bobdemo.c */

/* Dieses Beispiel kommt gleich nach der VSprite-Demo, um zu
   zeigen, daß das Gel-System sowohl VSprites als auch Bobs
   behandelt. Wenn Sie die beiden Beispiele nebeneinander
   legen, können Sie genau verfolgen, wo sich Bobs und
   VSprites unterscheiden. Bei beiden Beispielen sind die
   Grafikdaten identisch. Sie werden feststellen, daß Bobs
   weder einen Einfluß auf die Hintergrundfarbe, noch auf die
   Farbregister der Sprites haben, bis auf die Tatsache, daß
   sie den Hintergrund zwischenspeichern und später wieder
   herstellen.
*/

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"
#include "graphics/gels.h"
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 1)

```
/* System mit der Erzeugung und Verwaltung von MAXSP Bobs
   beauftragen */
#define MAXSP 8

/* Mögliche Geschwindigkeit für Bobs definieren (Angabe in
   Zahl der "vblank"-Interrupts). */

SHORT speed[] = { 1, 2, -1, -2 };

SHORT xmove[MAXSP], ymove[MAXSP];/* Bewegung */


/* changed for Bobs */
struct Bob *bob[MAXSP];          /* MAXSP Bobs */
struct VSprite *vspr;            /* Zeiger auf Sprite */


/* Ergänzung für Bobs */
/* Alle Kombinationen auf 2 Bit (inclusive 5) gibt an, welche
   Plane des Bobs die Farbauswahl trifft. */

BYTE pick[] = { 0x03, 0x05, 0x09, 0x11, 0x06,
                0x0c, 0x18, 0x0A, 0x12, 0x14 };


short maxgot;                    /* Maximal mögliche Bobs */

struct GelsInfo mygelsinfo;   /* Der RastPort des Windows
                                 braucht dies im Zusammenhang
                                 mit VSprites. */

struct Window *w;         /* Zeiger auf Window   */
struct RastPort *rp;      /* Zeiger auf RastPort */
struct Screen *s;         /* Zeiger auf Screen */
struct ViewPort *vp;      /* Zeiger auf ViewPort */

struct RastPort *srp;     /* Zeiger auf RastPort des Screens */

struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* 18 Worte Sprite-Daten = 9 Zeilen Sprite-Daten, jeweils in
   2 Planes */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 2)

```
UWORD bob_data[] = {
                        /* erste Plane, entspricht dem linken
                           Wort der VSprite-Grafikdaten */

                        0x0fc3,
                        0x3ff3,
                        0x30c3,
                        0x0000,
                        0x0000,
                        0x0000,
                        0xc033,
                        0xffc0,
                        0x3f03,

                        /* zweite Plane, entspricht dem
                           rechten Wort der VSprite-
                           Grafikdaten */

                        0x0000,
                        0x0000,
                        0x0000,
                        0x3c03,
                        0x3fc3,
                        0x03c3,
                        0xc033,
                        0xffc0,
                        0x3f03,
                        /* Plane 3 der Grafikdaten (in
                           Wirklichkeit bestehen nur 2 Planes
                           auf echten Daten, InitMasks, das
                           in MakeBob aufgerufen wird und
                           MakeVSprite benötigen diese
                           separate leere Plan, um die Maske
                           korrekt aufzubauen. */

                        0,0,0,0,0,0,0,0,0,
                        /* Plane 4 */
                        0,0,0,0,0,0,0,0,0,
                        /* Plane 5 */
                        0,0,0,0,0,0,0,0,0 };

UWORD *bobdata;
UWORD *sprdata;

/* Um einen Bob zu bewegen, verschiebt man das
   darunterliegende VSprite, so daß der Großteil der Routinen
   noch immer geeignet ist. */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 3)

```
movesprites()
{
   short i;

   for (i=0; i<maxgot; i++)
   {
      vspr = bob[i]->BobVSprite;        /* Für Bobs geändert */

      vspr->X = xmove[i]+vspr->X;
      vspr->Y = ymove[i]+vspr->Y;

      /* Sprites bewegen */

      if(vspr->X >= 300 || vspr->X <= 0) xmove[i]=-xmove[i];
      if(vspr->Y >= 190 || vspr->Y <= 0) ymove[i]=-ymove[i];

   }

   SortGList(srp);       /* Liste sortieren */
   DrawGList(srp, vp);   /* Spritebefehle erzeugen */

   MakeScreen(s);
   RethinkDisplay();

   return(0);
}


#define AZCOLOR 1
#define WHITECOLOR 2

#define WC WINDOWCLOSE
#de#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP)
/* Das Flag WINDOWDRAG wird nicht benutzt, da der Screen
   nicht verschoben werden soll. */

/* Erlaubt dem Benutzer die Größe des Fensters zu verändern.
   Zuerst kann es verkleinert und anschließend vergrößert
   werden, so daß der Text im Hintergrund gelöscht wird. Der
   Benutzer kann einfach erkennen, daß einige VSprites
   flackern, sobald zu viele Sprites innerhalb einer einzigen
   horizontalen Plane erscheinen, da das VSprite-System keine
   Sprites mehr zuweisen kann. */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 4)

```
/* myfont1 beschreibt die Eigenheiten des Default-Fonts. In
   diesem Falle wird der 80-Spalten-Font ausgewählt, der 40
   Spalten in niedriger Auflösung darstellt. */
 */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,       /* LeftEdge, TopEdge */
        320, 200,   /* Width, Height ... Größe des Screens */
        5,          /* Tiefe von 5 Planes, bedeutet, man hat
                       die Wahl von 2 aus 32 verschiedenen
                       Farben, die vom Screen bereitgestellt
                       werden. */
        1, 0,       /* DetailPen, BlockPen */
        SPRITES,    /* ViewModes ...
                       Wert 0 = niedrige Auflösung */
        CUSTOMSCREEN, /* Screen-Art */
        &myfont1,     /* Default-Font des Screens */
        "32 Color Test", /* DefaultTitle, Titelzeile */
        NULL,         /* User-Gadgets des Screens, immer Null
                         -> wird ignoriert */
        NULL };     /* Adresse der eigenen Bitmap für den
                       Screen, wird hier aber nicht benutzt. */


struct NewWindow myWindow = {
   0,          /* LeftEdge für das Fenster (in Pixeln) in der
                  aktuellen Auflösung von der linken Ecke des
                  Screens gerechnet. */
   0,          /* TopEdge für das Fenster (in Pixeln) von der
                  oberen Ecke des Screens gerechnet. */
   320, 185,   /* Breite und Höhe des Fensters */
   0,          /* DetailPen - Nummer des Stiftes, der zum
                  Zeichnen der Windowränder verwendet wird. */
   1,          /* BlockPen - Nummer des Stiftes, der zum
                  Zeichnen der systemeigenen Window-Gadgets */
               /* (Der Wert -1 bedeutet bei den zwei Einträgen
                  DetailPen and BlockPen, daß der Defaultwert
                  benutzt wird.) */
   CLOSEWINDOW, /* Simple-Sprite-Programm benutzt sowohl
                   INTUITICKS, als auch IDCMP-Flags */
   NORMALFLAGS | GIMMEZEROZERO | ACTIVATE | BACKDROP,
               /* Window-Flags */
   NULL,       /* FirstGadget */
   NULL,       /* CheckMark   */
   "Close-Gadget zum Abbruch anklicken",
               /* Titel des Windows */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 5)

```
   NULL, /* Pointer auf Screen, falls kein
                  Workbench-Screen */
   NULL, /* Zeiger auf BitMap, falls
                  SUPERBITMAP-Window */
   10, 10,          /* minimale Breite und Höhe */
   320, 200,        /* maximale Breite und Höhe */
   CUSTOMSCREEN
};

#include "graphics/gfxmacros.h"

/*#include "event1.c"  */
/* holt den Event-Handler */
/* event1.c */

HandleEvent(code)
   LONG code;    /* von main übergeben */
{
   switch(code)
   {
   case CLOSEWINDOW:
      return(0);
   case INTUITICKS:
      movesprites();  /* 10 Bewegungen pro Sekunde */
   default:
      break;
   }
return(1);
}

UWORD mycolortable[] = {

   0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
   0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
   0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

   0x0000, 0x0e30, 0x0fff, 0x0b40,
   0x0fb0, 0x0bf0, 0x05d0, 0x0ed0,
   0x07df, 0x069f, 0x0c0e, 0x0f2e,
   0x0feb, 0x0c98, 0x0bbb, 0x07df
   };

   /* Schwarz, Rot, Weiß, Feuerrot, Orange, Gelb, Blaßgrün,
      Grün, Hellblau, Dunkelblau, Purpur, Violett, Gelbbraun,
      Grau, Himmelblau (und alles noch einmal) */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 6)

```
UWORD colorset0[] = { 0x0e30, 0xffff, 0x0b40 };
                            /* Gleiches, wie Farben 17-19 */
UWORD colorset1[] = { 0x0bf0, 0x05d0, 0x0ed0 }; /* 21-23 */
UWORD colorset2[] = { 0x069f, 0x0c0e, 0x0f2e }; /* 25-27 */
UWORD colorset3[] = { 0x0c98, 0x0bbb, 0x07df }; /* 29-31 */
UWORD *colorset[] = {
         colorset0, colorset1,
         colorset2, colorset3 };
int choice;
char *numbers[] =
{
   "17","18","19",
   "20","21","22","23",
   "24","25","26","27",
   "28","29","30","31"
};

#include "ram:purgegels.c"
#include "ram:readygels.c"
#include "ram:makevsprite.c"
/* Ergänzung für Bobs */
#include "ram:makebob.c"

main()
{
   struct IntuiMessage *msg;
   LONG result;
   SHORT k, j, x, y, m, error;
   UWORD *src, *dest; /* Zum Kopieren der Sprite-Daten
                         ins RAM */

   GfxBase = OpenLibrary("graphics.library",0);

   IntuitionBase = OpenLibrary("intuition.library",0);
   /* Um das Programm kurz zu halten, wurde auf eine
      Fehlerbehandlung verzichtet */
   s = OpenScreen(&myscreen1);  /* Versuchen, den Screen zu
                                   öffnen */
   if(s == 0)
      {
         printf("Myscreen1 kann nicht geöffnet werden\n");
         exit(10);
      }
   myWindow.Screen = s;   /* Adresse des Screens mitteilen */

   ShowTitle(s, FALSE);   /* Screen darf nicht nach unten
                             gedraggt werden */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 7)

```
    w = OpenWindow(&myWindow);
    if(w == 0)
    {
       printf("Window konnte nicht geöffnet werden!\n");
       CloseScreen(s);
       exit(20);
    }
    vp = &(s->ViewPort);
    /* Farben für diesen Viewport setzen */

    srp = &(s->RastPort); /* Bobs direkt in den Screen
                             zeichnen*/

    LoadRGB4(vp, &mycolortable[0], 32);
    rp = w->RPort;
       /* Nun auf Nachricht von Intuition warten
        * (Task "schläft" während dieser Wartezeit)
        */

    /* Text mit den Sprite-Farben schreiben, so daß man an der
       Demo sieht, wie das System die Farben einlädt. Es sollte
       darauf geachtet werden, grundsätzlich bei Verwendung
       von VSprites auch die betreffenden Farbregister zu
       benutzen. Bobs brauchen keine dieser Farbspeicher. */

    for(j=8; j<180; j+=50)
    {
       for(k=0; k<15; k++)
       {
          Move(rp,k*20,j);
          SetAPen(rp,k+17);        /* Farben 17-31 zeigen */
          /* 16, 20, 24, 28 werden nicht von VSprites berührt,
             da sie nicht von den Hardware-Sprites benutzt
             werden. */

          Text(rp,numbers[k],2);
       }
    }
/*
    ScreenToBack(s);
*/
    /* ****************** */
    /* Bob-Demo           */
    /* ****************** */

    /* Um die gültigen Sprite-Daten aufzunehmen wird
       CHIP-Memory belegt. Notwendig, falls es auf einem
       Amiga mit RAM-Erweiterung gestartet werden soll. */
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 8)

```
bobdata = (UWORD *)AllocMem(90, MEMF_CHIP);

if(bobdata == NULL)
{
   /* Zu wenig Speicher für den Bob */
   printf("Kein freier Speicher für Bobdaten!\n");
   goto finish;
}
  /* Nun werden die Sprite-Daten in das CHIP-RAM kopiert */

src = bob_data;  /* Quelle */
dest = bobdata;  /* Ziel */

for( j=0; j<45; j++)
{
   *dest++ = *src++;
}

maxgot = 0;

/* GELS-System vorbereiten, um mit VSprites und Bobs zu
   arbeiten */

error = ReadyGels(&mygelsinfo, srp);

for(k=0; k<MAXSP; k++) /* Maximale Anzahl der BOBS */
{
   xmove[k]=speed[RangeRand(4)];
   ymove[k]=speed[RangeRand(4)];

   /* Position für Bobs einrichten */
   x = 10 + RangeRand(280);
   y = 10 + RangeRand(170);

   /* Demonstriert, daß ein Bob eine wesentlich größere
      Auswahl an Farben bietet, indem "planepick" und
      "planeonoff" benutzt werden, die andere, als die von
      VSprites benutzten anbieten. */

   /* Bob erzeugen */
   bob[k] = MakeBob( 16, 9, 5, bobdata, pick[RangeRand(10)]
                  , RangeRand(31),x,y,SAVEBACK | OVERLAY );

   /* 16 Bit breit, 9 Zeilen groß, 5 Planes tief (auch
      wenn die Grafikdaten nur 2 Panes tief sind, so
      werden doch 5 Planes für jeden gezeichneten Bob
      benötigt!). Bob-Daten im Chip-Memory unterbringen,
```

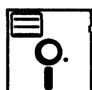
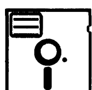
Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 9)

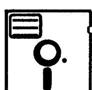
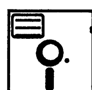
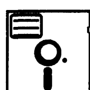
```
        plaziere die 2 Planes in zwei der fünf Bitplanes und
        fülle dann die ungebutzten Planes mit einem
        Zufallsmuster aus Einsen oder Nullen, als ob man die
        entsprechende Farbe in der Bitmaske des Bob-Objektes
        erzeugt; an Position x,y; den Hintergrund bei
        Bewegungen des Bobs zwischenspeichern und wieder
        herstellen. */

     if(bob[k] == 0)
     {
        printf("Kein Speicher mehr nach makebob\n");
        goto finish;
     }
     if(k > 0)
     {
        /* Reihenfolge für das Zeichnen vorgeben */
        m = k-1;
        bob[k]->After = bob[m]->Before;
        bob[k]->After->Before = bob[k];
     }

     AddBob(bob[k], srp);

     maxgot++;

     /* Bobs haben mit Farbpaletten "nichts am Hut" */
  }

  while(1)    /* "forever" */
  {
     WaitTOF();
     movesprites();

     result = -1;    /* Nachsehen, ob "CLOSEWINDOW" wartet */
     msg = (struct IntuiMessage *)GetMsg(w->UserPort);

     if(msg != 0)
     {
        result = HandleEvent(msg->Class);


        /* ermöglicht Intuition die Wiederverwendung
           von msg */
        ReplyMsg(msg);
     }
```

Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Teil 10)

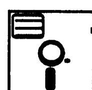
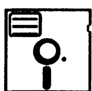
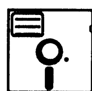
```
        if(result == 0)
        {
            break;  /* CLOSEWINDOW erhalten */
        }
    }
        /* Bevor wieder gewartet wird, muß der Port geleert
           werden */

    /* Alle erzeugten Bobs wieder freigeben */
finish:
    for(k=0; k<maxgot; k++)
    {
        if(bob[k])
        {
            DeleteGel(bob[k]->BobVSprite);
        }
    }
    /* Alles von ReadyGels Erstellte wieder löschen */

    PurgeGels(&mygelsinfo);
    if(bobdata)
    {
        FreeMem(bobdata, 90); /* Belegtes freigeben */
    }
    if(w)
    {
            CloseWindow(w);
    }
    if(s)
    {
        CloseScreen(s);
    }
}
```

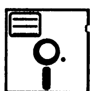
Listing 7.7: Beispielprogramm zur Anwendung von Bobs (Schluß)

8



Kapitel 8

Sound

In diesem Kapitel gehen wir genauer auf die Sound- und Klangmöglichkeiten des Amiga ein. Sie werden lernen, wie man Audio-Kanäle reserviert und wie man Daten direkt in den Audio-Registern ablegt. Um die Funktionsweise der Klangerzeugung durch die System-Software verstehen zu können, sollte man zuerst einmal einen Blick auf die Audio-Hardware des Amiga werfen.

Die Audio-Hardware

Bei vielen Anwendungen auf dem Amiga (z.B. Grafik) ist es so, daß zwischen der Hardware des Amiga und einem Programm Bausteine der Amiga-System-Software liegen, die die Anpassung der einzelnen Komponenten untereinander vornehmen. Bei der Klangerzeugung hingegen steht dem Programmierer relativ wenig System-Software zur Verfügung; um einen bestimmten Effekt hervorzurufen, muß man daher oft die zuständigen Register direkt manipulieren.

Die Sound-Hardware des Amiga besteht im wesentlichen aus vier voneinander unabhängigen Audio-Kanälen. Jeweils zwei davon sind mit einer der beiden Audio-Buchsen, die sich auf der Rückseite des Amiga befinden, verbunden. Die Kanäle 0 und 3 liefern die Ausgangsspannung der linken Buchse, die Kanäle 1 und 2 die der rechten. Jeder der vier Kanäle hat ein Lautstärkeregister, in dem Werte von 0 (min) bis 64 (max) abgelegt werden können.

Weiterhin hat jeder Kanal ein "Period"- und ein Datenregister. Mit dem Period-Register legt der Programmierer die Frequenz der aktuellen Note fest;

große Werte haben eine niedrige Frequenz zur Folge. Das Datenregister beinhaltet einen Zeiger auf den Speicherbereich, in dem die Sound-Daten für einen Kanal abgelegt sind. Beachten Sie bitte, daß ein solcher Speicherbereich in den untersten 512KB-Speicher (Chip-Mem) liegen muß, damit die CoMem-Prozessoren auf ihn zugreifen können. Die Größe des Speicherbereiches wird für jeden Kanal in einem weiteren Register abgelegt.

Jeder der vier Kanäle kann seine Daten auf zwei verschiedene Arten empfangen. Die am meisten verwendete Methode nutzt das DMA-Prinzip (DMA = Direct Memory Access, etwa: Direkter Datenzugriff), bei dem auf Daten zugegriffen wird, ohne den Prozessor zu beanspruchen. Der Amiga stellt insgesamt 26 DMA-Kanäle zur Verfügung; ein Programm kann also z.B. bereits die nächste Wellenform berechnen, während gleichzeitig auf Sound-Daten zugegriffen wird – und das alles ohne Hilfe des 68000-Prozessors.

Um Daten zu erzeugen, mit denen der Amiga einen Klang oder Musik produzieren kann, muß man einen Speicherbereich reservieren, in denen die gewünschten Tonfolgen und -lagen numerisch abgelegt werden. Wenn Sie nur mit der Hardware des Amiga arbeiten wollen, dann gestaltet sich die Klangerzeugung recht einfach: Man füllt den reservierten Speicherbereich mit den Klangdaten, setzt das Lautstärke-, Period-, Daten- und Längenregister mit den entsprechenden Werten und startet dann den DMA-Zugriff.

Die Kommunikation mit dem Audio-Device

Das Audio-Device verwendet – genau wie alle anderen Devices auch – einen "IORequest"-Datenblock, um mit einem Task kommunizieren zu können. Der verwendete Datenblock bzw. die Datenstruktur nennt man "IOAudio". Geöffnet wird das Audio-Device mit einem Aufruf der Funktion OpenDevice, der ein Zeiger auf eine "IOAudio"-Datenstruktur übergeben wird. Diese Struktur beinhaltet vor dem Aufruf nur Nullen, wird aber durch OpenDevice entsprechend initialisiert. Als Return-Wert erhalten Sie von der Funktion einen Zeiger auf den Speicherbereich, in dem die Datenstruktur ("io_device") abgelegt ist, die von fast allen Funktionen des Audio-Devices verwendet wird.

In dem Listing 8.1 finden Sie ein kleines Beispielprogramm, mit dem die "IO-Audio"-Datenstruktur initialisiert werden kann. Sie können die gezeigten

```
/* newaudio.c */

struct IOAudio *

CreateAudioIO()
{
   struct IOAudio *iob;

   iob = (struct IOAudio *) AllocMem(sizeof(struct IOAudio),
                            MEMF_CHIP | MEMF_CLEAR);

   return(iob);     /* Liefert 0, falls der Speicher
                       nicht ausreicht */
}

void FreeAudioIO(iob)
struct IOAudio *iob;
{
   FreeMem(iob, sizeof(struct IOAudio));
}
```

Listing 8.1: Routine zum Initialisieren eines IOAudio-Datenblocks

Funktionen in eigenen Programmen verwenden, um Datenstrukturen für das Audio-Device anzulegen bzw. zu initialisieren. Die Funktion CreateAudioIO liefert einen Zeiger auf einen "IORequest"-Datenblock; wie Sie diesen Datenblock verwenden, hängt von Ihren eigenen Vorstellungen ab. In Listing 8.1 finden Sie weiterhin eine kleine Routine, die den Speicherplatz wieder freigibt, der von einem "IORequest"-Datenblock belegt wird. Sie sollten darauf achten, daß vor dem Beenden eines Programms alle belegten Speicherbereiche – nicht nur die des Audio-Devices – freigegeben werden, damit andere Tasks wieder auf diesen Speicher zugreifen können.

Zum Öffnen des Audio-Device wird – wie bereits erwähnt – ein Zeiger auf den "IOAudio"-Datenblock verwendet. Das Listing 8.2 zeigt, wie das Audio-Device korrekt geöffnet wird.

Die globale Variable "auDevice" enthält nach dem Aufruf von OpenDevice den Wert des "io_Device"-Datenfeldes; Sie können diesen Wert dann einfach zur Initialisierung anderer "IOAudio"-Datenblöcke verwenden, indem Sie ihn in die entsprechenden Datenfelder kopieren.

```
int error;
struct Device *auDevice;
struct IOaudio *audioIOB;
WORD allocationKey;

audioIOB = CreateAudioIO();

/* Liefert <audioIOB> den Wert 0, dann ist nicht genug
   Speicherplatz vorhanden. In einem solchen Fall sollten
   Sie das Programm mit einer Fehlermeldung verlassen. */

error = OpenDevice("audio.device",0,audioIOB,0);
/* Ist <error> ungleich 0, dann liegt
   ebenfalls ein Fehler vor */

/* Deviceadresse speichern */
auDevice = audioIOB->ioa_Request.io_Device;

/* Key zwischenspeichern */
allocationKey = audioIOB->ioa_AllocKey;

/* An dieser Stelle kann der Rest des IOAudio-Datenblocks
   initialisiert werden und mittels SendIO,DoIO oder
   BeginIO an das Device übergeben werden. */
```

Listing 8.2: Routine zum Öffnen des Audio-Device

Wurde eine Reservierung erfolgreich durchgeführt, dann liefert das System einen sogenannten "Key" (Schlüssel), der zusammen mit dem Audio-Kanal zwischengespeichert wird. Der Wert dieses Schlüssels ist bei allen weiteren Funktionsaufrufen von großer Bedeutung; stimmt er aus irgendwelchen Gründen nicht mit dem Datenfeld "ioa_AllocKey" überein, dann wird ein Befehl vom System nicht ausgeführt. Aus diesem Grund wird in unserem Beispielprogramm der Wert des Schlüssels zwischengespeichert.

Die Audio-Software

Die System-Software des Audio-Device nimmt die Anpassung der Hardware an die Multitasking-Möglichkeiten des Amiga vor. Die Software stellt verschiedene Routinen zur Verfügung:

- Reservierung von Kanälen zur exklusiven Benutzung durch einen Task.

- Festlegung einer Priorität, so daß ein Klang höherer Priorität einen Kanal verwenden kann, auch wenn dieser bereits durch einen anderen Task belegt ist.

- Ein Task mit niedriger Priorität kann einen Kanal belegen, um die aktuelle Ein-/Ausgabeoperation zu beenden, bevor der Kanal für einen anderen Task freigegeben wird.

- Musik kann beliebig gestartet oder gestoppt werden. Weiterhin können mehrere Stücke in einer Liste vermerkt werden, so daß sie nacheinander gespielt werden.

- Es kann festgelegt werden, wie oft eine Wellenform gespielt werden soll. Ein Task kann vom System erfahren, welche Wellenformen bereits gespielt wurden bzw. welche Wellenform gerade abgearbeitet wird.

- Die Software erlaubt einen direkten Zugriff auf die Hardware-Register.

Reservierung von Kanälen

Durch die Multitasking-Fähigkeiten des Amiga ist es möglich, einen oder mehrere Kanäle für einen Task zu belegen, diese aber gleichzeitig durch andere Tasks benutzen zu lassen.

Der Amiga stellt zwei verschiedene Methoden zur Kanalreservierung zur Verfügung: durch den Aufruf von OpenDevice oder mit Hilfe der "ADCMD_ALLOCATE"-Funktion. Bei beiden Funktionen wird die Priorität der Kanäle für einen Task in einem Datenfeld des "IORequest"-Datenblocks abgelegt; der Name dieses Datenfeldes lautet "ioa_Request.io_Message.mn_Node.ln_Pri". Wenn Sie die bereits besprochene Routine CreateAudioIO verwenden, dann wird dieses Feld mit dem Wert 0 initialisiert. Nachfolgend finden Sie ein Beispiel, in dem die Priorität auf den Wert 127 gesetzt wird:

```
struct AudioIO *iob; /* Zeiger auf einen IOAudio-Datenblock */

/* Priorität setzen */
iob->ioa_Request.io_Message.mn_Node.ln_Pri = 127;
```

Der gültige Wertebereich für dieses Datenfeld reicht von –128 (Minimum) bis +127 (Maximum). Verwenden Sie das Maximum, dann erhalten Sie vom System den exklusiven Zugriff auf den entsprechenden Kanal. Hat das Datenfeld jedoch einen kleineren Wert, dann kann ein Task mit höherer Priorität diesen Kanal vom System zugeteilt bekommen, auch wenn Ihr Task die aktuelle Ein-/Ausgabeoperation noch nicht beendet hat. Meistens ist es jedoch so, daß Ihr Task vom System informiert wird, wenn ein anderer Task den Kanal belegen will.

Die Priorität eines Kanals wird beim Aufruf von OpenDevice festgelegt. Wollen Sie den Wert nachträglich ändern, dann stellt Ihnen das System die Funktion "ADCMD_SETPREC" zur Verfügung, die später in diesem Kapitel behandelt wird.

Es ist egal, wann und auf welche Art Sie Kanäle reservieren; das Ablaufschema für die Reservierung folgt immer denselben Regeln: Der Parameter "io_Data" zeigt auf das erste Byte eines reservierten Speicherbereiches, in dem die Klangdaten abgelegt wurden, "io_Length" gibt an, wie groß dieser Speicherbereich ist. Zur Kanalreservierung werden dem System vier Bits übergeben, die – je nach Kombination – festlegen, welche Kanäle belegt werden sollen. Die Bitpositionen (0 bis 3) entsprechen den vier möglichen Kanalnummern; die Tabelle 8.1 zeigt dies in einer Übersicht.

Sollen zwei Kanäle reserviert werden, dann wird dem System ein Wert übergeben, der die Kanalnummer des linken und des rechten Kanals beinhaltet. Dieser Wert wird "Allocation Mask" (Reservierungsmaske) genannt.

Kanal Nr.	Bitposition	Binärwert	Dezimalwert
0	0	0001	1
1	1	0010	2
2	2	0100	4
3	3	1000	8

Tab. 8.1: Zusammenhang zwischen Kanalnummern und Bitpositionen

Das System stellt dem Programmierer bei der Auswahl der Kanäle einen großen Freiraum zur Verfügung. Sind z.B. die Kanäle 0 und 1 bereits belegt, dann kann man dem System mitteilen, daß keine anderen Kanäle verwendet werden sollen. Ebenso ist es möglich, bei einem einzigen Funktionsaufruf das System anzuweisen, mehrere Kanalkombinationen daraufhin zu überprüfen, ob sie bereits von anderen Tasks belegt sind. Ist dies nicht der Fall, dann wird die entsprechende Kombination für den eigenen Task reserviert. Ein solcher Aufruf kann z.B. so lauten: "Reserviere Kanal 0 und 1 oder 0 und 2 oder 3 und 1 oder 3 und 2". Jede dieser Kombinationen führt bei der Reservierung zu einer späteren Stereowiedergabe, d.h. es wird jeweils ein linker und ein rechter Kanal zur Wiedergabe verwendet. Um eine Kanalreservierung der oben gezeigten Form durchzuführen, wird ein 4-Byte-Datenfeld verwendet, wobei jedes einzelne Byte eine der vier Kanalkombinationen darstellt. In der Tabelle 8.2 ist dies in einer Übersicht dargestellt.

Nach Initialisierung des "IORequest"-Datenblocks gibt das Datenfeld "io_Unit" darüber Auskunft, welche Kanäle für Ihren Task vom System reserviert wurden. Das Beispielprogramm im Listing 8.3 verwendet den "ADCMD_ALLOCATE"-Befehl (nachdem das Device geöffnet wurde), um entweder einen oder zwei (Stereo-) Kanäle zu reservieren. Jede Routine des Programms liefert als Resultat einen Wert, der die reservierten Kanalnummern beinhaltet.

Beachten Sie bitte, daß jeder Kanal, den Ihr Task reserviert, vor dem Programmende wieder freigegeben werden muß, damit andere Programme diese dann weiterverwenden können. Geben Sie einen Kanal nicht frei, dann geht das System davon aus, daß er weiterhin von Ihrem Programm verwendet wird; bis zum Neustart des Systems kann kein anderes Programm diesen Kanal verwenden.

Kanäle	**Wert (binär)**	**Wert (dezimal)**
0, 1	0011	3
0, 2	0101	5
3, 1	1010	10
3, 2	1100	12

Tab. 8.2: Werte zur Reservierung von Stereokanälen

Reserviert das System einen oder mehrere Kanäle bei einem einzigen Funktionsaufruf, dann liefert es weiterhin einen Wert – den "Key" –, der von Ihnen für weitere Zugriffe auf die Kanäle angegeben werden muß. Jeder Kanal verwaltet eine Liste, in der Ein-/Ausgabeoperationen eingetragen werden. Erreicht ein solcher Eintrag ("IORequest") das obere Ende der Liste, dann wird das Datenfeld "ioa_AllocKey" mit einem Wert verglichen, den das Audio-Device intern verwaltet. Anhand dieses Wertes bestimmt das Audio-Device, welcher Kanal gerade von welchem Task belegt ist.

Jedesmal, wenn ein Kanal reserviert oder freigegeben wird, berechnet das System den "Key" neu. Sendet ein Task einen "IORequest"-Datenblock an das Audio-Device, dann wird die Ein-/Ausgabeoperation nur dann durchgeführt, wenn der Wert des Key-Datenfeldes dem der internen Device-Variablen entspricht. In allen anderen Fällen wird der Datenblock mit einem Fehlerstatus an den Task zurückgesendet. Aus diesem Grunde benötigen die Routinen aus dem Listing 8.3 einen initialisierten "IOAudio"-Datenblock (es reicht, wenn die Datenfelder "io_Device" und "mn_ReplyPort" eindeutig definiert werden) sowie die Adresse der Variablen, in der der Key-Wert abgelegt werden kann. Alle Funktionen, die mit den Audio-Kanälen arbeiten, müssen einen korrekten Key-Wert übergeben, da sonst der gewünschte Befehl vom System nicht ausgeführt wird.

Kanäle können – wie bereits erwähnt – schon beim Aufruf der Funktion OpenDevice reserviert werden. Bei dieser Methode werden die gleichen Schritte wie im vorangegangenen Programm durchgeführt, allerdings braucht der Befehl "ADCMD_ALLOCATE" nicht ausgeführt zu werden. In Listing 8.4 finden Sie ein Beispielprogramm, das einen Kanal während des Aufrufs von OpenDevice reserviert und als Return-Wert die Nummer des reservierten Kanals liefert.

Beachten Sie bitte, daß Sie verschiedene Werte aus dem Datenblock des Device auslesen können, sobald es erfolgreich geöffnet wurde. Einen Zeiger auf das Device erhalten Sie durch das Auslesen des "io_Device"-Datenfeldes; den Wert der Key-Variablen finden Sie im "ioa_AllocKey"-Datenfeld. Um zu erfahren, welche Kanäle vom System reserviert wurden, können Sie das Feld "io_Unit" auslesen, in dem die Bitmaske der reservierten Kanäle abgelegt ist.

Um weitere Kanäle zu reservieren, nachdem das Device geöffnet wurde, wird der Befehl "ADCMD_ALLOCATE" verwendet; es ist nicht erforderlich, das Device mehrere Male zu öffnen.

```
/* getaudio.c */

WORD mykey;       /* Globale Variable, beinhaltet den "Key" */

GetAnyStereoPair(iob)
struct IOAudio *iob;
{
   /* Vor dem Aufruf der Routine müssen "iob" und "ReplyPort"
      bereits im Datenblock des Devices initialisiert worden
      sein. */

   UBYTE stereostuff[4];
   UBYTE mychan;
   int error;

   stereostuff[0] = 3;
   stereostuff[1] = 5;
   stereostuff[2] = 10;
   stereostuff[3] = 12;

   /* Priorität */
   iob->ioa_Request.io_Message.mn_Node.ln_Pri = 20;
   /* Befehl */
   iob->ioa_Request.io_Command = ADCMD_ALLOCATE;
   /* Zeiger auf Datenfeld */
   iob->ioa_Data = (UBYTE *)stereostuff;
   /* 4 Einträge */
   iob->ioa_Length = 4;

   /* Das Flag ADIOF_NOWAIT besagt, daß nicht auf die
      Reservierung von Kanälen gewartet werden soll, da wir
      davon ausgehen, daß die gewünschten Kanäle zur Verfügung
      stehen. */

   iob->ioa_Request.io_Flags = ADIOF_NOWAIT | IOF_QUICK;

   BeginIO(iob);

   error = WaitIO(iob);  /* error ungleich 0: Fehler */
```

Listing 8.3: Die getaudio-Routinen (Teil 1)

```
   if(!(iob->ioa_Request.io_Flags & IOF_QUICK))
      /* Ist das Flag nicht gesetzt, dann wurde die Nachricht
         an den Reply-Port angehängt. */
      GetMsg(iob->ioa_Request.io_Message.mn_ReplyPort);

   if(error) return(0);

   mychan = ((ULONG)(iob->ioa_Request.io_Unit)) & 0xFF;
   return(mychan);
}


GetAnyChannel(iob)
struct IOAudio *iob;
{
   UBYTE anychan[4];
   UBYTE mychan;
   int error;

   anychan[0] = 1;
   anychan[1] = 2;
   anychan[2] = 4;
   anychan[3] = 8;

   iob->ioa_Request.io_Message.mn_Node.ln_Pri = 20;
   iob->ioa_Request.io_Command = ADCMD_ALLOCATE;
   iob->ioa_Data = (UBYTE *)anychan;
   iob->ioa_Length = 4;
   iob->ioa_Request.io_Flags = ADIOF_NOWAIT | IOF_QUICK;

   BeginIO(iob);

   error = WaitIO(iob);

   if(!(iob->ioa_Request.io_Flags & IOF_QUICK))
      GetMsg(iob->ioa_Request.io_Message.mn_ReplyPort);

   if(error) return(0);

   mychan = ((ULONG)(iob_Request.io_Unit)) & 0xFF;
   return(mychan);
}
```

Listing 8.3: Die getaudio-Routinen (Schluß)

Auf die Reservierung von Kanälen warten

Bei allen Arten der Reservierung "schläft" Ihr Task so lange, bis die von Ihnen gewünschte Reservierung vom System vorgenommen werden konnte. Jedesmal, wenn ein Kanal freigegeben wird, prüft das Audio-Device alle noch anstehenden "IORequest"-Datenblöcke und versucht, dem Datenblock, der die höchste Priorität hat, die angeforderten Kanäle zur Verfügung zu stellen. Können die Kanäle nicht bereitgestellt werden, dann wird der Task vom System wieder "auf Eis" gelegt.

Wenn Sie vermeiden wollen, daß Ihr Task bis zur erfolgreichen Reservierung wartet, dann können Sie das Flag "ADIOF_NOWAIT" im "ioa_Flags"-Datenfeld setzen. Enthält dieses Flag einen Wert ungleich 0 und kann die Reservierung nicht durchgeführt werden, dann wird der Datenblock, den Ihr Task an das Audio-Device gesendet hat, zurückgeschickt. Sie erhalten in diesem Fall im Datenfeld "ioa_Request.io_Error" die Fehlernummer "IOERR_ALLOC-

```
/* openanyaudio.c */

BYTE
OpenAnyAudio(iob) struct IOAudio *iob;
{
   int error;
   BYTE anychan[4];
   BYTE mychannel;

   anychan[0] = 1;
   anychan[1] = 2;
   anychan[2] = 4;
   anychan[3] = 8;

   iob->ioa_Data = (UBYTE *)anychan;
   iob->ioa_Length = 4;

   error = OpenDevice("audio.device",0,iob,0);
   if(error != 0) return((BYTE)0);

   else return(((ULONG)(iob->ioa_Request.io_Unit)) & 0xFF);
}
```

Listing 8.4: Routinen zum Öffnen des Audio-Device

FAILED". Kann das System einen Kanal nicht reservieren, dann liefern alle Funktionen, die mit den Audio-Kanälen arbeiten, den Wert 0. Auch hier bekommen Sie als Fehlernummer den Wert "IOERR_ALLOCFAILED".

Nachfolgend ein Beispiel zum Setzen des "ADIOF_NOWAIT"-Flags:

```
iob->ioa_Request.io_Flags = ADIOF_NOWAIT;
```

Die Verwendung von reservierten Kanälen

Wenn Sie das Audio-Device öffnen, dann wird das "ioa_Request.io_Device"-Datenfeld des "IOAudio"-Datenblocks nach dem Aufruf der Funktion OpenDevice initialisiert.

Beim Reservieren von Kanälen – entweder durch OpenDevice oder später durch "ADCMD_ALLOCATE" – wird außerdem das "io_Unit"-Datenfeld mit der Bitmaske der reservierten Kanäle gefüllt. Das System generiert daraufhin den bereits besprochenen Key-Wert und legt diesen im Feld "ioa_AllocKey" ab. Wenn Sie die Routine CreateAudioIO verwenden, um mehrere "IORequest"-Datenblöcke zu erstellen, dann müssen Sie alle drei Werte – "io_Device", "io_Unit" und "ioa_AllocKey" – von einem bereits initialisierten Datenblock umkopieren, damit die entsprechenden Kanäle diese identifizieren können.

Haben Sie jedoch einen Stereokanal (zwei Kanäle) reserviert, dann müssen Sie den Wert von "io_Unit" entsprechend der Kanalnummer ändern; Befehle werden also separat an jeden der beiden Kanäle übergeben. Wenn Sie z.B. die Kanäle 1 und 2 reserviert haben, dann erhalten Sie vom System den Wert 0110 (binär) im "io_Unit"-Datenfeld. Da Befehle aber getrennt übermittelt werden, wird dieser Wert von Ihnen entsprechend der Kanalnummer geändert, d.h. um einen Befehl an den Kanal 1 zu übermitteln, wird der Wert 0010 an "io_Unit" übergeben; für Kanal 2 ändern Sie den Wert auf 0100. Vom Device erhalten Sie jedoch jedesmal den Return-Wert 0110 im "io_Unit"-Datenfeld; das System zeigt Ihnen so an, daß beide Kanäle angesprochen wurden. Für weitere Befehle an die Kanäle wird der Return-Wert dann entsprechend geändert.

Wenn Ihr Task nicht auf die erfolgreiche Reservierung von Kanälen warten soll, dann verwenden Sie das Flag "ADIOF_NOWAIT"; hierbei sollten Sie aber immer den Fehlerstatus überprüfen, damit Sie wissen, ob ein Kanal vom System reserviert wurde. Tun Sie dies nicht, dann ist ein Systemabsturz die Folge, wenn Sie einen Kanal ansprechen, der nicht reserviert wurde.

Das "Abschließen" von Kanälen

Durch das Setzen der Priorität im "AudioIO"-Datenblock verhindern Sie, daß ein Task mit kleinerer oder gleicher Priorität den von Ihnen reservierten Kanal verwenden kann. Wenn Sie den exklusiven Zugriff auf einen Kanal haben wollen, dann sollten Sie die Priorität 127 (Maximum) verwenden. In einem solchen Fall "gehört" der Kanal bis zur Freigabe durch Ihren Task Ihnen. Wollen Sie hingegen Ihren Kanal auch anderen Tasks zur Verfügung stellen, dann können Sie ihn für den von Ihnen beanspruchten Zeitraum "abschließen" – so wie dies auch bei Dateien unter AmigaDOS möglich ist.

Wenn ein anderer Task (mit hoher Priorität) einen Kanal anfordert, der gerade von Ihrem Programm verwendet wird, dann haben Sie zwei Möglichkeiten: Sie können den Kanal direkt freigeben, d.h. Ihre aktuelle Ein-/Ausgabeoperation wird sofort abgebrochen; wollen Sie dies vermeiden, dann können Sie den Kanal bis zum Ende der laufenden Ein-/Ausgabe "abschließen". Wählen Sie die zweite Methode, dann sendet das System eine Nachricht an Ihren Task und fordert ihn auf diese Weise auf, die Ein-/Ausgabe so schnell wie möglich zu beenden und den Kanal freizugeben. Verwenden Sie in Ihren Programmen nur die Standard-Ein-/Ausgabebefehle, dann wird es jedoch nicht nötig sein, einen Kanal abzuschließen.

Wenn Ihr Task direkt die Hardware-Register manipuliert, sollten Sie die Möglichkeit des Abschließens nutzen; Sie stellen auf diese Weise sicher, daß die Register und der Kanal korrekt freigegeben werden, d.h. ein anderer Task kann den freien Kanal ohne Zeitverzögerungen weiterverwenden. Ändern Sie hingegen die Registerwerte, ohne den Kanal abgeschlossen zu haben, dann kann es unter Umständen passieren, daß ein anderer Task ebenfalls versucht, die von Ihnen verwendeten Register zu manipulieren, was meistens einen Systemabsturz zur Folge hat.

Normalerweise wird ein Kanal direkt abgeschlossen, nachdem er vom System zugeteilt wurde. Kann er nicht abgeschlossen werden, dann wurde der Kanal bereits von einem anderen Task höherer Priorität reserviert. Im anderen Fall haben Sie exklusiven Zugriff auf diesen Kanal, selbst dann, wenn er von einem anderen Task angefordert wird.

In Listing 8.5 finden Sie ein Beispielprogramm, in dem das Abschließen eines Kanals demonstriert wird. Vor dem Aufruf der Routine müssen jedoch bereits einige Werte und Datenfelder initialisiert worden sein; der "IOAudio"-Datenblock muß den Wert des "Keys" und die Variable "auDevice" enthalten. Wei-

terhin wird zur Durchführung von Ein-/Ausgabeoperationen ein ReplyPort benötigt, das Datenfeld "ioa_Request.ReplyPort" muß also einen Zeiger auf einen solchen ReplyPort enthalten.

Beachten Sie bitte, daß der "IOAudio"-Datenblock, den wir in diesem Beispiel verwenden, vom Device nicht an unseren Task zurückgesendet wird, bis wir den Kanal freigeben oder ein anderer Task den Kanal beansprucht. Hier liegt auch ein weiterer Vorteil der CreateAudioIO-Funktion: Mit ihrer Hilfe können mehrere "IOAudio"-Datenblöcke erstellt werden, um mit dem Audio-Device kommunizieren zu können.

```
/* lockchan.c */

LockAChannel(lockiob,channel)
struct IOAudio *lockiob;
UBYTE channel;
{
   /* Zuerst wird dem System mitgeteilt, welcher Kanal
      reserviert werden soll. */

   lockiob->ioa_Request.io_Unit = (struct Unit *)channel;
   lockiob->ioa_Request.io_Command = ADCMD_LOCK;
   lockiob->ioa_Request.io_Flags = 0;

   /* Der übermittelte IORequest-Datenblock wird vom Device
      nicht zurückgesendet, bis unser Task den Kanal
      freigibt oder ein anderer Task den Kanal anfordert. */

   BeginIO(lockiob);
}

FreeAChannel(iob)
struct IOAudio *iob;
{
   /* Die oben erwähnten Datenfelder müssen bereits
      initialisiert sein */

   iob->ioa_Request.io_Command = ADCMD_FREE;
   iob->ioa_Request.io_Flags = IOF_QUICK;
   BeginIO(iob);
   WaitIO(iob);
}
```

Listing 8.5: Routine zum "Abschließen" eines Audio-Kanals

Setzen einer neuen Priorität

Bei vielen Anwendungen – z.B. bei Spielen – hat man einige "wichtige" Klänge und andere, die nur ab und zu eingespielt werden, z.B. Hintergrundgeräusche. Es ist daher wünschenswert, einen Kanal so lange exklusiv zu reservieren, bis ein "wichtiger" Klang komplett ausgegeben wurde; mit anderen Worten: Die Reservierung eines Kanals durch einen Task mit hoher Priorität soll für die Dauer der aktuellen Ein-/Ausgabeoperation unterbunden werden.

Mit dem Befehl "ADCMD_SETPREC" haben Sie die Möglichkeit, einem Kanal, der von Ihrem Task reserviert wurde, nachträglich eine andere Priorität zu verleihen; Sie können auf diese Weise temporär einen exklusiven Zugriff auf den Kanal erlangen, indem Sie seine Priorität auf den Wert 127 setzen.

In Listing 8.6 finden Sie eine Routine, die das Setzen dieses Wertes innerhalb eines Programms veranschaulicht. Beachten Sie bitte, daß die Priorität eines Kanals während des Aufrufs der OpenDevice-Funktion vom System gesetzt wird.

```
/* setprec.c */

SetChanPrecedence(iob,channel,prec)
struct IOAudio *iob;
BYTE channel;
BYTE prec;
{
   iob->ioa_Request.io_Command = ADCMD_SETPREC;
   iob->ioa_Request.io_Unit     = channel;
   iob->ioa_Request.io_Message.mn_Node.ln_Pri = prec;
   BeginIO(iob);
   WaitIO(iob);    /* Auf Ausführung warten */
   return(iob->ioa_Request.io_Error);
   /* Liefert 0, wenn kein Fehler auftrat */
}
```

Listing 8.6: Dynamisches Ändern der Kanalpriorität

Kontrolle der Audio-Ausgabe

Sie wissen jetzt, wie man Audio-Kanäle reserviert und wieder freigibt; in den folgenden Abschnitten werden Sie lernen, wie man diesen Kanälen Daten übermittelt, um auf diese Weise Klänge und Musik zu erzeugen. Der Amiga stellt Ihnen eine Reihe von Befehlen zur Verfügung, um mit dem Audio-Device arbeiten zu können; zwei Befehle dienen zur Synchronisierung zwischen Ihrem Task und der Wiedergabe durch die Audio-Kanäle. Nachfolgend finden Sie die Befehle zur Ausgabekontrolle aufgelistet.

CMD_WRITE — Mit diesem Befehl wird das Audio-Device veranlaßt, eine zuvor definierte Wellenform auszugeben bzw. diese in der Liste einzutragen, falls gerade eine andere abgearbeitet wird. Mit diesem Befehl können neue Werte für die Lautstärke und die Tonhöhe übergeben werden; fehlen diese Angaben, dann verwendet das System die Werte der zuvor gespielten Wellenform.

ADCMD_PERVOL — Mit diesem Befehl kann die Lautstärke und/oder die Tonhöhe der aktuellen Wellenform geändert werden. Die Änderung kann direkt erfolgen oder nachdem ein Durchlauf der Wellenform erfolgt ist.

ADCMD_FINISH — Hiermit wird die aktuell ausgegebene Wellenform gestoppt; auch hier kann der Programmierer entscheiden, ob dies sofort geschehen soll oder erst nach dem Ende der Wellenform. Dies wird durch das Flag "ADIOF_SYNCCYCLE" bestimmt, das eine Kombination aus "ADCMD_PERVOL" und "ADCMD_FINISH" darstellt. Ist das Flag gesetzt, dann wird die Ausgabe erst gestoppt, wenn die Wellenform ihr Ende erreicht hat.

CMD_STOP — stoppt die Ausgabe auf einem Kanal.

CMD_START — startet die Wiedergabe an der Stelle, an der sie durch "CMD_STOP" abgebrochen wurde.

CMD_FLUSH — löscht alle Einträge aus der Liste der noch zu bearbeitenden Datenblöcke.

CMD_RESET — initialisiert alle Hardware-Register und Vektoren mit vom System voreingestellten Werten. Dieser Befehl führt selbständig die Befehle "CMD_FLUSH" und "CMD_ START" durch, gibt jedoch keine abgeschlossenen Kanäle frei; die Freigabe muß weiterhin durch den entsprechenden Task erfolgen.

Nachfolgend die Befehle zur Synchronisierung:

CMD_READ — liefert einen Zeiger auf den "IOAudio"-Datenblock; auf diese Weise teilt das System mit, daß ein bestimmter Kanal gerade Klangdaten bearbeitet.

ADCMD_WAITCYCLE — wartet so lange, bis die aktuelle Ein-/Ausgabe mittels "CMD_WRITE" beendet wurde. (Dieser Befehl wird mit DoIO übergeben).

Wenn Sie eine Kombination aus "CMD_READ" und "ADCMD_WAITCYCLE" verwenden, dann können Sie Ihren Task mit der Klangwiedergabe synchronisieren. Dies ist dann sinnvoll, wenn Sound und Grafik gleichzeitig eingesetzt werden; Sie können so feststellen, ob ein Grafikobjekt z.B. den Bildschirmrand berührt, und dann ein entsprechendes Geräusch erzeugen.

Audio-Daten

Um Klänge erzeugen zu können, die Ihren Vorstellungen entsprechen, müssen Sie wissen, wie man diese Klangstrukturen in Daten umsetzt und wie diese Daten von der Hardware verarbeitet werden. Für die Ausgabe auf einen Audio-Kanal müssen drei Dinge definiert werden:

- Die Wellenform
- Die Frequenz (Tonhöhe)
- Die Lautstärke

Die Definition der Wellenform

Die Definition einer Wellenform erfolgt durch eine Anzahl von Werten, die in ihrer Gesamtheit den hörbaren Klang ergeben. Der folgende Abschnitt verwendet als Beispiel eine Wellenform, wie sie in Abb. 8.1 dargestellt ist.

Um diese Wellenform erzeugen zu können, benötigt man eine gerade Anzahl von positiven und negativen Werten, die den Grundcharakter der Wellenform wiedergeben können. Das beste Resultat erzielt man, wenn man den gesamten Wertebereich der Hardware ausnutzt (–127 bis +127).

Die Tabelle 8.3 zeigt für definierte Zeitintervalle die entsprechenden Werte an diesem Punkt der Wellenform. Auf diese Weise erhalten wir acht Werte, die die markanten Punkte der Wellenform wiedergeben.

Diesen Vorgang, in dem eine Wellenform in einzelne Werte eines definierten Bereiches umgewandelt wird, nennt man Analog-Digital-Wandlung.

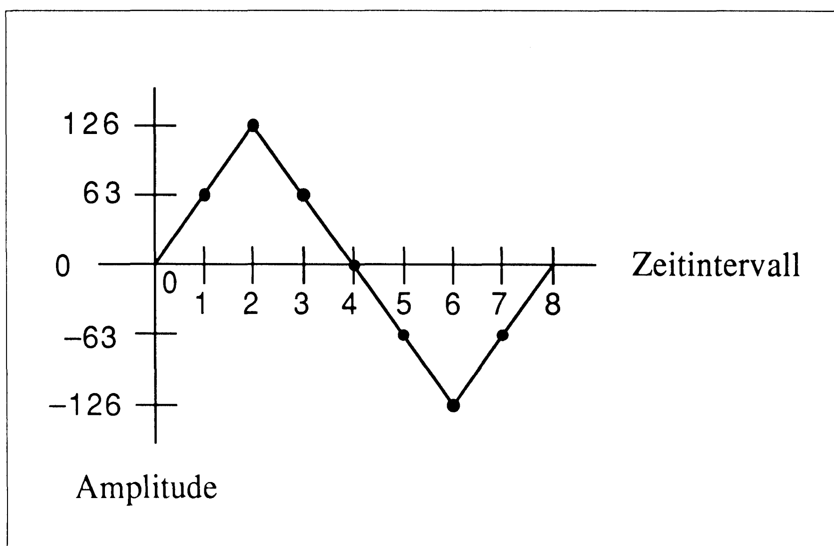


Abb. 8.1: Eine Audio-Wellenform

Die Tonhöhe (Sampling Period)

Die Sound-Hardware des Amiga übergibt die Werte der Wellenform einem Digital-Analog-Wandler, der aus den gelieferten Werten wieder die ursprüngliche Wellenform herstellt. Weiterhin werden von der Hardware fehlende Werte errechnet, da ja zwischen den übergebenen Werten große Sprünge liegen können.

Die "Sampling Period" bestimmt bei diesem Vorgang, mit welcher Geschwindigkeit die Daten verarbeitet werden. Je größer die Geschwindigkeit, desto öfter werden die Daten der Wellenform ausgegeben; es wird also eine Note mit hoher Frequenz erzeugt. Ist die Geschwindigkeit klein, dann erzeugt die Hardware eine tiefe Note, da ja auch die Frequenz klein ist.

Die Ausgabegeschwindigkeit – die Frequenz – wird beim Amiga nicht durch einen absoluten Wert repräsentiert, sondern sie wird in Form eines Divisors übergeben. Ein interner Zeitgeber liefert für Audio-Anwendungen einen bestimmten Wert, der dann durch den Wert des "Period"-Registers geteilt wird, um so die eigentliche Wiedergabefrequenz zu erhalten. Wollen Sie eine Note mit hoher Frequenz erzeugen, dann muß der Divisor einen kleinen Wert haben; soll eine Note mit tiefer Frequenz ausgegeben werden, hat das Period-Register einen großen Wert – die Werte zur Bestimmung der Tonfrequenz verhalten sich also umgekehrt proportional zueinander.

Der kleinste Wert, der dem Period-Register übergeben werden kann, ist 126. Dieses Minimum wird seitens der DMA-Hardware auferlegt. Im "Amiga Hardware Manual" finden Sie nähere Informationen zu diesen Begrenzungen.

Zeitintervall Nr.	Amplitude der Wellenform
0	0
1	63
2	126
3	63
4	0
5	–63
6	–126
7	–63

Tab. 8.3: Daten der Wellenform aus Abb. 8.1

Audio-Ausgabe löschen

Sie können dem System mittels "CMD_WRITE" mitteilen, wie oft eine Wellenform ausgegeben werden soll. In diesem Fall wird die gleiche Tabelle mit den digitalen Daten viel früher gelesen, bevor ein neuer "CMD_WRITE" auftaucht. Um das Knacken bei der Ausgabe zu verhindern, sollte man die Wellenform so festlegen, daß ein weicher Übergang zwischen den Wellen möglich ist. Dies gilt natürlich auch bei der Benutzung zweier verschiedener Wellenformen.

Der bestmögliche Übergang zwischen zwei Arten der Wellenformen ist im Nulldurchgang möglich. Das passiert genau dann, wenn die Wellenform bei einem Wert von Null beginnt und auch wieder endet. So ist es einfach, die verschiedensten Wellenformen miteinander zu verbinden, da jede bei Null endet, wenn die folgende dort beginnt.

Wenn beide Wellenformen beispielsweise mit einem Wert von 37 enden und starten, wird nicht zwangsläufig eine Verzerrung auftreten, solange nur der Verlauf der beiden Wellenformen ab dem betreffenden Punkt identisch ist. (Beispiel: Beide Wellenformen bewegen sich in den negativen Bereich, wobei die zweite dicht an dem Verlauf der ersten Wellenform liegt.) Der Wechsel am Nulldurchgang ist aber immer noch der beste Weg, da zu diesem Zeitpunkt der Audioausgang auf 0 Volt liegt und dadurch ein Minimum an Energie vorliegt. Das verhindert das Klicken, wenn die Wellenformen umgeschaltet werden.

Die Plazierung der Audiodaten

Die Audiodaten müssen sich im Speicher befinden, auf den die Custom-Chips zugreifen können. Das bedeutet, daß man zuerst Speicher des Typs "MEMF_CHIP" reservieren muß und in diesen dann die Daten schreibt. Der Speicher außerhalb des Zugriffs der Custom-Chips kann deshalb nicht von der Audio-Hardware gelesen werden.

Die Audiodaten müssen ein Vielfaches von zwei Bytes sein, da jeder Zugriff der Audio-Hardware über DMA zwei Bytes transportiert. Weiterhin muß das erste Byte stets an einer geraden Adresse plaziert sein. Um die Ausrichtung des Speichers braucht man sich keine Gedanken zu machen, wenn man die Funktion AllocMem() benutzt, da der gelieferte Speicherplatz immer auf geraden Adressen ausgerichtet wird.

Nun folgt die Zeile, mit deren Hilfe man Speicher für die Audio-Wellenform belegt:

```
BYTE *audata;
audata = (BYTE *)AllocMem(8, MEMF_CHIP);
/* Es werden 8 Bytes gebraucht */
```

Wenn der gelieferte Wert für "audata" nicht Null ist, können in diesen Bereich die Audiodaten kopiert werden, die dann von den Custom-Chips zur Ausgabe verwendet werden.

Lautstärkeregelung

Bei der Vorbereitung der Datentabelle für die Audioausgabe mußte die Erscheinung der Wellenform festgelegt werden. Außerdem ist anzugeben, wie laut der Ton sein soll. Jeder Kanal besitzt ein eigenes Lautstärke-Register, die einen Wert zwischen 0 (Minimum) und 64 (Maximum) annehmen können. Dieser Wert regelt linear die Lautstärke des betreffenden Kanals.

Alternativ kann die Lautstärke auch auf das Maximum eingestellt werden, und die genaue Abstufung erfolgt an einem externen Regler, der den wirklichen Pegel des Sounds bestimmt.

Das Audio-Programm

Das Listing 8.7 ist ein Audio-Programm, das alle vier Ausgabekanäle zum Spielen der Wellenformen nutzt. Dabei wird mit vier verschiedenen Frequenzen gleichzeitig operiert.

Das Programm schreibt die Informationen direkt in die dafür vorgesehenen Audio-Register. Darunter ist auch der Befehl, den Kanal zu reservieren, bevor man ihn zu benutzen versucht.

```
/* audio.c */

#include "exec/types.h"
#include "exec/memory.h"
#include "devices/audio.h"

extern struct IOAudio *CreateAudio(); /* fwd-Deklaration */

BYTE trianglewave[8] = { 0, 63, 126, 63, 0, -63, -126, -63 };

UWORD period[4] = { 508, 428, 339, 254 };

struct Device *auDevice=0;
BYTE *chipaudio;
struct IOAudio *audioIOB[4];
struct IOAudio *aulockIOB[4];
struct IOAudio *aufreeIOB[4];

extern struct MsgPort *CreatePort();
struct MsgPort *auReplyPort, *auLockPort;
extern struct IOAudio *CreateAudioIO();
extern UBYTE GetAnyChannel();
extern void FreeAudioIO();

main()
{
   int error,i,chan;
   BYTE *s, *d;
   struct Unit *auunit;

   for(i=0; i<4; i++)
   {
      audioIOB[i]  = CreateAudioIO();
      aulockIOB[i] = CreateAudioIO();
      aufreeIOB[i] = CreateAudioIO();

      if( audioIOB[i] == 0 ||
         aufreeIOB[i] == 0 ||
         aulockIOB[i] == 0)
      {
         finishup("Kein Speicher frei!");
      }

   }

   chipaudio = (BYTE *)AllocMem(8, MEMF_CHIP);
   if(chipaudio == 0)
```

Listing 8.7: Das Audio-Programm (Teil 1)

```
{
      finishup("Kein Speicher frei!");
}
d = chipaudio;  s = trianglewave;

for(i=0; i<8; i++)
{
   *d++ = *s++;     /* In's Chip-Memory kopieren */
}

error = OpenDevice("audio.device",0,audioIOB[0],0);

if(error)
{
   finishup ("Audio-Device läßt sich nicht öffnen!");
}
/* Device-Address für späteren Gebrauch merken */

auDevice = audioIOB[0]->ioa_Request.io_Device;

/* Ports für Antworten des Device erzeugen */

auReplyPort = CreatePort(0,0);
auLockPort  = CreatePoequest.io_Device;

if(auReplyPort == 0 || auLockPort == 0)
{
   finishup("Port kann nicht erzeugt werden!");
}

for(i=0; i<4; i++)
{
   /* Port-Adresse und Devicefelder für alle
      Audio-Request-Blocks initialisieren
   */
   audioIOB[i]->ioa_Request.io_Device = auDevice;
   aulockIOB[i]->ioa_Request.io_Device = auDevice;

   audioIOB[i]->ioa_Request.io_Message.mn_ReplyPort
         = auReplyPort;
   aulockIOB[i]->ioa_Request.io_Message.mn_ReplyPort
         = auLockPort;
}
for(i=0; i<4; i++)
{
   chan = GetAnyChannel(audioIOB[i]);
```

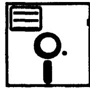
Listing 8.7: Das Audio-Programm (Teil 2)

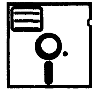
```
        printf("Kanal %ld erhalten\n",chan);

        /* Allocation-Keys zurechtbiegen */

        aulockIOB[i]->ioa_AllocKey = audioIOB[i]->ioa_AllocKey;

        /* dto. mit Unit-Numbers */
        auunit = audioIOB[i]->ioa_Request.io_Unit;
        aulockIOB[i]->ioa_Request.io_Unit = auunit;
              LockAChannel(aulockIOB[i],chan);

        /* Liefert CheckIO den Wert "wahr", so bedeutet dies,
           daß der Request zurückgeben wurde. Der Kanal wurde
           "gestohlen". */

        if(CheckIO(aulockIOB[i]))
        {
           finishup("Der Kanal wurde gestohlen!");
        }

     }
     for(i=0; i<4; i++)
     {
        /* Unter der Annahme, daß nichts gestohlen wurde, wird
           der Kanal eingerichtet und ein "Ausgabe-Antrag"
           gestellt. */
        audioIOB[i]->ioa_Data   = (UBYTE *)chipaudio;
        /* 4 WORDS in der Tabelle */
        audioIOB[i]->ioa_Length = 8/2;
        audioIOB[i]->ioa_Period = period[i]; /* aus Tabelle */
        audioIOB[i]->ioa_Volume = 64;        /* Maximum */
        audioIOB[i]->ioa_Cycles = 10000;     /* 10000mal */

        audioIOB[i]->ioa_Request.io_Command = CMD_WRITE;
        audioIOB[i]->ioa_Request.io_Flags = ADIOF_PERVOL;

         /* Audio-Block kopieren, um die Kanäle später wieder
            freizugeben */

        *aufreeIOB[i] = *audioIOB[i];

        printf("Sende einen Request\n");
        BeginIO(audioIOB[i]);
     }
     for(i=0; i<4; i++)
     {
        WaitIO(audioIOB[i]);
     }
```

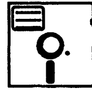
Listing 8.7: Das Audio-Programm (Teil 3)

```
    for(i=0; i<4; i++)
    {
        FreeAChannel(aufreeIOB[i]);
        printf("Kanal freigeben\n");
    }
    finishup("Erledigt!\n");
}


finishup(string)
char *string;
{
    int i;
    if(auDevice)
        CloseDevice(audioIOB[0]);
    if(chipaudio)
        FreeMem(chipaudio,8);
    for(i=0; i<4; i++)
    {
        if(audioIOB[i])
            FreeAudioIO(audioIOB[i]);
        if(aulockIOB[i])
            FreeAudioIO(aulockIOB[i]);
    }
    if(auReplyPort)
        DeletePort(auReplyPort);
    if(auLockPort)
        DeletePort(auLockPort);

    printf("%s\n",string);
    exit(0);
}

#include "ram:newaudio.c"
#include "ram:lockchan.c"
#include "ram:getaudio.c"
```

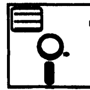
Listing 8.7: Das Audio-Programm (Schluß)

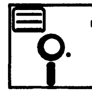
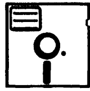
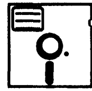
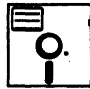






9



Kapitel 9

Multitasking

Wie bereits in Kapitel 3 beschrieben, kann Exec mehrere Tasks gleichzeitig verwalten. In diesem Kapitel wird Ihnen nun anhand von Beispielen gezeigt, wie man mit Exec, das das Multitasking überwacht, neue Tasks erzeugt oder mehrere Tasks zur selben Zeit ablaufen läßt.

Tasks

Exec verteilt die Leistung des Prozessors auf mehrere Tasks. Dabei hat jeder Task bei seiner Bearbeitung den alleinigen Zugriff auf alle Rechnerregister. Wenn man also einen neuen Task erzeugt, so wird die Rechnerleistung auf alle aktiven Tasks im System verteilt.

Jeder Task wird durch einen Task-Control-Block festgelegt. In diesem Block befinden sich alle Parameter, die den aktuellen Status des Tasks oder die Speicherbereiche festlegen, in denen die Prozessorregister zeitweilig zwischengespeichert werden können.

Sowie ein Interrupt ausgelöst wird, untersucht Exec den laufenden Task und die Liste mit allen startbereiten Tasks (Task-Ready-List). Dabei wird durch Vergleich der vergebenen Prioritäten entschieden, welcher der Tasks als nächstes bearbeitet werden soll. Sollte ein anderer Task eine Priorität besitzen, die größer oder gleich der des laufenden Tasks ist, so wird der aktuelle Status des MC68000 in den Task-Control-Block kopiert. Danach wird der komplette Sta-

tus aus dem Block des nächsten aktiven Tasks in den Prozessor geladen. Tasks mit gleicher Priorität wechseln sich gleichmäßig ab, so daß der Eindruck einer gleichzeitigen Bearbeitung entsteht.

Allerdings gibt es bei der Benutzung von Tasks einige Beschränkungen. So dürfen beispielsweise keinerlei AmigaDOS-Funktionen aufgerufen werden. Bedingt durch den Aufbau des AmigaDOS, werden weitere Variablen benötigt, die in Verbindung mit dem Task-Control-Block stehen. AmigaDOS benutzt diese Variablen, um Nachrichten zu empfangen oder um die Übersicht über die Daten- und Programmbereiche der einzelnen Programme zu behalten. Aber auch für den Zugriff auf das aktuelle Inhaltsverzeichnis und andere Möglichkeiten sind diese Variablen vonnöten.

Man sollte sich merken, daß nur der Task mit der höchsten Priorität die Chance erhält, bearbeitet zu werden. Sollten Sie also einen Task mit hoher Priorität starten, der weder Ein-/Ausgaben tätigt, noch Wait- oder Delay-Funktionen aufruft, so kann dieser den gesamten Rechner in Beschlag nehmen. Sobald ein Task mit hoher Priorität die Kontrolle übernommen hat, haben alle anderen Tasks mit niedriger Priorität keinerlei Möglichkeit, bearbeitet zu werden.

Die Priorität kann auf einen Wert zwischen –128 und +127 gesetzt werden. In der Regel werden die Tasks mit der Priorität Null gestartet. Wenn alle Tasks diesen Wert besitzen, der automatisch von AmigaDOS beim Start vom CLI vergeben wird, dann haben auch alle Tasks gleiche Chancen bearbeitet zu werden. In diesem Fall sieht es so aus, als ob alle Tasks gleichzeitig laufen. Alle Tasks werden dann nach dem "round-robin"-Verfahren an die Reihe kommen. Dies geht folgendermaßen vor sich: Der zuletzt aktivierte Task wird an das Ende der Liste aller startbereiten Tasks gesetzt. Der am Kopf der Liste stehende Task wird bis zur nächsten Unterbrechung bearbeitet und landet anschließend wieder am Ende der Liste. So kommen alle Tasks an die Reihe und können nach einem kompletten Durchlauf erneut bearbeitet werden.

Prozesse

AmigaDOS benutzt das Multitasking, indem es eine eigene Struktur dafür aufbaut, die sogenannte "Process-Data-Structure". Die meisten dieser Strukturen sind eng mit dem internen Aufbau des AmigaDOS verbunden. Deshalb werden auch die Einträge des "Process-Control-Block" in diesem Buch nicht erläutert. Die Struktur ist aber im "Amiga ROM Kernel Manual" im Includefile "libraries/dosextens.h" abgedruckt.

Prozesse schleppen eine größere Zahl an Daten als Task mit sich herum, da auch hierdurch die damit zusammenhängenden Möglichkeiten, die das AmigaDOS bietet, erweitert werden. Hier werden die Unterschiede zwischen Prozessen und Tasks nicht weiter ausgeführt, da sowohl die Verwendung von Prozessen als auch von Tasks in den Beispielen zu finden ist. Damit sollten auch Sie in der Lage sein, Ihre eigenen Applikationen zu schreiben.

Der einfache Weg, etwas Neues zu beginnen

Bevor wir den komplizierten Weg bei der Benutzung von Prozessen und Tasks beschreiten, hier zuerst einmal der einfache:

Sie können die Fähigkeiten des AmigaDOS nutzen, um einen separaten Prozeß zu starten. Während dieser Prozeß läuft, können Sie parallel dazu mit Ihrem Programm fortfahren.

Wenn Sie bereits das CLI benutzt haben, wissen Sie sicherlich auch, wie man Programme starten muß, damit sie gleichzeitig ablaufen. Tippen Sie beispielsweise ein:

```
RUN clock

RUN notepad
```

Diese Befehle erzeugen zwei neue Fenster, die zu den obigen Programmen gehören. Trotzdem bleibt das ursprüngliche CLI für weitere Kommandos eingabebereit. Eine andere Möglichkeit bietet

```
NEWCLI
```

mit dem man ein zusätzliches CLI-Fenster erhält und dadurch weitere Befehle abschicken kann. Um ein Programm einfach aus einem Ihrer eigenen starten zu können, benutzt man die Funktion `Execute`. Ihr wird ein Kommandostring übergeben, der genauso wie die Zeile aufgebaut ist, die man direkt per Hand im CLI eintippen würde. Ebenso wie man

```
RUN IRGENDWAS
```

im CLI eingeben könnte, müßte man für ein Programm diese Zeile nur folgendermaßen ergänzen:

```
erfolgreich = Execute("RUN IRGENDWAS", 0, 0);
```

Hierbei ist IRGENDWAS der Name des zu startenden Programms. Wenn Sie "RUN" im CLI eintippen, werden Sie feststellen, daß AmigaDOS die Kontrolle sofort wieder an das CLI übergibt und man dadurch auch ohne Verzögerung weiterarbeiten kann. Das gleiche passiert auch beim Execute-Aufruf. "RUN" lädt und startet IRGENDWAS und übergibt die Kontrolle an dieses Programm, so daß nun zwei Programme gleichzeitig ablaufen. Wenn Sie wollen, können Sie mehrere Programme in dieser Weise starten. Der Rückgabewert für die Variable "erfolgreich" ist stets "TRUE" (logisch wahr), da Execute das Kommando "RUN" immer problemlos aufrufen kann.

Außerdem kann man die Symbole für die Umleitung der Ein-/Ausgabe in den Kommandostring integrieren. Die Folge ist, daß die Ein- oder Ausgabe der unabhängigen Tasks aus einer Datei kommen oder in eine Datei gehen können. Dies würde man so machen:

```
erfolgreich = Execute("RUN <eingabedatei >ausgabedatei IRGENDWAS", 0, 0);
```

Die Namen Eingabedatei und Ausgabedatei sprechen für sich und gelten nur für das Programm IRGENDWAS, sobald es gestartet ist. Die Größer- und Kleiner-Zeichen sind die Umleitungssymbole des AmigaDOS und werden in Kapitel 2 ausführlich erklärt.

Sobald das RUN-Kommando bearbeitet wird, startet AmigaDOS einen separaten Prozeß für dieses Programm. Beide Prozesse (dieser und der eigene) dürfen nun ablaufen, indem sie sich die Leistung des Rechners teilen.

Dieses Beispiel zeigt, daß "RUN" für jedes neue Programm einen AmigaDOS-Prozeß startet. Werden keine Umleitungen der Ein-/Ausgabe im Kommandostring angegeben, so übernimmt dieser Prozeß die Einstellungen des CLI. Sie brauchen sich keine Gedanken zu machen, welche Funktionen Sie in Ihrem Programm verwenden wollen, weil alles, was mit dem "Process-Control-Block" in Verbindung steht, vom AmigaDOS eingestellt wird. Sie können in den gestarteten Programmen völlig ungezwungen AmigaDOS-Funktionen benutzen.

Zur Wiederholung: Wenn Sie lediglich Tasks starten möchten, sollten Sie vorsichtig bei der Verwendung bestimmter Arten von Funktionen sein. Mit anderen Worten: Sie sollten alles vermeiden, was direkt oder indirekt AmigaDOS

aufruft. Unter einem direkten Aufruf wird die Benutzung der Zeile mit AmigaDOS-Routinen (z.B. Open, Read, Write, Close oder Delay) verstanden, die im Kapitel 2 beschrieben werden. Zu den indirekten Funktionen gehören die IO-Routinen, die in der C-Library des Amiga integriert sind. IO-Funktionen sind definiert als etwas, das Daten zwischen einem Programm und einem Peripheriegerät austauscht. Dazu gehören unter anderem getchar, putchar, fopen, fclose, printf und fprintf. Ein Leitfaden, den man zum Erkennen einer IO-Funktion benutzen kann, wäre die Frage: "Überträgt die Funktion Daten zu einem Peripheriegerät wie z.B. der Tastatur, dem Bildschirm oder dem Diskettenlaufwerk?" Sollte dies zutreffen, kann es möglich sein, einen Task damit zu beauftragen. Da ein Prozeß aber alles kann, könnten Sie den Wunsch verspüren, anstelle des Tasks einen Prozeß zu verwenden.

Warum sollte man einen Task benutzen, wenn ein Prozeß mindestens das gleiche leistet? In einigen Fällen möchte man den Speicherbedarf eines Programms minimieren. Wenn bereits ein Task genau die Anforderungen erfüllt, wäre seine Verwendung doch die logische Konsequenz.

Warum befaßt man sich mit Tasks und Prozessen, wenn es so einfach ist, ein Programm mittels CLI oder eines anderen Programms zu starten? Manchmal möchte man vielleicht die Arbeitsweise des Systems ändern oder Tasks benutzen, die zusammenarbeiten und sich deshalb mit anderen verständigen müssen. Das folgende Beispiel zeigt, wie der Task-Control-Block erweitert werden muß, damit man mit anderen Task Informationen austauschen kann.

Ein Task-Beispiel

Das Beispiel in diesem Kapitel enthält ein Tool, um einen Task-Control-Block zu initialisieren und einen eigenen Task zu starten. Der Vorteil ist, daß das Hauptprogramm und der von ihm gestartete Task in demselben Codeblock untergebracht sind. Dies ist ein einfaches Beispiel, in dem weder das Hauptprogramm, noch der kleine, von ihm in die Welt gesetzte Task etwas wirklich Aufregendes machen. Es ist wirklich nur für Illustrationszwecke gedacht. Das Hauptprogramm initialisiert den einfachen Task, sendet eine Startmitteilung (startup message) und wartet so lange, bis es eine Antwort erhält.

Das Hauptprogramm kann einfach so umgeschrieben werden, daß es andere Aufgaben löst, während es auf eine Nachricht des Tasks wartet. Beispielsweise

könnte die main()-Funktion Benutzerdaten einlesen, während der Task Berechnungen mit bereits zur Verfügung stehenden Informationen durchführt. (An einem praktischen Beispiel: Stellen Sie sich vor, daß dieser einfache Task den nächsten Zug beim Schachspiel durchrechnet, gleichzeitig aktualisiert das Hauptprogramm die Uhr und wartet auf den nächsten Zug des Benutzers.) Sowie der Task auf die Startup-Message geantwortet hat, läuft er in einer Endlosschleife – benutzt die Wait()-Funktion. Wenn Sie z.B. ein Programm wie

```
main()
{
   printf("Hello world\n");
}
```

schreiben, so geht es in Ordnung, wenn man am Prorammende quasi "herausfällt". Mit anderen Worten: Sie können alle Ausgänge oder Return-Befehle übergehen, solange dies automatisch vom C-Compiler erkannt wird. Sobald das Programm beendet ist, gibt es den Startup-Code (AStartup oder LStartup) zurück, der beim Aufruf des Linkers an der ersten Position stand. Dieser Startup-Code überwacht zum einen das korrekte Initialisieren des Programms, zum anderen auch alle "Aufräumaktionen" nach Programmende.

Wenn man einen Task startet, ist es möglich, ihn einfach am Funktionsende "herausfallen" zu lassen, da das System keinen speziellen Startup-Code dieses Tasks erwartet und ohnehin weiß, was danach zu unternehmen ist. Es gibt zwei Möglichkeiten, einen Task zu beenden: Einerseits kann man den Task zu einem endlosen Warten veranlassen; dabei bricht AmigaDOS das Hauptprogramm nach Beendigung ab, gibt den Speicher für den Task frei und löscht ihn aus der systemeigenen Task-Liste. Andererseits kann sich der Task nach erledigter Arbeit auch selbst löschen. Diese Methode ist hier dargestellt.

Die Linkdatei für das Task-Beispiel

Die in Listing 9.1 abgedruckte Datei übernimmt das Linken des Task-Beispiels.

Um dieses Programm zu compilieren, gehen Sie folgendermaßen vor:

1. (von der eigenen Diskette)

```
COPY tasking.c ram:
COPY inittask.c ram:
COPY tasklink.lnk ram:
```

```
; tasklink.lnk
;
; assign lib df1:lib
; (df1: ist die C-Diskette)
; Hinweis: Compilieren Sie bei LC2 mit der Option -v, um die
; Stacküberwachung abzuschalten, ansonsten sucht der
; Linker nach den nicht definerten Symbolen _cxovf und _base.
; (Das Beispiel benutzt weder Lstartup.obj noch lc.lib.)
;
; Falls vom CLI gestartet:

FROM lib:Astartup.obj ram:tasking.o ram:inittask.o
TO ram:tasking
LIBRARY lib:amiga.lib
```

Listing 9.1: Die Datei "tasking.lnk"

2. (Diskette mit Amiga-C in Laufwerk 1)

   ```
   CD df1:examples
   EXECUTE make ram:tasking.c
   EXECUTE make ram:inittask.c
   df1:c/alink with ram:tasklink.lnk
   ```

   Als Ergebnis finden Sie dann eine Datei namens "tasking" vor.

3. Starten Sie das Programm durch Eintippen von

   ```
   RUN ram:tasking
   ```

   vom CLI und sehen Sie, was passiert.

Das Hauptprogramm inklusive einer kleinen Task

Das ausführlich dokumentierte Listing 9.2 zeigt das Hauptprogramm und den von ihm erzeugten kleinen Task.

Die InitTask()-Funktion

Listing 9.3 enthält die InitTask()-Funktion, die bereits in Listing 9.2 verwendet wurde. Sie unterscheidet sich von der CreateTask()-Funktion, die in der Ami-

ga-Funktions-Bibliothek bereitsteht, dadurch, daß der erschaffende Task nicht automatisch gestartet wird. Dieses Beispielprogramm wartet auf eine Startup-Message, anstelle sofort loszulegen. Zusätzlich muß das Hauptprogramm den Speicher für den Task belegen und wieder freigeben. CreateTask() übernimmt auch diese Speicheroperationen und ist somit der obigen Möglichkeit vorzuziehen, als daß dies von dem Prozeß gemacht wird, der auch den Task gestartet hat.

Ein Prozeß-Beispiel

Wenn ein Programm Ein-/Ausgabe unterstützt oder AmigaDOS-Funktionen auf irgendeine Art aufrufen soll, sollte man besser einen Prozeß als einen Task kreieren. Das fällt noch immer in den Bereich des Multitaskings, wird aber auf einer höheren Ebene erledigt, beispielsweise unter der Kontrolle von AmigaDOS.

In diesem Kapitel sind zwei Programme enthalten. Das Programm "proctest" lädt und startet "littleproc" und löscht dessen Code und Daten nach Beendigung.

```
/* tasking.c */

/* Einfaches Tasking-Programm
      - Erzeugt Vater- und Kind-Task

      - Hauptprogramm belegt Speicher für den
      - Task-Control-Block und den Message-Port, der sich um
      - den Kind-Task kümmert

      - Hauptprogramm initialisiert port und tcb und fügt
      - anschließend den Kind-Task hinzu.

      - Kind-Task wartet nach der Erzeugung auf eine Nachricht,
      - die über den Message-Port kommt.

      - Nach der Übermittlung der Nachricht wartet wiederum
      - der Vater-Task auf die Antwort des Kindes.
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 1)

```
      - Sobald die Nachricht des Vaters auf dem Port des
      - Kindes angekommen ist, erwacht das Kind, wiederholt
      - die Nachricht, antwortet und geht dann in eine
      - Endlos-Schleife.

      - Vater erwacht, sobald er die Nachricht auf seinem
      - Reply-Port empfangen hat, gibt den Speicher des Kindes
      - frei und verabschiedet sich. */

/* Betriebssystemversion 1.1 oder größer */

/* Programmabhängige Informationen:

      Linken Sie mit folgenden Dateien
         Astartup.obj, InitTask.o und amiga.lib,
      um "tasking" zu erhalten. */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/tasks.h"
#include "exec/execbase.h"

#include "exec/io.h"

#define PRORITY 0
#define STACKSIZE 500

extern struct Message *GetMsg();
extern struct MsgPort *CreatePort();
extern struct MsgPort *FindPort();
extern APTR            AllocMem();
extern struct Task    *FindTask();
extern int             InitTask();

struct MyExtendedTask
{
   struct Task      met_Task;     /* Ein Task-Control-Block */
   struct MsgPort   met_MsgPort;  /* auf einen Message-Port */
   int              met_Status;   /* Statuswert für info */
};

littletask()
{
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 2)

```
int signalbit;   /* Wert des Signal-Bit zur
                    Fehlerüberprüfung */

/* Zeiger auf den Message-Port des kleinen Tasks */
struct MsgPort *mp;
/* Zeiger auf eine Nachricht des kleinen Tasks */
struct Message *msg;
/* Zeiger auf einen Task */
struct MyExtendedTask *met;

met = (struct MyExtendedTask *)FindTask(0);

/* bis hier ist alles glatt gelaufen */
met->met_Status = 0;
/* Zeigt auf den Message-Port */

mp = &(met->met_MsgPort);

/* Nun muß der Message-Port vorbereitet werden, damit der
   Task schlafen kann, während er auf Nachrichten
   wartet.

   Teilt dem Port mit, welcher Task von dem Empfang einer
   Nachricht informiert wird. Es passiert solange nichts,
   bis mp_Flags vom übergeordneten Task auf PA_IGNORE
   gesetzt wird. */

mp->mp_SigTask = (struct Task *)met;

/* Nun erhält man eine Signal-Bit-Nummer.
   Dieses Vorhaben wird nicht schiefgehen, da es sich um
   einen brandneuen Task handelt und dadurch noch eine
   Menge Signal-Bits bereitstehen. Die Fehlerbehandlung
   fehlt hier völlig. Es liegt an Ihnen, was Sie bei einem
   Fehler unternehmen. */

/* beantrage irgendein Signal-Bit */
signalbit = AllocSignal(-1);

if(signalbit != -1)
{
    /* Es wurde ein gültiges Signal-Bit belegt. */
    /* Sage dem Port, daß er uns von einer ankommenden
       Nachricht informieren soll. */
    mp->mp_Flags = PA_SIGNAL;
}
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 3)

```
    else
    {
        /* Kein gültiges Signal-Bit verfügbar. */
        met->met_Status = -1;
        goto finish;
    }
    /* Ist ein Fehler aufgetreten, so kann die Flag-Variable
       nie auf einen anderen Wert als PA_IGNORE gesetzt werden.
       Das Ergebnis ist, daß der kleine Task vielleicht für
       immer schläft und auf ein Signal wartet, das niemals
       kommt. Folgendermaßen kann der Master-Task/-Prozeß eine
       Nachricht an diesen kleinen Task senden, wartet dann
       eine bestimmte Zeit und überprüft anschließend, wenn
       keine Antwort  von dem kleinen Task kam, ob der kleine
       Task auch tatsächlich gestartet wurde. Wenn dies der
       Fall sein sollte, so findet man die Fehlernummer im
       aktuellen Status oder in der Status-Variablen. Anstelle
       dieser Möglichkeit haben wir uns für das Beenden des
       Tasks entschieden (goto finish). */

    WaitPort(mp);  /* Warten auf gesetztes Signal-Bit */

    /* Ist bereits eine Nachricht vorhanden, schläft der Task
       überhaupt nicht */

    msg = GetMsg(mp);

    /* Erfolgreich geantwortet */
    met->met_Status = 1;
    /* Hier können Sie irgend etwas anderes einsetzen, was der
       Task auch immer tun soll */

finish:
    Forbid();       /* Task-Switching ausschalten */
    ReplyMsg(msg); /* Nachricht zurück an Absender schicken */
    /* Sie werden erwarten, daß dieses Signal-Bit mittels
       FreeSignal an genau dieser Position freigegeben wird.
       Solange der Task nicht wieder gelöscht ist, brauchen
       wir uns damit aber nicht herumzuärgern. */

    RemTask(0);  /* Permit() wird automatisch ausgeführt,
                    wenn der Task gelöscht wird. */

    /* Task aus der Task-Liste löschen. Der nächste Task kann
       ausgeführt werden. Dies ist durch den MemList-Eintrag
       in InitTask() möglich. Der gesamte, in "memlist" der
       Task eingetragene Speicher wird beim Aufruf von RemTask
       automatisch an das System zurückgegeben. */
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 4)

```
   /* Beachten Sie, daß "littletask" in einer Art Wait()
      oder in einer Endlosschleife enden muß, zumindest aber
      auf die hier beschriebene Weise gelöscht werden kann.
      Sollte es ganz normal beendet werden, gäbe es keine
      Stelle, zu der es zurückkehren könnte. Es muß so lange
      weiterlaufen, bis es später durch den übergeordneten
      Task gelöscht wird. */
} /* Ende von "littletask" */

/* Hinweis
   In diesem Beispiel sollte in "littletask" kein printf()-
   Aufruf getätigt werden, da dies nur ein Task und kein
   Prozeß ist. Wenn Sie ein Programm mit RUN starten oder
   es einfach nur vom CLI aufrufen, wird es als Prozeß
   gestartet, ein Überbegriff des Tasks. Die im "ROM Kernel
   Manual" beschriebenen Funktionen sind von Task aus
   aufrufbar. Die im "AmigaDOS Developers' Manual" oder im
   Lattice-C-Handbuch aufgeführten Funktionen müssen aus
   main() oder aus eigenen Unterfunktionen aufgerufen werden.
   Einzelaufgaben, die als Task und nicht als Prozesse
   gestartet werden, dürfen nur "task-taugliche" Funktionen
   benutzen. (Delay(xx) ist eine DOS-Funktion, so daß sie
   nicht von "littletask" benutzt werden kann.)

   (printf veranlaßt das AmigaDOS eine Ausgabe in ein CON:-
   oder RAW:-Fenster vorzunehmen. Daher erfordert dieser
   Befehl einen Prozeß anstelle eines Tasks.)

   Der Benutzer von Tasks muß bei den Dingen sehr vorsichtig
   sein, die von einem Task kontrolliert werden.

   Primäre Funktionen, die völlig selbständigen Code
   enthaltenen, sind wahrscheinlich am einfachsten zu
   benutzen. Alles, was irgendwie AmigaDOS auf den Plan
   rufen könnte, sollte man vermeiden. Dazu gehört sowohl
   das Öffnen einer Library, einer Device als auch eines
   Fonts, da sie allesamt einen Diskettenzugriff nach sich
   ziehen (um nicht-residenten Code oder Devices zu laden).
   Allgemein kann man empfehlen, lieber einen Prozeß als
   einen Task zu benutzen, wenn der Programmcode irgend
   etwas mit AmigaDOS zu tun hat. Ein separates Beispiel
   verdeutlicht, wie man einen Prozeß (im Gegensatz zum
   Task) zum Leben erweckt.

   Dieses Task-Beispiel arbeitet mit residentem Code, bei
   dem sowohl main() als auch sein Task gemeinsam geladen
   werden. Dieses Beispiel für einen Prozeß lädt ein anderes.
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 5)

```
   Programm als separaten Prozeß und gibt dessen Code nach
   der Beendigung wieder frei.
*/

main()
{
   /* aktuelle Datenstruktur für eine Message */
   struct Message mymessage;
   /* Zeiger auf den Reply-Port von main() */
   struct MsgPort *mainmp;
   /* Zeiger auf einen Extended-Task-Control-Block */
   struct MyExtendedTask *met;
   struct MsgPort *mp;
   int result;

   printf("\nmain() gestartet!\n");

   mainmp = CreatePort(0, 0);

   /* Da nur als Reply-Port benutzt, braucht er keinen
      Namen. Adresse der Antwort ist bereits in der Nachricht
      enthalten. */

   if(mainmp == 0) exit(20); /* Fehler bei CreatePort() */

   /* Belegen der Message-Data-Structure, damit PutMsg benutzt
      werden kann, um die Nachricht an die Task zu übermitteln
   */

         mymessage.mn_Node.ln_Type = NT_MESSAGE;
         mymessage.mn_Length       = sizeof(struct Message);
         mymessage.mn_ReplyPort    = mainmp;

   /* Nun den Speicher für einen Extended-Task-Control-Block
      belegen und initialisieren */

   met = (struct MyExtendedTask *)AllocMem(sizeof(struct
         MyExtendedTask), MEMF_PUBLIC | MEMF_CLEAR );

   if(met == 0)
      {
         /* Fehler bei AllocMem */
         DeletePort(mainmp);
         exit(40);
      }

   /* Jetzt den Task-Control-Block als Teil des
      Extended-Task-Control-Block initialisieren */
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 6)

```
result = InitTask( (struct Task *)met, "littletask",
                   PRIORITY, STACKSIZE );

if (result == 0)
   {
      /* Fehler bei InitTask...
         Kein Speicher für den Stack */
      DeletePort(mainmp);
      exit(45);
   }
/* Adresse des Message-Port ermitteln */

mp = &(met->met_MsgPort);

/* Message-Port für den Task initialisieren */

mp->mp_Node.ln_Type = NT_MSGPORT; /* Message-Port */
mp->mp_Flags = PA_IGNORE;    /* Wenn Nachricht angekommen
                                ist, keine Mitteilung */
/* Initialisieren der Message-List */
NewList(&(mp->mp_MsgList));

/* Der Message-Port ist nun in der Lage, Nachrichten
  abzuschicken. Man kann sie vor oder nach dem Hinzufügen
  einer neuen Task senden. */

PutMsg(mp, &mymessage);

AddTask(met, littletask, 0);

printf("\nTask erzeugt und hinzugefügt");

WaitPort(mainmp); /* Auf Antwort warten */

/* Dieses Beispiel geht davon aus, daß alles glatt geht.
   Sollte irgendwo ein Fehler auftreten, würde der Task
   ewig warten, was zur Folge hätte, daß die Hauptroutine
   ebenfalls blockiert würde. Die Nachricht käme also
   niemals zum Reply-Port zurück. Eine Alternative
   dazu wäre ein Timer, der folgendermaßen miteingebunden
   wird:

 wakeupmask = Wait(TIMER_SIGNAL_BIT|REPLYPORT_SIGNAL_BIT);

   Sollte man dann nach einer bestimmten Zeit keine
   Antwort erhalten, so untersucht man met_Status, um die
   Fehlerursache herauszufinden und etwas dagegen zu
   unternehmen. */
```

Listing 9.2: Das Hauptprogramm und der Task (Teil 7)

```
   GetMsg(mainmp); /* Nachricht abholen */
   printf("\n main: Task hat meine Nachricht empfangen\n");

cleanup:
   if(met) {
      /* Task vor dem Programmende löschen */
      FreeMem(met, sizeof(struct MyExtendedTask));
   }

   printf("\n main: Speicher des Tasks gelöscht");

   if(mainmp) {
      /* Löschen des Message-Reply-Port */
      DeletePort(mainmp);
   }

   /* Delay(250);      * 5 Sekunden warten, damit Benutzer
                         die Nachricht auch lesen kann, die
                         in das Lattice-Workbench-Window
                         geschrieben wurde. */

}  /* Ende von main */
```

Listing 9.2: Das Hauptprogramm und der Task (Schluß)

```
/************************************************************/
/* inittask.c -

        1.  Benutze Speicher, der von anderen angefordert
            wurde und ihn auch später wieder freigibt.
        2.  Task-Control-Block so weit wie möglich
            initialisieren
            (Gleiche, wie bei CreateTask).

************************************************************/

/* Originalprogramm von Carl Sassenrath und Neil Katin.
   Modifiziert, damit das Hauptprogramm etwas mehr als nur
   die Initialisierung des Extended-Task-Control-Blocks
   vornimmt, bevor der Task gestartet wird. */
```

Listing 9.3: Die Funktion "inittask" (Teil 1)

```
#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/tasks.h"
#include "exec/execbase.h"


/* Initialisiere Task mit vorgegebenen Namen, Priorität und
   Stackgröße. Es werden die Standard-Exception und die
   Trap-Handlers benutzt. */

/* Hinweise für die Einträge der Speicherbereiche (mementry):
   Unglücklicherweise ist dies mit C schwierig durchzuführen:
   Mementries beinhalten unions und können daher nicht
   "static" initialisiert werden.

   Um es einfach zu halten, habe ich die Mem-Entry-Struktur
   mit entsprechenden Größen ausgestattet. Diese wird in
   eine lokale Variable kopiert. Danach wird die Stackgröße
   auf dem vom Benutzer gewünschten Wert gesetzt und
   schließlich der Speicher belegt. */


#define ME_STACK      0
#define NUMENTRIES    1

struct FakeMemEntry {
    ULONG fme_Reqs;
    ULONG fme_Length;
};

struct FakeMemList {
    struct Node fml_Node;
    UWORD       fml_NumEntries;
    struct FakeMemEntry fml_ME[NUMENTRIES];
} TaskMemTemplate = {
    { 0 },          /* Knoten */
    NUMENTRIES,     /* Zahl der Einträge */
    {               /* Aktuelle Einträge */
        { MEMF_CLEAR,   0 }   /* Stack */
    }
};
```

Listing 9.3: Die Funktion "inittask" (Teil 2)

```
int
InitTask( task, name, pri, stackSize )
struct Task *task;
char *name;
UBYTE pri;
ULONG stackSize;
{
    struct Task *newTask;
    struct FakeMemList fakememlist;
    struct MemList *ml;


    /* Stack auf Langwort aufrunden... */
    stackSize = (stackSize +3) & ~3;


    /* Einen Speicherblock als Privatstack belegen. */
    fakememlist = TaskMemTemplate;

    fakememlist.fml_ME[ME_STACK].fme_Length = stackSize;

    ml = (struct MemList *) AllocEntry( &fakememlist );

    if( ! ml ) {
        return( 0 );
    }

    /* Stackinformationen setzen */
    newTask = task;

    newTask->tc_SPLower = ml->ml_ME[ME_STACK].me_Addr;
    newTask->tc_SPUpper = (APTR)((ULONG)(newTask->tc_SPLower)
                          + stackSize);
    newTask->tc_SPReg = newTask->tc_SPUpper;

    /* Misc-Task-Data-Structures */
    newTask->tc_Node.ln_Type = NT_TASK;
    newTask->tc_Node.ln_Pri = pri;
    newTask->tc_Node.ln_Name = name;

    /* an die Tasks-Memory-List anhängen */
    NewList( &newTask->tc_MemEntry );
    AddHead( &newTask->tc_MemEntry, ml );

    return( 1 );
}
```

Listing 9.3: Die Funktion "inittask" (Schluß)

"littleproc" wird von "proctest" gestartet. Der "mitgelinkte" Startup-Code wartet automatisch auf die Workbench-Startup-Message, bevor er loslegt. Der Prozeß benutzt den gleichen Message-Port, mit dem er auch bei seiner Initialisierung ausgestattet wurde. Er wartet dann auf eine Nachricht, die bestimmte Informationen enthält, in diesem Fall die Parameter, die das übergeordnete Programm benutzt. Im einzelnen sind dies die Dateihandles von stdout und stderr. Daher kann der gestartete Prozeß seine Ausgaben in das gleiche Fenster tätigen, das auch vom ursprünglichen Programm benutzt wird.

Ein Prozeß ist der Oberbegriff für einen Task. Die verschiedenen AmigaDOS-Routinen verlangen, daß ein Prozeß-Control-Block und die damit in Verbindung stehenden Informationen zur Laufzeit verfügbar sind. Dieser Code ist so ausgestattet, daß es dem Programmierer, der eher einen Prozeß als einen Task benötigt, ermöglicht wird, auf diesem Beispiel aufzubauen.

Die Linkfiles für das Prozeß-Beispiel

Um dieses Beispiel auszuprobieren, müssen beide Programme einzeln compiliert werden. Die Linkfiles, die zusammen mit dem Programm "alink" verwendet werden, sind hier abgedruckt. Folgende Schritte sind auszuführen:

1. (Von der Source-File-Diskette)

   ```
   COPY proctest
   COPY littleproc.c ram:
   ```

2. (Amiga-C-Diskette in Laufwerk 1)

   ```
   CD df1:examples
   ```

3. Benutzen Sie nun Ihren gewohnten Editor, um die Datei "make" im Verzeichnis "examples" zu verändern. Der Anfang der Zeile, die mit "lc2" beginnt, wird in "lc2 -v" geändert. Dies verhindert die Stacküberprüfung bei diesem Beispiel.

4. Geben Sie die folgenden Zeilen ein:

   ```
   EXECUTE make ram:proctest
   EXECUTE make ram:littleproc

   df1:c/alink with ram:proc.with
   df1:c/alink with ram:littleproctest.with
   ```

Das erste Link-File "proctest.with", welches in diesem Zusammenhang benutzt wird, beinhaltet die folgenden Befehle:

```
FROM lib:Astartup.obj proctest.o
TO proctest
LIBRARY lib:amiga.lib
```

Die zweite Link-Datei "littleproc.with":

```
FROM lib:Astartup.obj littleproc.o
TO littleproc
LIBRARY lib:amiga.lib
```

Die Prozeß-Programme

Um die Programme aus den Listings 9.4 und 9.5 zu starten, stellen Sie bitte sicher, daß sich beide Dateien im gleichen Unterverzeichnis befinden. "proctest" sucht das File "littleproc" nämlich stets im selben Directory. Starten Sie nun "proctest". Danach wird automatisch "littleproc" geladen und gestartet, der Speicher wieder freigegeben und das Programm beendet.

```
/* proctest.c */

/* Betriebssystemversion: V1.1 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"
```

Listing 9.4: Das Programm "proctest" (Teil 1)

```
#define PRIORITY 0
#define STACKSIZE 5000

extern struct Message *GetMsg();
extern int LoadSeg();
extern struct MsgPort *CreateProc();
extern struct MsgPort *CreatePort();

struct MyMess {
   struct Message mm_Message;
   int            mm_OutPointer;
   int            mm_ErrPointer;
};

extern int stdout;
extern int stderr;

main()
{
   struct Message *reply;
   struct Process *myprocess;

   /* Nachricht an den Prozeß, um ihn zu aktivieren */

   struct WBStartup *msg;

   /* Die Nachricht an den zu startenden Prozeß enthält
      eigene Parameter (nur für dieses Beispiel). Nachdem
      lediglich geprüft wurde, ob der Prozeß korrekt
      gestartet werden konnte, erhält er seine Handles für
      stdout and stderr (normalerweise nil:). In
      Wirklichkeit geben wir ihm unsere Werte, so daß das
      Ausgabefenster von beiden benutzt wird. */

   struct MyMess *parms;

  /* Da main() selbst als Prozeß gestartet wurde, wurde für
     ihn auch automatisch ein Message-Port zur Verfügung
     gestellt. Er ist durch
     &((struct Process *)FindTask(0))->pr_MsgPort
     zu finden */

   int littleSeg;

   /* In Wirklichkeit ist littleSeg ein BPTR, der Compiler
      freut sich aber über die int-Deklaration. Da wir den
      Wert nicht benutzen... nehmen Sie es einfach so hin! */
```

Listing 9.4: Das Programm "proctest" (Teil 2)

```
   char *startname, *parmname;

   struct MsgPort *mainmp;  /* Zeiger auf Message-Port von
                               main */
   struct MsgPort *littleProc; /* Zeiger auf Message-Port des
                                  gestarteten Prozesses */

/* Versorgt die Nachrichten mit Namen, so daß wir deren
   Übermittlung kontrollieren können. */
   startname = "startermessage";
   parmname  = "parameterpass";

/* das zu startende Programm laden */

   littleSeg = LoadSeg("littleproc");
   if(littleSeg == 0)
   {
      printf("\nlittleproc nicht gefunden");
      exit(999);
   }

/* Einen Prozeß für das andere Programm erzeugen */

   littleProc = CreateProc("littleguy",PRIORITY, littleSeg,
                STACKSIZE);
   if( littleProc == 0 )
   {
      printf("\Prozeß kann nicht erzeugt werden");
      UnLoadSeg(littleSeg);
      exit( 1000 );
   }

/* Message-Port ausfindig machen, der als Teil des Prozesses
   reserviert und vom Hauptprogramm zuerst gestartet wurde. */

   myprocess = (struct Process *)FindTask(0);

   mainmp = CreatePort(0,0;

/* Der folgende Block startet den Prozeß, als würde er von
   der Workbench aufgerufen werden. Da der Prozeß in dieser
   Weise erzeugt wird, erwartet er eine Startup-Message. (Es
   gibt noch eine andere Möglichkeit, ein geladenes Programm
   aufzurufen, was aber nicht den Start des Prozesses zur
   Folge hat. Es wird eine direkter Aufruf wie bei einem
   Unterprogramm auf den geladenen Code verwendet. Das
   andere Programm benutzt ebenfalls den eigenen Stack, so
   daß er so bemessen sein muß, um beide zu verwalten. Da
```

Listing 9.4: Das Programm "proctest" (Teil 3)

```
    dieses Programm auch als eigener Prozeß läuft, kann Ihr
    Programm darüber keinerlei Kontrolle erlangen, solange es
    nicht beendet ist. Befehle wie return() oder exit()
   versorgen Sie dann mit den entsprechenden Rückmeldungen. */

    /* Dieser Nachrichtenblock ist ein Weckruf an den Prozeß,
       den wir erschaffen haben. */
    msg = (struct WBStartup *)AllocMem(sizeof(struct
          WBStartup), MEMF_CLEAR);
    if(msg)
       {
       /* Stellt die nötigen Argumente der zu übertragenden
          Nachricht dar. */

       msg->sm_Message.mn_ReplyPort = mainmp;
       msg->sm_Message.mn_Length = sizeof(struct WBStartup);
       msg->sm_Message.mn_Node.ln_Name = startname;

       /* Keine Workbench-Argumente an diesen Prozeß
          übergeben; wir sind nicht WBench. Natürlich können
          wir, wenn wir wollen, die Argumente in
          "Workbench-Art" übermitteln. */

       msg->sm_ArgList = NULL;

       /* Wird ein Prozeß ohne "ToolWindow" geöffnet (Workbench
          erledigt das), so macht der Sklave einfach mit
          seinem main() weiter (s. Astartup.asm). */

       msg->sm_ToolWindow = NULL;

       /* Startup-Message senden */

       PutMsg(littleProc,msg);
    }
    else
    {
       printf("\nKein Speicher für WBStartup!\n");
       goto aarrgghh;  /* Oh nein, ein "goto"! */
    }
    /* Nur eine Beispielnachricht senden, die die gleichen
       Message- und Reply-Ports nutzt.
       Littleproc ist ein kooperierender Prozeß. Er weiß, daß
       er auf eine Nachricht auf seinem Port warten muß, die
       die Parameter für seine Ausgabe enthält.
       Die Startup-Message wird von dem Standard-Startup-Code
       übernommen. Diese Parameternachricht wird aber vom
       Programm selbst gehandhabt.
```

Listing 9.4: Das Programm "proctest" (Teil 4)

```
    Die Startup-Message wird an den Replyport durch den
    Startup-Code zurückgeschickt, nachdem das Programm
    beendet ist. */

 parms = (struct MyMess *)AllocMem(sizeof(struct
             MyMess),MEMF_CLEAR);
 if(parms)
 {
    parms->mm_Message.mn_ReplyPort = mainmp;
    parms->mm_Message.mn_Length = sizeof(struct mymess);
    parms->mm_Message.mn_Node.ln_Name = parmname;

    /* Hinweis: Dies sind stdout und stderr von
       AStartup.asm. Das Beispiel läuft nur dann
       einwandfrei, wenn das Hauptprogramm und das von
       ihm aufgerufende Unterprogramm mit dem gleichen
       Startup-Code. */

    parms->mm_OutPointer = (int)stdout;
    parms->mm_ErrPointer = (int)stderr;
    /* Unsere Parameters senden */

    PutMsg(littleProc,parms);

    /* Auf Antwort der Parameter-Nachricht warten */

    WaitPort(mainmp);

    reply = GetMsg(mainmp);

    /* Der Name der Nachricht sollte die Adresse des
       Strings "parms" enthalten, falls eine
       Fehlerüberprüfung eingebaut werden soll.
       Man sollte zur Übertragung von Parametern einen
       anderen als den vom System zur Verfügung gestellten
       Port belegen. Dies erleichert die Handhabung
       von mehreren verschiedenen Nachrichtenarten.

       Nun könnte das Hauptprogramm mit etwas Sinnvollem
       weitermachen und später nochmal nachschauen, ob
       mittlerweile der gestartete Prozeß seine Arbeit
       erledigt hat. Danach wird er gelöscht.

       Wartet auf die Wbstartup-Message, bevor das
       Hauptprogramm beendet werden darf. */

    WaitPort(mainmp);

    reply = GetMsg(mainmp);
```

Listing 9.4: Das Programm "proctest" (Teil 5)

```
      /* Name des Message-Nodes sollte die Adresse der
         "Startnachricht" */

      /* Hinweis: Hier sollte eine Abfrage stehen, die
         überprüft, ob die empfangene Nachricht der
         betreffende String oder der Weckruf war. */

         Dieses Programm erwartet, daß der String zuerst
         gesendet und wiederholt wird, und dann erst der
         Weckruf erfolgt, wenn der kleine Task seine Arbeit
         erledigt hat. */

      UnLoadSeg(littleSeg);
      printf("\nTask ist fertig! main eliminiert ihn!\n");
   }
   else
   {
      printf("\nkein Speicher für Parameter-Message\n");
   }
aarrgghh:
   /* erreicht diesen Punkt sowohl bei korrektem als auch
      bei fehlerhaftem Ablauf */

   if(mainmp)  DeletePort(mainmp);
   if(parms)   FreeMem( parms, sizeof(struct MyMess));
   if(msg)     FreeMem( msg,   sizeof(struct WBStartup));

}  /* Ende von main */
```

Listing 9.4: Das Programm "proctest" (Schluß)

```
/* littleproc.c */

/* Beispielcode für Prozeß-Test */

/* System-Software Version: V1.1 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"
```

Listing 9.5: Das Programm "littleproc" (Teil 1)

```
#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"

/* Diese werden dem Task vom Starter zur Verfügung gestellt
   und sind im Startup-Code definiert (Astartup.asm). */

extern int stdout;
extern int stderr;

struct MyMess {
   struct Message mm_Message;
   int mm_OutPointer;
   int mm_ErrPointer;
};

extern struct Message *GetMsg();
extern struct Task *FindTask();
extern struct FileHandle *Open();

main()
{
   struct MyMess *msg;
   struct MsgPort *myport;
   struct Process *myprocess;

   struct FileHandle *myOwnOutput;
   myprocess = (struct Process *)FindTask(0);

   myport = &myprocess->pr_MsgPort;

/* Auf Starter für die Absendung einer Message warten.
   Message hat eigenes stderr und stdout, so daß wir den
   Kram zum CLI-Fenster schicken, von dem gestartet wurde. */

   WaitPort(myport);
   msg = (struct MyMess *)GetMsg(myport);
   stdout = msg->mm_OutPointer;

/* printf verwenden, um zu beweisen, daß es wirklich ein
   Prozeß ist. Ein Task würde abstürzen! */

   printf("\nHier bin ich, der von Ihnen gestartete Prozeß!");
   printf("\nWerde nun mein eigenes Fenster öffnen.\n");
```

Listing 9.5: Das Programm "littleproc" (Teil 2)

```
    /* Mach' nun etwas Nützliches, bearbeite die Aufgaben,
       für den der Prozeß erzeugt wurde. */

    myOwnOutput = Open("CON:10/10/320/150/Kind-Prozeß",
                       MODE_NEWFILE);
    if(myOwnOutput == 0)
    {
       ReplyMsg(msg);  /* main mitteilen, daß fertig */
       exit(0);    /* Kann keinen Fehlercode zurückgeben */
    }
    else
    {
       stdout = (int)myOwnOutput;
       /* Output-File-Handle zurücksetzen */
       printf("Siehste, ich kann AmigaDOS!");
       Delay(250);      /* 250/50 = 5 Sekunden */
       stdout = msg->mm_OutPointer;
       Close(myOwnOutput);
       ReplyMsg(msg);
    }

 /* Nun am Ende alles fallen lassen,
  * zum Startup-Code zurückkehren und beenden */

 }
```

Listing 9.5: Das Programm "littleproc" (Schluß)

Beachten Sie, daß diese Programme so geschrieben sind, daß der Amiga-Startup-Code "AStartup.obj" und nicht der Lattice-Startup-Code "LStartup.obj" verwendet wird. Es wurden keine Versuche unternommen, dies an LStartup anzupassen. Der einzige Grund der Inkompatibilität sollte die Benutzung von stdout und stderr sein, da dies von Lattice etwas unterschiedlich gehandhabt wird. Lattice definiert stdout und stderr als die Adresse eines IO-Blocks. Wenn aber sowohl das Hauptprogramm als auch der von ihm zu startende Prozeß mit Lattice übersetzt wurden, so sind die Werte für stdout und stderr wieder kompatibel.

Stdout und stderr von AmigaDOS sind Zeiger der Sprache BCPL auf eine AmigaDOS-Datenstruktur. Wenn Sie "amiga.lib" zur Versorgung der printf()-Funktion zur Linkzeit verwenden, so wird die in "amiga.lib" enthaltene Ver-

sion der printf()-Routine intern die AmigaDOS-Version stdout benutzen. Wenn beim Linken die Datei "lc.lib" zuerst angegeben wird, dann interpretiert das Linkfile stdout auf seine Weise. Das ist dann natürlich nicht mehr zu den von der Open()-Funktion (Open für AmigaDOS, open für Lattice) gelieferten Werten kompatibel.

Beachten Sie weiterhin, daß dieses Beispiel den Message-Port nutzt, den AmigaDOS für diesen Prozeß zur Verfügung stellt, um IO-Messages des AmigaDOS zu empfangen. Wenn dieser Code für Sie interessant und nützlich ist, dann sollten Sie sich einen separaten Message-Port für eigene Mitteilungen einrichten, anstatt sich den Port mit AmigaDOS zu teilen.

Den Message-Port dem AmigaDOS einfach zu stehlen, ist ein einfacher und bequemer Weg, die Dinge ans Laufen zu bringen. Falls Ihr Prozeß jedoch intensiven Gebrauch von den Ein-/Ausgaberoutinen macht, ist es vorteilhaft, einen eigenen Message-Port für den privaten Gebrauch zu öffnen.

Als abschließende Bemerkung dieses Prozeß-Beispiels hier noch der Hinweis, wie man mittels Execute-Kommando des AmigaDOS eigene Prozesse erschaffen und starten kann:

```
success = Execute("irgendeinprogramm",0,0);
```

Dieses Beispiel wurde aber dazu geschrieben, Daten und Nachrichten zwischen verschiedenen Prozessen auszutauschen.

Kommunikation zwischen einzelnen Tasks

Für die Kommunikation zwischen Prozessen und Tasks haben die bisher gezeigten Programme speziell geschaffene Message-Ports verwendet. Wenn Sie völlig unabhängig gestartete Prozesse haben, dann gibt es verschiedene Möglichkeiten, einen Prozeß feststellen zu lassen, ob ein anderer Prozeß oder Task bereits läuft, so daß Sie mit ihm kommunizieren oder von ihm geschaffene Datenstrukturen verwenden können.

Das Aufspüren von Tasks

Das System führt verschiedene Listen, einschließlich einer Liste für Tasks und einer für Ports. Wenn Sie einen Task über die CreateTask-Funktion starten, dann trägt die Funktion den Namen des Tasks im List-Node des Task-Kontroll-Blocks ein. Daher können Sie die FindTask-Funktion verwenden, um den Task zu lokalisieren.

In Listing 9.3 wurde der InitTask-Funktion ein Name übergeben, anhand dessen der Task-Kontroll-Block gefunden werden konnte. Der Name war Littleguy.

Statt die Adresse des vom Hauptprogramm erzeugten Task-Kontroll-Blocks zu verfolgen, hätte main auch den Task zum System hinzufügen und den Task (und darüber auch den Message-Port) wie folgt finden können:

```
struct Task *taskblock; /* Ein Zeiger auf
                           Task-Kontroll-Block */

taskblock = FindTask("littleguy");
```

So kann alles aufgespürt werden, was Sie verändert haben und was im Zusammenhang mit einem Task-Kontroll-Block steht, und zwar über den Eintrag in der System-Task-Liste. Auf jeden Fall wußte das Programm in Listing 9.3, wo der Task war, da main diesen Block erzeugt hatte.

Das Aufspüren von Prozessen

Unglücklicherweise ist das Aufspüren von Prozessen nicht so einfach zu lösen wie das Aufspüren von Tasks. Obwohl AmigaDOS (ab Version 1.1) die Task-Liste verwendet, um die Prozesse zu verfolgen, initialisiert AmigaDOS trotzdem nicht das Task-Namen-Feld, um es in Beziehung zum Namen des laufenden Prozesses zu bringen.

Sollte Ihr Programm ein laufendes Programm identifizieren müssen, um dorthin Daten zu übergeben, so ist zu empfehlen, einen Task statt eines Prozesses zu verwenden.

Das Aufspüren von Ports

Wenn Sie einen Message-Port erzeugen, benennen und in die Systemliste eintragen, dann kann Ihr Task oder ein anderer Task oder Prozeß später diesen Port über die FindPort-Funktion lokalisieren:

```
myport = FindPort("thisport");
```

Wenn die Funktion einen Wert von Null zurückgibt, dann gab es keinen Port dieses Namens in der System-Port-Liste. Wenn der zurückgegebene Wert ungleich Null ist, dann zeigt "myport" auf den List-Node des Message-Ports mit dem entsprechenden Namen.

Ich benötigte diese Eigenschaft für ein Grafik-Demonstrationsprogramm, in dem sechs unabhängige Programme einen Custom-Screen verwendeten, um einige der vielen Grafik-Funktionen des Amiga zu demonstrieren. Eines dieser Programme heißt "Rectangles" (Rechtecke), es erzeugt mehrfarbige Rechtecke in einem Grafik-Fenster. Ein anderes heißt "Lines" (Linien), es zeichnet zufällige, farbige Linien. Ein drittes hieß "Wallpaper" (Tapete), es erzeugt vielfarbige Rechtecke.

Custom-Screens können eine Menge Speicherplatz belegen, daher entschied ich mich dafür, jedes Programm überprüfen zu lassen, ob ein Programm dieses Typs bereits läuft. Wenn ein anderes Programm bereits einen Custom-Screen erzeugt hatte, dann öffnete jedes meiner Programme einfach ein Fenster auf dem Custom-Screen. Jedes Fenster hat sein eigenes Schließsymbol, das auch das Programm beendet. Jedes Fenster hat seine eigene Titelzeile, die die Namen aller anderen Programme aufführt, die bekanntermaßen den Custom-Screen nutzen.

Um zwischen den laufenden Programmen zu kommunizieren, erzeugte ich eine spezielle Version eines Message-Ports:

```
typedef struct {
   struct MsgPort normalMsgPort;
   int usersOfScreen;
   struct Screen *screen;
   } MYCUSTOMPORT;
```

Durch die korrekte Initialisierung von "normalMsgPort" konnte er über die AddPort-Funktion zur System-Port-Liste hinzugefügt werden. Rufen Sie sich

in Erinnerung, daß das System sich nicht darum kümmert, wie groß jeder Listeneintrag ist, solange die Listen-Strukturelemente korrekt initialisiert sind.

Das erste Programm (egal welches aus der Gruppe der untereinander kompatiblen Grafikprogramme) reservierte Speicherplatz für diesen Custom-Port, öffnete einen Custom-Screen, kopierte die Adresse des Custom-Screen in die Custom-Port-Datenstruktur und setzte "usersOfScreen" auf Eins. Es konnte dann diesen Port zur System-Port-Liste hinzufügen und ein Fenster auf diesem Custom-Screen öffnen.

Später folgende Programme können diesen Port über FindPort lokalisieren und über den Wert für "userOfScreen" berechnen, wo ein neu zu öffnendes Fenster plaziert werden mußte, um nicht bereits vorhandene Grafik oder Fenster zu verdecken. Jedes neue Programm erhöhte bei der Aufspürung des Ports und der Öffnung seines Fensters den Wert für "usersOfScreen" um Eins. Jedesmal, wenn ein Programm sein Fenster schloß, wurde der Wert für "usersOfScreen" um Eins heruntergezählt. War der Wert auf Null gesunken, so löschte das letzte Programm den Port und beendete seine Arbeit.

Listing 9.6 zeigt zwei Code-Segmente, die Teil aller oben erwähnten Grafikprogramme sind. Diese Segmente demonstrieren, wie der Custom-Port allen Programmen auf der Suche nach einem Custom-Screen als Treffpunkt dient, auf dem Sie ihr Fenster öffnen könnten.

```
/* Code-Fragment 1 - Custom-Port */

/* Achtung: Nicht alle Deklarationen sind hier enthalten.
   Dieser Code ist nur dazu gedacht, um dem Programmierer
   zukünftiger Applikationen einige Ideen zu geben. */

struct MyPort {            /* Custom-Message-Port */
   struct MsgPort mp;      /* Standard-Message-Port */
   struct Screen *Screen; /* Wo ist der Screen, der mir
                              gehören soll  */
   int Users;            /* Wie viele Programme verwenden diesen
                             Screen */
```

Listing 9.6: Ausschnitt für einen eigenen Port (Teil 1)

```
   int RangeSeed; /* Was ist der aktuelle Wert des
                     Zufallszahlengenerators für den letzten
                     User... sonst beginnt RangeSeed bei 0,
                     und zwar für alle */
   char portname[64];
   char screenname[64];
};

struct Screen *
Setup()  /* Gibt Pointer auf neuen oder existierenden Screen
            zurück */
{
    int junk;
    struct MyPort *myp, *sysmyp;
    struct Screen *screen;
    GfxBase = OpenLibrary("graphics.library", 0);
    if (GfxBase == NULL)
    {
       problem = 1001; /* Kann Gfx-Bibliothek nicht öffnen */
       return(0);
    }
    IntuitionBase = OpenLibrary("intuition.library", 0);
    if (IntuitionBase == NULL)
    {
       problem = 1002; /* Kann Intuition-Bibliothek nicht
                          öffnen */
       CloseLibrary(GfxBase);
       return(0);
    }


   /* Hat eine andere Applikation bereits den Custom-Screen
      geöffnet? Wenn ja, dann verwenden. Wenn nicht, einen
      öffnen. Task-Wechsel verhindern, falls beide
      gleichzeitig starten.  Nur einer sollte überprüfen
      dürfen, ob ein mit Informationen besetzter Port
      existiert und ihn andernfalls erzeugen dürfen. */

   /* Beginnen, den Custom-Message-Port zu erzeugen, falls
      das System noch keinen hat. Auf diese Weise ist er
      zur Hinzufügung vorrätig und muß nicht zu lange
      deaktiviert bleiben. */

    myp = (struct MyPort *)AllocMem(sizeof(struct MyPort),
                           MEMF_CLEAR);
    if (myp == 0)
```

Listing 9.6: Ausschnitt für einen eigenen Port (Teil 2)

```
    {
       CloseLibrary(IntuitionBase);
       CloseLibrary(GfxBase);
       return(0);
    }
    else
    {
        /* Setup der Port-Parameter */

        myp->mp.mp_Node.ln_Pri = 0;
        myp->mp.mp_Node.ln_Type = NT_MSGPORT;
        NewList(&(myp->mp.mp_MsgList));

        /* Titel für Port, innerhalb des Ports */

        strcpy( &(myp->portname[0]) , "lowres.16.color" );
        myp->mp.mp_Node.ln_Name = &(myp->portname[0]);

        /* Titel für Test-Screen, innerhalb des Ports */

        strcpy( &(myp->screenname[0]), "TestScreen" );
        ns.DefaultTitle = (UBYTE *)&(myp->screenname[0]);

        /* Anzahl der User eines Custom-Screens */

        myp->Users = 1;    /* Ein Benutzer; nur öffnen */

        /* Dummy-Wert berechnen, um aktuellen Wert für
           RangeSeed zu ändern */

        junk = RangeRand(100);

        /* Startwert für Zufallszahlen-Generator (für
           nächsten User) */

        myp->RangeSeed = RangeSeed;

        /* Anfang des Teils mit deaktivierten Interrupts */
        Disable(); /* Task-Wechsel während dieser Operation
                      verhindern */

/* Wir können Custom-Ports dem System hinzufügen. Gibt es
   bereits so einen?  Wenn ja, dann Speicher nicht
   reservieren. Wenn nicht, Custom-Screen öffnen und
   Initialisierung des Ports mit seiner Adresse abschließen.
   Dann in die System-Port-Liste eintragen. */
```

Listing 9.6: Ausschnitt für einen eigenen Port (Teil 3)

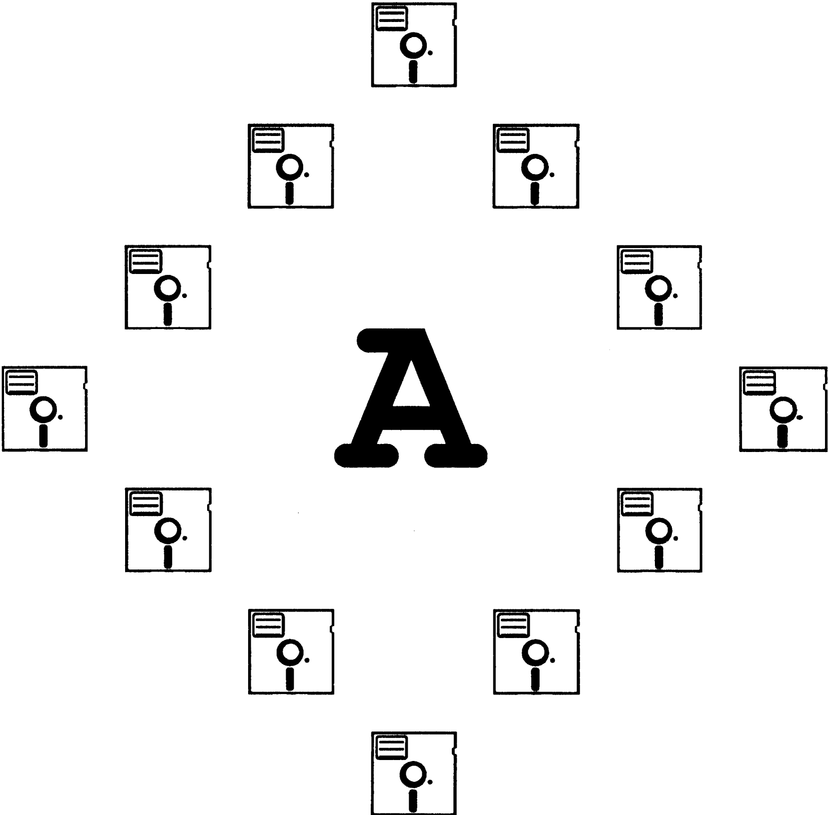
```
      sysmyp = (struct MyPort *)FindPort("lowres.16.color");

      if(sysmyp == 0)
      {
          screen = OpenScreen(&ns);
          if (screen == NULL)
          {
              problem = 1003; /* Kann keinen Custom-Screen
                                 öffnen */
              Enable();   /* Interrupts und Task-Wechsel
                             ermöglichen */
              FreeMem(myp, sizeof(struct MyPort));
              CloseLibrary(IntuitionBase);
              CloseLibrary(GfxBase);
              return(0);
          }

  /* Anzeigen, wo der gerade geöffnete Custom-Screen zu
     finden ist. */

          myp->Screen = screen;

          /* Zum System, damit ihn auch andere finden */

          AddPort(myp);
          Enable(); /* Interrupts und Task-Wechsel
                       ermöglichen */
          return(screen);
      }
      else /* System hat diesen Custom-Port bereits! */
      {
          /* Wenn System einen Port wie diesen hat, dann
             braucht es Ihren nicht */

          FreeMem(myp, sizeof(struct MyPort));

          /* Anzahl der Anwender um Eins erhöhen */
          sysmyp->Users += 1;

          /* Nimmt den Wert des vorigen Users für
             RangeSeed */
          RangeSeed = sysmyp->RangeSeed;
          junk = RangeRand(100);   /* Ändert Wert für
                                      RangeSeed */

          /* Neuen Wert für nächsten User sichern */
          sysmyp->RangeSeed = RangeSeed;
          Enable();   /* Task-Wechsel ermöglichen */
```

Listing 9.6: Ausschnitt für einen eigenen Port (Teil 4)

```
                /* Screen-Position zurückgeben */
                return(sysmyp->Screen);
            }
        }
}

/* Code-Fragment 2 */

/* Custom-Window schließen und Anzahl der User des
   Custom-Screens dekrementieren. Wenn Wert für User auf
   Null fällt, dann auch den Screen-Message-Port löschen,
   seinen Speicher freigeben und Custom-Screen löschen. */

int
EndTest()
{
    struct MyPort *myp;
    if (w != NULL) {
        ClearMenuStrip(w);
        CloseWindow(w);
    }
    Forbid();  /* Task-Wechsel kurz stoppen */
    myp = (struct MyPort *)FindPort("lowres.16.color");
    /* Das könnte klappen */
    if(myp)
    {
        if((myp->Users -= 1)==0)
        {
            if (myp->Screen != NULL) CloseScreen(myp->Screen);
            RemPort(myp);
            FreeMem(myp, sizeof(struct MyPort));
        }
    }
    Permit();  /* Task-Wechsel ermöglichen */
    return(TRUE);
}
```

Listing 9.6: Ausschnitt für einen eigenen Port (Schluß)

Die Multitasking-Fähigkeiten des Amiga eröffnen dem Entwickler ungeahnte Möglichkeiten. Der hier gezeigt Code hat diese gerade nur grob angerissen. Ich hoffe, Sie haben in diesem Kapitel einen Einblick in die Arbeitsweise des Multitasking-Systems erhalten und ein Sprungbrett für Ihre eigenen Programme gefunden.



Anhang A

Der Texteditor ED

Das Programm ED ist ein einfacher Texteditor, der zum Erzeugen von Quelldateien für den C-Compiler dient. Für diejenigen unter Ihnen, die bei der Verwendung eines Editors zum Erstellen von Programmen noch nicht so vertraut sind, ist dieser Anhang gedacht. Er führt Sie anhand eines kurzen Programms in die Bedienung ein.

Wenn ED gestartet ist, präsentiert er Ihnen einen leeren Bildschirm auf dem Sie nun Textzeilen eintippen können. Nachdem die Zeilen eingegeben sind, können Sie den Text in eine Datei abspeichern.

ED ist ein Zeileneditor, keine Textverarbeitung. Benutzt man ED, sollte man sich merken, daß er immer einen kompletten Text-Bildschirm anzeigt und zeilenorientiert arbeitet. So muß ein Block beispielsweise aus einer oder mehreren Zeilen bestehen. Die Blockmarke kann deshalb nicht in der Mitte einer Zeile stehen. Freundlicherweise fügt ED einen zu verschiebenden oder zu kopierenden Textblock zwischen der aktuellen Zeile, in der der Cursor steht, und der direkt darüberliegenden Zeile ein.

Wenn Sie etwas mit ED herumprobieren wollen, werden Sie sich für seine Kommandos interessieren. Auf jeden Fall braucht sich ein unerfahrender Benutzer nur ein paar fundamentale Regeln und Befehle zu merken, um das Programm effektiv zu nutzen. Hier nun einige Gedächtnisstützen:

- Man befindet sich stets im Einfüge-Modus. Wo der Cursor auch immer steht, sowie man ein gültiges Zeichen (kein Kommando) eintippt, fügt ED dieses Zeichen an der aktuellen Cursorposition ein und schiebt den Rest um eine Stelle nach rechts. Die Betätigung der Return-Taste teilt die Zeile an der Cursorposition.

- Die Cursortasten arbeiten wie gewohnt. Sie können sich nur mit den Cursortasten durch den ganzen Text bewegen.

- Die Backspace-Taste löscht das Zeichen, welches sich direkt links neben dem Cursor befindet.

- Die Escape-Taste wird zum Aufruf von weitergehenden Befehlen benötigt. Wenn Sie die Escape-Taste drücken, wird der Cursor kurzfristig in die Statuszeile bewegt, die sich am unteren Rand des Bildschirms befindet. Dann können Sie den erweiterten Befehl vervollständigen.

Um ED zu starten, sollten Sie sicher gehen, daß sich die Workbench- oder CLI-Diskette im internen Laufwerk befindet. Diese Diskette darf nicht schreibgeschützt sein, damit es ED gestattet wird, im Directory "SYS:T" eine Arbeitsdatei zu erzeugen. Tippen Sie in das CLI-Window

```
ED hello.c
```

und betätigen Sie die Return-Taste. Ein neues Fenster wird geöffnet und ED zeigt folgendes an:

```
Creating a new file
```

Der Cursor steht oben im Fenster. Tippen Sie nun die folgenden Zeilen genau so ab, wie sie hier abgedruckt sind, und betätigen Sie am Ende jeder Zeile die Return-Taste.

```
main()
{
   printf("\nHello world\n");
}
```

Tippen Sie nun <Esc>, X und <Return>. Das Programm ist nun gespeichert, so daß der Compiler es später verwenden kann.

Tabelle A.1 faßt die ED-Befehle zusammen. Bei Verwendung der Tastenkombination eines Buchstabens mit der <Ctrl>-Taste wird <Ctrl> gedrückt gehalten, während der betreffende Buchstabe eingetippt wird, der das Kommando beschreibt. Alle Befehlsbuchstaben sind hier in Großbuchstaben angegeben, Kleinbuchstaben führen aber zum gleichen Ergebnis.

Befehle zur Cursor-Bewegung

Befehl	Beschreibung
↑	Eine Zeile nach oben bewegen.
↓	Eine Zeile nach unten bewegen.
→	Ein Zeichen nach rechts bewegen.
←	Ein Zeichen nach links bewegen.
<Ctrl>I	(Tabulatur) Zum nächsten Tabulatorstop bewegen.
<Ctrl>R	Zum Ende des vorangehenden Wortes bewegen.
<Ctrl>T	Zum Anfang des nächsten Wortes bewegen.
<Ctrl>D	Den Text eine Zeile nach unten verschieben.
<Ctrl>U	Den Text eine Zeile nach oben verschieben.
<Ctrl>E	An den Anfang oder das Ende des Bildschirms bewegen.
<Ctrl>]	Den Text eine Zeile nach unten verschieben.
<Ctrl>M	Den Text eine Zeile nach unten verschieben.
<Esc>B	An das Ende des Textes bewegen.
<Esc>T	An den Anfang des Textes bewegen.
<Esc>N	Zum Anfang der nächsten Zeile bewegen.
<Esc>P	Zur vorhergehenden Zeile bewegen.
<Esc>CE	An das Ende der aktuellen Zeile bewegen.
<Esc>CB	An den Anfang der aktuellen Zeile bewegen.
<Esc>M *Zeilennummer*	In die angegebene Zeile innerhalb der Datei bewegen.

Tab. A.1: ED-Kommandos (Teil 1)

Einfüge- und Lösch-Befehle

<Ctrl>A	Die Zeile hinter der aktuellen einfügen.
<Ctrl>B	Die aktuelle Zeile löschen.
<Ctrl>H	(Backspace) Das Zeichen links vom Cursor löschen. Alles wird um ein Zeichen nach links verschoben.
<Del>	Das Zeichen unter dem Cursor löschen.
<Ctrl>O	Befindet sich der Cursor über einem Leerzeichen, werden alle Leerzeichen bis zum nächsten Wort in dieser Zeile gelöscht. Andernfalls werden dieses und alle anderen rechts vom Cursor stehenden Zeichen bis zum nächsten Space eliminiert.
<Ctrl>Y	Bis zum Ende der Zeile alles löschen (inklusive des Zeichens, auf dem der Cursor zu finden ist).
<Esc>A/*Zeichenkette*/	Die Zeichenkette als Zeile vor der aktuellen einfügen.
<Esc>I/*Zeichenkette*/	Die Zeichenkette als Zeile nach der aktuellen einfügen.
<Esc>D	Die aktuelle Zeile löschen.
<Esc>DC	Das Zeichen unter dem Cursor löschen.
<Esc>IF!*Dateiname*!	Die Datei mit dem angegebenen Dateinamen an der momentanen Cursorposition einfügen.

Tab. A.1: ED-Kommandos (Teil 2)

Rand- und Tabulator-Befehle

<Esc>SL *Nummer*	Den linken Rand auf die angegebene Spaltennummer setzen (voreingestellt ist 1).
<Esc>SR *Nummer*	Den rechten Rand auf die angegebene Spaltennummer setzen (voreingestellt ist 80, Maximum ist 255, das ED und AmigaDOS zulassen).
<Esc>ST *Nummer*	Die Tabulatorentfernung auf den angegebene Wert setzen, die vom Standardtabulator benutzt wird.
<Esc>EX	Den rechten Rand erweitern; mit Randfunktion beim Typenraddrucker vergleichbar.

Befehle für Suchen und Ersetzen

<Esc>F/*Zeichenkette*/	Vorwärts nach dem nächsten Auftreten der angegebenen Zeichenkette suchen.
<Esc>BF/*Zeichenkette*/	Rückwärts nach dem nächsten Auftreten der angegebenen Zeichenkette suchen.
<Esc>E/*Zeichenkette1*/*Zeichenkette2*/	Vorwärts nach dem nächsten Auftreten der "Zeichenkette1" suchen und durch "Zeichenkette2" ersetzen. Es wird beim Ersetzen nicht nochmals nachgefragt.
<Esc>EQ/*Zeichenkette1*/*Zeichenkette2*/	Vorwärts nach dem nächsten Auftreten der "Zeichenkette1" suchen und durch "Zeichenkette2" ersetzen, falls die Abfrage bestätigt wurde.
<Esc>LC	Groß-/Kleinschreibung soll beim Suchen unterschieden werden.

Tab. A.1: ED-Kommandos (Teil 3)

Block-Befehle	
<Esc>BS	Diese Zeile als Blockanfang markieren.
<Esc>BE	Diese Zeile als Blockende markieren.
<Esc>DB	Den markierten Block löschen.
<Esc>ID	Den markierten Block an der aktuelle Cursorposition einfügen.
<Esc>SD	Die erste Zeile des markierten Blocks in der obersten Zeile des Bildschirms anzeigen; erlaubt das schnelle Bewegen zu der markierten Position innerhalb der Datei.
<Esc>WF!*Dateiname*!	Den markierten Block in eine Datei schreiben. Wenn die Datei nicht in das aktuelle Verzeichnis soll, kann auch der komplette Pathname inklusive Slashes mit angegeben werden. (Die Ausrufezeichen dienen dazu, den Pfadnamen zu begrenzen.)
Speichern und Beenden	
<Esc>SA	Die Datei unter dem aktuellen Dateinamen abspeichern und mit dem Editieren fortfahren.
<Esc>Q	Das Programm beenden, ohne die gemachten Änderungen seit dem letzten Speichern zu sichern.
<Esc>X	Das Programm verlassen und die Datei unter dem aktuellen Dateinamen abspeichern.
Diverse Befehle	
<Esc>J	Diese und folgende Zeile zu einer vereinigen.
<Esc>S	Die aktuelle Zeile an der Cursorposition splitten. Das Zeichen unter dem Cursor wird zum ersten Zeichen der neuen Zeile.
<Esc>SH	Den Status des Editors anzeigen.
<Esc>V	Den Bildschirm neu aufbauen.

Tab. A.1: ED-Kommandos (Schluß)

Beachten Sie, daß sowohl <Esc>SA als auch <Esc>X einen optionalen Dateinamen akzeptieren, der anstelle des beim Öffnen der Datei verwendeten benutzt wird. Dies entspricht

```
<Esc>X!Dateiname!
```

oder

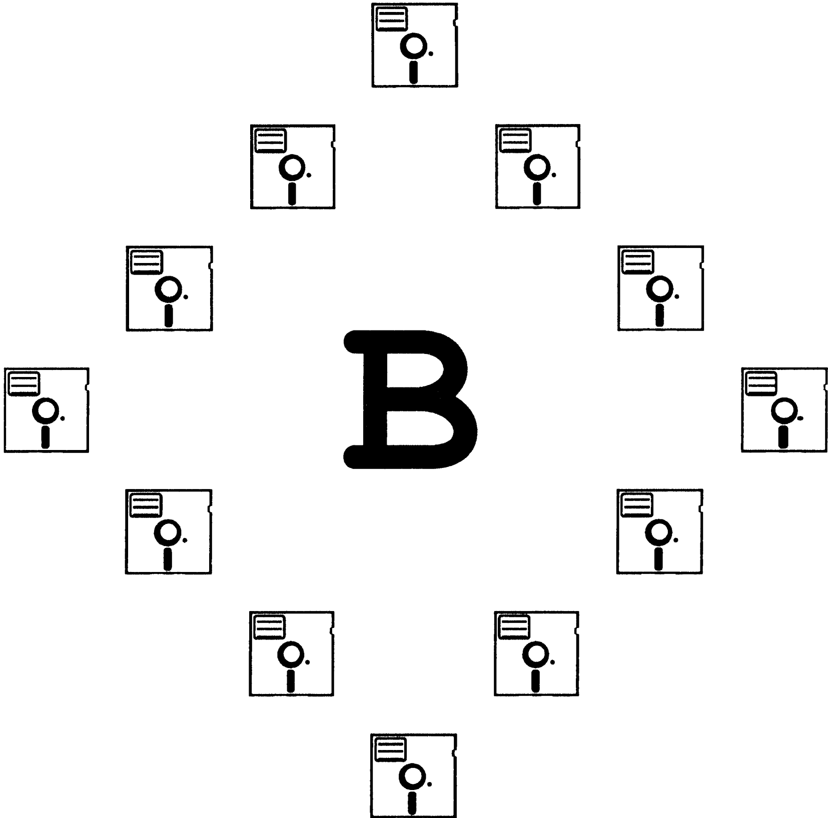
```
<Esc>SA!Dateiname!
```

Diese Kommandos können dann hilfreich sein, wenn man zwischendurch einmal speichern möchte.

Wenn Sie erst einmal die Esc-Taste betätigt haben, können in die Befehlszeile mehrere, von Kommata getrennte Kommandos eingegeben werden. Zum Beispiel:

```
F/irgendwas/;E/vorher/nachher/
```

Zuerst wird "irgendwas" gesucht, dann wird "vorher" beim nächsten Auftreten durch "nachher" ersetzt. Sie können auch bestimmen, wie oft ein Befehl oder gar die gesamte Befehlszeile ausgeführt werden soll, indem Sie dem oder den Befehlen eine Nummer voranstellen. Alternativ kann auch "RP" angegeben werden, damit der Befehl solange ausgeführt wird, bis das Dateiende erreicht ist.


B

Anhang B

Der Amiga-C-Compiler

Dieser Anhang erklärt, wie man den Amiga-C-Compiler, der ein Abkömmling des Lattice-C ist, zum Laufen bringt. Leser, die den Lattice-C-Compiler bereits von anderen Systemen her kennen, werden einige Probleme haben, ihn auf diese Version anzupassen.

Ich habe versucht – wo dies möglich war –, alle compilerabhängigen Details aus diesem Buch fernzuhalten. Für eine gemeinsame Basis, wurde von mir für alle Bespiele der Amiga-C-Compiler verwendet. Wenn Sie aber einen anderen Compiler benutzen, kann es notwendig sein, das Programm für Ihren Compiler umzuschreiben. Die Beispiele wurden so kurz und präzise wie möglich gehalten, damit der Aufwand für eine solche Aktion minimiert wird.

Starten des Amiga-C-Compilers

Wenn Sie erst einmal ein Programm geschrieben haben, sei es mit ED, Micro-Emacs oder einem anderen Texteditor, müssen drei Schritte durchgeführt werden, bevor das Programm ablaufbereit ist.

Erste Compilerphase

Die erste Compilerphase heißt LC1. Damit wird Ihr Programm von C in einen compilerabhängigen Zwischencode übersetzt. In diesem Durchgang werden die Syntax und die Struktur geprüft.

Ein typischer Aufruf aus dem CLI sieht folgendermaßen aus:

```
CD DF1:C ;in's Directory des Amiga-C-Compilers wechseln
LC1 -i:include/ DF0:hello.c TO DF0:hello.q
```

Dabei nehme ich an, daß Sie mit einem 512KB-Amiga und einem externen Laufwerk arbeiten, in dem sich die Amiga-C-Compiler-Diskette befindet. Auf der Diskette im internen Laufwerk sollte das zu compilierende Programm namens "hello.c" zu finden sein.

Diese Befehlsfolge erzeugt eine Datei "hello.q" im Hauptverzeichnis des internen Laufwerks. Der Teil

```
-i:include/
```

teilt dem Compiler mit, in welchem Verzeichnis er nach den Include-Dateien suchen soll, die bei der Übersetzung eingebunden werden sollen. Der Doppelpunkt weist auf das Hauptinhaltsverzeichnis dieser Diskette (df1:) hin, wobei allen in Ihrem Programm benutzten Include-Dateien noch "include/" vorangestellt wird.

Die Ausgabedatei erhält automatisch den Namen "hello.q", sofern Sie keinen anderen gewählt haben. Es wurde hier so angegeben, daß man leicht sehen kann, was der Compiler erzeugt. Die Dateikennung ".q" steht für Quad-File, der Name, den Amiga-C für Zwischendateien der ersten Phase verwendet.

Zweite Compilerphase

Die zweite Phase heißt LC2. Hier wird die Zwischendatei der ersten Phase in ein Object-File übersetzt. Unter der Voraussetzung, daß die erste Phase korrekt abgearbeitet wurde, führen Sie bitte Phase zwei aus, indem Sie die folgende Zeile eintippen:

```
LC2 df0:hello.q TO df0:hello.o
```

Sie erzeugt die Objektdatei für Ihr Programm.

Dritte Compilerphase

Die Objektdatei ist nicht die letzte Version des Programms. Es enthält alle von Ihnen gewünschten übersetzten Anweisungen, aber – zumindest nicht zu diesem Zeitpunkt – den Code der zu benutzenden Systemroutinen. Um ein vollständiges Programm daraus zu machen, müssen Sie die dritte Phase, den Alink aufrufen. Er verbindet Ihr Programm mit dem gewünschten Object-Code aus einer oder mehreren Funktions-Bibliotheken (debug.lib oder amiga.lib) oder einem oder mehreren bereits zuvor übersetzten Objektdateien.

Das Arbeiten mit einer Hochsprache – hier C – ist in diesem Fall ein Vorteil. Das Ergebnis nach der Übersetzung des Programms ist eine Datei mit verschiebbarem Code, den AmigaDOS überall in den Systemspeicher laden und als Prozeß oder Task ausführen kann. Das Programm ist in der Lage, alle Fähigkeiten des Rechners mit anderen Tasks zu teilen.

Wenn die beiden Phasen eins und zwei erfolgreich waren, benutzen Sie bitte hier den typischen Aufruf von Alink, der "hello" in ein lauffähiges Programm verwandelt:

```
ALINK FROM df0:hello.o+Lstartup.obj TO df0:hello
LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
```

Bitte alles in eine Befehlszeile eintippen! Sollte auch dieser Durchgang erfolgreich sein, können Sie nun Ihr Programm durch folgende Zeilen aufrufen:

```
cd df0:
hello
```

Zusammenfassung der Compileraufrufe

Hier sind nun alle Schritte zusammengefaßt, die zum Compilieren eines Programms mit Amiga-C nötig sind:

1. Programm mit dem gewohnten Texteditor erstellen

2. Phase 1 des Compilers (LC1) starten, der dann eine Datei mit der Endung ".q" produziert

3. Zweite Phase aufrufen, die auf der ".q"-Datei ein File mit der Endung ".o" erzeugt.

4. Phase 3 (Alink) starten, der das Programm mit anderen Object-Files und Libraries verbindet.

Das Ergebnis ist ein lauffähiges Programm.

Erstellen und Verwendung einer Make-Datei

Um die drei unterschiedlichen Compilerphasen zu starten ist eine ganze Menge Tipparbeit vonnöten. Die Amiga-C-Diskette enthält aber einen Befehlsinterpreter (execute), der Sie davon entbindet. Das teilweise ausführbare Programm wird Make-Datei genannt, da es zum Erzeugen anderer Dateien benutzt werden kann. Der Name der Datei, die alle drei Phasen des Compilers übernimmt, ist "make" und befindet sich im Directory "examples" auf der C-Diskette. In einigen früheren Versionen hieß diese Datei "makesimple", da dort die Datei "make" nur die ersten zwei Phasen des Compilers aufgerufen hat. Ein File namens "link" kümmerte sich dann um die dritte Phase. Wenn Sie ein umfangreiches Programm schreiben, das aus mehreren C-Dateien besteht, sollten Sie die Dateien einzeln übersetzen. Erst wenn alle Compilierungsvorgänge abgeschlossen sind, werden die ".o"-Dateien zu einem Programm gelinkt.

Hier ist so ein typischer Aufruf, der auf alle in diesem Buch gezeigten Programme angewendet werden kann:

1. Legen Sie die CLI-Diskette in das interne, die Diskette, die das zu übersetzende Programm enthält, in das externe Laufwerk ein. Dann geben Sie ein:

   ```
   copy hello.c to ram:
   ```

2. Sobald die LED am Laufwerk ausgeht, entnehmen Sie bitte die Programmdiskette und legen dafür die Amiga-C-Diskette in das interne Laufwerk. Dann

   ```
   cd df1:
   ```

   eintippen, das AmigaDOS dazu veranlaßt, das externe Laufwerk als das Verzeichnis für alle Befehle und benötigten Datenfiles zu benutzen.

3. Folgende Zeile eingeben:

```
execute examples/makesimple ram:hello
```

Mit diesem einzigen Kommando, wird AmigaDOS durch die Make-Datei damit beauftragt, alle drei Compilerphasen durchzuführen.

AmigaDOS übersetzt die Datei "hello.c" auf der RAM-Disk und erzeugt, falls die Compilierung erfolgreich war, eine ausführbare Datei namens "hello" auf der RAM-Disk. Um das Programm zu testen, brauchen Sie nur

```
run ram:hello
```

oder

```
ram:hello
```

einzutippen. Um nun die Datei, die gerade übersetzt und gelinkt wurde, zu speichern, entnehmen Sie wieder die C-Compiler-Diskette und legen dafür die Programmdiskette ein. Nun noch

```
copy ram:hello to df1:hello
```

eintippen und schon wird diese Datei auf Diskette gesichert.

Lassen Sie uns noch etwas näher auf den Aufbau einer typischen Make-Datei eingehen. Wenn Sie nämlich deren Arbeitsweise verstanden haben, ist es für Sie auch kein Problem, die Datei auf die eigenen Bedürfnisse anzupassen.

Inhalt einer Make-Datei

Die Beispiel-Datei ist eigentlich nur eine Textdatei mit Befehlen, die Amiga-DOS ausführen soll. Wenn Sie den Befehl

```
execute Dateiname
```

geben, lädt AmigaDOS das Programm Execute, das die angegebene Datei ausließt und deren Inhalt so behandelt, als würden Sie die Befehle im CLI direkt eintippen. Excecute können auch noch optionale Parameter übergeben wer-

den, die es ermöglichen, verschiedenste Programme mit der gleichen Make-Datei zu compilieren. Dateien, die Befehle enthalten, die von Execute ausgeführt werden sollen, werden Execute-Files genannt.

Diese Dateien enthalten eine Reihe unterschiedlicher Eingabezeilen. Darunter sind Kommentare, Ersetzungen von Parametern und Befehle, die andere Programme starten.

Beachten Sie, daß die Zeile, die in Spalte 1 ein Semikolon enthält, von AmigaDOS als Bemerkung behandelt wird. Auch Execute betrachtet Zeilen, die ein Semikolon in der ersten Spalte aufweisen als Kommentar, Zeilen mit einem Punkt als Befehl.

Parameterersetzung in einem Execute-File

Sie können mit dem Key-Befehl Execute anzeigen, daß Sie bestimmte Parameter ersetzen möchten. Wenn Sie dieses Kommando verwenden, erzeugt Execute eine Kopie der Datei in dem Verzeichnis ":T" des aktuellen Laufwerks, übernimmt die dem Kommando folgenden Parameter aus der Befehlszeile und ersetzt die Werte der Zeichenkette an den von Ihnen festgelegten Stellen.

Betrachten wir die Datei "typeit", die die folgenden Zeilen enthält:

```
.key ersterpar, zweiterpar
IF EXISTS "<ersterpar>"
TYPE "<ersterpar>"
ENDIF
IF EXISTS "<zweiterpar>"
TYPE "<zweiterpar>"
ENDIF
```

Tippen Sie nun

```
execute typeit dateil datei2
```

ein, so erzeugt Execute eine Kopie der Befehlsdatei auf Diskette, ersetzt den String "ersterpar" durch den ersten Parameter – hier "dateil" – überall dort, wo <ersterpar> auftritt (die in Größer-/Kleinerzeichen gefaßten Namen des Key-Befehls). Entsprechendes gilt für den zweiten Parameter.

Das Ergebnis ist:

```
IF EXISTS DATEI1
TYPE DATEI1
ENDIF
IF EXISTS DATEI2
TYPE DATEI2
ENDIF
```

Die Datei wird ausgeführt, als hätten Sie alle Befehle über Tastatur eingegeben. Diese Beispieldatei geht nach dem Schema vor: "Sieh' nach, ob die Datei vorhanden ist. Ist dies der Fall, zeige sie auf dem Bildschirm an."

Das gleiche Prinzip wird mit dem C-Compiler verbunden. Sie verwenden ED, MicroMacs oder einen anderen Texteditor, um sich den Inhalt der Datei "examples/make" auf der Amiga-C-Diskette anzeigen zu lassen. Sie werden feststellen, daß das Key-Kommando Parameter für den Aufruf von LC1, LC2 und Alink enthält.

Listing B.1 ist ein typisches Beispiel einer Make-Datei. Die Nummern, die man an der linken Seite erkennt, sind nur zu Verweisen aus dem Text nötig und nicht Bestandteil dieses Files.

In Zeile 1 werden zwei Namen für zu ersetzende Parameter angegeben: "sourcename" und "listingname". Wenn die Datei "make "heißt, wird sie durch

```
execute make meinedatei
```

aufgerufen und compiliert die Datei "meinedatei". Für den Fall, daß der Compiler Fehler entdeckt, sollte die Make-Datei abbrechen.

Zeile 6 prüft, ob ein Name für die Source-Datei angegeben wurde. Trifft dies nicht zu, werden die Zeilen 23–25 mit dem entsprechenden Hinweis ausgeführt. In Zeile 9 prüft man, ob die angegebene Datei vorhanden ist. Wenn nicht, wird dies durch die Zeile 28 mitgeteilt.

Zeilen 13 und 15 zeigen in Abhängigkeit, ob ein Listingname bekannt ist, zwei verschiedene Möglichkeiten den LC1 aufzurufen. Die Option -i zeigt dem Compiler, wo er beim Auftreten von Include-Dateien zu suchen hat. Include-Dateien enthalten Konstantendefinitionen, Makros und Datenstrukturen, die man benötigt, wenn das Amiga-Betriebssystem benutzt werden soll. Als Pfadname zur den Include-Dateien ist hier als "C1.0:include" angegeben, was bedeutet, daß sich diese auf der C-Compiler-Diskette befinden. Wenn Sie die Dis-

```
 1  .key sourcename,listingname
 2  ;sourcename ist der Name des C-Programms
 3  ;listingname ist der Name der Datei, in die das Listing,
 4  ; falls es erzeugt wurde, geschrieben wird
 5  ;
 6  IF "<sourcename>" EQ ""
 7  SKIP USAGE
 8  ENDIF
 9  IF NOT EXISTS <sourcename>.c
10  SKIP NOTFOUND
11  ENDIF
12  IF "<listingname>" EQ ""
13     LC1 > <Listingname> -iC1.0:include/ <sourcename>.c
14  ELSE
15     LC1 -iC1.0:include/ <sourcename>.c
16  ENDIF
17  ;     Nun Phase 2
18  LC2 <sourcename>.c
19  ;     nur noch linken
20  ALINK FROM <sourcename>.o+Lstartup.obj TO <sourcename>
       LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
21  SKIP DONE
22  LAB USAGE
23  ECHO "Aufruf: make sourcename listingname
24  ECHO "sourcename ist zwingend, stellt sourcename.c dar"
25  ECHO "listingname ist optional"
26  SKIP DONE
27  LAB NOTFOUND
28  ECHO "<sourcename>.c nicht gefunden!"
29  LAB DONE
```

Tab. B.1: Beispiel für eine Make-Datei

kette mit dem AmigaDOS-Befehl Relabel umbennen, müssen Sie hier dann den aktuellen Namen einsetzen. Besitzen Sie beispielsweise C1.1 anstelle von C1.0, so können Sie den angegebenen Namen überprüfen, indem Sie das CLI-Kommando INFO aufrufen. Es zeigt Ihnen dann den Namen der Diskette. Er würde z.B melden:

```
C1.0[mounted]
```

Zeile 20 zeigt einen möglichen Aufruf von Alink. Diese Anweisung linkt ein einzelnes File mit dem Startup-Code und durchsucht die verschiedenen Bibliotheken nach fehlendem Objekt-Code, der zur Komplettierung benötigt wird.

Wenn Sie verschiedene Programmteile einzeln compilieren wollen, wie es auch bei einigen Beispielen in diesem Buch nötig ist, brauchen Sie vielleicht eine Version der Make-Datei, die die Zeile 20 nicht aufweist. Dann benutzt man eine weitere Datei, vielleicht "link" getauft, die etwas Ähnliches wie die folgenden Zeilen enthält:

```
.key linkwas,object
ALINK FROM <linkwas>.o+Lstartup.obj TO <object>
      LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
```

Sie starten die Link-Datei, indem Sie die Zeile eintippen:

```
execute link "fa.o+fb.o+fc.o" gesamtname
```

Dabei werden die Objektdateien direkt in Anführungszeichen dem Execute-Befehl übergeben, damit diese Zeichenkette als ein einziger Parameter bei der Ersetzung berücksichtigt wird. "gesamtname" ist der Name der ausführbaren Datei, die von "link" erzeugt wird.

Im "AmigaDOS User's Manual" finden Sie viele weitere Beispiele, die das Execute-Kommando verwenden.

Wie man MAKESIMPLE.A erstellt

In Kapitel 2 habe ich auf eine Datei namens "makesimple.a" verwiesen, die ein Abkömmling der auf der C-Diskette enthaltenen Datei "examples/make" oder "examples/makesimple" ist. Um "makesimple.a" zu erzeugen, führen Sie bitte folgende Schritte durch:

1. Kopieren Sie makesimple (oder Make) in ein File mit dem Namen "makesimple.a". Tippen Sie nun im CLI ein:

   ```
   copy df1:examples/makesimple to df1:examples/makesimple.a
   ```

2. Verwenden Sie einen Editor, um die folgenden Änderungen durchzuführen: Die Zeile

   ```
   :c/lc2 <file>
   ```

in

```
:c/lc2 -v <file>
```

umschreiben. Dies verhindert die Kontrolle des Stacks, das sonst zum Programm hinzugebunden würde.

3. Ändern Sie des weiteren

```
:c/alink FROM LIB:LStartup.obj+<file>.o TO <file>
   LIBRARY LIB:lc.lib+LIB:amiga.lib
```

in

```
:c/alink FROM LIB:AStartup.obj+<file>.o TO <file>
   LIBRARY LIB:amiga.lib+LIBRARY LIB:lc.lib
```

Hinweis: Betätigen Sie erst dann <Return>, wenn Sie alles eingegeben haben. Es ändert das Startup-File und die Reihenfolge der Libraries. Dies bedeutet, daß man einige Einschränkungen der Routinen aus "amiga.lib" in Kauf nimmt. (Sehen Sie dazu im "Amiga ROM Kernel Manual" nach.) Oft wird dann die maximale Größe des Execute-Files von 12000 Bytes auf weniger als 2000 Bytes vermindert.

Wenn Sie "makesimple" anstelle von "make" verwenden, werden die printf() und sprintf()-Versionen aus amiga.lib eingebunden, die eingeschränkte Formatierungmöglichkeiten bieten. (Erwähnenswert ist hier, daß keine Fließkommazahlen bearbeitet werden können.) Sonst würden printf() und sprintf() aus der Lattice-Bibliothek gelinkt werden. Dies ist auch Grund dafür, daß die Länge des entstandenen Programms verringert ist. Beachten Sie, daß dies für einige Programmarten Konsequenzen hat, deren Sie sich bewußt sein müssen, bevor diese Fähigkeiten benutzt werden.

Zur Erinnerung: Die Zielsetzung des Make-Files war, den Code so portabel wie möglich zu halten, so daß er von den verschiedenen Compilern (Lattice-C, Aztek-C oder andere) übersetzt werden kann. Benutzen Sie deshalb lieber die AmigaDOS-Version Open(), Read() oder Close() anstelle der compilerabhängigen open(), read() oder close() aus Kapitel 2.

Denjenigen Lesern, die in C mit den IO-Routinen experimentieren, sei gesagt, daß zwischen den von AmigaDOS und den von den unix-ähnlichen Funktionen aus den Compiler-Bibliotheken gelieferten Handles ein Unterschied besteht. Sie dürfen nicht vertauscht werden.

Stichwortverzeichnis

AbortIO 128
ADCMD_ALLOCATE 378
AddFont 187
AddHead 90
AddTail 90
AllocMem 60, 83, 390
AllocRaster 177
AllocSignal 93
Amiga-C-Compiler 441
  Starten des 441
AmigaDOS 31, 401
  Arbeiten mit 49
  Fehlernummern von 54
Animation 313
APen 143
AreaDraw 175
AreaEnd 175
AreaMove 175
Argumentübergabe 33
AskFont 188
AskSoftStyle 193
Audio-
  Ausgabe
    löschen 390
    Kontrolle der 386
  Daten 387
  Device
    Kommunikation mit dem 372
  Hardware 371
  Programm 391
  Sòftware 374
  Daten, Plazierung der 390
Auflösung 23
Ausgabe auf den Drucker 42
AvailFonts 188

Befehle
  für Suchen und Ersetzen 437
  senden 126
  zur Cusor-Bewegung 435
BeginIO 127
Bibliothek 25, 106
  öffnen 113
  schließen 117
  Namen und Basisadressen der 114
  Struktur einer 109
Bildpunkte 178
Bildschirm öffnen 166
Bildschirmbereiche löschen und
  verschieben 194
Bit-Map 196
BitMap-Daten kopieren 198
Blitter Objects 251
Block-Befehle 438
BltBitMap 198
BltBitMapRastPort 198
BOBS 251
Bobs
  Nachteile eines 358
  Vorteile eines 358
Boot-Diskette bestimmen 75
BPen 144

CD 48

ChangeSprite 316, 321
CheckIO 128
Checkmarks 250
ClearEOL 194
ClearScreen 194
CLI 24, 31
CLI-Fenster öffnen 39
ClipBlit 198
ClipBlitTransparent 204
Close 36
CloseDevice 133
CloseLibrary 117, 165
CMD_WRITE 390
CON 39
Console Device 293
Coprozessoren 84
CreateAudioIO 375
CreateDir 69
CreatePort 98
CreateTask 403, 404
CurrentDir 54, 56
Custom Messages 105
Custom Screen 166
  Definition 167
  Öffnen 167
  Öffnen eines Fensters auf dem 168

Datei 35
  lesen 37
  löschen 68
  öffnen 36
  schließen 36
  schreiben 38
  schützen 69
Datum auslesen 71
Delay 39, 75
DeleteGels 356
DeletePort 99
Device 23, 118, 285
  -namen 123
DeviceProc 76
DIR 52
Directory 47
Diskettenname ändern 76
DMA 372
DoIO 127
Draw 146
Drucker 42

ED 433
ED-Kommandos 435
Editor 433
Ein-/Ausgabeoperation 119
Einfüge- und Lösch-Befehle 436
Enqueue 90
Examine 54
Exec 79
Execute 49
ExNext 54

Farben 143
  Auswahl der 170
  Bestimmen der aktuellen 172
Farbregister 353
Fenster
  öffnen 136
  Ausgabe von Grafik in ein 142
  Initialisierung und Definition 136
  -Ein-/Ausgabe 38
File 35
file handle 35
FILENOTE 70
FindName 90
FindPort 425
FindTask 424
Flood 173
FollowPath 64
FreeMem 85
FreeRaster 197
FreeSprite 320
Freigabe von Dateien 55
Füllen mit eigenen Mustern 174
Füllroutine 173
Funktionsbibliotheken 25

Gadgets 223, 257
  Anwahl von 260
  Aussehen eines 257
  Automatisches Freigeben von 259
  Boolean- 263

Größe eines 257
Identifikation von 258
Position eines 257
Proportional- 266
String- 264
Gadgettypen 259
Gadgetverarbeitung 270
Gameport 306
Gel-System 313
Gel-Systems, Intitialisierung des 332
Gerät
virtuelles 118, 285
schließen 133
GetMsg 97
GetRGB4 172
GetSprite 315
GIMMEZEROZERO 194
Grafik 135, 278
bewegte 313
-daten kopieren 200
-en, farbige 247

HandleEvent 140
Hauptverzeichnis 48

IDCMP 223
IDCMPFlags 136
Image 250
Include-Dateien 29
Info 71
INFO 448
Inhaltsverzeichnis 47
InitArea 176
InitBitMap 196
InitRequester 255
InitTask 403
InitTimer 289
InitTmpRas 177
Insert 90
IntuiMessage 219
Intuition 23, 215
Kommunikation mit 216
IoErr 37, 54
IORequest-Blocks, Struktur des 122
IsInteractive 45

JOIN 43
Joystickport 306

Kanal 371
abschließen 383
Reservierung eines 375
Verwendung eines 382
Kommandosequenzen 301
Kommentar 70
Kommunikation zwischen Tasks 423
Kommunikationsleitung 98
auffinden 100
hinzufügen 100
Verwendung einer 101
wegnehmen 99
Kontrollsequenz 294

LastColors 336
Lautstärkeregelung 391
Linien
punktierte 156
Zeichnen mehrerer 157
Linkfile 414
List Node 88
Liste, Kopfzeile einer 87
Listen 80
LoadRGB4 171
Lock 51
-Funktion 54

make 414
Make-Datei
Erstellen und Verwendung einer 444
Inhalt einer 445
MakeBob 251
MAKEDIR 67
MakeVSprite 337
Malprogramm 271
Maus
-port 306
-taste 238
-zeiger lokalisieren 236
Menü
Erstellung und Verwendung 239
Text oder Grafik 245

-komponenten 239, 279
Initialisierung von 244
-punkten
Automatisches Freigeben von 252
Hervorheben von 253
Kombination von Text und Grafik 279
Positionierung von 246
Initialisierung 240
-verarbeitung 269
Message 93, 219
Port 93
MoveSprite 321
Multitasking 397
Muster, Allgemeines über 154
Mutual Exclusion 252

Nachricht 93, 102, 219
Inhalt einer 103
eigene 105
NewList 87
NewScreen 168, 230
NewWindow 169, 232

Open 36
OPen 145
OpenDevice 118, 376
OpenDiskFont 185
OpenFont 185
OpenLibrary 113
OpenScreen 168
OpenWindow 139

PAL 23
PAR 43
Parameterersetzung in einem Execute-File 446
ParentDir 54, 61
Pixel
lesen 178
setzen 178
PlaneOnOff 354
PlanePick 354
PolyDraw 157
Ports aufspüren 425
printf 450
Priorität 89
setzen 385
Programmierung in
68000-Assembler 28
C 28
PROTECT 67
Prozeß 82, 398
-Programme 415
aufspüren 424
PRT 43
PurgeGels 356
PutMsg 98

Rand- und Tabulator-Befehle 437
RASSIZE 177
RastPort 140
RAW 41
RAWKEY 224, 298
Read 37
ReadPixel 178
ReadyGels 332
Rechtecke
mit mehrfarbigen Mustern 153
mit zweifarbigen Mustern 151
zeichnen 150
RectFill 150
RELABEL 76
RemHead 90
Remove 90
RemTail 90
RENAME 67
Reply Ports 131
ReplyMsg 99
Requester 255
Root 47
RUN 49, 400

Sampling Period 389
Scan-Codes 299
Schreib-/Lesezeiger 45
Schriftarten 192
Screen 23, 136
Scrollen 194
ScrollRaster 195

Seek 45
SendIO 126
SER 43
SetAfPt 152
SetAPen 143
SetBPen 144
SetComment 70
SetDrMd 145
SetFont 186
SetOPen 145
SetProtection 69
SetRast 194
SetRGB4 170
SetSignal 95
SetSoftStyle 192
Signal-Bits
  reservieren 93
  setzen 95
  Verwendung mehrerer 95
Signal-Erkennung 102
Signale 92
  beim Multitasking 93
Simple-Sprites 313
SimpleSprite 314
Slider 266
SMART_REFRESH 161
Sondertasten 298
Sound 371
Speicherfreigabe 85
Speichern und Beenden 438
Speicherreservierung 83
SpriteHead 333
Sprites
  Ändern eines 316
  Daten eines 317
  Farben eines 318
  Farbregister eines 319
  Freigabe eines 320
  reservierte 333
  Reservierung eines 315
SpriteTail 333
Stack 450
Stapelverarbeitung 131
Startup-Code 402
Stereokanal 377
SUPER_BITMAP 162
Systemsoftware 20, 22

Task 82, 397
  aufspüren 424
Tastatur 306
  -codes 298
Text 148, 183, 278
  Darstellungsweise 191
TextAttr 185
Textausgabe 32
Texteditor 433
TextLength 148
Timer Device 286
Tonhöhe 389

Umleitung der Standard-
  Ein-/Ausgabe 34
UnLock 55
Untermenü 280

VANILLAKEY 224
Verzeichnis
  anlegen 69
  wechseln 56
  aktuelles 61
Virtual Sprites 330
  Nachteile eines 331
  Vorteile eines 331
VSprite 337

Wait 93
WaitForChar 45
WaitIO 128
Warteschleife 75
Wellenform, Definition der 388
WindowFlags 138
Workbench 24
Write 38
WritePixel 178

Zeichen
  -codes 294
  -modus, Auswahl des 145
  -satz

bestimmen 187
in RastPort einbinden 186
in Zeichensatzliste eintragen 187
öffnen 184
Zeitgeber 286

Fordern Sie ein Gesamtverzeichnis unserer Verlagsproduktion an:

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
4000 Düsseldorf 30
Tel.: (02 11) 61 802-0
Telex: 8 588 163

SYBEX INC.
2021 Challenger drive, NBR 100
Alameda, CA 94501, USA
Tel.: (4 15) 523-8233
Telex: 287 639 SYBEX UR

SYBEX
6–8, Impasse du Curé
75018 Paris
Tel.: 1/4203-95-95
Telex: 211.801 f

Amiga
Das Programmierbuch

Dieses Buch ist ein Muß für Programmierer, Anwender und Software-Entwickler – besonders für die Arbeit in C, Pascal oder Maschinensprache. Alle Beispielprogramme wurden in Amiga C geschrieben.

Aus dem Inhalt

- Organisation des Amiga-Systems
- die Hauptkomponenten (AmigaDOS, Exec, Grafik, Intuition, Devices, Sound, Animation)
- Programmierung auf dem Amiga mit Details der AmigaDOS-Funktionen und des Dateien-Systems
- schrittweise aufeinander aufbauende Programmbeispiele, die den Gebrauch der System-Routinen veranschaulichen
- gebrauchsfertige Routinen, die die Programmierung von Grafik, Animation, Geräte-Kontrolle und die Bearbeitung von Anwender-Eingaben vereinfachen
- eine Einführung in *Multitasking* und die Kommunikation zwischen Tasks und Prozessen
- vollständige Anleitung für Compiler und Texteditor im Anhang.

Über den Autor

Robert A. Peck, ein Ingenieur und Programmierer, leitete das Amiga-Dokumentations-Team. Er hat bereits zahlreiche Computer-Handbücher, Lehrbücher und Referenz-Handbücher verfaßt, einschließlich des Amiga Hardware Manuals und des Amiga ROM Kernel Manuals, außerdem zahlreiche Artikel für Fachzeitschriften. Er lebt in Los Gaos, Kalifornien.

ISB N 3-88745-520-7

DM 49,—
sFr 45,10
öS 382,—