

**Das Standardwerk
zum Amiga**

SYBEX

Amiga Profibuch



Garry Glendown / Roland Haas

mit
Programmdiskette



Das Amiga Profibuch

Das Amiga Profibuch

Garry Glendown
Roland Haas



DÜSSELDORF • SAN FRANCISCO • PARIS • LONDON • ARNHEIM

Anmerkungen:

MS-DOS und Amiga BASIC sind eingetragene Warenzeichen von Microsoft Corp., USA.

Amiga, Amiga 500/1000/2000 und AmigaDOS sind eingetragene Warenzeichen von Commodore-Amiga Inc., USA.

IBM, PC-DOS und PC/XT/AT sind eingetragene Warenzeichen von International Business Machines Corp., USA.

Deluxe-Paint ist ein geschütztes Warenzeichen von Electronic Arts, USA.

Z 80 ist ein geschütztes Warenzeichen von ZILOG Inc., USA.

Macintosh ist ein geschütztes Warenzeichen von Apple Computer Inc., USA.

Centronics ist ein geschütztes Warenzeichen von Centronics Data Computer Corp.

ATARI ST und TOS sind eingetragene Warenzeichen von Atari Corp., USA.

68000, 68008, 68010, 68020, 68030 und 68881 sind eingetragene Warenzeichen von Motorola Inc., USA

Fast alle Software- und Hardwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen und sollten als solche betrachtet werden.

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.

Satz: Textservice Brigitte Witzer, Ratingen

Belichtung: Softtext Computersatz-Service GmbH, Düsseldorf

Titelfoto: Studio Gladow, Düsseldorf

Farbproduktionen: Rheinische Reprotechnik, Düsseldorf

Druck und buchbinderische Verarbeitung: Bercker, Kevelaer

ISBN 3-88745-580-0

1. Auflage 1990

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 © 1990 by SYBEX-Verlag GmbH, Düsseldorf

Inhaltsverzeichnis

Einleitung	1
------------	---

Teil 1 – Die Hardware

Kapitel 1: Die Systemarchitektur

Vom Exoten zum "Chartbreaker" – die Entwicklungsgeschichte des Amigas	8
Die Hüllen fallen – grundlegende Aspekte der Systemarchitektur	9
DMA – der Turbo für die Busse	10
Fat Agnus – Schaltstelle des Systems	11
Elegant gelöst – die Realisierung des DMA-Systems	11
Kommunikationsfreudig – Prozessor und Systembausteine	13
Die verschiedenen Modelle der Amiga-Familie – ein Überblick	14
Der Familienvater – A1000	14
Der Bestseller – A500	15
Flexibilität ist Trumpf – der Amiga 2000	16

Kapitel 2: Die CPU

Die Pinbelegung des MC68000	24
Die Stromversorgung: Vcc + GND	26
Der Takteingang: CLK	26
Der Datenbus: D0...D15	26
Der Adreßbus: A1...A23	27
Die Bussteuerung	29
Die Buszugriffssteuerung: BR*, BG*, BGACK*	31
Die Unterbrechungssteuerung: IPL0*...IPL2*	
(IPL = Interrupt-Pending-Level)	33
Die Funktionscodes: FC0...FC2	34

Die Systemsteuersignale: BERR*, RESET*, HALT*	35
Die Betriebszustände des MC68000	37
Ausnahmereverarbeitung mit dem MC68000	37
Herzverpflanzung – Systemtuning mit anderen Prozessoren aus der 68000-Familie	40
Die erste Hürde – Systemsoftware und 680xx	40
Der Weg des geringsten Widerstandes – der 68010 im Amiga	42
Der Amiga mit Workstationambitionen – der MC68020	44
Warum nicht träumen? – der MC68030	45

Kapitel 3: Die Customchip-Hardware

Registerarchitektur und Programmiermodell	48
Die Adreßregister	50
Die Daten- oder Parameterregister	51
Die Kontrollregister	51
Impulsgeber" – die Strobe-Register	54
Die Customchip-Register im Überblick	54
Fat Agnus – der Herr der Busse	62
Fat Agnus, der DMA-Controller	65
Der "schlanke" Vorgänger – Agnus	67
Erläuterungen zur Pinbelegung von Fat Agnus	69
Denise – mehr als nur ein Videoshifter	74
Pixel-Adressierung contra Bitplane-Adressierung	74
Metamorphose – wie Bits zu Pixeln werden	78
Erläuterungen zur Pinbelegung von Denise	79
Paula – Tor zur Außenwelt	82
Der Interrupt-Controller	84
Aufgaben und Funktion eines Interrupt-Controllers	85
Die Programmierung des Interrupt-Systems	85
"Weiche Unterbrechungen" – die SOFT-Interrupts	88
Erläuterungen zur Pinbelegung von Paula	89

Kapitel 4: Die CIA-Hardware

Der parallele Port	97
Die Timer	98
Troubleshooting – Übertragungsfehler sind vermeidbar	101

Ein Timer für alle Fälle – die Programmierung	101
"Timer an alle" – der Ausgabemodus	102
Der serielle Port	104
Bits im Gleichschritt – der Synchronisationstakt	105
Der Event-Counter	106
Der Ereigniszähler als Taktgeber	107
"Break Dance" – die CIA-Interrupts	108
Die Systemeinbindung der CIA-Hardware	111
Die CIAs – Mädchen für alles	111

Kapitel 5: Vom DMA-System zur Speicherorganisation

Das DMA-System unter der Lupe	116
Der Amiga ohne DMA? – ein paar theoretische Überlegungen	116
"Platzhalter" – das Speicher-Interface des MC68000	116
"Slotmaschine" – das Timing des DMA-Systems	118
"Weichenstellung" – wenn der Prozessor auf die "schiefe Bahn" gerät	119
"Rush Hour" – Verkehrsplanung auf dem CHIP-RAM-Bus	119
"Wanderdüne" – die Bitplane-DMA	120
Die PAL-Raserverzeile als Timing-Grundlage	121
Die Speicherbelegung	122
"Verkabelung" – die Arbeitsweise der Adreßdecodierung	122
Fallstudie 1 – Adreßeinbindung der Customchip-Hardware	125
Fallstudie 2 – Adreßeinbindung der CIA-Hardware	126
Dürfen es auch ein paar Byte mehr sein? – die Kapazität des RAM-Speichers	128
Ein Ausflug in die Technologie dynamischer RAM-Chips	130
"Innenansichten" – die Architektur eines RAM-Bausteins	130
Von Zugriffszeiten, Zyklusdauer und Refreshmechanismen	134
Die Speicherorganisation	137

Kapitel 6: Die Interface-Hardware Teil 1

Die Centronics-Schnittstelle	142
Die Signale der Parallel-Schnittstelle	144
Aufbau und technische Realisierung der Centronics-Schnittstelle	144
Die serielle Schnittstelle	146
Die Signale der seriellen Schnittstelle	148
Aufbau und technische Realisierung des RS232-Interfaces	148
Asynchrone Datenübertragung am RS232-Port	150

Die Programmierung der UART-Schaltung	151
Nicht nur zum Spielen geeignet – die Game-Ports	155
Aufbau und Schaltung der Game-Ports	155
Flinkes Nagetier am Game-Port – die Maus	157
Der Joystick am Game-Port	161
Paddles, XY-Joysticks, Potentiometer	163

Kapitel 7: Die Interface-Hardware Teil 2

Die Tastatur	170
Tastatur mit "Köpfchen" – der Tastaturprozessor	173
"Tastsinn" – die Schaltung der Tastaturplatine	174
"Small-Talk" – die Kommunikation zwischen Rechner und Tastatur	177
Informationen bitte – die Zustandcodes	179
Vor Mithörern wird gewarnt – RESET über die Tastatur	180
Das Floppy-Disk-Interface	181
Die Signale der Floppy-Schnittstelle	183
Die Signale des Floppy-Controllers: DKWD*, DKRD*, DKWE*	183
Selektion, Motorsteuerung und Initialisierung:	
SELX*, MTRX*, DRES*, RDY*	183
Sputreue – die Positionierung des Kopfes:	
SIDE*, STEP*, DIR*, TK0*	184
Die Zustandsinformationen: WPRO*, CHNG*, INUSE*, INDEX*	185
Gesichtskontrolle am Floppy-Port – der Identifikationsmodus	185
Die Aufzeichnungsformate	186
MFM	187
GCR	187
Die Programmierung des Floppy-Controllers	188

Kapitel 8: Das Audio-System

Einführung in die Akustik	194
Tonhöhe und Frequenz	194
Schalleistung und Lautstärkeempfindung	195
Partialtöne, Resonanz und Klangfarbe	198
Wellensalat – von Rausch- und Knacksgeräuschen	199
Theoretische Grundlagen der Analog-/Digitalwandlung	200
Auflösung und Quantisierungsfehler	201
Die Abtastfrequenz	203

Die Aussteuerung	205
Die Audio-Hardware des Amigas	206
Der D/A-Wandler	206
Der Tiefpaßfilter	208
Die Audio-Register	210
Der Ausgabezyklus	213
Frequenz- und Amplitudenmodulation	215
Soft-Tuning oder: wie man der Hardware auf die Sprünge hilft	216
Longplay – Samples mit mehr als 128 KByte	216
Höhere Ausgabefrequenz, größere Auflösung	218
Sampeln wie ein CD-Player	219
Der Amiga als Synthesizer	221
Rekonstruieren statt Digitalisieren – Wellenformen selbst entworfen	221
Alles streng nach Noten	223
Arbeitserleichterung – eine universelle Initialisierungsroutine	224

Kapitel 9: Die Grafik-Hardware

"Alles im Griff" – der Copper	228
Die Copper-Befehle	228
Befehlsformat und Ausführungszeit	229
Der MOVE-Befehl	230
Der Skip-Befehl	234
Aufbau einer Copper-Befehlsliste	235
Die Initialisierung der Copper-Hardware	235
Die Copper-Programmierung	238
Programmbeispiel 1: Schleifen in der Copper-Befehlsliste	239
Programmbeispiel 2: Copper-Liste im Interlace-Modus	240
Programmbeispiel 3: Color-Cycling mit dem Copper	241
Modifikation der Copper-Liste	242
Rasterstrahl-Interrupts	243
Synchronisation zwischen Copper und Blitter	243
Sprites	244
Aufbau der Datenliste	244
Farbselektion	244
Multi-Color-Sprites	246
Definition der Bildschirmposition	247
Die Programmierung der Sprite-Hardware	249
Die Register	249
Initialisierung	251

Animation	253
Definition der Bildschirmpriorität	254
Die Kollisionserkennung	255

Teil 2 – Die Systemsoftware

Kapitel 1: Exec

Am Anfang ist die ExecBase	262
Listen	267
Listenverwaltung	267
Routinen zur Listenmanipulation	269
Die Libraries	270
Wie man mit Libraries umgeht	270
Libraries Marke "Eigenbau"	271
Eine Anleitung im SingleStep...	271
Die Library-Einsprünge	272
Der Library-Knoten	273
Legen wir los...	274
Die Tasks	283
Zustände eines Tasks	284
Steuerung der Tasks	285
3 – 2 – 1 – Feuer! Wir starten einen Task...	286
... und halten ihn wieder an	287
Interprozeßkommunikation oder: CB-Funk für Tasks	287
Wir beantragen einen Kanal ...	288
Warten auf das Signal ...	289
Sohn an Vater: bitte kommen... oder:	
Wie man Signale erzeugt	289
Ampeln im Amiga oder: Wir stoppen Tasks ...	290
Ausnahmen – aber nur nach den Regeln!	290
Vorsicht Falle!	291
Interrupts	292
Bitte nicht unterbrechen...	293
Bearbeiten von Interrupts	294
Software-Interrupts	295
"Störung unerwünscht"	295
Messages und Ports	296
Kommunikation im System	296

Wir bauen einen Hafen...	297
Ab geht die Post...	298
Abschicken und Empfangen	299
IO	300
Allgemeine Ein- und Ausgaben	300
Zugriff auf die Devices	302
Standard-IO-Befehle	303
Memory	304
AllocMem	304
Allocate	307

Kapitel 2: Intuition

Intuition	310
Windows – und was dazu gehört	310
Gadgets	318
Proportional-Gadgets	324
String-Gadgets	326
Menüs	327
60543 Menüpunkte – und wie man damit umgeht	331
Requester und Alerts	332
Intuition – IO für Anfänger?	333
Eingaben leicht gemacht!?	333
Warten – aber richtig!	335
Grafik unter Intuition	337
Border	338
Intuition – Texte	339
Images	339
Was es sonst noch so gibt ...	341
"Ordentliche" WB-Programmierung	342

Kapitel 3: Das Disk-Operating-System

Disketten-Format	344
Der Root-Block	345
Der Directory-Block	346
Der File-Header-Block	347
Der File-List-Block	349
Der Data-Block	349

Das FastFileSystem	350
Die FileSystem-Kommandos	352
Die Datenorganisation auf Disk	357
Wie man ein File findet...	358
Hashing beim AmigaDOS/FileSystem	358
Struktur der Binärfiles	359
Kapitel 4: Die Library-Routinen	371
Schlußwort	843
Anhang A: Befehlsausführungszeiten des MC 68000	845
Anhang B: Alphabetische Auflistung der Customchip-Register	853
Anhang C: Die Guru-Meditation-Fehlermeldungen	887
Anhang D: Library-Routinen nach Offset sortiert	893
Anhang E: Liste der Tastencodes	927
Anhang F: Das IFF-Grafikformat	933
Anhang G: Die C-Operatoren und ihre Prioritätsverteilung	953
Anhang H: Die Diskette zum Buch – Programme	957
Stichwortverzeichnis	1001

Einleitung

Programmieren auf dem Amiga

– aller Anfang ist schwer oder: Wie alles losging...

Der Amiga hat in seinem "Leben" ja schon so einiges durchgemacht. Geplant wurde er von einer Firma um Jay Miner, die sich als Joystick-Hersteller tarnte. Und dann sollte er auch nicht dem "normalen" User, sondern Spielhallen-Freaks zur Verfügung stehen. Doch dies konnte glücklicherweise durch den drohenden Konkurs abgewendet werden...

Im Januar '84 wurde der Amiga, noch mehr Schein als Sein, einigen Entwicklern vorgestellt, die prompt von diesem Rechner begeistert waren. Doch die Geldgeber waren nicht bereit, mehr Geld in das Projekt Amiga zu stecken. Außer dem Verkauf an eine andere Firma blieb kein anderer Ausweg.

Da Atari der Firma Geld geliehen hatte, lag der Verkauf an sie nahe. Beschleunigt wurde das, nachdem Jack Tramiel von Commodore, die Firma, in der er den C64 groß gemacht hatte, zu Atari überwechselte. Durch den Kauf der Rechte am Amiga wollte er Commodore "den Garaus" machen. Leider (für ihn) war er aber sehr knausrig, er setzte Amiga Inc. unter Druck und wollte nur 70 Cent pro Aktie zahlen (Nennwert: \$1).

Als der Verkauf knapp bevorsteht, schaltete sich der Commodore-Chef Gould ein und bot Amiga Inc. pro Aktie \$4 (obwohl er nur wenig vom Amiga gehört hatte). Amiga nahm an, Commodore übernahm Amiga für 27 Millionen Dollar.

Von da ab stellte Commodore einige Bedingungen, darunter eine besonders wichtige: der Amiga sollte keine Spielekonsole, sondern ein PC werden. Dabei entstand ein Problem: Eine Spielekonsole, was der Amiga bis dahin war, braucht kein DOS, ein PC

kommt ohne es nicht aus. Eine von der Amiga-Firma engagierte Programmierer-Truppe schaffte es nicht, das DOS fertigzustellen und kündigte Ende '84 den Vertrag. Bereits kurz darauf wird den Programmierern des Amigas von Commodore "Tripos" ein DOS der Firma Metacomco, übergeben. (Dies ist auch der Grund, warum das DOS der einzige Teil des Amigas ist, der in BCPL programmiert ist.)

Zu diesem Zeitpunkt machten sich auch die ersten Gerüchte über zwei neue Supercomputer von Commodore und Atari breit. Im P.M. Computerheft vom Dezember '84 fand man folgende Meldung:

Kommt Commodore mit 32 Bit ?

Die Gerüchte über bahnbrechende Neuheiten aus dem Hause Commodore kochen derzeit auf großer Flamme. Für Zündstoff in der Branche sorgte vor allem Commodores Übernahme der kalifornischen Firma Amiga Corporated. In der Entwicklungsabteilung von Amiga steht bereits der Prototyp eines 32-Bit-Computers, in dem – wie im Apple Macintosh – als Zentralprozessor der Motorola 68000 sitzt. Die Grafikfähigkeiten sollen denen des Macintosh sogar überlegen sein.

Insider schließen bereits Wetten ab, daß das Gerät schon 1985 von Commodore vermarktet wird.

Unbestätigt wird Commodore-Präsident Marschall Smith zitiert, der einen dem Macintosh ähnlichen Computer ankündigte – für einen Preis von unter 1000 Dollar. Man kann gespannt sein, denn auch die Konkurrenz schläft nicht: Atari – nun unter (dem ehemaligen Commodore-Boss) Jack Tramiel – arbeitet angeblich fieberhaft an einem vergleichbaren Rechner. Tramiel hatte sich übrigens auch für den Kauf von Amiga interessiert.

Soweit der Bericht von '84. Zum Vergleich: Im gleichen Heft wurde ein Testbericht von dem neuen Zugpferd von Atari, dem Atari 800XL, abgedruckt.

Doch Moment, hier geht es auch um's Programmieren. Wie lief das damals mit der Programmierung des Amigas überhaupt ab? Da der Rechner, wie schon erwähnt, nicht einmal ein DOS hatte, mußte man, damals wie auch heute bei neuen Rechnern nötig, auf größere Rechner ausweichen, auf denen alles programmiert, kompiliert bzw. assembliert und gelinkt wurde, und dann auf den Amiga übertragen wurde. Auch gab es später noch keine Programmiersprachen, so daß trotz DOS mit größeren Rechnern gearbeitet werden mußte. So wurde z.B. damals für Entwickler ein Paket für Sun-Rechner ange-

boten, mit dem die Programmierung vorgenommen wurde. Die fertigen Programme wurden dann über die serielle oder parallele Schnittstelle übertragen. Dabei wurden Programme wie "binload" (Sun) und "download"/"upload" (Amiga) verwendet.

Doch zurück zur Hardware. Mitte '85 war der Amiga dann endlich fertig und bereit zur ersten öffentlichen Vorstellung. Und der Erfolg war umwerfend. Die gesamte Presse und Prominenz waren begeistert von dieser Maschine, die alles bisher Dagewesene in den Schatten stellte. Und auch heute noch ist die Gesamtleistung dieser Maschine von kaum einem anderen Rechner erreicht.

Von dieser ersten Präsentation bis zum Ladenverkauf verging aber immer noch ein gutes halbes Jahr, und entgegen den ersten Gerüchten (s. Bericht) kostete er leider nicht um 1000 Dollar, sondern (in Deutschland) etwa das Drei- bis Vierfache, nämlich 7000 DM. Dies ist natürlich für viele Anwender, die sich diesen "Traumcomputer" gern hätten kaufen wollen, zu hoch, und so trieb Commodore viele potentielle Kunden zu Atari und sorgte für den großen Anfangserfolg des Atari ST. Erfolg versprach erst die "Sonderpreis-Aktion", bei der der Amiga 1000 Mitte '86 für "nur" 4000 DM angeboten wurde. Sehr viele Leute nahmen diese "einmalige Chance" wahr, den Amiga noch zum hohen Preis zu kaufen, denn am Ende der Aktion fiel der Amiga wieder um einige hundert Mark.

Soweit zu dieser Hardware-Episode. Wie sah es zu der Zeit bei der Software aus? Zuerst einmal wurde natürlich AmigaBasic mitgeliefert (anfangs noch ohne Handbuch), was allerdings mit der Zeit nicht mehr als Programmiersprache angesehen wurde. Als Einsteiger hoffte man dann auf etwas Neues. Und so war ich dann sehr gespannt, als ein Freund sich den Lattice-Compiler anschaffte. Die anfängliche Euphorie schwand jedoch schnell, da es kaum möglich war, diesen Compiler mit einem Laufwerk und 512 KByte Ram zu betreiben.

Doch wieder Erwarten schaffte man es dann doch, und man hatte schließlich eine Diskette, auf der zum Programmieren noch etwa 70 KByte frei waren – Wahnsinn, was waren wir damals froh darüber.

Doch schon kurz danach kam die Alternative: der Aztec Compiler, Version 3.2 war endlich erhältlich, und ein Test bei einem Computerhändler offenbarte es: Manx war "diskettentauglich", sprich man konnte auch mit den beschränkten Mitteln meines Rechners damit etwas anfangen. So stand also der Weg offen zu neuen Welten. Übrigens kam etwa zur selben Zeit auch die erste Version der Kickstart 1.2 heraus, deren Sinn anfangs nicht einzusehen war.

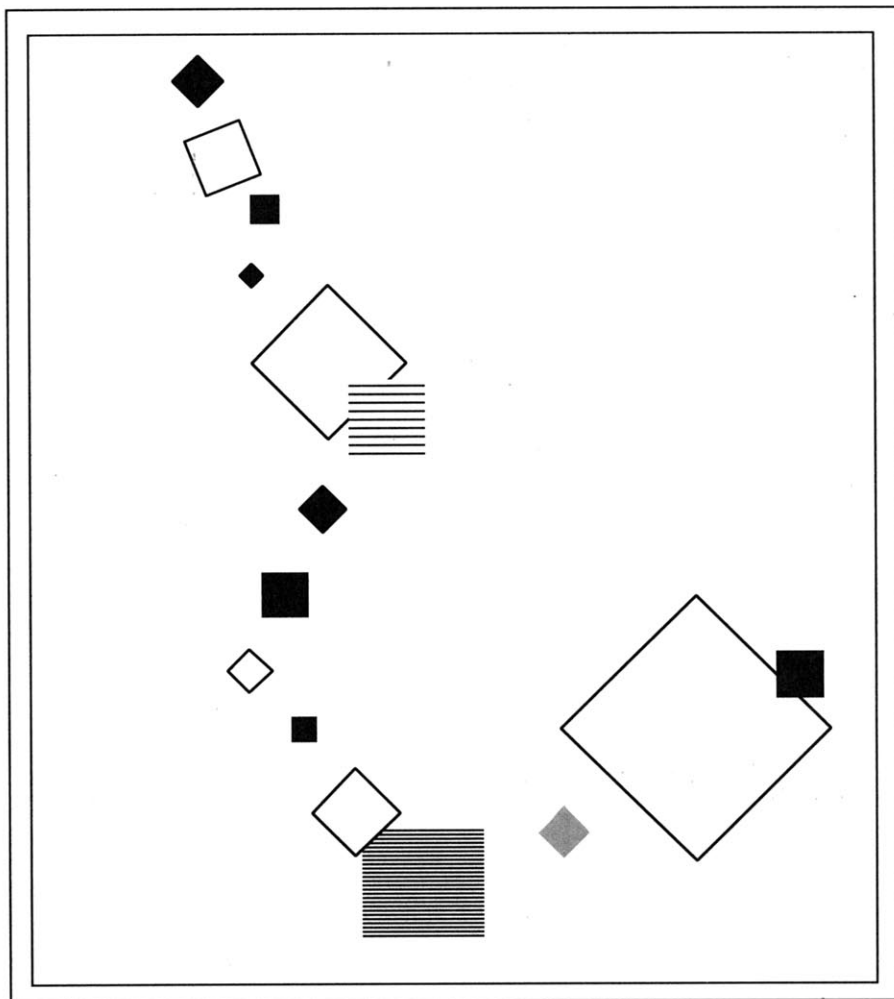
Im März des darauffolgenden Jahres kam dann Commodore auf der CeBIT mit dem 500er heraus, der 2000er war angekündigt. Einher damit ging der rapide Preisverfall. Langsam konnte sich der Amiga auch bei kleineren Geldbeuteln ausbreiten.

Von da an ging es Schlag auf Schlag. Immer neue Versionen von Compilern, neue Programmiersprachen (z.B. Modula II) und massenweise Unterhaltungssoftware stürmten auf den Markt, der Erfolg des Amigas war nicht mehr aufzuhalten.

Und inzwischen ist man ja auch nicht mehr so eingeschränkt... Speicher und externe Laufwerke haben den Weg zu meinem Rechner gefunden, eine Platte hat Verbindung aufgenommen, und zum wahren Glück fehlt eigentlich nur noch eine 68030-Karte.

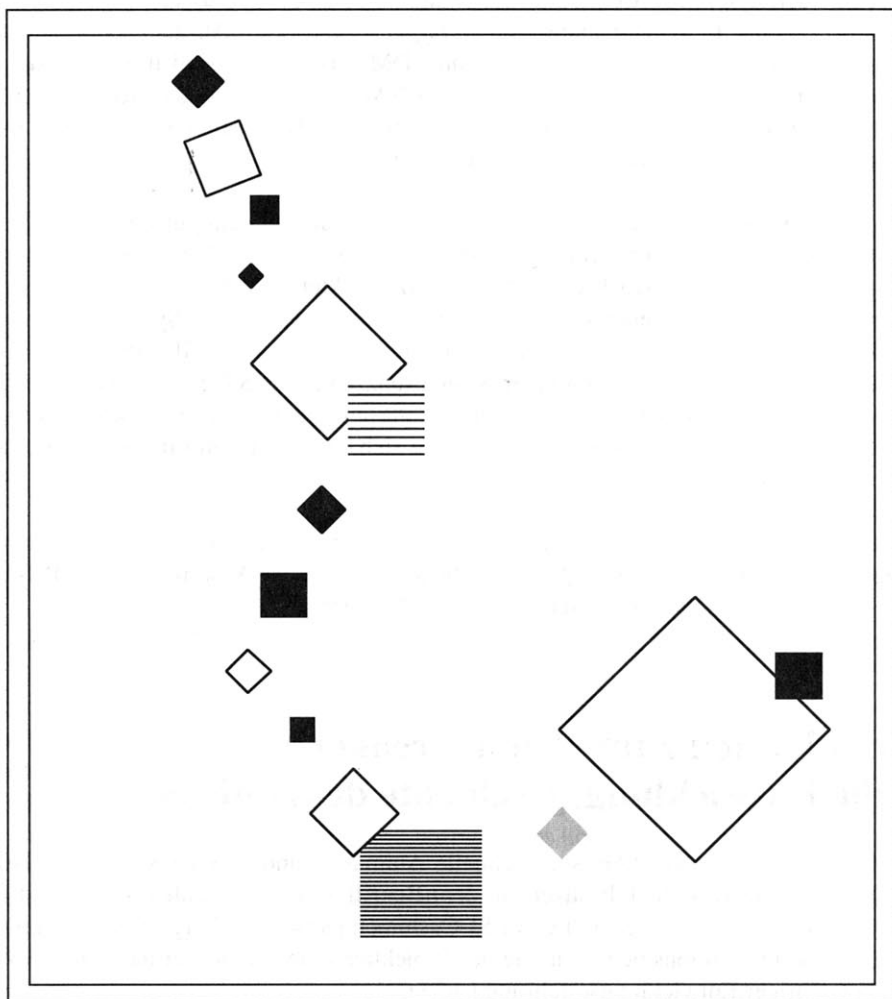
Teil 1

Die Hardware



Kapitel 1

Die Systemarchitektur



Um ein modernes Computerkonzept richtig einzuschätzen, reicht es nicht mehr aus, lediglich die Leistungsfähigkeit der verwendeten Mikroprozessoren zu vergleichen – derartige Untersuchungen bleiben zwangsläufig oberflächlich. Hochtrabende MIPS-Angaben (MIPS = Million Instructions Per Second, Maßeinheit für den Befehlsdurchsatz eines Prozessortyps) vernebeln den Blick auf grundlegende Aspekte der Systemarchitektur; gerade diese erweist sich aber im positiven Sinne als "Katalysator", im negativen jedoch als größter "Bremsklotz" der theoretischen Leistung einer CPU.

Globale Konzepte wie Architektur und Datendurchsatz der Busse, Realisierung des Speichersystems, I/O-Handhabung (Input/Output = E/A = Ein-/Ausgabe), Einsatz hochspezialisierter Coprozessoren und effizienter DMA-Techniken sind in der Praxis von entscheidender Bedeutung! Natürlich führt auch kein Weg an der genauen Analyse der Systemsoftware vorbei – schließlich kann sich die beste Hardware nicht entfalten, wenn sie durch ein träges Betriebssystem gesteuert wird.

Mit dem MC68000 entschieden sich die Entwickler des Amigas für eine schnelle 16-Bit-CPU mit hohem Programmierkomfort; solide Basis für den Aufbau eines leistungsfähigen Grafiksystems. Doch ein mit 7,1 MHz getakteter 68000er könnte den Amiga kaum aus der Masse neuer Computermodelle hervorheben; das System bliebe eine blaße Erscheinung am Rande des öffentlichen Interesses. Ein 7,1 MHz Prozessortakt ist selbst im eigenen Lager der 68000er-Konkurrenz (ATARI ST & Macintosh, beide 8 MHz) eher am unteren Ende der Frequenzskala angesiedelt; ganz zu schweigen von dem Heer der XT- bzw. AT-Kompatiblen, die sich gegenseitig mit immer neuen Taktrekorden übertrumpfen.

Trotzdem braucht sich der Amiga vor diesen Systemen nicht zu verstecken – ganz im Gegenteil –, seine besonderen Qualitäten liegen nicht in der Ausreizung des CPU-Taktes, sondern dem rundum durchdachten Systemkonzept.

Vom Exoten zum "Chartbreaker" – die Entwicklungsgeschichte des Amigas

Grafik in Farbfernsehqualität, superschnelle Animation und digitale Klangerlebnisse – Stichwörter, die auch Nichtbesitzer mit dem Begriff Amiga verbinden –, verdankt dieser Computer im wesentlichen drei VLSI-Customchips (= Very Large Scale Integration = sehr hohe Integrationsdichte), in die das Projektteam um Jay Miner das geballte Wissen langjähriger Entwicklungserfahrung packte.

In dieser Hinsicht profitierte der Computer davon, daß er ursprünglich als eine Art Super-Spielkonsole für den Einsatz anspruchsvoller Flugsimulatoren geplant war und sich erst später zu einem seriösen System mit leistungsfähigem Betriebssystem mauserte. Seit der Vorstellung im Sommer 1985, einer langen Phase wilder Gerüchte und der sehnsüchtig erwarteten Einführung der ersten Amiga-Modelle auf dem deutschen Markt, hat dieser Computer eine kontinuierliche Entwicklungsphase durchgemacht; anfangs noch als exotischer Ausflug Commodores belächelt, ist die Amiga-Palette mittlerweile zum wichtigsten Standbein der Firma geworden. Als stärkste Triebfeder für den Ausbau der Modellreihe erwies sich das Bestreben, die Produktionskosten zu reduzieren, um den Computer wirksam als Massenprodukt vermarkten zu können. Auf der anderen Seite zielte die Vorstellung des 2000er auf den professionellen Markt; auch hier wollte man das innovative Amiga-Konzept etablieren.

Im Zuge dieser Entwicklung wanderten viele diskrete Logikbausteine, PALs, Taktgeber und Treiber in die Customchips. Agnus, der sich ursprünglich mit einem 48-Pin-DIL-Gehäuse begnügte, platzte aus allen Nähten; bei den 84 Anschlüssen des Nachfolgers Fat Agnus mußte schon ein platzsparendes Chip-Carrier-Gehäuse her. Das Triumvirat Agnus, DENISE und PAULA bekam Unterstützung durch zwei neue, ebenfalls hochintegrierte Bausteine:

Gary ersetzte viele PAL-Chips (Programmable Array Logic = Programmierbare Gatterlogik) für die Adreßdecodierung und BUSTER übernahm die Steuerung der Expansion-Slots im B2000.

Die Hüllen fallen – grundlegende Aspekte der Systemarchitektur

Schnelle hochauflösende Grafik, komplexe Animationen und digitalisierter Sound mit Samplingfrequenzen von bis zu 28,6 KHz, stellen besonders hohe Anforderungen an die Leistungsfähigkeit der Bussysteme. Konventionelle Konzepte, bei denen ausschließlich die CPU den Adreß- und Datenbus steuert (also alle Datentransfers zwangsläufig über den Hauptprozessor laufen), wären damit hoffnungslos überfordert.

Derartige Architekturen stoßen schnell an die Grenzen des eingesetzten Prozessortyps.

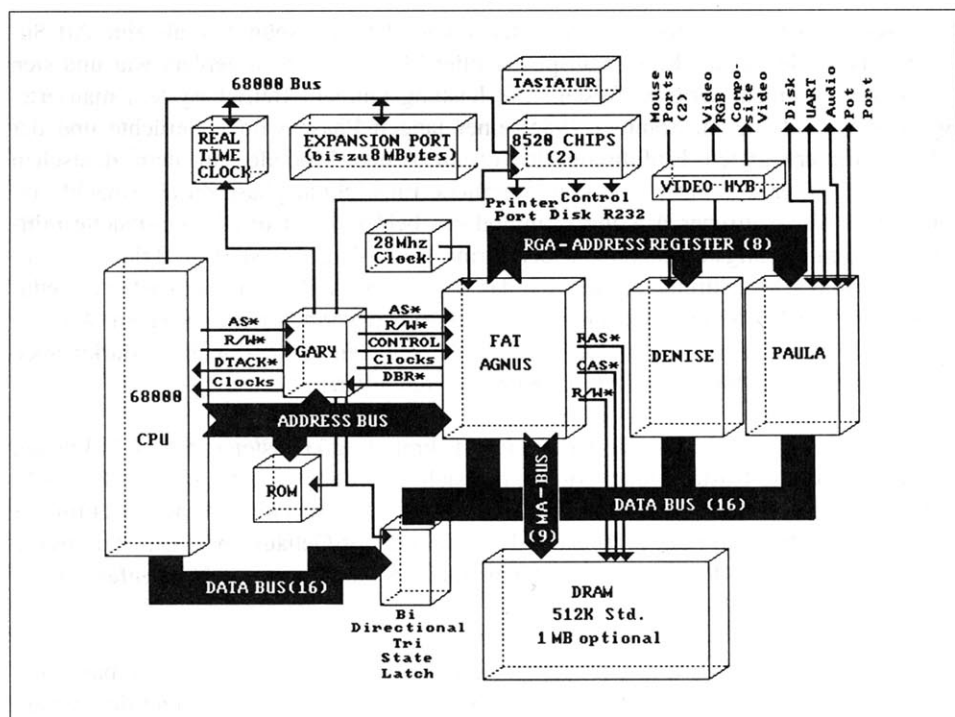


Abb. 1.1: Der Systemaufbau am Beispiel des A500

DMA – der Turbo für die Busse

Der Amiga verdankt deshalb seine überragende Leistungsfähigkeit in erster Linie einer ausgefeilten DMA-Technik (Direct Memory Access = Direkter Speicherzugriff), die es den Customchips erlaubt, unabhängig von der CPU auf die unteren 512 KByte des Systemspeichers zuzugreifen (das sogenannte CHIP-RAM ab Adresse \$0) – und zwar in einem Tempo, das selbst Hochleistungsprozessoren vom Schlage eines MC68020 auf die Plätze verweist. Die Customchiphardware arbeitet zwar nur mit der halben CPU-Taktfrequenz (also im PAL-System mit ungefähr 3,55 MHz), dafür ist der DMA-Controller aber in der Lage (dank eines optimierten Speicher-Interfaces) mit jedem Taktzyklus (= 280 ns) ein Datenwort aus dem CHIP-RAM-Speicher in die Zielregister der Spezial-Hardware zu transportieren.

Durch diesen 1-Takt-Buszyklus und die Umgehung der prozessortypischen Engpässe ist der Geschwindigkeitszuwachs eines DMA-Zugriffs gegenüber der entsprechenden CPU-Instruktion derartig fundamental, daß sich spontan der Vergleich mit einem Turbolader aufdrängt. Übertragungsraten von 3,55 MWorte/s bzw. 7,1 MByte/s sind Werte, mit denen sich nur noch ein 25-Mhz-68020er messen kann!

Fat Agnus – Schaltstelle des Systems

Nun ist DMA in der Computertechnik längst keine exotische Luxusausstattung mehr. Doch während viele Systeme den zentralen Prozessor nur beim Zugriff auf die Massenspeicher entlasten, macht der Amiga davon intensiveren Gebrauch – sämtliche Funktionen der Customchips, die auf eine hohe Übertragungsrate angewiesen sind, werden durch spezielle DMA-Kanäle unterstützt.

Der Customchip Fat Agnus (bzw. Agnus) spielt dabei eine Schlüsselrolle. Er enthält die komplette DMA-Logik für sechs verschiedene Quellen (Bitmap-, Sprite-, Copper-, Blitter-, Audio- und Disk-DMA), die über insgesamt 25 Kanäle kontinuierlich mit Datenworten versorgt werden.

Während eines DMA-Transfers generiert er die Speicheradresse der Daten im CHIP-RAM und legt gleichzeitig die Zieladresse eines Customchip-Registers auf den unidirektionalen Register-Adreßbus (kurz RGA, siehe Blockdiagramm 1.1), der ihn mit den Customchips DENISE und PAULA verbindet. Die internen Register-Adreß-Decoder interpretieren die 8-Bit-Binärinformation und aktivieren das zugehörige Register (mehr dazu im Kapitel 3, "Die Anatomie der Customchips"). Auch die Zugriffe des Hauptprozessors auf die Customchip-Register laufen über den Register-Adreßbus. Ein auf Fat Agnus integrierter Adreßpuffer schaltet den CPU-Adreßbus wahlweise zum RGA-Bus durch bzw. trennt ihn ab, je nachdem, ob dieser gerade frei ist oder nicht. Mit dem Memory-Address-Bus (= Speicher-Adreßbus) besitzt Fat Agnus bereits ein gemultiplextes Speicher-Interface für den direkten Anschluß dynamischer RAM-Chips vom Organisationstyp 256K x 1, so daß der externe Schaltungsaufwand extrem gering bleibt.

Elegant gelöst – die Realisierung des DMA-Systems

Da auch der 68000er das CHIP-RAM adressiert, kann es bei unkoordinierten Zugriffen zu Konflikten zwischen DMA und CPU kommen. Ein wesentlicher Teil der Steuerungsaufgaben entfällt deshalb auf die Vermeidung solcher Datenkollisionen – Fehler wären im wahrsten Sinne des Wortes "tödlich" und würden das System zerstören.

Die flexible Busschnittstelle des 68000ers unterstützt bereits von Hause aus einen Multiprozessor-Betrieb. Durch die vollständige Übergabe der Buskontrolle ist gewährleistet, daß CPU und intelligenter Peripheriechip (wie zum Beispiel ein DMA-Controller) niemals gleichzeitig dasselbe Bussystem benutzen. Obwohl sich die Nutzung dieser komfortablen Einrichtung geradezu anbietet, gingen die Entwickler des Amigas einen anderen Weg.

Zwei Überlegungen spielten dabei eine wichtige Rolle:

Jeder Verlust der Buskontrolle reißt die CPU jäh aus ihrem Programmfluß. Bei längeren Unterbrechungen muß deshalb der erhebliche Geschwindigkeitsvorteil eines DMA-Zugriffs durch kostbare Prozessorleistung erkaufte werden (mit gravierenden Folgen für die Reaktionsfähigkeit des Betriebssystems). Andererseits erweist sich das DMA-Konzept auf dem Amiga als außerordentlich komplex. 25 (teilweise sogar vollkommen unabhängige) DMA-Kanäle sind in einem System wechselnder Busherrschaften schwer zu handhaben. Sie provozieren eine "Wechselorgie" der Buszuständigkeiten und überhäufen die CPU mit Anforderungsprotokollen – alles droht unter ihrer Last zu ersticken. Die Amiga-Entwickler erkannten die Problematik und umgingen sie auf elegante Weise:

Die CHIP-RAM-Speicherzugriffe von Prozessor und DMA werden auf Basis einer Rasterzeile so harmonisch miteinander verzahnt, daß beide über weite Strecken parallel arbeiten können. Dabei machte man sich die Trägheit des 68000er-Speicher-Interfaces zunutze, das mit seinem 4-Takt-Buszyklus genügend Raum für die schnellen DMA-Zyklen läßt. In der Tat funktioniert die rasterstrahlsynchronisierte Verzahnung von DMA- und CPU-Zugriffen so gut, daß Audio-, Sprite-, Sound- und mit gewissen Einschränkungen auch die Bitplane-DMA ohne Behinderung des Prozessors ablaufen.

Nur bei Nutzung des Blitters und hochauflösender bzw. extrem vielfarbiger Grafikdarstellung kann es zu Konflikten kommen. In diesem Falle hat der 68000er das Nachsehen; ein von der Buszugriffslogik verzögertes DTACK*-Signal gaukelt einen langsameren Speicher vor und verlängert den Speicherzyklus so weit, bis der DMA-Transfer beendet ist. Während die Customchips Daten von und zum CHIP-RAM transportieren, trennt ein (von Gary gesteuerter) Tristate-Latch (Schlüsselspeicher) den Prozessordatenbus von diesem Speicherbereich ab.

Die enormen Fähigkeiten, die in der DMA-Architektur des Amiga-Systems schlummern, zeigen sich bei einer Speicheraufrüstung im Adreßbereich von \$200000 bis \$A00000. Dieses sogenannte FAST-RAM ist vollständig vom CHIP-RAM-Bussystem getrennt und unterliegt deshalb nicht dessen typischen Zugriffsengpässen. Egal wie viele DMA-Quellen mit dem Prozessor um das CHIP-RAM konkurrieren, das FAST-

RAM garantiert immer ungebremsen Zugriff. Der asynchrone Betrieb erlaubt echte Parallelverarbeitung und größten Datendurchsatz.

Leider sind die 0,5 MByte zusätzliches RAM ab \$C00000 sowohl beim A500 (interne Speichererweiterung), als auch beim B2000 (standardmäßig auf der Systemplatine) kein echtes FAST-RAM, sondern eine mißlungene Mischung aus beiden RAM-Arten. Da es von Fat Agnus synchron zum CHIP-RAM verwaltet wird, senkt jeder Speicherzyklus der Customchips die Zugriffsgeschwindigkeit, obwohl diese gar nichts mit dem Zusatz-RAM anfangen können – eine ärgerliche Situation, die sich nicht durch die einfache Architektur einer derartigen Speichererweiterung entschuldigen läßt.

Kommunikationsfreudig – Prozessor und Systembausteine

Alle programmierbaren Systemkomponenten sind "memory-mapped", daß heißt, ihre Register erscheinen als normale Speicherstellen in der Memory-Map (= Speicherbelegungsplan) des Systems. Eine kombinierte decodierte und lineare Adreßverwaltung reagiert auf hardwaremäßig festgelegte Adreßkombinationen und selektiert die zugewiesenen Bausteine (siehe Chipselect-Leitungen im Blockdiagramm 1.1 von Gary zu den Systemkomponenten); die eigentliche Registerauswahl läuft linear über drei, vier oder acht Leitungen des Prozessor-Adreßbusses.

In den neueren Amiga-Modellen ist der Customchip Gary für die Adreßdecodierung zuständig. Er ersetzt eine Reihe von PAL-Chips, die beim alten 1000er und den frühen 2000er-Versionen die Decodierlogik simulierten. Gary teilt den 16-MByte-Adreßraum des MC68000 in überschaubare Sektionen, ordnet den Systembausteinen feste Adreßpositionen zu und sorgt als Bus-Controller für eine konfliktfreie Abwicklung der Customchip-DMA. Wegen der Vielfältigkeit seiner Aufgaben ist dieser Customchip neben dem Anschluß an die Adreßleitungen A17 bis A23 (sie werden für die Adreßdecodierung benötigt) auch mit den Bussteuersignalen der CPU verbunden (weitere Informationen zu Gary im Kapitel 5).

Eine wichtige Komponente des E/A-Systems sind die 8520-CIA-Chips. In der Abbildung 1.1 werden sie zwar nur von einem kleinen, unscheinbaren Rechteck symbolisiert, besitzen aber eine zentrale Stellung im System. Als universell einsetzbare Schnittstellen-Bausteine sind sie für die Realisierung wichtiger Funktionen verantwortlich und regeln so unterschiedliche E/A-Prozesse wie die Steuerung der Schnittstellen, die Kommunikation zwischen Rechner und Tastatur oder die Abfrage der Game-Ports (nicht umsonst ist ihnen ein ganzes Kapitel gewidmet: Kapitel 4, "Die CIA-Hardware").

Die verschiedenen Modelle der Amiga-Familie – ein Überblick

Der Familienvater – A1000

Der A1000 hob sich bei seiner Vorstellung im Sommer 1985 wohltuend von den ausdruckslosen Kastendesigns anderer Computer ab. Bescheiden präsentierte er sich in einem dezent blaßgrauen Gehäuse, dessen kleine Ausmaße die wahre Leistungsfähigkeit nur erahnen ließen. Liebevolle Details, darunter die Signaturen des Entwicklungsteams auf der Innenseite der Gehäuseabdeckung oder die Aussparung unterhalb der Zentraleinheit (in der die Tastatur bei Bedarf einfach verschwinden kann) unterstreichen die Sorgfalt, mit der man sich um eine harmonische Verbindung "nackter Computer-Power" mit ansprechendem Design bemühte.

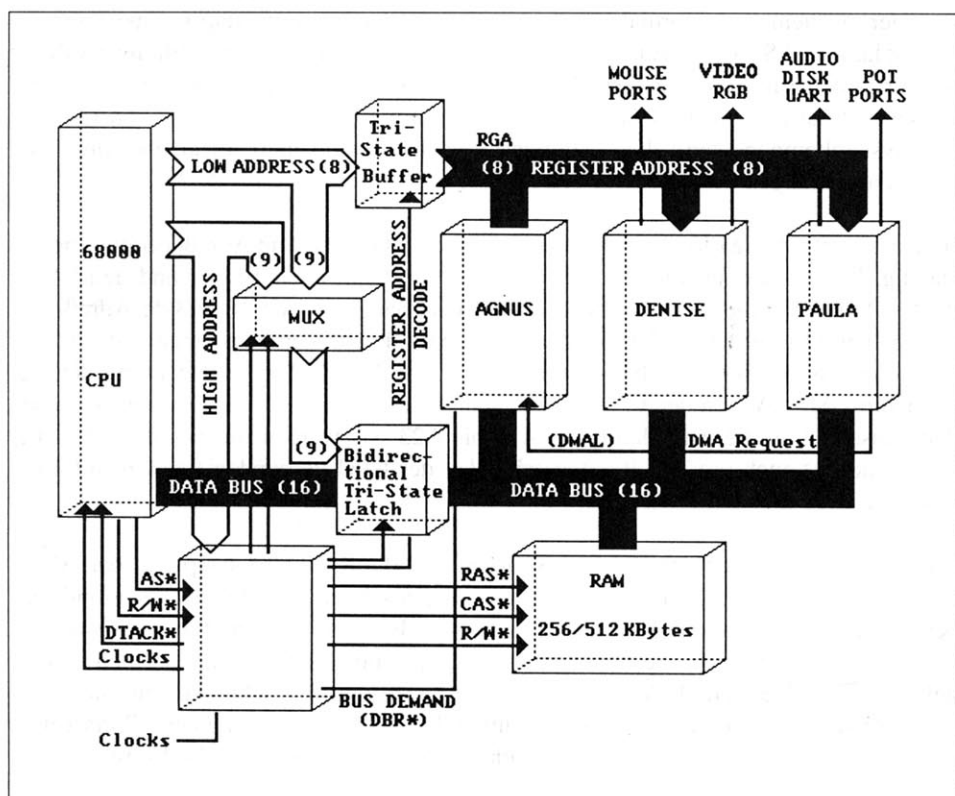


Abb. 1.2: Blockdiagramm A1000

Die Unterschiede im Blockdiagramm der alten 1000er-Modelle zu den Nachfolgern 500 und 2000 sind weniger funktioneller Art, als vielmehr Ergebnis der höheren Integrationsdichte der zweiten Customchip-Generation. Während sich die Chips im A500 (abgesehen von den DRAMs) an zehn Fingern abzählen lassen, enthält die Platine des A1000 noch viele Logik-Bausteine, die für Takterzeugung, Adreßdecodierung und Refreshsteuerung der dynamischen RAM-Chips zuständig sind.

Da die Testphase der Betriebssystemfunktionen zur Markteinführung noch nicht abgeschlossen war, reservierten die Entwickler einen RAM-Bereich von 256 KByte, der das Kickstart beim Hochfahren (Booten) des Systems aufnimmt. Dieser Speicherbereich bekam die Bezeichnung WOM (Write-Once-Memory = Einmal-Schreib-Speicher), weil nach dem Laden des Betriebssystems die Schreib-/Leseleitung automatisch deaktiviert wird, so daß das Kickstart vor Schreibzugriffen sicher geschützt ist. Weil das Layout der Systemplatine schon stand, aber nur noch Platz für die Boot-ROMs ließ, lieferte man die erste Serie des A1000 mit einer Huckepack-Platine aus, die neben dem WOM-Speicher vier weitere PAL-Bausteine für diverse Steuerfunktionen enthielt. Im Zuge der Weiterentwicklung des 1000er-Modells wurde die Huckepack-Platine überflüssig und das WOM verschwand auf der Systemplatine.

Obwohl die PAL-Version des Agnus-Chips bereits Mitte 1986 verfügbar war, versäumte man, die Generierung des FBAS-Videosignals auf PAL-Verhältnisse umzustellen. Zum Ärger der enttäuschten Anwender konnten aber PAL-Modulatoren und Videorekorder mit dem PAL/NTSC-Gemisch nichts anfangen (wenigstens erlaubt ein leichter Eingriff die Nutzung des AV-Ausgangs als monochromes BAS-Signal).

Als die vollständig eingedeutschte Version des A1000 im Winter '86 endlich fertig war, stellte man die Produktion ein. Neue, kostengünstigere Modelle warteten bereits in den Startlöchern – das ungewöhnliche Outfit und die besondere Ausstrahlung des A1000 wuch im harten Kampf um Marktanteile einer aggressiven Verkaufspolitik, die für gewisse Extravaganzen keinen Platz ließ. Verpackt in einem biedereren Alltagsdress sollte das Amiga-Konzept die Konkurrenz im Personal- und Heimcomputerbereich erschüttern.

Der Bestseller – A500

Äußerlich verbindet den A500, Anfang 1987 als Nachfolger des C64 vorgestellt, nichts mehr mit der 1000er-Modellreihe.

Die Entwicklung neuer Customchips (Fat Agnus, Gary) reduzierte die Produktionskosten, und so zielte der Neue sowohl vom Preis als auch vom Design her (Tastaturcomputer, externes Netzteil) auf den Heimcomputermarkt. (Gehört dazu auch die ergono-

mische Meisterleistung, den Ein/Aus-Schalter am Netzteil ("Bügeleisen") anzubringen? – na ja, ein bißchen Gymnastik kann nicht schaden.) Leider geriet das Netzteil etwas "schwachbrüstig", ein stärkeres Modell hätte sicherlich nicht das Produktionsbudget gesprengt.

Positiv dagegen ist die Normung der Schnittstellen:

Centronics und RS232C sind nun endlich zum IBM-Standard kompatibel; eine unnötige Eigenart des alten 1000ers erforderte ein spezielles (natürlich entsprechend teures) Verbindungskabel für den Anschluß handelsüblicher Drucker.

Obwohl die Serienbezeichnung 500 Leistungseinbußen gegenüber dem 1000er vermuten läßt, stellt der Neue keinen Abstieg dar. Im Gegenteil – die preisgünstige, interne Erweiterungsoption auf insgesamt 1 MByte RAM-Speicher macht den A500 als Komplettsystem recht attraktiv.

Flexibilität ist Trumpf – der Amiga 2000

Mit dem A2000 wollte man das Amigakonzept auch im Personalcomputerbereich salonfähig machen. Die Voraussetzungen dazu waren günstig:

Die innovative Systemarchitektur des Amigas brauchte den Vergleich mit konventionellen PCs nicht zu scheuen; in vielen Punkten war sie sogar AT-Kompatiblen weit überlegen. Dem alten 1000er-Modell fehlten eigentlich nur noch interne Erweiterungsslots, um Festplatten und Speichererweiterungen, kurz Steckkarten aller Art, bequem installieren zu können.

Daß der Erfolg einer neuen Serie zum großen Teil von dieser Erweiterungsfähigkeit abhängt, wußte man bei Commodore nur zu gut. Also spendierten die Entwickler dem 2000er ein ausgeklügeltes Slotsystem, das mit seiner Flexibilität und Anpassungsfähigkeit selbst höchsten Ansprüchen genügt:

- 1 MMU-Slots (ungepuffert) für weitere CPU- und Coprozessoren
- 5 Amiga-Slots (100-Pin, Zorro-kompatibel) für RAM-Erweiterungen, Interface-Karten, Festplatten-Controller usw.
- 2 PC-Slots (8-Bit, erweiterbar auf 16 Bit)
- 2 AT-Slots (16-Bit, überlappen sich mit den Amiga-Slots) für die sogenannten Bridgeboards

- 2 Videoslots für Grafikkarten, Genlock, Digitizer etc.

Der aufgestockte Speicher (1 MByte, Sockel für weitere 0,5 MByte bereits vorhanden) und die großzügig ausgelegte Tastatur (96 Tasten nach DIN, 10 Funktionstasten, separate Cursor- und numerische Tasten) rundeten den positiven Gesamteindruck ab.

Die offene Systemarchitektur OSA (= Open System Architecture) macht den 2000er zu einem Allroundtalent, das (mit entsprechenden Karten) jeweils optimal auf die verschiedensten Einsatzgebiete angepaßt werden kann. Eine Spezialität dieses Computers:

Die Verbindung zweier Welten über die PC/AT-Slots und das PC-Bridgeboard bzw. die brandneue AT-Karte. Zwar gibt es schon seit längerem Systeme mit unterschiedlichen Mikroprozessoren (eine beliebte Kombination: i8086/88 und Z80), doch genauso gut hätte man einen ZX81 neben den IBM-PC stellen können, um sich jedesmal zu entscheiden, auf welchem Computer man nun arbeiten möchte.

Geradezu revolutionär dagegen die Wandlungsfähigkeit des Amigas, der den PC- bzw. AT-Teil als physikalisch existenten Task (= Prozeß) in die Multitasking-Umgebung der Systemsoftware einbettet. Diese intensive Einbindung eines Fremdprozessors, die echte Parallelverarbeitung erlaubt, war bisher Domäne extrem teurer Workstations.

Hier setzt der Amiga neue Maßstäbe!

Den 2000er gibt es mittlerweile in zwei Ausführungen (ungezählt die vielen kleinen Änderungen an der Systemplatine oder Updates der ROM-Software), einer A- und einer B-Version. Der A-Typ war von der deutschen Commodore-Tochter in Braunschweig parallel zu dem amerikanischen A500-Projekt entwickelt worden und basierte noch auf dem Layout des alten 1000er-Modells.

Mit der Verfügbarkeit der neuen Customchips, wurde der A2000 komplett überholt und auf die kostengünstigere A500-Architektur umgestellt; ergänzt durch einen weiteren Customchip (Spitzname: BUSTER), der für die Steuerung des komplexen Slotsystems konzipiert wurde.

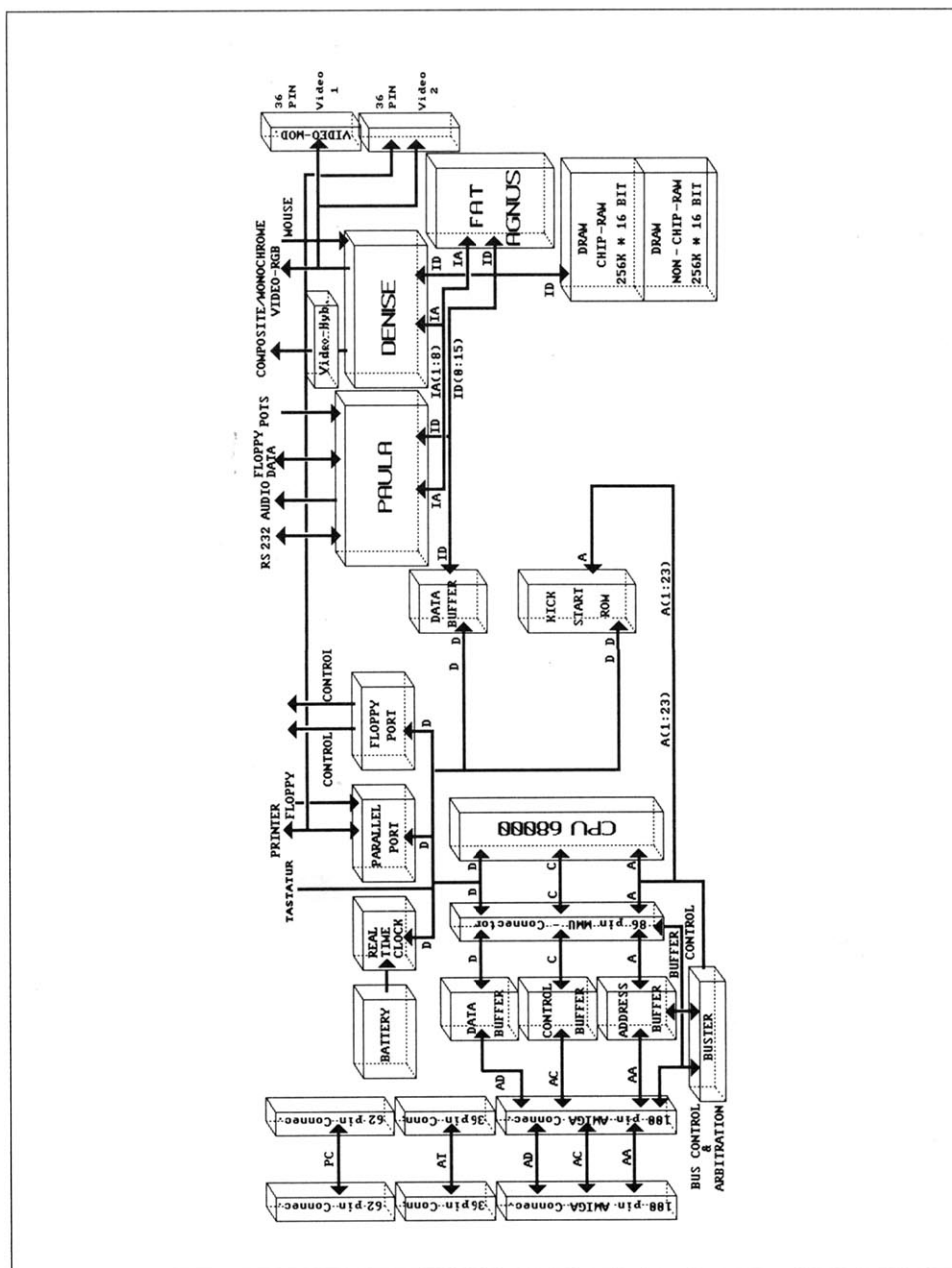
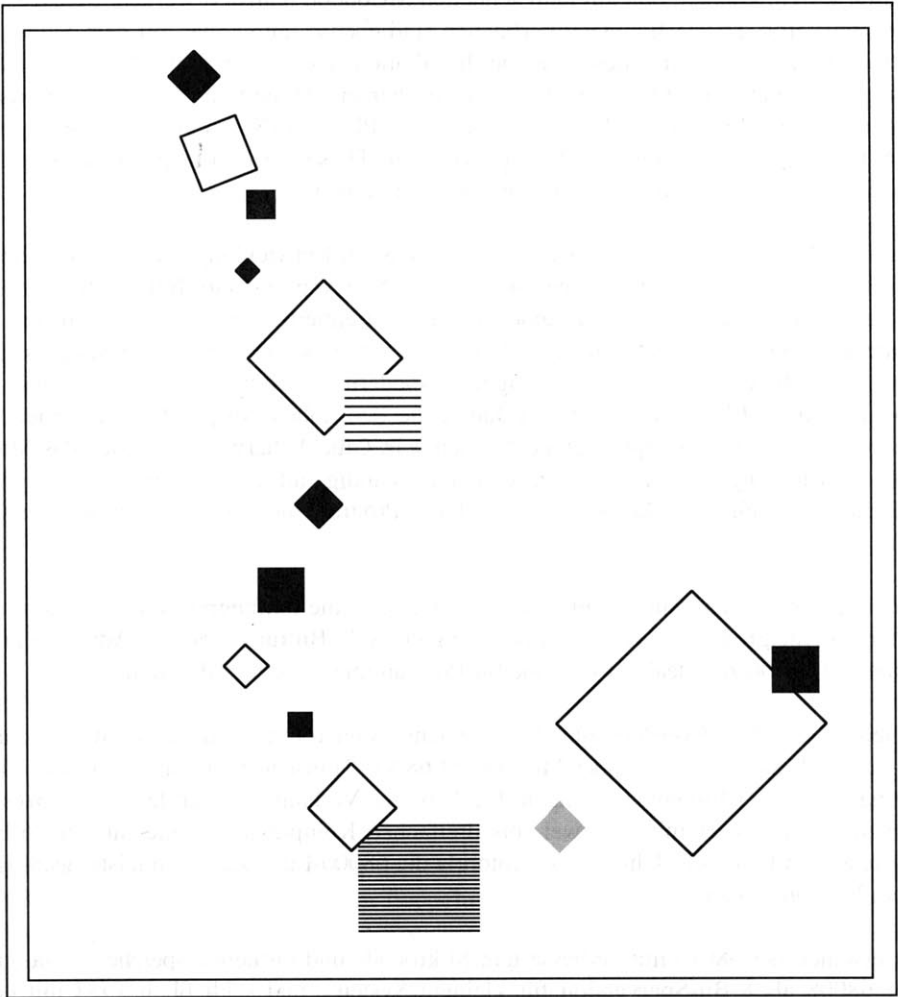


Abb. 1.4: Blockdiagramm B2000

Kapitel 2

Die CPU



Obwohl der MC68000 schon 1979 als Prototyp verfügbar war, ist er noch heute einer der leistungsfähigsten, universell einsetzbaren 16-Bit-Mikroprozessoren. Der Weg in den PC- und Heimcomputerbereich war beschwerlich; jahrelang blieb diese CPU Domäne teurer Minis und Workstations. Erst mit dem Macintosh hielt der 68000 Einzug in die PC-Welt und löste schließlich mit dem ATARI ST und dem Amiga einen Boom im Heimbereich aus.

Viele Anwendungen, die vorher auf den alten 8-Bit-CPU-Generationen undenkbar gewesen wären, konnten nun auf dem schnellen MC68000 realisiert werden. Mausgesteuerte, fensterorientierte Benutzeroberflächen sind ebenso untrennbar mit dem Siegeszug dieses Prozessors verbunden wie die Installation von UNIX auf Mikrocomputern. Lange Zeit hatte der 68000 im PC-Bereich eher ein Mauerblümchendasein gefristet, während seine direkten Konkurrenten, die Intel-CPU's 8086/88, mit dem Betriebssystem MS-DOS den professionellen Markt eroberten. Diese Entwicklung hängt sicherlich auch mit seinem ungewöhnlichen Konzept zusammen.

In einer Zeit, in der andere Hersteller bewährte 8-Bit-Entwicklungen einfach mit einem 16-Bit-Datenbus aufbohrten, aber nur wenig Neues in puncto Befehlsstruktur und Adressierungsmechanismen zustande brachten, orientierte sich der 68000 an der Architektur von Großcomputern und Minis. Der entscheidende Entwicklungsimpuls ging diesmal nicht von den Hardwaredesignern, sondern den Softwarespezialisten aus. Sie suchten eine CPU für die achtziger Jahre, die den hohen Ansprüchen moderner Betriebssysteme und Hochsprachen gewachsen war. Glücklicherweise ließ sich das Motorola-Entwicklungsteam nicht dazu verleiten, ständig auf das (technisch) momentan Machbare zu schießen. Das Konzept stand, der Programmierkomfort blieb oberstes Gebot.

Ein überschaubarer, sehr symmetrischer Befehlsatz, die umfangreichen Adressierungsarten und die großzügige Registerausstattung (16 x 32 Bit für Daten und Adressen), machen den 68000 zur idealen Maschine für Programmierer (siehe Abbildung 2.1).

Allen notorischen Zweiflern zum Trotz, gelang es auch mit der neuen HMOS-Technologie (High Density N Channel MOS) rund 68000 Transistorfunktionen auf einem fingernagelgroßen Chip unterzubringen; für damalige Verhältnisse war der neue Prozessor ein Siliziummonster mit der zwei- bis dreifachen Komplexität vergleichbarer 16-Bit-Systeme. Im Laufe der Jahre baute Motorola die 68000-Linie zu einem leistungsfähigen Familienkonzept aus:

Es erschien der 68010 mit verbessertem Mikrocode und virtueller Speicherverwaltung, der 68008 als 8-Bit-Sparversion für kleinere Systeme und schließlich 1984 mit dem MC68020 der erste echte 32-Bit-Prozessor aus dem Hause Motorola.

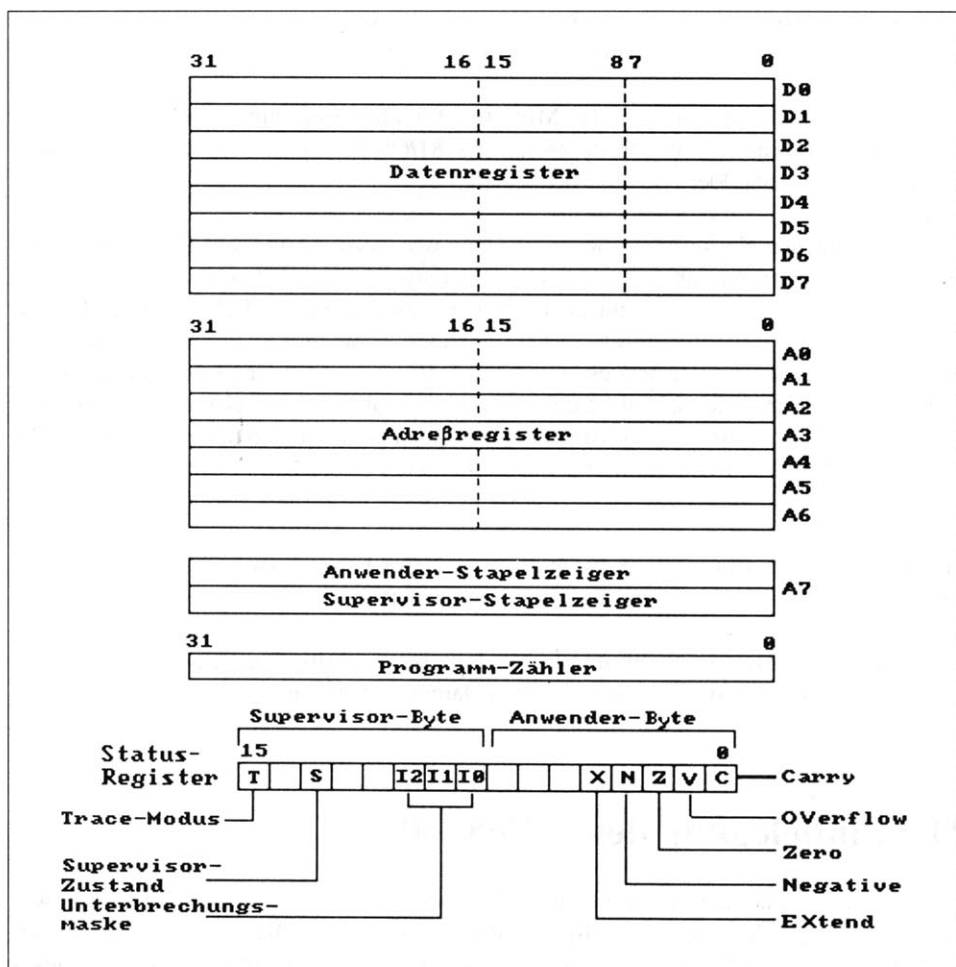


Abb. 2.1: Registerarchitektur des 68000

Heute ist der 68030 das Topmodell der 68000er-Familie, ein auf Höchstleistung getrimmter 32-Bit-Prozessor mit intern doppeltausgeführten Bussystemen (Harvard-Architektur), jeweils 256 Byte Cache-Speicher für Daten und Befehle sowie einem chipinternen Memory-Management-Unit (kurz: MMU). Parallel zu den rasanten Fortschritten auf dem Gebiet der Integrationsdichte, erhöhte man kontinuierlich die verfügbaren Taktfrequenzen; so gibt es den 68030 mittlerweile in einer 33Mhz-Version!

Daß sich Motorolas Prozessorfamilie durchsetzen konnte, lag wohl nicht zuletzt auch an der hervorragenden Unterstützung durch Peripheriechips:

Vom Multifunktionsbaustein (z.B. MFP 68901) über Interruptcontroller bis hin zu MMUs und arithmetischen Coprozessoren (68881/68882) ist alles für den Aufbau eines leistungsfähigen 68000er-Systems vorhanden.

Das nun folgende Kapitel dient in erster Linie der Auffrischung von Basiswissen. Tabellen, Flußdiagramme und schematische Darstellungen sollen dem Programmierer geballte Informationen schnell und übersichtlich präsentieren. Im Rahmen einer Funktionsbeschreibung des Pin-Layouts werden wichtige Themen zu diesem Prozessor erörtert und so weit analysiert, wie es zum Verständnis seiner Systemeinbindung erforderlich ist. Besonderes Gewicht liegt dabei auf einer realistischen Leistungseinschätzung. Lösen wir uns vom einseitigen MIPS-Denken, auch wenn die Computerwerbung gerne Durchsatzraten, Taktfrequenzen und Busbreiten wie PS und Nm in der Automobilbranche anpreist.

Im Anhang dieses Buches findet sich ein besonderer Leckerbissen für alle, die nie mit der Geschwindigkeit ihres Maschinencodes zufrieden sind:

Die Tabelle der Befehlsausführungszeiten hilft bei zeitkritischen Anwendungen, dem Prozessor mit optimierter Syntax ordentlich Dampf zu machen.

Die Pinbelegung des MC68000

Auf der Amiga-Platine braucht man nicht lange nach der CPU zu suchen. Das erste, was einem ins Auge fällt, ist das bullige 64-Pin-DIL-Gehäuse (DIL = Dual In Line) des 68000. Trotz hohen Platzbedarfes ist es noch immer die Standard-Gehäuseform dieses Prozessors. (Wenn's zu eng wird, gibt es allerdings auch den platzsparenden Chip-Carrier.)

Wozu ein so großes Gehäuse, wenn andere 16-Bit-Produkte (i8088, i8086, Z8000) mit nur 40 Pins auskommen?

Die Erklärung liegt in der kompromißlosen Auslegung der Bussysteme. Beim 68000 sind Daten- und Adreßbus einzeln herausgeführt, benötigen zusammen also bereits 41 Leitungen ($23 + 16 + \text{UDS}^* + \text{LDS}^*$); weitere 23 Signale gehören zur Minimalausstattung für die wichtigsten Systemfunktionen. Ein gemultiplextes Bussystem schafft zwar kleinere (und dadurch entsprechend leichter zu handhabende) Gehäuse, das Umschalten und Sortieren der Signale kostet aber zusätzliche Zeit und verlängert den Buszyklus.

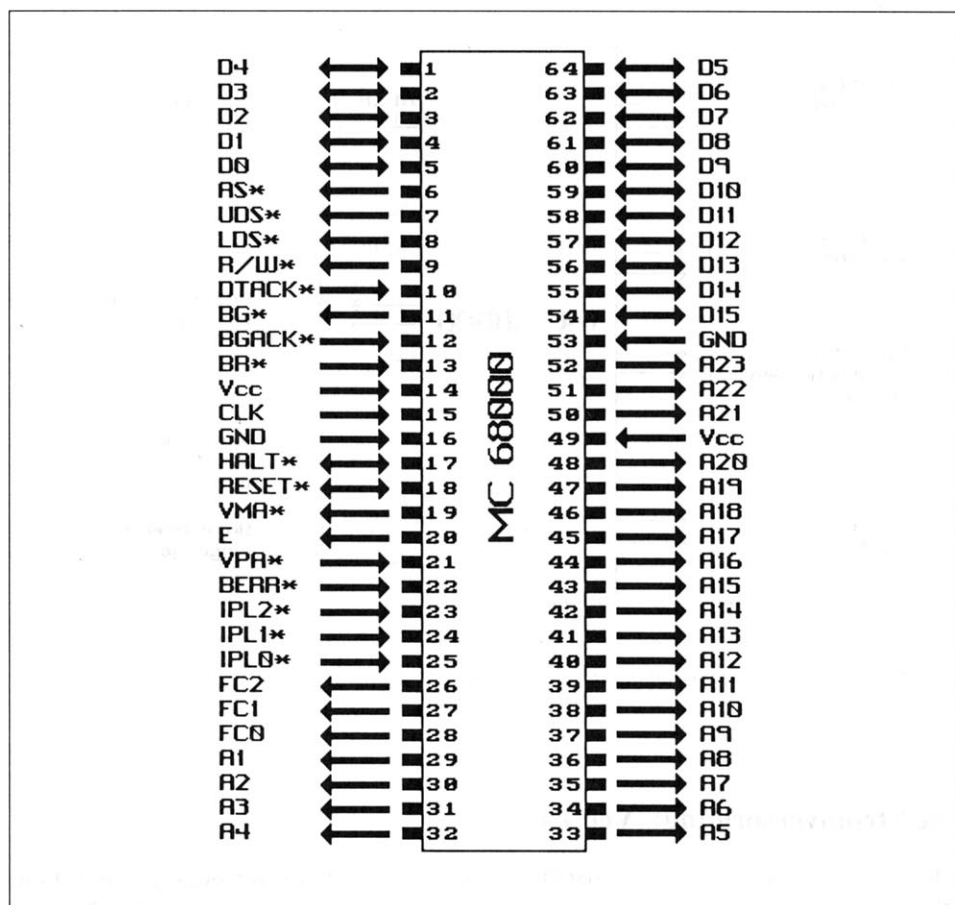


Abb. 2.2: Pinbelegung des 68000

Obwohl die Gehäusegröße während der Markteinführungsphase des 68000 ein ausschlaggebendes Kriterium war, entschied man sich bei Motorola bewußt gegen das verbreitete Multiplex-Verfahren.

Geschwindigkeit und Leistungsfähigkeit vor Kostendenken – eine Philosophie, die sich durch die gesamte Entwicklungsgeschichte dieses Prozessors zieht.

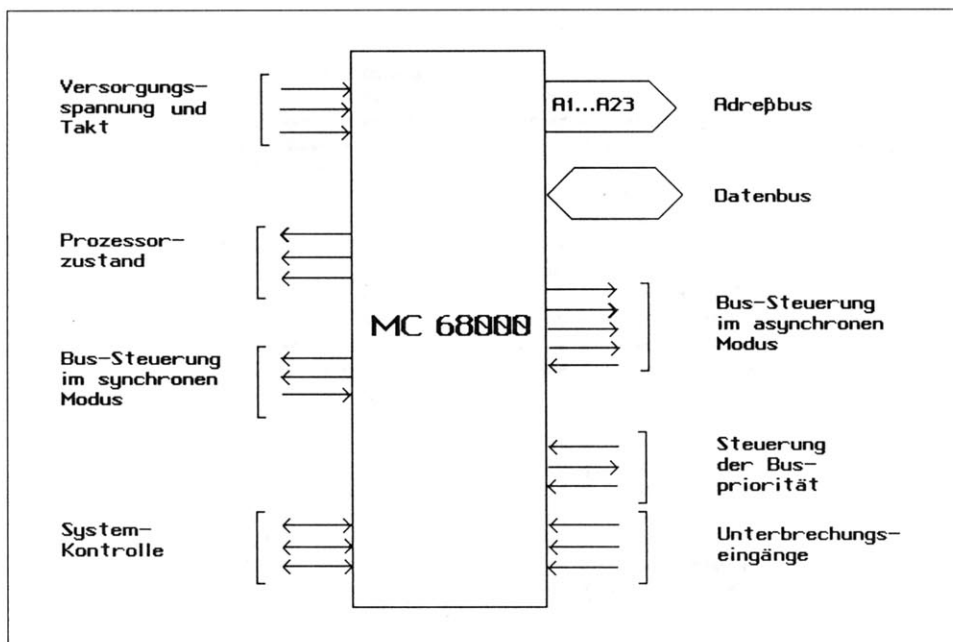


Abb. 2.3: Strukturierte Darstellung des Pin-Layouts

Die Stromversorgung: Vcc + GND

Als moderne CPU benötigt der 68000 nur eine einzige Stromversorgung von 5 Volt. Die doppelte Ausführung von Vcc und GND erleichtert eine Optimierung des Platinen-Layouts. Beide Leitungen sind so zentral platziert, daß der Stromverlust durch Leitungswege im Gehäuse minimal bleibt.

Der Takteingang: CLK

Das CPU-Timing basiert auf einem einphasigen, TTL-kompatiblen Taktsignal (TTL = Transistor Transistor Logic).

Der Datenbus: D0...D15

Die Datenleitungen D0 bis D15 bilden ein 16 Bit breites, bidirektionales Tristate-Bus-system (Tristate = 3 Zustände).

Damit gehört der 68000 eindeutig in die Kategorie der 16-Bit-Prozessoren, obwohl seine Registerarchitektur bereits in Richtung 32 Bit zielt.

Das flexible Businterface unterstützt eine Reihe von Datenformaten:

- BCD-Werte (4 Bit)
- Bytes (8 Bit)
- Words (Worte, gemäß Motorola-Definition 16 Bit)
- Longwords (Längsworte, 32 Bit)

Zusätzlich existieren Befehle für die gezielte Manipulation einzelner Bits (Bitverarbeitungsbefehle auf Byte-Speicherstellen oder 32-Bit-Register). Die 16 Datenleitungen übertragen Bytes und Words in nur einem Buszyklus; Langworte werden in zwei Hälften à 16 Bit aufgespalten und benötigen dementsprechend auch zwei Buszyklen.

Der maximale Datendurchsatz eines Prozessortyps leitet sich aus dem Quotienten von Taktfrequenz und Buszykluszeit ab; die Breite des Datenbusses bestimmt, wieviel Bytes in einem Zugriff transportiert werden können. Legt man eine Buszykluszeit von vier Taktzyklen zugrunde (so lange dauert die vollständige Abwicklung eines Speicherzugriffs), erreicht die Standard-8Mhz-Version des 68000 eine maximale Durchsatzrate von 4 MByte/s ($8 \text{ MHz} / 4 \text{ Tz} * 2 \text{ Byte pro Zugriff}$).

Bei der 7,1-MHz-CPU im Amiga-System sind das immerhin noch ca. 3,5 MByte/s ($7,1 \text{ MHz} / 4 \text{ Tz} * 2 \text{ Byte}$).

Der Adreßbus: A1...A23

Der unidirektionale Adreßbus (Tristate) umfaßt 23 Adreßleitungen, mit denen der Prozessor 8 MWorte ($= 2^{23}$) bzw. 16 MByte Speicherraum ansprechen kann. Die Leitung A0 ist nicht herausgeführt, sondern wird durch die Strobes UDS* (Upper-Data-Strobe = Signal für höherwertige Daten) und LDS* (Lower-Data-Strobe = Signal für niederwertige Daten) ersetzt, die den Zugriff auf die obere (D15 – D8) bzw. untere Hälfte (D7 – D0) des 16-Bit-Datenbusses markieren. Je nach Datenformat der adressierten Speicherstelle (Byte, Wort, Langwort) und Zustand des A0-Bits im Programmzähler (PC = Program-Counter), aktiviert der Prozessor die Datenstrobes UDS* (A0 = 0) bzw. LDS* (A0 = 1).

Die Bausteine der Speichersteuerung interpretieren die Zustände der Daten-Strobes und legen die Informationen der byteweise organisierten Speicherbänke auf den entsprechenden Teil des Datenbusses. Durch die Aufspaltung des Bussystems in ein höher-

bzw. niederwertiges Datenbyte, sind die Adressierungsmechanismen nicht auf das 16-Bit-Wort als kleinste Informationseinheit festgelegt, sondern können gezielt auf einzelne Bytes im Speicher zugreifen.

Beginnend mit der Speicherstelle 0 (die definitionsgemäß gerade sei) entsteht eine lange Kette aufeinanderfolgender Bytes mit abwechselnd gerader bzw. ungerader Adresse.

Wortzugriffe sind beim 68000 an gerade Adressen gebunden. Um 16-Bit-Werte auch auf ungeraden Speicherstellen anzusprechen, müßte der Prozessor den Zugriff automatisch in zwei Adreßzyklen aufspalten; eine Technik die beispielsweise vom Intel 80286 unterstützt wird, für den 68000 aber nicht vorgesehen ist. Deshalb dürfen Befehle (die ja wortweise codiert sind) und Langwörter (als aus 16-Bit-Wörtern zusammengesetztes Datenformat) auch nur auf geraden Adressen liegen. Diese Einschränkung der Adressierungsmechanismen spiegelt sich in einer charakteristischen Speicherorganisation wider:

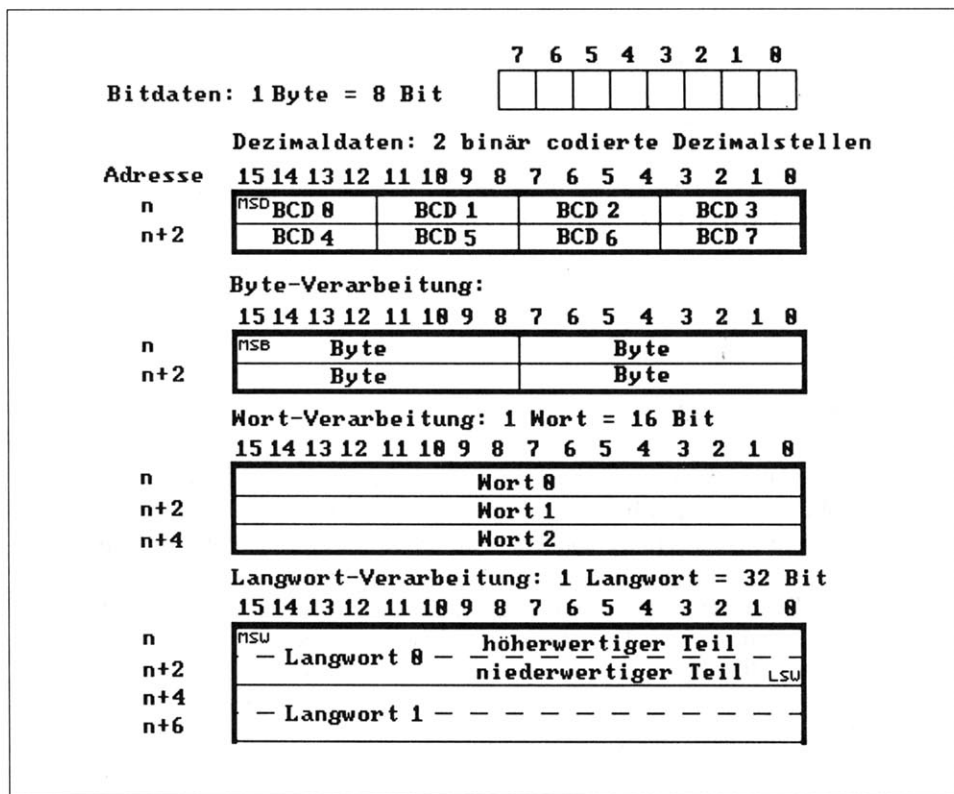


Abb. 2.4: Die Speicherbelegung der diversen Datenformate

Ein großer Vorteil des 68000-Systems ist die lineare Aufteilung des Adreßraums; jede logische Speicherstelle kann direkt über den Programmzähler erreicht werden. Andere 16-Bit-Produkte (Intel, Zilog) greifen für die Verwaltung eines viel kleineren Speichervolumens auf eine segmentierte Adressierung zurück. Dabei wird der Speicherraum in eine Reihe gleich großer Segmente (Seiten) unterteilt und ihre Lage in einer Offsettabelle vermerkt. (Der 8088/8086 zerhackt den vergleichsweise bescheidenen Maximal-Speicher von 1 MByte in zwölf 64-KByte-Segmente ... das ruft Erinnerungen an die Adressierungskapazität gängiger 8-Bit-Prozessoren wach.) Bei jeder Überschreitung der Segmentgrenzen (und das kommt häufiger vor) müssen die Tabelleneinträge zum Programmzähler hinzuaddiert werden; ein Vorgang, der nicht nur umständlicher ist, sondern auch noch wesentlich mehr Zeit kostet als eine lineare Speicheradressierung.

Die Bussteuerung

Die Busschnittstelle des 68000 unterstützt zwei grundsätzlich verschiedene Arten der Kommunikation mit Peripheriebausteinen, den synchronen und den asynchronen Betrieb.

Der asynchrone Busmodus stellt sich dynamisch auf die Geschwindigkeit der angesprochenen Peripheriechips ein und garantiert damit eine reibungslose Kommunikation zwischen CPU und Systembausteinen. So legt der Prozessor im Schreibzugriff nicht einfach "blind" Daten auf die Busse, sondern wartet bis die adressierten Bausteine bereit sind, diese auch zu übernehmen. Die Strobesignale AS* (Address-Strobe = Adreßsignal) und DTACK* (Data Transfer Acknowledgement = Bestätigung der Datenübertragung) sorgen für das richtige Bustiming.

Sobald der Adreßbus eine gültige Adresse führt, geht AS* auf Low und markiert damit den Beginn eines Buszugriffs. Die eigentliche Datenübertragung wird so lange zurückgehalten, bis der Prozessor das Quittungssignal DTACK* empfängt. Diese Technik läßt den Systembausteinen genügend Zeit, die Datenübernahme vorzubereiten bzw. gültige Daten bereitzustellen. Reagieren die adressierten Bausteine schnell genug, um DTACK* rechtzeitig zu bestätigen, ist der Buszyklus in der kürzest möglichen Zeit von vier Taktzyklen abgeschlossen. Kommt DTACK* dagegen nicht innerhalb einer gewissen Toleranzspanne nach Aktivierung der AS*-Leitung, dehnt die CPU den Buszugriff durch Wartezyklen und setzt ihn erst nach der Quittierung ordnungsgemäß fort.

Theoretisch läßt sich ein Buszyklus also beliebig in die Länge ziehen. Im Extremfall hängt das System bei der Adressierung physikalisch nicht existenter Speicherbereiche in einem Endloszyklus fest. Wirksamen Schutz bietet ein sogenannter Watchdogtimer (= Wachhund), der die Zeit zwischen dem AS*-Impuls und der Bestätigung des

DTACK*-Signals mißt; trifft DTACK* nicht innerhalb der tolerierten Verzögerungszeit ein, fängt der Baustein den aktuellen Buszyklus in einer Busfehlerausnahme ab.

Steuersignale im asynchronen Modus: AS*, R/W*, UDS*, LDS*, DTACK*

- AS*: (Address Strobe = Adressierungs-Signal) kennzeichnet eine gültige Adresse auf dem Adreßbus.
- R/W*: (Read/Write = Lesen/Schreiben) steuert die Zugriffsrichtung (Low = Schreiben, High = Lesen).
- UDS*, LDS*: (siehe Funktionserläuterung der Adreßleitungen) ersetzen die nicht ausgeführte Leitung A0 und markieren den Zugriff auf die obere bzw. untere Hälfte des Datenbusses.
- DTACK*: (Data Transfer Acknowledgement = Bestätigung des Datentransfers) quittiert die Kommunikationsbereitschaft eines adressierten Systembausteins. Während einer Interruptverarbeitung signalisiert die Leitung eine autovektorielle Unterbrechungsanforderung.

Obwohl die asynchrone Busarchitektur durch Einbindung von Systemkomponenten mit unterschiedlichsten Reaktionszeiten bereits ein großes Maß an Flexibilität bietet, implementierten die Entwickler noch einen weiteren Übertragungsmodus.

Der synchrone Busbetrieb ist ein Zugeständnis an die Verwendbarkeit preiswerter 8-Bit-Peripheriechips, die mit den Dialogmechanismen (AS*, DTACK*) der asynchronen Busschnittstelle nichts anfangen können. Diese Bausteine sind auf eine exakte Synchronisation von Sender und Empfänger angewiesen, da die Datenübertragung an feste Positionen innerhalb des Buszyklus gebunden ist. Außerdem sind 6800- bzw. 6502-Peripheriechips typischerweise auf erheblich niedrigere Taktfrequenzen ausgelegt als der 68000-Prozessor. Während letzterer mit 8 MHz davonschnellt, ist auf dem 8-Bit-Sektor häufig schon bei 1 MHz Schluß.

Trotz dieser Hürden gelang es mit relativ geringem Aufwand, beide Welten zu verbinden. Der 68000 stellt auf dem E-Ausgang ein durch zehn geteiltes Taktsignal für synchrone Peripheriechips zur Verfügung. Als Schnittstelle zwischen Prozessor und Systembausteinen reagiert die Adreßdecodierung auf das Erkennen einer gültigen Geräteadresse mit VPA* (Valid Peripheral Address = Gültige Peripheriegerätadresse), woraufhin die CPU den gesamten Busbetrieb auf den E-Takt synchronisiert. Sobald dieser Vorgang abgeschlossen ist, quittiert der Prozessor mit VMA* (Valid Memory Address = Gültige Speicheradresse; Funktion entspricht AS* im asynchronen Busbetrieb) und

signalisiert damit den Beginn des synchronen Buszugriffs. So wird aus dem rasanten Sprinter ein gemütlicher Spaziergänger im 8-Bit-Gleichschritt.

Da der 68000 (wie die meisten Neuentwicklungen) erst einmal "nackt" auf den Markt kam, erwies sich der Brückenschlag zu bewährten Bausteinen aus der 8-Bit-Generation als ausgezeichnete Starthilfe.

Steuersignale im synchronen Modus: E*, VPA*, VMA*

E*: (Enable = Freigabe) Taktsignal für 8-Bit-Peripherie. Entspricht einem Zehntel der Prozessortaktfrequenz.

VPA*: (Valid Peripheral Address = gültige Peripheriegerätadresse) wird von der Adreßdecodierung erzeugt und initiiert einen Datentransfer im synchronen Busbetrieb. Nach Empfang dieses Signals synchronisiert der Prozessor seine Busschnittstelle auf den E-Takt, so daß die Peripheriechips problemlos mit der CPU kommunizieren können.

VMA*: (Valid Memory Address = Gültige Speicheradresse) Reaktion auf VMA*. Die CPU bestätigt den Abschluß des Synchronisationsprozesses und signalisiert gleichzeitig das Anliegen einer gültigen Adreßinformation.

Die Buszugriffssteuerung: BR*, BG*, BGACK*

Die Busschnittstelle des MC68000 ist durch die komfortable Unterstützung eines Multiprozessorbetriebs auch für komplexe Bussysteme geeignet. Über die Signale BR* (Bus Request = Busanforderung) und BGACK* (Bus Grant Acknowledgement = Bestätigung der Busfreigabe) kann ein intelligenter Baustein dem Prozessor die Buskontrolle entziehen und selbst übernehmen.

Normalerweise ist der Hauptprozessor alleiniger Busmaster; über ihn laufen alle Datentransfers von und zu den Peripheriechips. Moderne Systemarchitekturen versuchen aber möglichst viele Aufgaben zu dezentralisieren und entlasten den Zentralprozessor durch spezialisierte, anwendungsoptimierte Chips. DMA-Controller beispielsweise besitzen eine einfache Adreßlogik mit eigenem Speicher-Interface, so daß sie in der Lage sind, Daten selbständig aus dem Speicher zu holen. Ihr großer Vorteil liegt in der vergleichsweise simplen Architektur der Speicherschnittstelle und den festverdrahteten Funktionen. Bei einfachen Datentransporten sind sie den Zentralprozessoren weit überlegen.

Die Steuersignale der Buspriorität: BR*, BG*, BGACK*

BR*: (Bus Request = Busanforderung) signalisiert der CPU, daß ein intelligenter Peripheriechip den Bus übernehmen will.

BG*: (Bus Grant = Busfreigabe) quittiert den Empfang von BR*. Nach Ablauf des aktuellen Buszyklus gibt der Prozessor den Bus frei.

BGACK*: (Bus Grant Acknowledgement = Bestätigung der Busfreigabe) markiert die endgültige Übernahme der Buskontrolle durch den Peripheriechip und bleibt während der gesamten Busmasterphase aktiv. Die CPU hat jetzt bereits alle Busaktivitäten eingestellt und wartet auf die Negierung von BGACK*; erst dann setzt sie den normalen Busbetrieb fort.

Ein Multiprozessorbetrieb stellt hohe Anforderungen an die Buszugriffslogik. Datenkollisionen als Folge unkontrollierter Zugriffe können das System zerstören. Es muß gewährleistet sein, daß immer nur ein Master die Busse bedient. Vor einer endgültigen Übergabe der Buskontrolle an einen neuen Masterchip wird die Steuerlogik deshalb eine Reihe wichtiger Voraussetzungen überprüfen, die sie anhand einer Art Checkliste Punkt für Punkt abhakt.

Zu den wichtigsten Bedingungen gehören:

- Der Prozessor hat BR* durch BG* bestätigt und signalisiert damit seine Bereitschaft, den Busbetrieb nach Ablauf des aktuellen Buszyklus einzustellen.
- AS* ist inaktiv. (Der Prozessor unternimmt keinen weiteren Adressierungsversuch.)
- DTACK* ist ebenfalls inaktiv. (Keine Systemkomponente steht mit der CPU in Verbindung.)
- BGACK* muß vor der Aktivierung im inaktiven Zustand gewesen sein. (Der Prozessor ist Busmaster und hat die Kontrolle noch auf keinen anderen Chip übertragen.)

Die Unterbrechungssteuerung: IPL0*...IPL2* (IPL = Interrupt-Pending-Level)

Über die IPL*-Leitungen unterscheidet der 68000 sieben verschiedene Unterbrechungsebenen. Ebene 0 stellt eine Art Pseudopriorität dar, denn sie reflektiert den Zustand vor dem Auftreten der ersten Interruptanforderung. Der Maskierungsmechanismus ist so ausgelegt, daß Unterbrechungsanforderungen einer höheren Priorität die niedrigeren automatisch ausblenden. Die höchste Ebene (7) ist für die Systeminitialisierung reserviert und nicht maskierbar (NMI = Non Maskable Interrupt), kann also jederzeit von Anforderungen gleicher Priorität überrannt werden – das garantiert schnelle Reaktionszeiten und erweist sich als wirksamer Schutz gegen ein Festfahren auf dem höchsten Unterbrechungsniveau.

Eine Unterbrechungsanforderung an den Prozessor liefert ihre Priorität als 3-Bit-codierte Binärinformation auf den IPL*-Eingängen. Sie wird nicht sofort bearbeitet, sondern befindet sich dann in der schwebenden Phase ("pending"); erst der Vergleich mit der aktuellen Unterbrechungsmaske im Statusregister (Bits I0, I1, I2 repräsentieren die Priorität der höchsten, bisher eingetroffenen Unterbrechungsanforderung) entscheidet über die weitere Verarbeitung. Ist ihre Priorität größer als der Maskenwert, beginnt die CPU unverzüglich mit der Bearbeitung, modifiziert das Statusregister und setzt die Interruptbits auf das neue Niveau. Bei Gleichstand oder kleinerer Unterbrechungspriorität ignoriert der 68000 die Interruptanforderung und geht in den normalen Programmbetrieb zurück.

Der 68000 kennt zwei Arten der Vektorgenerierung als Antwort auf eine gültige Unterbrechungsanforderung, die autovektorielle und die non-autovektorielle (= nicht-autovektorielle). Der betroffene Systembaustein wählt den jeweiligen Modus über die Signale VPA* (für autovektoriell) bzw. DTACK* (für nicht-autovektoriell).

Bei VPA* erzeugt der Prozessor den Zeiger auf eine Unterbrechungsroutine, indem er die Priorität der Anforderung mit 4 multipliziert und zur Basisadresse (\$064) der Interruptvektoren in der Ausnahmevektor-Tabelle addiert. Empfängt die CPU dagegen DTACK* übernimmt sie die Ordnungszahl eines Anwendervektors auf den Datenbusleitungen D0 bis D7. Nach dem Lesevorgang wird die Nummer wieder mit 4 multipliziert und diesmal zur Basisadresse der 192 Anwendervektoren ab \$100 summiert.

Die non-autovektorielle Unterbrechung bietet dem Betriebssystem einen größeren Spielraum und erlaubt (durch die erweiterte Zahl von Interruptvektoren) entsprechend differenziertere Reaktionen; im Systemkonzept des Amigas spielt sie jedoch keine Rolle.

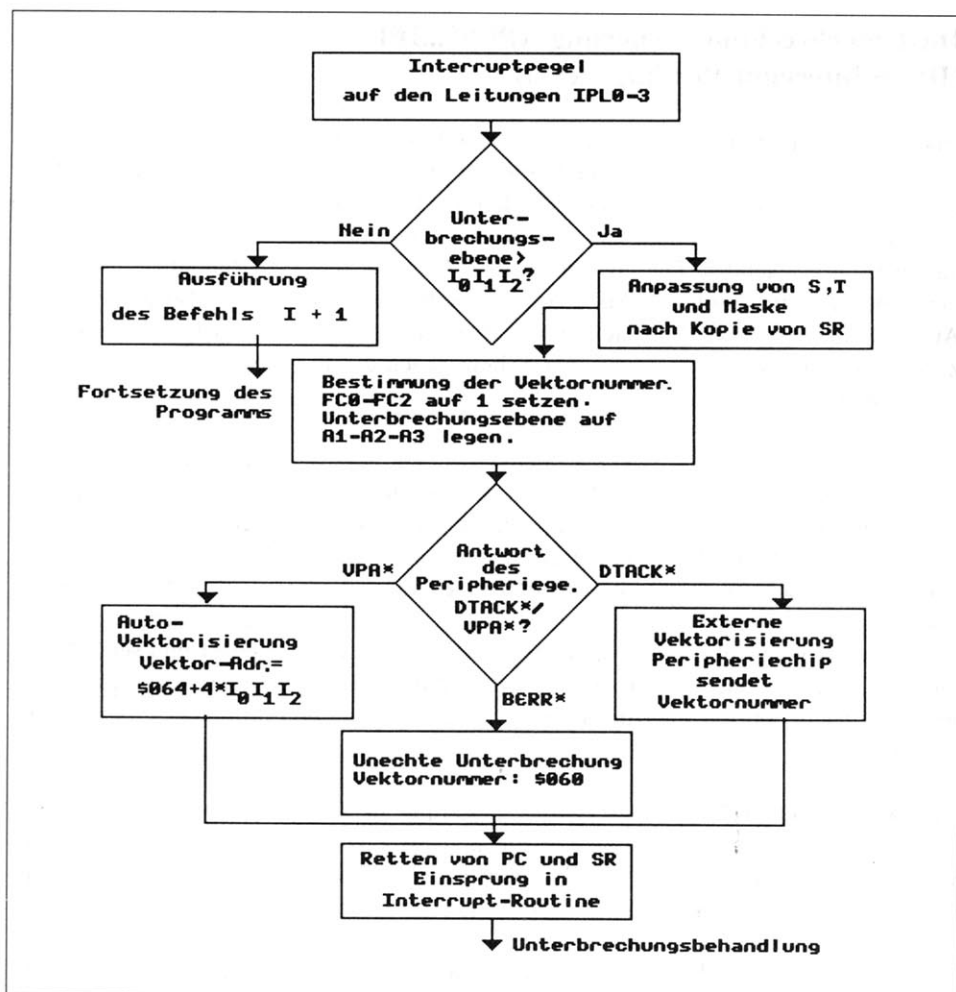


Abb. 2.5: Der Ablauf einer Unterbrechungsverarbeitung im Detail

Die Funktioncodes: FC0...FC2

Die Funktioncodes reflektieren den aktuellen Prozessorzustand und sind damit eine wichtige Schnittstelle zwischen CPU und Systemkomponenten. Hier können Informa-

tionen über den Betriebsmodus (Supervisor = Überwacher, User = Anwender), die Art des Speicherzugriffs (Befehle oder Daten) und den Interruptstatus abgegriffen werden.

Der 68000 läßt sich in die Karten schauen:

FC2	FC1	FC0	Funktion
0	0	0	unbenutzt
0	0	1	Anwender-Daten
0	1	0	Anwender-Code
0	1	1	unbenutzt
1	0	0	Überwacher-Daten
1	0	1	Überwacher-Code
1	1	0	unbenutzt
1	1	1	Interrupt

Ein solches Maß an Transparenz erschließt interessante Anwendungsgebiete. So lassen sich bei entsprechender Auswertung durch die Adreßdecodierlogik wirkungsvolle Konzepte zur Speichersicherung realisieren. Denkbar wäre sogar eine Erweiterung des Adreßraums auf bis zu 64 MByte (durch konsequente Daten/Befehls-Trennung von Betriebssystem und Anwenderprogramm).

Die Systemsteuersignale: BERR*, RESET*, HALT*

RESET*, HALT* und BERR* bilden die Schnittstelle zwischen Systemsteuerung und CPU. Ihre Funktionen sind nicht auf einzelne Leitungen fixiert, sondern werden durch diverse Kombinationen der Signale untereinander bzw. mit weiteren Leitungen aktiviert. Diese Technik spart Anschlüsse und entlastet das Pin-Layout.

BERR*: (Bus Error = Busfehler, unidirektional)

Dieses Signal wird von einer externen Logik generiert und informiert den Prozessor über Störungen des normalen Busbetriebs.

Hier ein Auszug der potentiellen Problemquellen:

- das adressierte Peripheriegerät antwortet nicht (z.B. Zugriff auf einen physikalisch nicht vorhandenen Speicherbereich).
- ein PIC (Programmable Interrupt Controller) initiiert eine vektorielle Unterbrechungsverarbeitung, kann aber keine Vektoradresse liefern.

- die Adreßlogik hat eine ungültige Adresse erhalten, die sich nicht zuordnen läßt.

Beim Empfang des BERR*-Signals springt der Prozessor über den Ausnahmevektor Nr. 2 in eine Betriebssystemprozedur, die den aufgetretenen Fehler analysiert und mit entsprechenden Gegenmaßnahmen reagiert.

RESET*: (bidirektional)

Die gleichzeitige Aktivierung von RESET* und HALT* löst einen kompletten System-RESET aus. Prozessor und Systembausteine gehen in einen klar definierten Ausgangszustand. Der RESET-Befehl setzt die gleichnamige Leitung für 124 Taktzyklen auf Low-Pegel. Der Prozessorzustand bleibt unverändert, aber alle direkt oder indirekt (über zusätzliche Logik angeschlossenen) Systembausteine werden initialisiert.

HALT*: (bidirektional)

Erkennt der Prozessor ein Low auf dieser Leitung, geht er automatisch in den "Tiefschlaf", alle Steuersignale werden inaktiv, die Bussysteme hochohmig.

Im Ausgabemodus signalisiert HALT* den totalen Abbruch der Befehlsausführung nach einem doppelten Busfehler. Dieser extreme Zustand ist die Folge eines erneuten Busfehlers während der Verarbeitung einer BERR*-Ausnahme. Dabei gehen wichtige Informationen auf dem Supervisor-Stack verloren, der Prozessor muß endgültig kapitulieren – in Ermangelung einer Alternative hilft nur noch der rigorose Neustart.

Ein doppelter Busfehler, der "GAU" unter den Ausnahmezuständen, ist der einzige Systemfehler, der einen 68000 wirklich aus der Bahn wirft; alle anderen Störungen können (bei entsprechenden Aufwand durch das Betriebssystem), zumindestens theoretisch abgefangen werden.

Die Praxis sieht allerdings ganz anders aus. Obwohl der 68000 ein recht fehlertoleranter Prozessor ist, neigen gerade Systeme mit dieser CPU zu überdurchschnittlich häufigen Systemabstürzen.

Offensichtlich gilt das aber nicht als Manko, denn an Stelle einer simplen Fehlermeldung wird der Absturz mit einem mehr oder weniger aufwendigen Bildschirmspektakel zelebriert. Da gibt es Bömbchen, Pilze oder gar Guru-Meditationen, die die frohe Botschaft verkünden, daß wieder einmal eine Ausnahmeprozedur in den Tiefen des Betriebssystems versunken ist.

Für geplagte Amiga-Besitzer ist der Griff zur RESET-Tasten-Kombination längst zu einer instinktiven Handlung geworden.

Die Betriebszustände des MC68000

Der Prozessor kennt drei verschiedene Zustände:

- Normalbetrieb
- Haltezustand
- Ausnahmeverarbeitung

Im Normalbetrieb differenziert die CPU zwischen System- und Anwendersoftware.

Das Betriebssystem läuft im Supervisor-Modus (= Überwacher) und darf als höchste Instanz sämtliche Befehle und Hardwareressourcen des Prozessors nutzen. Der User-Modus unterliegt einigen Beschränkungen. Die sogenannten privilegierten Befehle (RESET, RTE, STOP) sind ausschließlich für Systemsoftware reserviert, das Statusregister darf nicht modifiziert werden. Die besondere Stellung des Betriebssystems und die Privilegierung wichtiger Steuerbefehle garantieren ein hohes Maß an Betriebssicherheit.

Ausnahmeverarbeitung mit dem MC68000

Der Ausnahmezustand ist die Reaktion auf ein besonderes Ereignis, das eine individuelle Behandlung durch die Systemsoftware erforderlich macht.

Auslöser kann eine interne (beispielsweise Division durch Null) oder externe (z.B. Unterbrechungsanforderung), soft- (RESET-Befehl) bzw. hardwarebedingte Ausnahme-situation sein:

- Unterbrechungen (non-/autovektoriell)
- RESET-Befehl
- TRAP-Befehle
- TRACE-Modus
- Illegaler Befehl (Instruktionen, die mit 1111 bzw. 1010 beginnen)
- Privilegverletzung (Statusregister-Manipulationen, STOP, RESET, RTE im User-Modus)
- Division durch Null
- Busfehler
- Adreßfehler
- Unehnte Unterbrechung

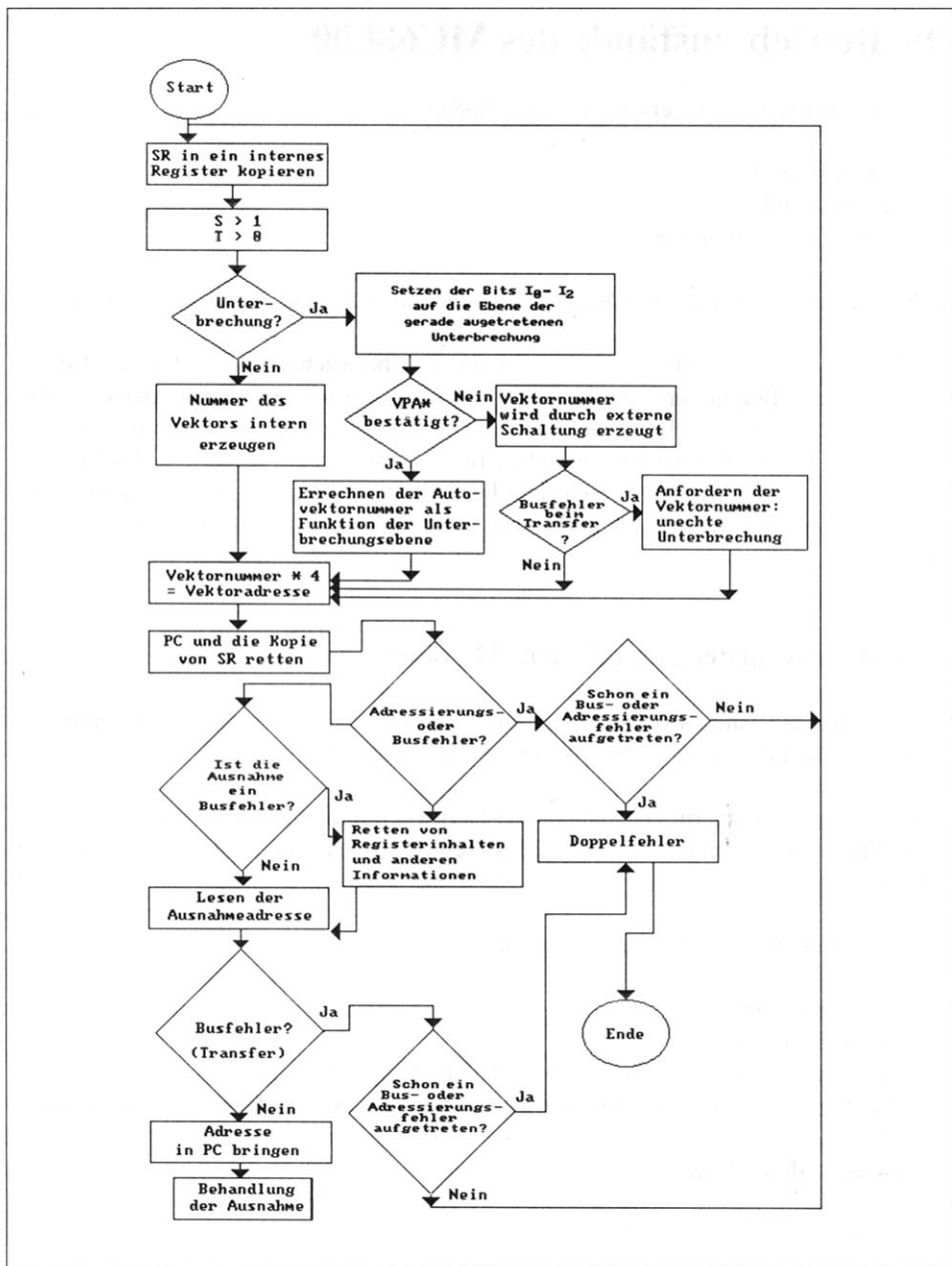


Abb. 2.6: Schematischer Ablauf einer Ausnahmeverarbeitung

Die Steuerlogik des 68000 weist jedem Ausnahmezustand eine feste Position in der Ausnahmevektor-Tabelle zu (mit Einschränkung der non-autovektoriellen Unterbrechung, die ja eine externe Beeinflussung erlaubt).

Die Vektoren werden vom Betriebssystem initialisiert und zeigen auf spezielle Abfangroutinen.

Nummer des Vektors	Adresse		Betroffene Ausnahme
	Dez.	Hex.	
0	0	000	Reset: Initialisierung von SSP
	4	004	Reset: Initialisierung von PC
2	8	008	Busfehler
3	12	00C	Adreßfehler
4	16	010	Illegaler Befehl
5	20	014	Division durch Null
6	24	018	Befehl CHK
7	28	01C	Befehl TRAPV
8	32	020	Privilegverletzung
9	36	024	Ablaufverfolgung
10	40	028	Befehl mit 1010 am Anfang
11	44	02C	Befehl mit 1111 am Anfang
12	48	030	Reserviert
13	52	034	Reserviert
14	56	038	Reserviert
15	60	03C	Nicht initialisierter Unterbrechungsvektor
16-23	64	040	Reserviert
	96	05F	
24	96	060	Unechte Unterbrechung
25	100	064	autovektorielle Unterbrechung, Ebene 1
26	104	068	autovektorielle Unterbrechung, Ebene 2
27	108	06C	autovektorielle Unterbrechung, Ebene 3
28	112	070	autovektorielle Unterbrechung, Ebene 4
29	116	074	autovektorielle Unterbrechung, Ebene 5
30	120	078	autovektorielle Unterbrechung, Ebene 6
31	124	07C	autovektorielle Unterbrechung, Ebene 7
32-47	128	080	TRAP-Befehlsvektoren
	191	0BF	-
48-63	192	0C0	Reserviert
	255	0FF	-
64-255	256	100	Anwender-Unterbrechungsvektoren
	1023	3FF	-

Abb. 2.7: Tabelle der Ausnahmevektoren

Herzverpflanzung – Systemtuning mit anderen Prozessoren aus der 68000-Familie

Deluxe-Paint, perspektivische Darstellung, Antialiasing ... endloses Warten. Einfache Mandelbrot-Menge, relativ grobe Rekursion ... nervenzerrende Rechenarbeit. Simplex Raytracing-Bild ... extrem lange Aufbauzeit, tschüß bis Morgen!

Situationen, die die Grenzen eines 68000-Systems deutlich machen. Wer liebäugelt da nicht mit dem mittlerweile recht umfangreichen Angebot diverser Prozessor-Upgrades? Ein arithmetischer Coprozessor beschleunigt Gleitkommaberechnungen bis zum Faktor 500 und ist in der 12,5-MHz-Version bereits für unter 300 DM zu haben. Die Werbung verspricht gigantische Leistungssteigerungen, echte 32-Bit-Power zum Greifen nahe – der heimische Amiga zum Supercomputer ausgebaut!

Doch wie kompatibel sind diese Erweiterungen? Vertragen sie sich auch mit dem Betriebssystem? Läuft die alte Software? Wie groß sind die zu erwartenden Leistungssteigerungen wirklich? ... Fragen, die trotz aller Euphorie in den Vordergrund rücken.

Die erste Hürde – Systemsoftware und 680xx

Da die Prozessoren der 68000-Familie untereinander aufwärtskompatibel sind (d.h. der Maschinencode des Vorgängers läuft auch auf dem Nachfolger), steht einer Prozessoraufrüstung unter dem Aspekt der Softwareverträglichkeit eigentlich nichts im Wege.

Der Haken dabei liegt in der Privilegierung des MOVE-SR,<ea>-Befehls (Kopie des gesamten Statusregisters), der zwar vom 68000 im Anwender-Modus anstandslos geschluckt wird, ab 68010 jedoch nur im Überwacher-Betrieb erlaubt ist.

Nun ist das Kompatibilitätsproblem bereits seit der Markteinführung des 68010 vor ca. fünf Jahren bekannt; Zeit genug – so sollte man meinen – die Softwareentwicklung darauf einzustellen, zumal die Umgehung der unverdaulichen Anweisung mit dem MOVE-CCR,<ea>-Befehl (CCR = Condition Code Register, dieser Teil des Statusregisters darf vom Anwender modifiziert werden) nicht aufwendiger ist.

Um so erstaunlicher, daß selbst Betriebssysteme mit den neueren Mitgliedern der 68000-Serie zu kämpfen haben. So weigert sich die Systemsoftware des ATARI ST, bekannt unter dem Kürzel TOS, hartnäckig gegen eine Installation auf dem 68020.

Wer dennoch nicht auf 32-Bit-Power verzichten will, muß entsprechende Abfangroutinen schreiben oder ein entsprechend angepaßtes TOS erwerben. (Mittlerweile gibt es auch die 32-Bit-Version für den TT.)

Im Gegensatz dazu verdienen die Betriebssystementwickler des Amigas ein großes Lob. Mit Blick auf zukünftige Innovationen der Amiga-Technologie, bemühte man sich um eine möglichst harmonische Systemeinbindung stärkerer Vertreter der 680xx-Linie. Während der Initialisierungsphase prüft EXEC die Prozessorkonfiguration und vermerkt das Ergebnis in den "AttnFlags" der ExecBase-Systemstruktur.

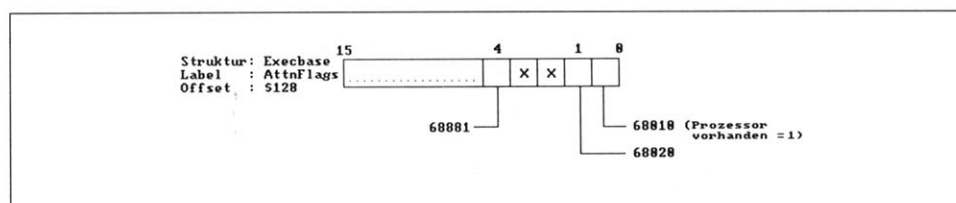


Abb. 2.8: Was ist denn nun drin? – die "AttnFlags"

Die genaue Kenntnis der vorhandenen Prozessorkonfiguration ist unabdingbare Voraussetzung für den Multitasking-Betrieb; schließlich stellt das Gespann 68020/68881 besondere Anforderungen an die Systemsoftware. Greifen beispielsweise mehrere Tasks quasi parallel auf den 68881 zu, müssen die Inhalte der Floatingpoint-Register bei jedem Taskwechsel gesichert werden. Ähnliches gilt auch für die Behandlung des 68020-Caches; hier hat man gar an die Aktivierung nach einem System-RESET gedacht. All diese Vorgänge laufen verdeckt und vollkommen automatisch – das Betriebssystem ist bestens gerüstet.

Und die 1.3-Version der Betriebssoftware geht noch weiter. Ist ein 68881er vorhanden, ersetzt sie die langsame Software-Emulation der Fließkomma-Library durch direkte Befehlssequenzen für den arithmetischen Coprozessor.

Rechenintensive Programme, die von der Library Gebrauch machen, kommen wie selbstverständlich in den Genuß des schnellen Rechengenies; eine aufwendige (und teuer bezahlte) Modifikation vorhandener Softwarepakete entfällt.

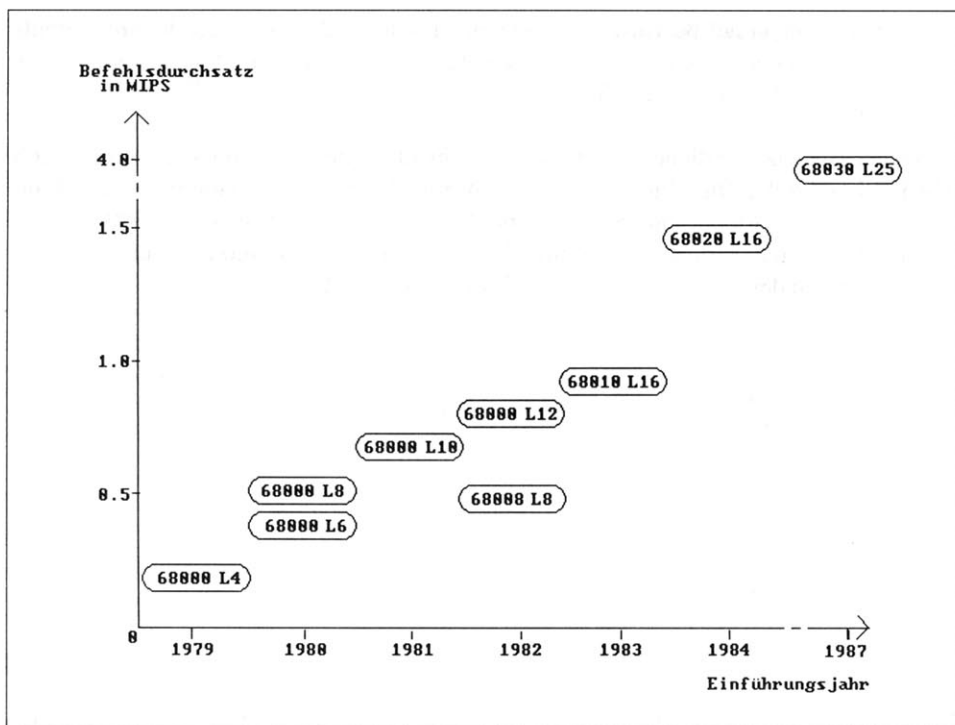


Abb. 2.9: Markteinführung und Leistungsfähigkeit der 680xx-Prozessoren

Der Weg des geringsten Widerstandes – der 68010 im Amiga

Die 68010-CPU ist die preiswerteste Alternative einer Prozessorausrüstung. Vollständig pin-kompatibel zum 68000, reduziert sich die Arbeit auf das Aushebeln des alten Prozessors und Einstecken des neuen ... fertig.

Der 68010 war Motorolas Antwort auf die gestiegenen Anforderungen von Betriebssystemen. Sein virtuelles Speicherkonzept qualifiziert ihn für den Multiuser/Multitasking-Betrieb. Trotzdem hat sich an der internen Architektur des Prozessors nicht viel geändert; lediglich drei neue Register, vier privilegierte und zwei geänderte Befehle ergänzen das Programmiermodell. Echter Multiuserbetrieb verlangt die Verwaltung getrennter Informationsbereiche für jedes Betriebssystem. Der 68010 trägt dem Rechnung, indem er die Ausnahmevektor-Tabelle nicht mehr an eine feste Position (ab Adresse 0) bindet, sondern mit Hilfe des Vektorbasisregisters eine freie Verschiebung im Speicher

erlaubt. Die Umschaltung zwischen den einzelnen Vektortabellen erfolgt durch Änderung der Basisadresse im VBR-Register.

Beim Einsatz des 68010 im Amiga-System sind Eigenschaften wie virtuelles Speicherkonzept und frei-positionierbare Vektortabelle nur schmückendes Beiwerk, brachliegende Fähigkeiten. Die Erwartungen eines derartigen Prozessoraustausches knüpfen sich an die Geschwindigkeitsvorteile des 68010, schließlich optimierten die Entwickler (quasi als Bonbon) den Mikrocode (wovon besonders arithmetisch/logische Operationen mit 32-Bit-Datenformat profitieren) und implementierten einen speziellen Loop-Modus für Mini-Schleifen.

Die beschleunigten Instruktionen des MC68010 (+ = addiere die Berechnungszeit der effektiven Adresse):

<i>Befehl</i>	<i>MC68000</i>	<i>MC68010</i>	<i>Steigerung</i>
DIVS	158	122+	30%
DIVU	140	108+	23%
MULS	70	max. 42+	40%
MULU	70	40+	43%
ADDX.L Dn,Dn	8	6	25%
SUBX.L Dn,Dn	8	6	25%
ANDI.L #x,Dn	16	14	12,5%
ANDI.L #x,Dn	16	14	12,5%
CMPI.L #x,Dn	16	14	12,5%
EORI.L #x,Dn	16	14	12,5%
ORI.L #x,Dn	16	14	12,5%
SUBQ.W #x,An	8	4	50%
ANDI #x,CCR	20	16	20%
ANDI #x,SR	20	16	20%
EORI #x,SR	20	16	20%
ORI#x,CCR	20	16	20%
ORI#x,SR	20	16	20%
CHK	10	8	20%
MOVE SR,<ea>	6	4	33%
Bcc.B	8	6	25%
Bcc.W	12	10	17%

DBcc.B	12	10	17%
DBcc.W	16	14	12,5%

Der Loop-Modus arbeitet mit einem kleinen 6-Byte-On-Chip-Cache (schneller Zwischenspeicher auf dem Prozessorchip). Während der Ausführung werden die Schleifenbefehle gepuffert; die CPU muß lediglich zum Lesen und Schreiben der Operanden auf den Speicher zugreifen.

Kurze Schleifen (maximal zwei Befehle), die im Bedingungssteil einem DBxx-Befehl und als Rumpf eine, der insgesamt 33 "loopable" Instruktionen enthalten, erreichen mühelos Ablaufsteigerungen von 150%. Einfaches Verschieben eines Speicherbereichs von A nach B ist im Loop-Modus ein rasantes Schauspiel:

```
start: move.l (a0)+, (a1)+ ;MOVE ist loopable
       dbra    d0, start  ;d0 dient als Zähler für die
                           Schleifendurchläufe
```

Ein neuer, optimierter Algorithmus bringt die Multiplikation auf Trab und erzielt Geschwindigkeitssteigerungen von über 40%, die Division legt mit 23% ebenfalls merklich zu.

Da sich die Verbesserungen allerdings nur auf einige Befehle beziehen (Datentransportbefehle beispielsweise sind nicht schneller geworden), bleiben die im Alltagsbetrieb erzielten Ergebnisse hinter den Erwartungen zurück. Gerade bei Standardsoftware ist die Geschwindigkeitssteigerung kaum spürbar (< 5%); erst Programme mit viel 32-Bit-FFP (Fast-Floating-Point = schnelle Gleitkomma-Arithmetik) und speziell auf den 68010 zugeschnittenen Routinen, rechtfertigen den Aufwand eines Prozessorausstausches.

Der Amiga mit Workstationambitionen – der MC68020

Wesentlich effektiver, wenn auch erheblich teurer, ist eine Aufrüstung mit dem MC68020. 1984 vorgestellt, war diese CPU – Motorolas erster echter 32-Bit-Prozessor – kompromißlos auf diese Datenbreite ausgelegt.

Trotz eines erheblich erweiterten Befehlssatzes, zusätzlicher Datenformate und noch mächtigerer Adressierungsarten bleibt er (mit Einschränkung des MOVE-SR,<ea>-Befehls) vollkommen aufwärtskompatibel zu seinen Vorgängern (68000, 68008, 68010, 68012). Neben einer höheren Taktfrequenz und dem 32-Bit-Bussystem machen sich der verkürzte Buszyklus (drei Taktzyklen anstatt vier beim 68000) und der schnelle 256

Byte-Befehlscache angenehm bemerkbar. Der universelle Datentransportbefehl MOVE zeigt den fundamentalen Anstieg des Prozessordurchsatzes:

```
move.l $C00000,$E00000 ; Adressierungsart: Absolut/Lang  
nach Absolut/Lang
```

Diese Sequenz benötigt auf der 68000-CPU 36 Taktzyklen; dabei werden neun 16-Bit-Worte über den Datenbus geschaufelt (eins für das 16-Bit-Befehlswort, jeweils zwei für Quell- und Zieladresse, schließlich noch vier für die Verschiebung des 32-Bit-Operanden). Auf Basis eines 4-Takt-Buszyklus entspricht das einem Befehlszyklus von $9 * 4 = 36$ Taktzyklen.

Dank des 32-Bit-Datenbusses braucht der 68020 für diese Instruktion nur fünf Buszyklen (Befehlswort, $2 * 32$ -Bit-Operandenadresse, Datentransport). Der 3-Takt-Buszyklus reduziert die Befehlsausführungszeit auf 15 Taktzyklen ($= 5 * 3$), so daß der 68020 bei gleicher Taktfrequenz mehr als doppelt so schnell wie sein Vorgänger arbeitet!

So weit die Theorie – in der Praxis sieht das oft ganz anders aus. Denn ohne 32-Bit-Speicher-Interface und entsprechend schnelle RAM-Bausteine kann der 68020 seine überlegenen Fähigkeiten überhaupt nicht ausspielen. Muß er seine Daten durch ein "16-Bit-Nadelöhr quetschen" und sich mit den vergleichsweise langsamen RAM-Bausteinen des Amiga-Systems begnügen, arbeitet der 32-Bitter kaum schneller als sein 16-Bit-Vorgänger. Bemerkenswerte Ergebnisse liefern Benchmarks, bei denen der 68020 noch auf seinen schnellen Befehlscache verzichten muß. Eingezwängt in das enge "Amiga-Korsett" schneidet er teilweise sogar schlechter ab als der alte 68000.

Diese Tatsache wirft ein anderes Licht auf das inzwischen recht umfangreiche Angebot diverser Speed-Boards und 68020-Karten. Der Prozessoraustausch gegen ein höher-getaktetes Modell oder gar eine vollwertige 32-Bit-CPU allein, bringt noch nicht den erhofften Geschwindigkeitszuwachs. Erst die richtige Systemumgebung (z.B. schneller 32-Bit-Speicherbus, RAM-Puffer für das Kickstart) macht aus dem Amiga einen Hochleistungscomputer mit Workstationambitionen.

Warum nicht träumen? – der MC68030

Das Topmodell der 68000-Familie vereint so ziemlich alles auf einem Chip, was die 32-Bit-Mikroprozessortechnologie momentan zu bieten hat:

- 256 Byte-Cache für Befehle
- 256 Byte-Cache für Daten

- Harvard-Architektur (Daten und Adreßbus sind intern doppelt ausgeführt)
- MMU (Memory-Management-Unit) auf dem Prozessorchip integriert
- 2-Takt-Buszyklus
- Burst-Mode (mit Nibble und Page-Mode-DRAMs) zum schnellen Nachladen des Cachespeichers (5 Taktzyklen für 128 Bit)
- ca. 7 MIPS Leistung bei 33-MHz-Takt

Es ist nicht erstaunlich, daß dieser Prozessor noch empfindlicher auf seine Systemumgebung reagiert als der 68020; nur unter optimalen Bedingungen kann er die volle Leistungsfähigkeit zur Geltung bringen.

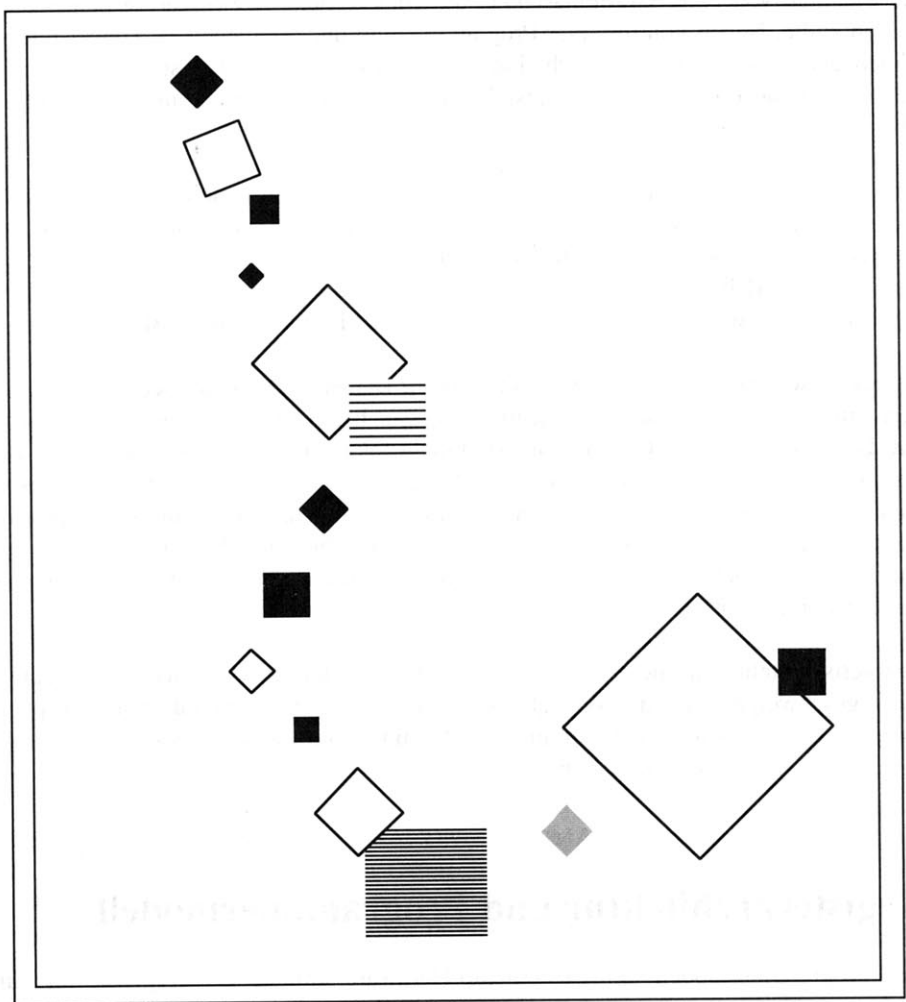
Viele, der in Mode geratenen Adapter mit dem 68030-Prozessor, erreichen selbst in 68020-Systemen kaum spürbare Leistungssteigerungen.

In letzter Zeit häufen sich aber die Ankündigungen (zum Teil sind sie auch schon erhältlich) durchaus vielversprechender 68030-Karten für das Amiga-System, die mittels diverser architektonischer Kniffe, einiges aus diesem Superprozessor herauskitzeln.

Wir dürfen gespannt sein!

Kapitel 3

Die Customchip - Hardware



Die Fachbegriffe Customchip (= Kundenbaustein) bzw. ASIC (Application Specific IC = anwendungsspezifisches IC) beschreiben einen Chiptyp, der gezielt auf Kundenwünsche abgestimmt ist und in vergleichsweise geringen Stückzahlen produziert wird – kein Massenprodukt von der Stange. Derartige Bausteine gewinnen in der modernen Computertechnik eine zunehmende Bedeutung, da die verfügbaren Chips den gestiegenen Leistungsanforderungen spezieller Anwendungsgebiete nicht mehr gewachsen sind.

Das galt auch für das Amiga-Projekt. Die Entwickler hatten sich hohe Ziele gesteckt; sie träumten von einer Grafikmaschine, die alles bisher Dagewesene in den Schatten stellen sollte. Neben komfortabler Programmierung und weitgehender Transparenz der Videogenerierung hatte die Geschwindigkeit höchste Priorität. Ein schneller Prozessor bildet zwar die Basis eines leistungsfähigen Grafiksystems, reicht aber noch nicht für überzeugende Ergebnisse aus.

Vor diesem Hintergrund entstanden die Amiga-Customchips, deren eigenwillige Namensgebung so etwas wie Persönlichkeit, ja sogar Charisma ausstrahlt (Die ausdruckslosen Seriennummern 836x waren den Entwicklern schlichtweg zu blaß!). Es sind diese VLSI-Bausteine, die den Amiga aus der Masse konventioneller Computersysteme hervorheben und ihm den gewissen Touch des Extravaganten verleihen.

Agnus (bzw. Fat Agnus im A500 und B2000), Denise und Paula erledigen eine Fülle von Aufgaben und entlasten den Zentralprozessor bei zeitaufwendigen Alltagsgeschäften. Die Funktionen der Customchips sind nicht lokal auf den einen oder anderen Baustein fixiert; das Meiste ist Gemeinschaftsarbeit mit bunt verteilten Zuständigkeitsbereichen. Logisch verschmelzen sie ohnehin zu einer Einheit, obwohl die Chips physikalisch getrennt, in verschiedenen Gehäusen untergracht sind. Warum integrierte man dann nicht sämtliche Transistorfunktionen auf einem einzelnen Chip – quasi einer Art Super-Coprozessor?

Theoretisch steht dem nichts im Wege, technisch ist der Aufwand aber noch heute zu groß (geschweige denn zum Vorstellungstermin Mitte 1985), einen derartig komplexen Baustein herzustellen (er hätte immerhin die Integrationsdichte eines MC68020). Für den Massenmarkt erscheint die Drei-Chip-Lösung geeigneter.

Registerarchitektur und Programmiermodell

Das Programmiermodell der Customchip-Hardware umfaßt 197 Register mit einer Breite von 16 Bit. Sie sind ab Adresse \$DFF000 (in den Datenblättern empfohlene Anfangsadresse) als zusammenhängende Speicherbank (ohne Adreßsprünge) in die Memory-Map des Systems eingeblendet. Der Adreßwert \$DFF000 bildet damit die Basis-

adresse einer geschlossenen Registertabelle, deren einzelne Einträge (die Registeradressen selbst) in jeweils 2-Byte-Abständen (16-Bit-Register!) an geraden Adressen liegen. Um auf ein Register der Tabelle zuzugreifen, muß lediglich sein Abstand (= Offset) zur Basisadresse bekannt sein. Dazu ein Beispiel:

Ein Spielprogramm benötigt Informationen über den Zustand von Game-Port 0. Das übernimmt eine kleine Maschinenroutine, die das JOY0DAT-Register abfragt (es registriert die Bewegungen eines Joysticks am Port 0). JOY0DAT ist das fünfte Register der Tabelle und liegt an der Position \$00A (dezimal $10 = 5 * 2$ -Byte-Registerabstand).

So könnte die Routine aussehen:

```
lea $DFF000,a5          ;Basisadresse in das
                        ;Adreßregister a5
move $00A(a5),d0        ;JOY0DAT nach d0 (Adressierung:
                        ;indirekt m. Offset)
jsr Test                ;Sprung zur Testroutine
```

Für den Programmierer stellen die memory-mapped Customchip-Register normale Speicherstellen dar, die er mit Datentransportbefehlen wie MOVE problemlos manipulieren kann. Allerdings entspricht der Registersatz der Customchip-Hardware keinem dynamischen Schreib-/Lese-Programmiermodell à la MC68000. Um die Registerarchitektur möglichst einfach zu halten, verzichtete man auf Adressen mit wahlfreier Zugriffsrichtung; sie können entweder nur gelesen oder nur beschrieben werden. Trotzdem gibt es Register, die beide Zugriffsrichtungen erlauben. (Ansonsten wäre der Programmierer ja gezwungen, jede Veränderung der Registerzustände zwischenspeichern – ein unzumutbarer Zustand.)

Und hier verläuft die Trennungslinie zwischen physikalisch existenten Registern und ihrer Zugriffsadresse. Ein Schreib-/Lese-Register ist im Programmiermodell der Customchips durch Aufspaltung der Zugriffsadresse in Schreib- und Lese-Bereich realisiert; dabei stellt die lesbare Registeradresse eine Art Kopie (eventuell leicht modifiziert) des Nur-Schreib-Registers dar.

So liegt das DMACON-Register (DMACONtrol = DMA-Kontrollregister, aktiviert den DMA-Zugriff) im Schreibzugriff an Offset \$096, während der Leseteil DMACONR (das R steht für Read) am Anfang der Registertabelle unter Offset \$002 zu finden ist (weitere Register dieses Typs: INTREQ, INTREQR, INTENA, INTENAR, ADKCON, ADKCONR usw.).

Zentrale Steuerregister mit globalen Daten, die nicht nur für einen einzelnen Customchip von Bedeutung sind, gibt es gleich in mehrfacher Ausführung; das erspart einen

Die Daten- oder Parameterregister

Rein quantitativ bilden sie die größte Gruppe. Der Zugriff ist unproblematisch; die CPU kann Parameter setzen oder abfragen, je nachdem, ob es sich um ein Nur-Lese bzw. Nur-Schreib-Register handelt. Zu diesem Typ gehören die Farbbregister des Color-Lookup-Table (Farbtabelle ab Offset \$180, 12-Bit-RGB-Farbinformation), die Steuerregister der Audio-Hardware (Lautstärke, Ausgabefrequenz) oder lesbare Portregister wie JOYxDAT (Maus- und Joystickabfrage am Port x).

Eine besondere Stellung innerhalb dieser Gruppe nehmen die Datenregister des DMA-Systems ein. Sie puffern die wortweise übertragenen Daten von und zum CHIP-RAM (z.B. BLTADAT für die Blitter-Quelldaten von Source A, AUD0DAT für Sampledaten des Audiokanals 0) und werden normalerweise ausschließlich vom DMA-Controller angesprochen. Im Übertragungsmodus Speicher/Register kann auch der Prozessor diese Aufgabe übernehmen.

Die Kontrollregister

Die Kontrollregister (DMACON, ADKCON = Audio/Disk Control, INTENA = Interrupt Enable usw.) erlauben eine gezielte Programmierung der Customchip-Hardware.

Man kann ihre Bits als Schalter auffassen, mit denen einzelne Funktionen in den Steuerungsprozeß ein- bzw. ausgeblendet werden. Die Initialisierung eines derartigen Registers (in dem jedes Bit eine bestimmte Steuerfunktion ausübt) wirft die Frage auf:

Wie kann man gezielt einzelne Binärstellen ändern, ohne dabei den gesamten Registerinhalt zu zerstören?

Bei den Allzweckregistern des 68000 ist das kein Problem. Da gibt es Bitmanipulationsbefehle oder logische Operationen, die nur bestimmte Bits verändern, den Rest des Registers aber nicht beeinflussen.

Eine potentielle Lösung für das Programmiermodell der Customchip-Hardware liegt in der Auswertung bzw. Modifikation des Lese-Teils eines Registers (sofern das betreffende Kontrollregister auch eine lesbare Zugriffsadresse besitzt) und dem Zurückschreiben des Ergebnisses in die korrespondierende Nur-Schreib-Adresse. Um beispielsweise im DMACON-Register Bit 0 auf den Wert 1 zu setzen, eignet sich folgende Sequenz:

```
lea $DFF000,a5          ;Basisadresse nach a5 move
$002(a5),d0             ;DMACONR nach d0 or.w
```

```

$0002,d0                ;Bit 1 einblenden
move d0,$096(a5)        ;Ergebnis ins DMACON-Register

```

Diese Methode erscheint allerdings sehr unkomfortabel – man ahnt schon, daß es auch wesentlich einfacher geht. Sehen wir uns dazu das DMACON-Register genauer an:

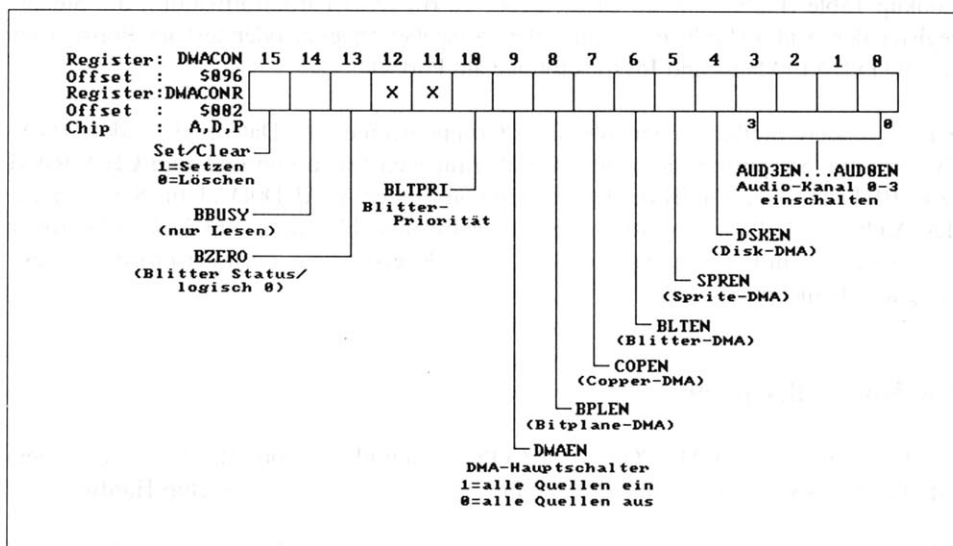


Abb. 3.1: Das DMACON-Register

DMACON aktiviert die Datenversorgung der logisch unabhängigen DMA-Systeme (so wird die Bitplane-DMA sinnvollerweise nur komplett aktiviert, während die einzelnen Audiokanäle getrennt angesprochen werden). Jede DMA-Quelle wird durch ein zugehöriges Bit repräsentiert, dessen Zustand den DMA-Transfer wahlweise startet (= 1) bzw. beendet (= 0).

Für die gezielte Modifikation einzelner Bits statteten die Entwickler das Register mit einem speziellen Zugriffsmechanismus aus, bei dem der Operand ein Maskenwort darstellt und das höchstwertige Bit den Verknüpfungsmodus (Löschen = 0, Setzen = 1) bestimmt. Da nur die 1er-Bits der Maske das Zielregister beeinflussen, alle anderen Bits aber unverändert bleiben, ist die Initialisierung genauso flexibel wie die eines CPU-Datenregisters.

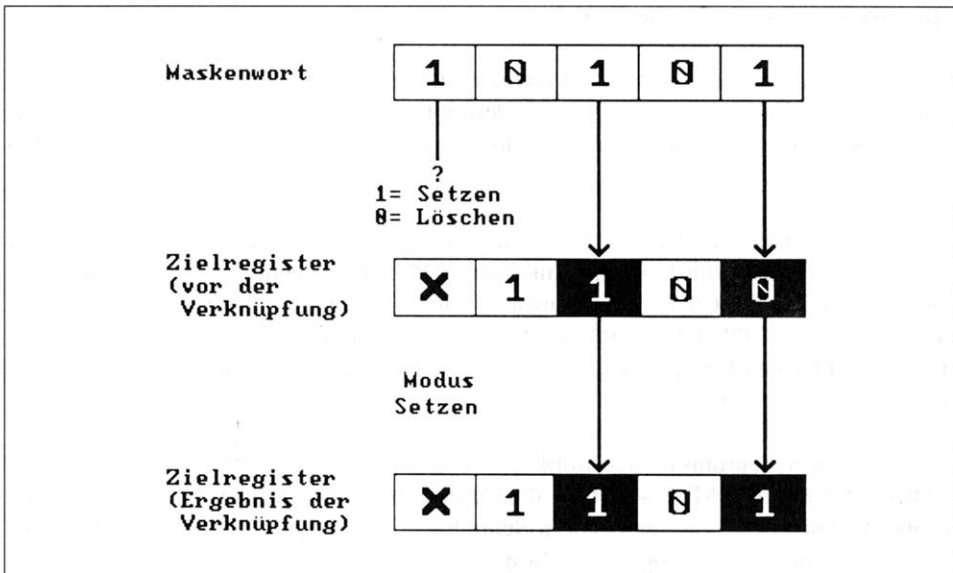


Abb. 3.2: Grundlage des Programmiermodells: Der Set/Clear-Mechanismus

Zurück zum DMACON-Register; die Aktivierung von Audiokanal 0 erfordert beispielsweise folgende Überlegungen:

Modus (Set)	↓ Bit 15 = 1
Hauptschalter DMAEN setzen	↓ Bit 9 = 1
Audio-DMA für Kanal 0	↓ Bit 0 = 1
alle anderen Bits nicht verändern	↓ = 0

Die entsprechende Sequenz sieht dann so aus:

```
move #$8201,DMACON           ;schaltet die Audio-DMA ein!
```

Zum Ausschalten reicht der Befehl:

```
move #$0001,DMACON
```


"Impulsgeber" – die Strobe-Register

Strobe-Register spielen in vielen Systemarchitekturen eine wichtige Rolle. Sie sind keine Register im klassischen Sinne, sondern interne Signalleitungen (Strobes), bei deren Adressierung ein Steuerimpuls ausgelöst wird (deshalb speichern sie auch keine Parameter).

Das Registerkonzept der Customchips umfaßt insgesamt sechs Register dieses Typs. Sie ersetzen extern ausgeführte Kommunikationsleitungen und entlasten das ohnehin recht enge Pin-Layout der Customchip-Hardware. So wird der Grafikchip Denise über vier Strobe-Register (STREQU, STRVBL, STRHOR, STRLONG) auf die Impulse der Rasterstrahlzähler von Fat Agnus synchronisiert; für die externe Verbindung beider Chips war keine Leitung mehr frei.

Neben diesen Synchronisations-Strobes gibt es noch zwei weitere Register, (COPJMP1 – Offset \$088, COPJMP2 – Offset \$08A), die ebenfalls typische Anwendungen der Strobe-Architektur repräsentieren. Ein Schreibzugriff auf die Adressen \$DFF088 bzw. \$DFF08A veranlaßt die Copper-Hardware, den internen Programmzähler mit dem Inhalt des COPxLC(H/L)-Adreßregisters zu laden (zeigt auf den Beginn einer Copper-Befehlsliste).

Die Customchip-Register im Überblick

Die nun folgende Tabelle schließt die Besprechung des Programmiermodells mit einer tabellarischen Auflistung sämtlicher Customchip-Register ab. Sie sind, gemäß ihres relativen Abstands (= Offset) zur Basisadresse, in aufsteigender Reihenfolge sortiert. Die jeweils kurze Funktionsbeschreibung erhebt keinen Anspruch auf besonderen Tiefgang, sondern dient zur schnellen Orientierung und bietet damit wichtige Anhaltspunkte für die Einordnung des betreffenden Registers.

Eine ausführliche Beschreibung (komplett mit Adresse, Tabellenoffset, Zugriffsrichtung, Verteilung auf die Customchips und Funktionserläuterung) befindet sich im Anhang des Buches.

Kürzel und Bezeichnungen:

Label:	Registerbezeichnung (gemäß Datenblätter)
Offset:	relativer Abstand zur Basisadresse; Effektive Adresse $\langle ea \rangle = \text{Basisadresse} (\$DFF000) + \text{Registeroffset}$

Chip: Lage des Registers (A = Fat Agnus bzw. Agnus, D = Denise, P = Paula)

Zugriff: Zugriffsrichtung (R = Lesen, W = Schreiben) und Zugriffsberechtigung (d = DMA-Controller, p = Prozessor, c* = Copper bei COPCON = 1, c = Copper ohne Einschränkung)

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
BLTDDAT	\$000	A	R(d)	DMA-Datenregister des Blitters (Übertragungsrichtung Register->CHIP-RAM)
DMACONR	\$002	AP	R(p)	DMA-Kontrollregister (lesen)
VPOS	\$004	A	R(p)	Vert. Rasterstrahlpos. (höchstw. Bit)
VHPOS	\$006	A	R(p)	Vert. und horiz. Rasterstrahlposition
DSKDATR	\$008	P	R(d)	Lesedaten der Diskette (Register->CHIP-RAM)
JOY0DAT	\$00A	D	R(p)	Joystick-/Maus-Bewegungen Game-Port 0
JOY1DAT	\$00C	D	R(p)	Joystick-/Maus-Bewegungen Game-Port 1
CLXDAT	\$00E	D	R(p)	Kontrollregister (Grafikobjekt-Kollisionen)
ADKCONR	\$010	P	R(p)	Audio-/Disk-Kontrollregister (lesen)
POT0DAT	\$012	P	R(p)	Potentiometer (Game-Port 0)
POT1DAT	\$014	P	R(p)	Potentiometer (Game-Port 1)
POTGOR	\$016	P	R(p)	Datenregister (Pot.-Port, lesen)
SERDATR	\$018	P	R(p)	Datenregister (Serieller Port u. Status, lesen)
DSKBYTR	\$01A	P	R(p)	Datenbyte und Status des Disketten-Laufwerks
INTENAR	\$01C	P	R(p)	Interrupt-Kontrollregister (lesen)
INTREQR	\$01E	P	R(p)	Interrupt-Anforderung (lesen)
DSKPTH	\$020	A	W(p)	Disk-DMA-Adreßregister (High, Bits 16 bis 18)
DSKPTL	\$022	A	W(p)	Disk-DMA-Adreßregister (Low, Bits 0 bis 15)
DSKLEN	\$024	P	W(p)	Blocklänge der Disk-DMA
DSKDAT	\$026	P	W(d)	Disk-DMA-Datenregister (Register ->CHIP-RAM)
REFPTR	\$028	A	W(dp)	Refreshpointer (Zeilenadressierung der DRAMs)
VPOSW	\$02A	A	W(p)	Vert. Rasterstrahlpos. (höchstw. Bit)
VHPOSW	\$02C	A	W(p)	Vert. und horiz. Rasterstrahlposition
COPCON	\$02E	A	W(p)	Copper-Kontrollregister
SERDAT	\$030	P	W(p)	Datenregister des UART
SERPER	\$032	P	W(p)	Kontrollregister des UART
POTGO	\$034	P	W(p)	Datenregister Pot.-Port
JOYTEST	\$036	D	W(p)	schreibt in beide Mausezähler

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
STREQU	\$038	D	W(d)	Strobe-Register (synchronisiert Denise auf die Rasterstrahlzähler von Fat Agnus)
STRVBL	\$03A	D	W(d)	dito
STRHOR	\$03C	D	W(d)	dito
STRLONG	\$03E	DP	W(d)	dito
BLTCON0	\$040	A	W(pc*)	Blitter-Kontrollregister 0
BLTCON1	\$042	A	W(pc*)	Blitter-Kontrollregister 1
BLTAFWM	\$044	A	W(pc*)	Maskenregister für das erste Datenwort von A
BLTALWM	\$046	A	W(pc*)	Maskenregister für das letzte Datenwort von A
BLTCPTH	\$048	A	W(pc*)	Adreßregister (High) für Blitterkanal C
BLTCPTL	\$04A	A	W(pc*)	Adreßregister (Low) für Blitterkanal C
BLTBPTH	\$04C	A	W(pc*)	dito für Blitterkanal B
BLTBPTL	\$04E	A	W(pc*)	dito
BLTAPTH	\$050	A	W(pc*)	dito
BLTAPTL	\$052	A	W(pc*)	dito
BLTDPTH	\$054	A	W(pc*)	dito
BLTDPTL	\$056	A	W(pc*)	dito
BLTSIZE	\$058	A	W(pc*)	StartBlit und Größe des Blitterfensters
xxxx	\$05A	—	—	Zugriffsadresse nicht belegt
xxxx	\$05C	—	—	nicht belegt
xxxx	\$05E	—	—	nicht belegt
BLTCMOD	\$060	A	W(pc*)	Blitter-Modulo für Quelldaten C
BLTBMOD	\$062	A	W(pc*)	Blitter-Modulo für Quelldaten B
BLTAMOD	\$064	A	W(pc*)	dito für Quelldaten A
BLTDMOD	\$066	A	W(pc*)	dito
xxxx	\$068	—	—	nicht belegt
xxxx	\$06A	—	—	nicht belegt
xxxx	\$06C	—	—	nicht belegt
xxxx	\$06E	—	—	nicht belegt
BLTCDAT	\$070	A	W(d)	Blitter-DMA-Datenregister (Kanal C)
BLTBDAT	\$072	A	W(d)	Blitter-DMA-Datenregister (Kanal B)
BLTADAT	\$074	A	W(d)	Blitter-DMA-Datenregister (Kanal A)
xxxx	\$076	—	—	Zugriffsadresse nicht belegt
xxxx	\$078	—	—	nicht belegt

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
xxxx	\$07A	–	–	nicht belegt
xxxx	\$07C	–	–	nicht belegt
DSKSYNC	\$07E	P	W(pc*)	Synchronisationsmaske des Floppy-Controllers
COP1LCH	\$080	A	W(pc)	Adreßregister (High) für CopperList 1
COP1LCL	\$082	A	W(pc)	Adreßregister (Low) für CopperList 1
COP2LCH	\$084	A	W(pc)	dito für CopperList 2
COP2LCL	\$086	A	W(pc)	dito
COPJMP1	\$088	A	W(pc)	Strobe-Register; initialisiert den Copper- Programmzähler mit der ersten Befehls-Liste
COPJMP2	\$08A	A	W(pc)	Strobe-Register; initialisiert den Copper- Programmzähler mit der zweiten Befehls-Liste
COPINS	\$08C	A	W(pc)	Befehlsregister des Copper
DIWSTRT	\$08E	A	W(pc)	Linke, obere Ecke des Grafik-Displays
DIWSTOP	\$090	A	W(pc)	Rechte, untere Ecke des Grafik-Displays
DDFSTRT	\$092	A	W(pc)	Beginn der Bitplane-DMA (X-Koordinate)
DDFSTOP	\$094	A	W(pc)	Ende der Bitplane-DMA (Y-Koordinate)
DMACON	\$096	ADP	W(pc)	DMA-Kontrollregister
CLXCON	\$098	D	W(pc)	Kontrollregister (Grafikobjekt-Kollisionen)
INTENA	\$09A	P	W(pc)	Interrupt-Kontrollregister
INTREQ	\$09C	P	W(pc)	Interrupt-Anforderungen
ADKCON	\$09E	P	W(pc)	Audio-/Disk-Kontrollregister
AUD0LCH	\$0A0	A	W(pc)	Adreßregister (High) für Audiokanal 0
AUD0LCL	\$0A2	A	W(pc)	Adreßregister (Low) für Audiokanal 0
AUD0LEN	\$0A4	P	W(pc)	Sample-Länge Audiokanal 0
AUD0PER	\$0A6	P	W(pc)	Abtastfrequenz Kanal 0
AUD0VOL	\$0A8	P	W(pc)	Lautstärkeregelung
AUD0DAT	\$0AA	P	W(dpc)	DMA-Datenregister für Audiokanal 0
xxxx	\$0AC	–	–	nicht belegt
xxxx	\$0AE	–	–	nicht belegt

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
AUD1LCH	\$0B0	A	W(pc)	siehe Audiokanal 0
AUD1LCL	\$0B2	A	W(pc)	dito
AUD1LEN	\$0B4	P	W(pc)	dito
AUD1PER	\$0B6	P	W(pc)	dito
AUD1VOL	\$0B8	P	W(pc)	dito
AUD1DAT	\$0BA	P	W(dpc)	dito
xxxx	\$0BC	–	–	nicht belegt
xxxx	\$0BE	–	–	nicht belegt
AUD2LCH	\$0C0	A	W(pc)	siehe Audiokanal 0
AUD2LCL	\$0C2	A	W(pc)	dito
AUD2LEN	\$0C4	P	W(pc)	dito
AUD2PER	\$0C6	P	W(pc)	dito
AUD2VOL	\$0C8	P	W(pc)	dito
AUD2DAT	\$0CA	P	W(dpc)	dito
xxxx	\$0CC	–	–	nicht belegt
xxxx	\$0CE	–	–	nicht belegt
AUD3LCH	\$0D0	A	W(pc)	siehe Audiokanal 0
AUD3LCL	\$0D2	A	W(pc)	dito
AUD3LEN	\$0D4	P	W(pc)	dito
AUD3PER	\$0D6	P	W(pc)	dito
AUD3VOL	\$0D8	P	W(pc)	dito
AUD3DAT	\$0DA	P	W(dpc)	dito
xxxx	\$0DC	–	–	Zugriffsadresse nicht belegt
xxxx	\$0DE	–	–	nicht belegt
BPL1PTH	\$0E0	A	W(pc)	Adreßregister (High) für Bitplane 1
BPL1PTL	\$0E2	A	W(pc)	Adreßregister (Low) für Bitplane 1
BPL2PTH	\$0E4	A	W(pc)	dito für Bitplane 2
BPL2PTL	\$0E6	A	W(pc)	dito
BPL3PTH	\$0E8	A	W(pc)	dito
BPL3PTL	\$0EA	A	W(pc)	dito
BPL4PTH	\$0EC	A	W(pc)	dito
BPL4PTL	\$0EE	A	W(pc)	dito
BPL5PTH	\$0F0	A	W(cp)	dito
BPL5PTL	\$0F2	A	W(pc)	dito

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
BPL6PTH	\$0F4	A	W(pc)	dito
BPL6PTL	\$0F6	A	W(pc)	dito
xxxx	\$0F8	–	–	Zugriffsadresse nicht belegt
xxxx	\$0FA	–	–	nicht belegt
xxxx	\$0FC	–	–	nicht belegt
xxxx	\$0FE	–	–	nicht belegt
BPLCON0	\$100	AD	W(pc)	Bitplane-Kontrollregister 0 (Auflösung,Darstellungsmodus)
BPLCON1	\$102	D	W(pc)	Kontrollregister 1
BPLCON2	\$104	D	W(pc)	Kontrollregister 2
xxxx	\$106	–	–	nicht belegt
BPL1MOD	\$108	A	W(pc)	Modulo-Wert für ungerade Bitplanes
BPL2MOD	\$10A	A	W(pc)	Modulo-Wert für gerade Bitplanes
xxxx	\$10C	–	–	nicht belegt
xxxx	\$10E	–	–	nicht belegt
BPL1DAT	\$110	D	W(d)	DMA-Datenregister für Bitplane 1
BPL2DAT	\$112	D	W(d)	dito für Bitplane 2
BPL3DAT	\$114	D	W(d)	dito
BPL4DAT	\$116	D	W(d)	dito
BPL5DAT	\$118	D	W(d)	dito
BPL6DAT	\$11A	D	W(d)	dito
xxxx	\$11C	–	–	nicht belegt
xxxx	\$11E	–	–	nicht belegt
SPR0PTH	\$120	A	W(pc)	Adreßregister (High) für Spritedaten (Sprite Nr. 0)
SPR0PTL	\$122	A	W(pc)	Adreßregister (Low) für Spritedaten (Nr. 0)
SPR1PTH	\$124	A	W(pc)	dito für Sprite 1
SPR1PTL	\$126	A	W(pc)	dito
SPR2PTH	\$128	A	W(pc)	dito
SPR2PTL	\$12A	A	W(pc)	dito
SPR3PTH	\$12C	A	W(pc)	dito
SPR3PTL	\$12E	A	W(pc)	dito

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
SPR4PTH	\$130	A	W(pc)	dito
SPR4PTL	\$132	A	W(pc)	dito
SPR5PTH	\$134	A	W(pc)	dito
SPR5PTL	\$136	A	W(pc)	dito
SPR6PTH	\$138	A	W(pc)	dito
SPR6PTL	\$13A	A	W(pc)	dito
SPR7PTH	\$13C	A	W(pc)	dito
SPR7PTL	\$13E	A	W(pc)	dito
SPR0POS	\$140	AD	W(dpc)	Vert. und horiz. Startposition Sprite 0
SPR0CTL	\$142	AD	W(dpc)	Kontrollregister für Sprite 0
SPR0DATA	\$144	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 0)
SPR0DATB	\$146	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 0)
SPR1POS	\$148	AD	W(dpc)	Vert. und horiz. Startposition Sprite 1
SPR1CTL	\$14A	AD	W(dpc)	Kontrollregister für Sprite 1
SPR1DATA	\$14C	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 1)
SPR1DATB	\$14E	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 1)
SPR2POS	\$150	AD	W(dpc)	Vert. und horiz. Startposition Sprite 2
SPR2CTL	\$152	AD	W(dpc)	Kontrollregister für Sprite 2
SPR2DATA	\$154	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 2)
SPR2DATB	\$156	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 2)
SPR3POS	\$158	AD	W(dpc)	Vert. und horiz. Startposition Sprite 3
SPR3CTL	\$15A	AD	W(dpc)	Kontrollregister für Sprite 3
SPR3DATA	\$15C	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 3)
SPR3DATB	\$15E	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 3)
SPR4POS	\$160	AD	W(dpc)	Vert. und horiz. Startposition Sprite 4
SPR4CTL	\$162	AD	W(dpc)	Kontrollregister für Sprite 4
SPR4DATA	\$164	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 4)
SPR4DATB	\$166	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 4)
SPR5POS	\$168	AD	W(dpc)	Vert. und horiz. Startposition Sprite 5
SPR5CTL	\$16A	AD	W(dpc)	Kontrollregister für Sprite 5
SPR5DATA	\$16C	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 5)
SPR5DATB	\$16E	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 5)

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
SPR6POS	\$170	AD	W(dpc)	Vert. und horiz. Startposition Sprite 6
SPR6CTL	\$172	AD	W(dpc)	Kontrollregister für Sprite 6
SPR6DATA	\$174	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 6)
SPR6DATB	\$176	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 6)
SPR7POS	\$178	AD	W(dpc)	Vert. und horiz. Startposition Sprite 7
SPR7CTL	\$17A	AD	W(dpc)	Kontrollregister für Sprite 7
SPR7DATA	\$17C	D	W(dpc)	1. Sprite-DMA-Datenregister (Sprite 7)
SPR7DATB	\$17E	D	W(dpc)	2. Sprite-DMA-Datenregister (Sprite 7)
COLOR00	\$180	D	W(pc)	Farbregister 0 (Startadresse des Color-Lookup-Table)
COLOR01	\$182	D	W(pc)	Farbregister 1
COLOR02	\$184	D	W(pc)	Farbregister 2
COLOR03	\$186	D	W(pc)	Farbregister 3
COLOR04	\$188	D	W(pc)	Farbregister 4
COLOR05	\$18A	D	W(pc)	Farbregister 5
COLOR06	\$18C	D	W(pc)	Farbregister 6
COLOR07	\$18E	D	W(pc)	Farbregister 7
COLOR08	\$190	D	W(pc)	Farbregister 8
COLOR09	\$192	D	W(pc)	Farbregister 9
COLOR10	\$194	D	W(pc)	Farbregister 10
COLOR11	\$196	D	W(pc)	Farbregister 11
COLOR12	\$198	D	W(pc)	Farbregister 12
COLOR13	\$19A	D	W(pc)	Farbregister 13
COLOR14	\$19C	D	W(pc)	Farbregister 14
COLOR15	\$19E	D	W(pc)	Farbregister 15
COLOR16	\$1A0	D	W(pc)	Farbregister 16
COLOR17	\$1A2	D	W(pc)	Farbregister 17
COLOR18	\$1A4	D	W(pc)	Farbregister 18
COLOR19	\$1A6	D	W(pc)	Farbregister 19
COLOR20	\$1A8	D	W(pc)	Farbregister 20
COLOR21	\$1AA	D	W(pc)	Farbregister 21
COLOR22	\$1AC	D	W(pc)	Farbregister 22
COLOR23	\$1AE	D	W(pc)	Farbregister 23
COLOR24	\$1B0	D	W(pc)	Farbregister 24
COLOR25	\$1B2	D	W(pc)	Farbregister 25
COLOR26	\$1B4	D	W(pc)	Farbregister 26
COLOR27	\$1B6	D	W(pc)	Farbregister 27
COLOR28	\$1B8	D	W(pc)	Farbregister 28

<i>Label</i>	<i>Offset</i>	<i>Chip</i>	<i>Zugriff</i>	<i>Funktionsbeschreibung</i>
COLOR29	\$1BA	D	W(pc)	Farbregister 29
COLOR30	\$1BC	D	W(pc)	Farbregister 30
COLOR31	\$1BE	D	W(pc)	Farbregister 31
xxxx	\$1C0	–	–	nicht belegt
xxxx	\$1FE	–	–	nicht belegt

Fat Agnus – der Herr der Busse

Unter den ICs auf der Amiga-Platine hält dieser Baustein gleich mehrere Rekorde:

Er ist mit über 80000 Transistorfunktionen der komplexeste Chip (selbst ein MC68000 kann da nicht mehr mithalten), verfügt mit 84 Pins über die meisten Anschlüsse und stellt mit 97 Registern (exklusive der für den Programmierer nicht zugänglichen Register) auch den größten Registersatz. Seine zentrale Stellung im System zeigt sich bereits an der Fülle unterschiedlicher Aufgaben; dieser Chip ist ein wahres Multitalent:

Als DMA-Controller verwaltet er sechs verschiedene DMA-Quellen mit insgesamt über 25 Kanälen, als DRAM-Controller übernimmt er das Multiplexen der CPU-Adressen, die Refreshsteuerung und Generierung der Zeilen- bzw. Spalten-Adreßimpulse für bis zu 1 MByte RAM-Speicher (DRAM-Typ 256K x 1), um schließlich in der Rolle des Taktgebers aus einem 28,38-Mhz-Mastertakt (gilt für die PAL-Version, NTSC hat 28,64 Mhz) sämtliche Systemtakte zu erzeugen!

Damit sind seine Fähigkeiten allerdings noch lange nicht erschöpft, die besonderen Qualitäten kommen erst auf dem Grafiksektor zur Geltung:

Zwei Stichworte werden immer wieder mit den überragenden Grafikfähigkeiten des Amigas assoziiert – Copper und Blitter – beides Bestandteile von Fat Agnus.

Der Copper (den Spitznamen verdankt er seinen umfangreichen Kontroll- und Steuerungsaufgaben im System), ein Spezialprozessor mit RISC-ähnlicher Struktur (RISC = Reduced-Instruction-Set-Computer; Archimedes läßt grüßen), erlaubt die Manipulation eines Customchip-Registers in Abhängigkeit von der Rasterstrahlposition.

Endlich ist der Grafik-Programmierer nicht mehr ausschließlich auf die Blanking-Interrupts am Beginn einer vertikalen Austastlücke angewiesen, sondern kann die Parameter

einer Grafikdarstellung praktisch an jeder beliebigen Bildschirmposition verändern. Viele Spezial-Effekte wie beispielsweise das Mischen verschiedener Grafikauflösungen, lokal begrenzte Änderungen der Farbpalette oder superweiches Scrolling lassen sich dank Copper-Hardware problemlos realisieren.

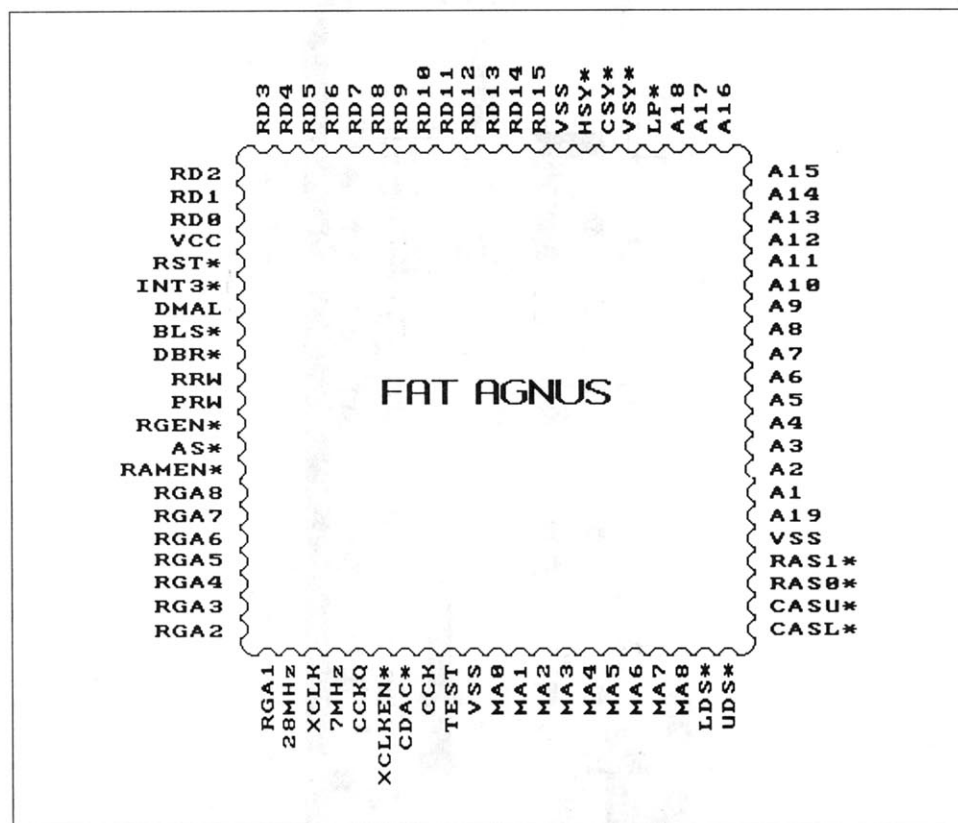


Abb. 3.3: Pinbelegung Fat Agnus

Der Blitter (ein Zusammenschchnitt aus der Bezeichnung Block-Image-Transferrer = Blockgrafik-Kopierer) ist in der Lage, rechteckig-organisierte Speicherdaten (sogenannte Blöcke) mit hoher Geschwindigkeit zu manipulieren. Sie werden über drei DMA-Kanäle eingelesen, logisch miteinander verknüpft und mittels eines vierten Kanals an die Zieladresse geschrieben.

Parallel dazu verschiebt ein integrierter Barrelshifter die 16-Bit-Datenworte um 1 bis 15 Bits (seine Aktivitäten haben keinen Einfluß auf die Geschwindigkeit der Blitteroperationen). Zwei besondere Modi für schnelles Linienziehen und das Füllen von Flächen (beide übertreffen die Möglichkeiten des 68000 um ein Vielfaches), runden das Leistungsspektrum dieser hochinteressanten Schaltung ab.

Die Bobs (Blitter-Objects, eine Spezialität des Animationssystems), die in puncto Größe und Farbauswahl die engen Grenzen der Sprite-Hardware sprengen, sind ohne den Blitter – als reine Software-Emulation – nicht denkbar. Obwohl in erster Linie für schnelle Grafikmanipulationen entwickelt, beschränkt sich das Einsatzspektrum des Blitters nicht nur auf diesen Sektor – überall dort, wo große Datenmengen anfallen, wird er schnell zu einem unentbehrlichen Hilfsmittel.

Da der Blitter von Anfang an in das Hard- und Softwarekonzept des Amigas eingebunden war, profitieren auch die unterschiedlichsten Systemkomponenten von seiner Existenz. Seien es die komplexen Refreshmechanismen der Intuition-Benutzeroberfläche, die Aufbereitung der codierten Lesedaten vom Diskettenlaufwerk oder der blitzschnelle Zugriff auf die RAM-Disk – der Blitter ist dabei.

Fat Agnus, der DMA-Controller

Im Blockdiagramm des Customchips ist der prinzipielle Aufbau des DMA-Systems deutlich zu erkennen.

Jede der sechs DMA-Quellen verfügt über eine eigene Kontrollogik, der jeweils eine Reihe von Registern zugeordnet ist. Sie wertet die Informationen der Kontrollregister aus (BPLCON0 = BitPLaneCONTROL beispielsweise steuert Bildschirmauflösung und Darstellungsmodus – je nach Einstellung schwankt das Datenvolumen erheblich), ermittelt den individuellen Datenbedarf eines DMA-Kanals und regelt den internen Ablauf des Datenzugriffs.

Die Pfeile im Blockdiagramm 3.4 symbolisieren Verbindungen mit den Adreß-Generatoren; die Nummern entsprechen der Anzahl verwalteter Kanäle.

Die formale Abwicklung eines DMA-Transfers ist bereits im ersten Kapitel kurz angerissen worden. Ein DMA-Zugriff erfolgt durch Adressierung eines Speicherbereichs im CHIP-RAM und Aktivierung des Zielregisters in der Customchip-Hardware. Sobald die Daten auf dem Bus erscheinen, wandern sie automatisch in ihr Bestimmungsregister.

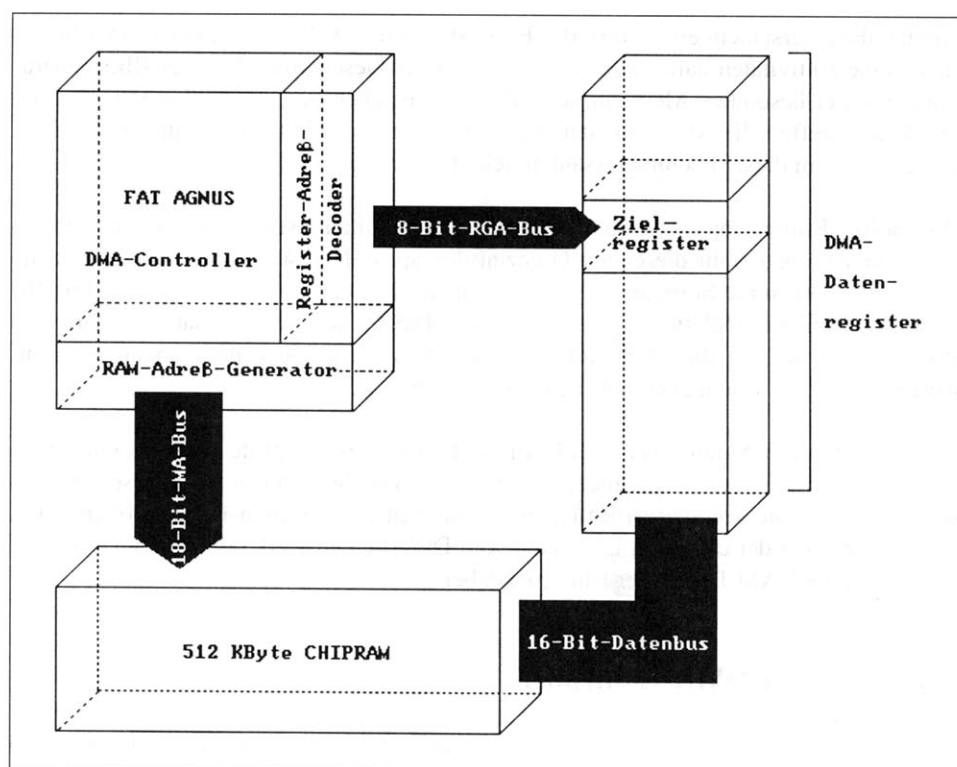


Abb. 3.5: Das Prinzip des DMA-Zugriffs

Dazu besitzt Fat Agnus einen RAM-Adreßgenerator (vergleichbar mit dem Address-Unit (kurz AU) eines Mikroprozessors); komplett mit Adreß- und Backupregistern (Speicherung des Adreßinhalts für zyklisch wiederkehrende DMA-Zugriffe, siehe Audiokanäle), Jump-Registern für Adreßsprünge und einem 18-Bit-Addierer zur Adreßinkrementierung (Hochzählen der Adreßregister).

Der Register-Adreß-Encoder erzeugt die Zieladresse eines Registers der Customchip-Hardware.

Als einfache Kodierschaltung mit PLA-Struktur (Programmable-Logic-Array) ordnet er jedem DMA-Kanal sein Datenregister zu und legt die Adresse auf den Register-Adreßbus. Man kann seine Arbeitsweise gut mit der Auswertung einer Tabelle vergleichen, in der die Lage der korrespondierenden DMA-Datenregister vermerkt ist.

Die Register-Adreß-Decoder der Customchips kehren den Zuordnungsprozeß um, entschlüsseln die Adreßinformation auf dem Register-Adreß-Bus (kurz RGA) und aktivieren das Zielregister. Diese Aktivierung läuft immer nach dem gleichen Schema ab:

Alle für den Programmierer sichtbaren Register sind über eine Schalterstufe mit dem internen 16-Bit-Datenbus verbunden. Die dünnen Pfeile vom Register-Adreß-Decoder zu den Funktionsblöcken (Abbildung 3.4) symbolisieren die Steuerleitungen, mit denen diese Schalter bedient werden. Normalerweise sind sie geschlossen, trennen also die Register vom Bussystem.

Liegt allerdings eine gültige Registeradresse auf dem RGA-Bus, öffnet die zugehörige Steuerleitung den Trennschalter, und die Daten gelangen in das Zielregister.

Im Zentrum des DMA-Konzepts steht die Prioritäts-Kontrollogik (= Priority-Control-Logic). Sie übernimmt die Hauptlast der Verwaltungsarbeit, sortiert die eintreffenden DMA-Anforderungen nach ihrer Priorität und weist den Kanälen freie Timeslots für den Zugriff auf das CHIP-RAM zu.

Das Timing orientiert sich an dem Verlauf einer PAL-Rasterzeile mit 225 potentiellen DMA-Zyklen (deshalb ist die Kontrollogik auch mit den Synchronisationszählern VPOS und VHPOS verbunden, kennt also jederzeit die aktuelle Bildschirmposition des Rasterstrahls).

Intensives Pipelining erlaubt eine schnelle Verarbeitung der eintreffenden DMA-Anforderungen und garantiert kürzeste Reaktionszeiten. Um die Buszugriffslogik ständig über die Aktivitäten des DMA-Controllers zu informieren, signalisiert die DBR*-Leitung (Data Bus Request = Datenbusanforderung), daß Fat Agnus die Busse im nächsten Zyklus übernehmen wird. So kann der Prozessor rechtzeitig vom CHIP-RAM-Bussystem abgetrennt werden.

Der "schlanke" Vorgänger – Agnus

Programmtechnisch besteht zwischen beiden Agnus-Chips kein Unterschied – effektiver Datendurchsatz, Copper- und Blitterfunktionen sind gleich geblieben. Schließlich beziehen sich die Erweiterungen von Fat Agnus ausschließlich auf die Integration wichtiger Systemfunktionen wie Takterzeugung oder Refreshsteuerung und sind damit ein wichtiger Faktor in der Reduzierung der Produktionskosten.

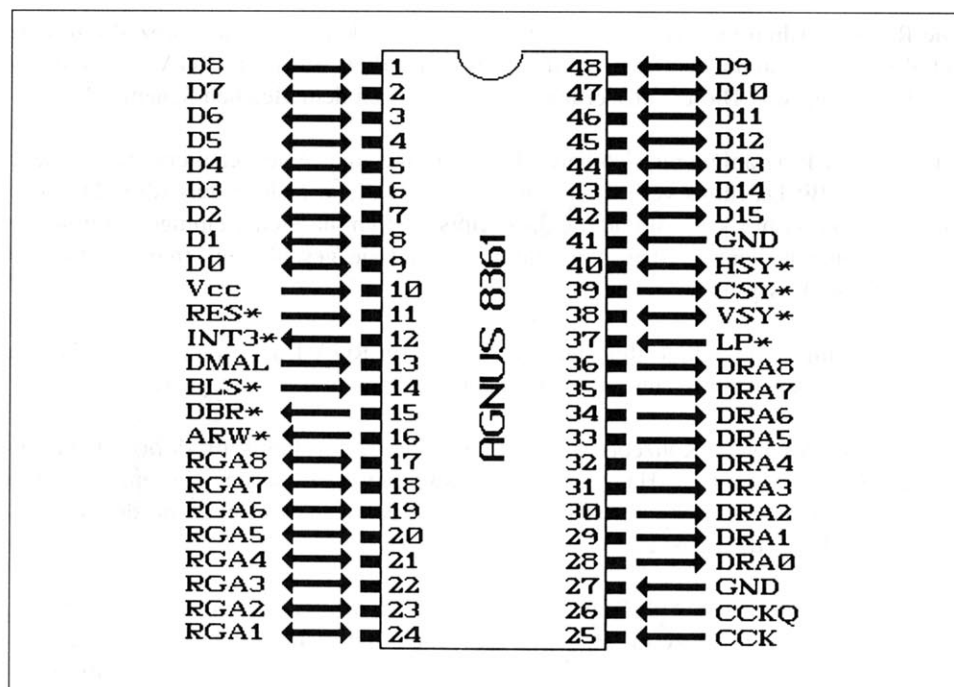


Abb. 3.6: Pinbelegung Agnus

Im Pin-Belegungsschema des schlanken Vorgängers tauchen viele bekannte Leitungen auf; lediglich das ARW-Signal (Agnus Read/Write) scheint aus dem Rahmen zu fallen. Es ist aber – leichte Namensverwirrung – mit der RRW-Leitung identisch und kennzeichnet die Richtung eines CHIP-RAM-Zugriffs. Unterschiede bestehen allerdings in der Schaltung des RGA-Busses. Agnus verfügt über keine speziellen Adreßanschlüsse für den Prozessor, sondern wickelt auch CPU-Zugriffe (auf die Customchip-Register) über den Register-Adreßbus ab.

Der Bus ist bidirektional; liegt eine gültige Registeradresse an (CPU-Adreßleitungen A1 bis A8), wird die Adreßinformation gepuffert und (falls das betreffende Register nicht in Agnus liegt) auf dem gleichen Bussystem an die anderen Customchips übermittelt.

Erläuterungen zur Pinbelegung von Fat Agnus:

Pin Pin-Nr. Eing./Ausg. Funktionsbeschreibung

A1-A19	59-77	E	<p>Adreßbus der CPU. Zur Adressierung der Customchip-Register verwendet der Prozessor die Leitungen A1 bis A8; mit der vollen Busbreite greift er auf das CHIP-RAM zu. Fat Agnus übernimmt das Adreß-Multiplexing, die Generierung der RAS*- bzw. CAS*-Signale und die komplette Refresh-Steuerung für insgesamt 1 MByte dynamischen RAM-Speicher. Der Zustand der Adreßleitung A19 dient zur Unterscheidung zwischen den unteren 512 KByte an reinem CHIP-RAM (A19 = 0) und den oberen 512 KByte, das nur die CPU ansprechen kann (A19 = 1). Auf der System-Platine lenkt ein Jumper wahlweise A23 bzw. A19 vom Prozessorbus auf den A19-Anschluß des Customchips um (in der normalen Konfiguration liegt hier A23). Diese Einstellung ist für die korrekte Adressierung der oberen Speicherbank (ab \$C00000) notwendig. Der von Fat Agnus verwaltete RAM-Speicher erscheint also normalerweise in zwei getrennten 512KByte-Blöcken, die bei Bedarf (durch Umschaltung des Jumpers) zusammengelegt werden können. Das ist in der neuesten Generation des B2000 mit 1-MByte-CHIP-RAM (neuer Fat Agnus) geschehen. Die Umrüstung auf die aktuelle Customchip-Generation ist dank dieser Technik kein Problem.</p>
RD15- RD0	1-14/ 83-84/	E/A	<p>Bilddirektionaler Datenbus (intern gepuffert) für die Einbindung in die 16-Bit-Busarchitektur des Systems.</p>
AS*	24	E	<p>Address Strobe (= Adreßimpuls) der CPU; kennzeichnet das Anliegen einer gültigen Adresse auf dem Prozessor-Adreßbus.</p>

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
REGEN*	23	E	Register Enable (= Registerzugriff erlauben); wird von der Adreßdecodierung (Gary im A500, B2000) erzeugt und signalisiert einen Prozessorzugriff auf die Register der Customchip-Hardware (A1 bis A8 führen die Registeradresse, AS* ist Low).
RAMEN*	25	E	RAM Enable (= RAM-Zugriff erlauben). Steuerleitung der Adreßdecodierung. Bei Low-Pegel interpretiert, Fat Agnus die Adreßinformation auf den Leitungen A0 bis A19 als gültige Speicheradresse.
PRW	22	E	Processor Read Write (= Schreib-/Leseleitung des Prozessors); markiert die Richtung eines Prozessorzugriff (Low = Schreiben / High = Lesen).
RRW	21	A	Reflektiert den Zustand der PRW-Leitung bzw. signalisiert die Richtung eines CHIP-RAM-Zugriffs von Fat Agnus.
MA0-MA8	43-51	A	Memory-Address (= Speicheradresse). Gemultiplexer Speicher-Adreßbus für den direkten Anschluß von 256K x 1-DRAMs. Die vollständige Adreßinformation (18 Bit) wird in zwei Schüben à 9 Bit in die Zugriffslogik der RAM-Chips geschoben.
LDS*	52	E	Lower Data Strobe; wird von der CPU aktiviert und markiert eine Datenübertragung auf dem unteren Teil des Datenbusses (D0...D7). Fat Agnus benötigt diese Information, um die korrespondierende Speicherbank zu selektieren (CASL* geht auf Low).
UDS*	53	E	Upper Data Strobe des Prozessors für die oberen Datenleitungen (D8...D15). Beim Zugriff auf die Speicherchips wählt Fat Agnus die obere Speicherbank mit der CASU*-Leitung.
CASL*	54	A	Column Address Strobe-Low (= Spalten-Adreßimpuls, Low-Byte). Der Adreßbus dynamischer RAM-Chips arbeitet zweiphasig.

Pin Pin-Nr. Eing./Ausg. Funktionsbeschreibung

Dabei bestimmen die Zeilen- (RAS*) und Spalten-Strobes (CAS*) das Adreßtiming und signalisieren das Anliegen einer gültigen Adreßhälfte. Gleichzeitig dienen diese Leitungen als Chipselect für die Auswahl der korrespondierenden Speicherbank. Mit CASL* spricht Fat Agnus die Speicherchips auf dem Low-Byte des Datenbusses an.

CASU* 55 A Column Address Strobe-Upper (= Spalten-Adreßimpuls, High-Byte). Entsprechendes für die obere Hälfte des Datenbusses (D15 bis D8).

RAS0* 57 A Row Address Strobe-0 (= Zeilen-Adreßimpuls 0). Sobald die Spaltenadresse für die unteren 512 KByte RAM (CHIP-RAM ab Adresse \$0) auf dem MA-Bus erscheint, geht RAS0* auf Low-Pegel.

RAS1* 56 A Row Address Strobe-1 (= Zeilen-Adreßimpuls 1). Diese Leitung wird nur bei einem CPU-Zugriff auf die höheren 512 KByte RAM (ab \$C00000) aktiv (-> Adreßeingang A19 = 1).

(Mehr über die Verwaltung der Speicher-Chips in Kapitel 5)

DBR* 20 A Data Bus Request (= Datenbus-Anforderung); bei Low-Pegel signalisiert Fat Agnus, daß er das Bussystem im nächsten Zyklus übernehmen wird. Auf diese Weise ist die Buszugriffslogik ständig über die aktuellen Busaktivitäten informiert.

RGA8 26-33 A Unidirektionaler 8-Bit-Register-Address-Bus –
-RGA1 RGA1 (kurz: RGA); führt die Registeradressen der Customchip-Hardware. Während eines Prozessorzugriffs schaltet Fat Agnus die Adreß-Leitungen A1 bis A8 auf den RGA-Bus durch; spricht er die externen Customchip-Register selbst an, legt er die vom Register-Adreß-Decoder erzeugte Tabellenposition auf den RGA-Bus.

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
			Die Decoder der Customchips entschlüsseln die Adreßinformationen und aktivieren das Zielregister.
HSY*	81	E/A	Horizontal SYNchronisation (= Horizontaler- bzw. Zeilen-Synchronisationsimpuls) kennzeichnet den Beginn einer neuen Rasterzeile. Die HSY*- und VSY*- Synchronisationsimpulse werden von den Rasterzeilen- bzw. Spaltenzählern im Fat-Agnus-Chip erzeugt und entsprechen der jeweiligen Fernsehnorm (PAL, NTSC). Beide sind bidirektional und erlauben im Eingabemodus die Steuerung der internen Zähler durch ein externes Signal. (Hardwaremäßige Voraussetzungen für das Mischen von Computergrafik mit einer externen Videoquelle (Genlocking) schon in der Grundausstattung!)
VSY*	79	E/A	Vertical SYNchronisation (= Vertikaler- bzw. Bild-Synchronisationsimpuls) signalisiert den Beginn eines neuen Videobildes. (Funktion und Realisierung wie HSY*).
CSY*	80	A	Composite SYNchronisation (Mischsignal aus HSY* und VSY*); wird von der Video-Hybrid zur Generierung des (F)BAS-Videosignals benötigt.
LP*	78	E	Light Pen (= Lichtgriffel). Sobald der Elektronenstrahl die lichtempfindliche Spitze des Lichtgriffels trifft, geht der LP*-Eingang auf Low und Fat Agnus friert die Strahlpositionszähler VPOS bzw. VHPOS (Offset \$004, \$006) ein. Ihr Wert bleibt jetzt bis zum Ende der vertikalen Austastlücke erhalten (Zeile 26), so daß sie innerhalb des Vertical-Blanking-Interrupts bequem ausgelesen werden können.
RST*	18	E	RESET-Leitung, setzt den Baustein in einen definierten Ausgangszustand zurück.
INT3*	17	A	Interrupt Level 3 (korrespondiert mit dem Unterbrechungsniveau der Ebene 3); direkt mit dem gleichnamigen Eingang von Paula verbunden.

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
			Signalisiert Unterbrechungsanforderungen des Blitters (Blitter finished = Blitteroperation beendet).
DMAL	18	E	DMA-Anforderung eines externen Bausteins. Die Leitung informiert Fat Agnus über den DMA-Status der Audio- bzw. Disk-Hardware (des Customchips Paula). Im Gegensatz zu den anderen DMA-Quellen kann der Controller bei diesen Komponentenden den Datenbedarf nicht vorausbestimmen!
BLS*	19	E	Blitter Slow Down (= Blitter verlangsamen); wird von der Buszugriffslogik erzeugt und zwingt den Blitter, die Busse für einen Buszyklus freizugeben. Ohne die BLS*-Leitung würde die CPU im Extremfall vollständig vom CHIP-RAM abgeschnitten, da die Blitteraktivitäten immer Vorrang haben. Um diese Situation zu verhindern, werden die DMA-Zugriffe überwacht; wartet die CPU bereits seit drei Buszyklen, geht BLS* auf Low. Der Blitter unterbricht seine Arbeit und überläßt den folgenden Buszyklus dem Prozessor.
28MHz	34	E	Haupttakt (PAL-Version = 28,375 MHz, NTSC = 28,636 MHz). Aus diesem Impuls generiert Fat Agnus sämtliche Systemtakte (C1, C4, 7M etc.).
XCLK	35	E	Extern-CLoCK (= Externer Takt). Alternativer Takteingang, erlaubt die Einspeisung eines externen Taktimpulses. Auf diese Weise kann das System wahlweise beschleunigt oder abgebremst werden. Einer beliebigen Erhöhung der Taktfrequenz sind hardwaremäßig enge Grenzen gesetzt; mehr als 40 MHz vertragen die Customchips in keinem Fall! Außerdem driftet der Bildaufbau des Amiga-Monitor schon bei geringen Abweichungen vom 28MHz-Takt aus der Synchronisation, gleiches gilt für den Disk-Controller.
CCK	40	A	Color-CloCk (= Farb-Takt, Frequenz: 3,55 MHz).

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
XCLKEN*	36	E	External-CLoCK-ENable (= Externer Takt). Bei Low-Pegel wird 28-MHz-Takt blockiert, der am XCLK-Eingang anliegende Impuls wird durchgeschaltet.
CCKQ	39	A	Color-CloCk-Quadrature (Frequenz wie CCK, Phasenlage um 90° verschoben).
7MHz	38	A	CPU-Takt (PAL = 7,09 MHz, NTSC = 7,16 MHz).
CDAC*	37	A	7-Mhz-Takt invertiert und 90° verschoben.
TEST	41	E	unterbricht aktuellen CPU-Zyklus, verzweigt in Testmodus.

Denise – mehr als nur ein Videoshifter

Denise ist der Grafikspezialist unter den Amiga-Customchips. Seine Hauptaufgabe liegt in der Generierung der Videodarstellung; zusätzlich enthält er noch die Sprite-Hardware, die Kollisionserkennung des Animationssystems und die Mauspositions-Zähler.

Eine wesentliche Grundlage der herausragenden Grafikeigenschaften des Amigas ist die Technik der Grafikdatenorganisation. Denise verwendet keine Pixel-Adressierung, sondern ein als Bitplane-Adressierung bekanntes Verfahren.

Pixel-Adressierung contra Bitplane-Adressierung

Bei der ersten Methode enthält bereits das Datum eines Grafikpunktes (Pixels) die binärcodierte Nummer des zugehörigen Farbregisters im Color-Lookup-Table (= Farbtabelle). Die Grafikdaten bilden einen kompakten Block und liegen als zusammenhängende Kette im Speicher; der Zugriff auf einzelne Pixels dieser großen Bitmap ist aufwendig, die Modifikation kompliziert und zeitintensiv.

Ganz anders geht die Bitplane-Adressierung vor. Hier sind die Pixels zu Bitplanes mit einer Tiefe von einem Bit organisiert (jedes Plane entspricht einer Art Monochromabbildung), die jeweils eine Binärstelle der Registernummer enthalten. Die Bitplanes sind unabhängig; erst im Moment der Videogenerierung werden die korrespondierenden Pixelbits zu einem Zeiger auf die Farbtabelle zusammengefaßt. Dadurch bietet die Bit-

plane-Adressierung neben einer gesteigerten Speichereffizienz für Farbcodes mit ungeradzahlgiger Bitbreite auch größere Flexibilität in der Grafikprogrammierung, da die einzelnen Bitplanes individuell manipuliert und untereinander beliebig kombiniert werden können.

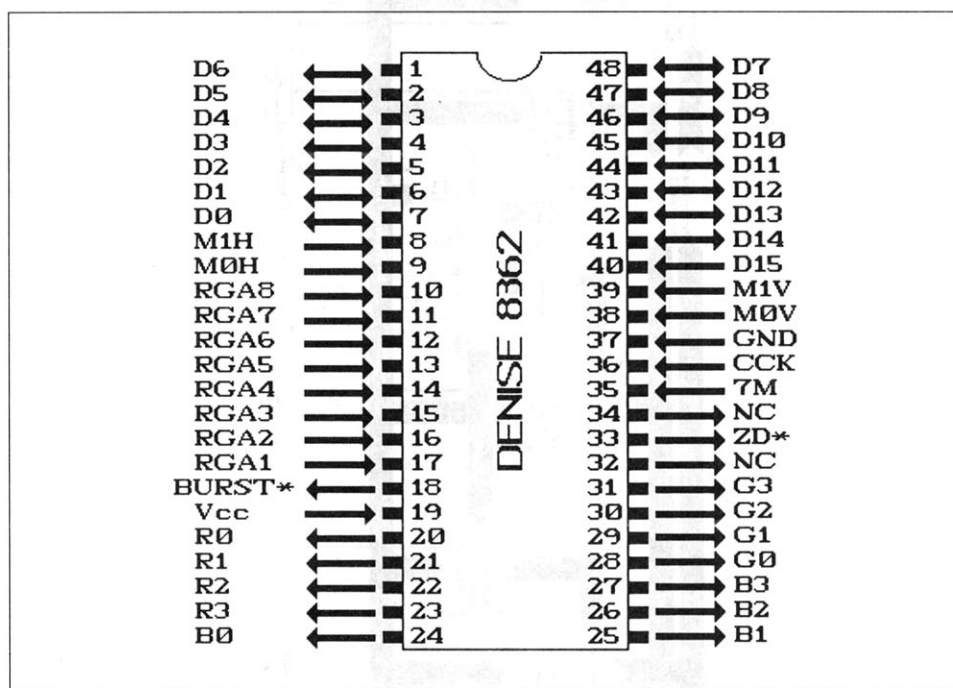


Abb. 3.7: Pinbelegung Denise

Dieser innovativen Datenorganisation verdankt der Amiga auch den effektvollen Dual-Playfield-Modus, bei dem die aktiven Bitplanes zwei, sich überlagernde Darstellungen (im Amiga-Jargon Playfields genannt) bilden. Die Hintergrundfarbe des vorderen Playfields ist transparent, so daß das hintere durchscheint. Die Attraktivität des Dual-Playfield-Modus liegt in der der Unabhängigkeit beider Playfields; jedes für sich stellt einen eigenen Grafikbildschirm dar – der Phantasie sind keine Grenzen gesetzt (mehr dazu im Grafik-Kapitel).

Ein Blick auf das Blockdiagramm (Abbildung 3.8) zeigt eine vertraute Grundstruktur, die die enge Zusammenarbeit von Fat Agnus, Denise und Paula unterstreicht: Da gibt es wieder den gepufferten 16-Bit-Datenbus, den ebenfalls gepufferten Register-Adreßbus mit den Decoder-PLAs und Steuerleitungen für die die Register-Aktivierung (dünne Pfeile im Blockschaltbild).

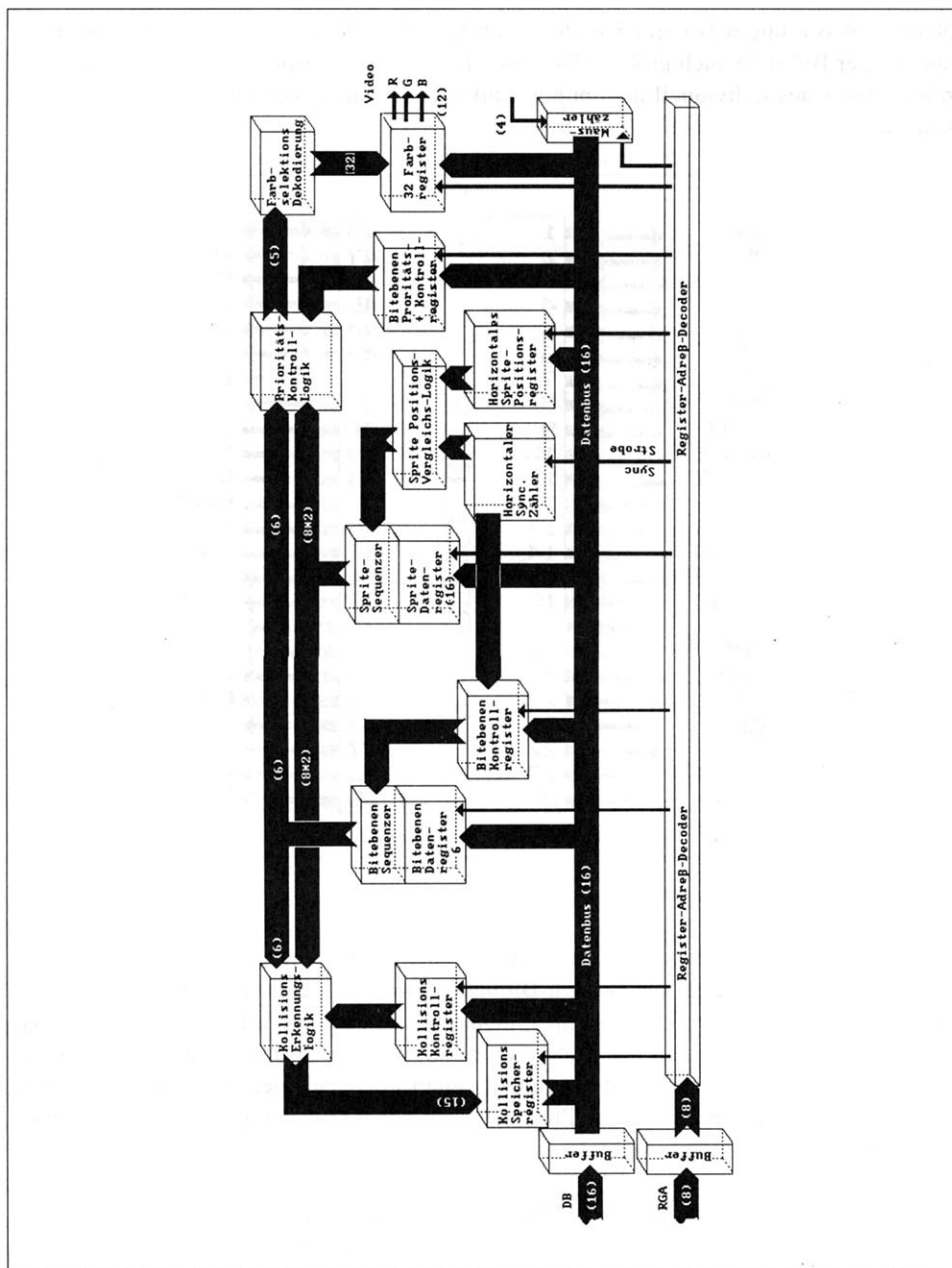


Abb. 3.8: Blockdiagramm Denise

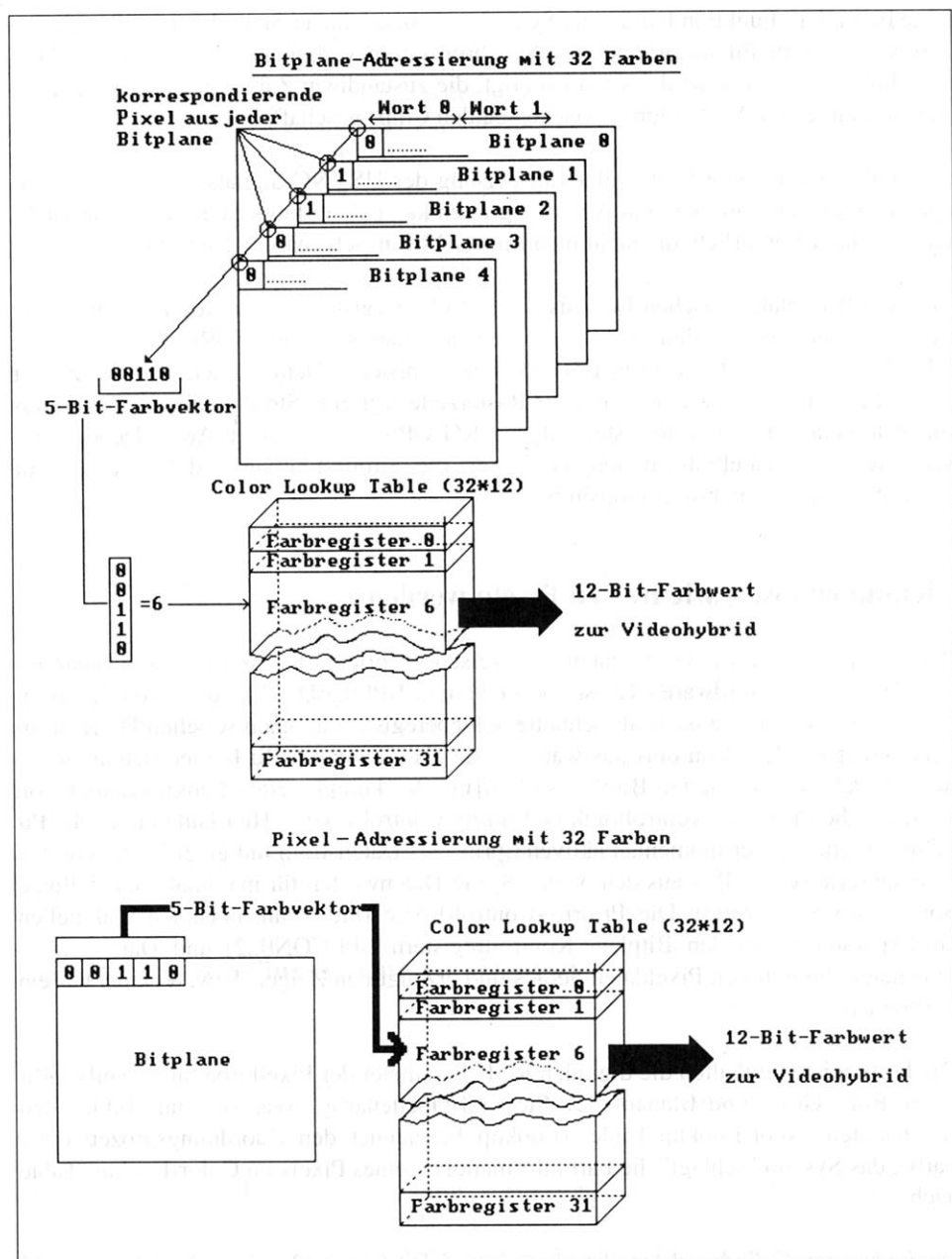


Abb. 3.9: Die Grafikdatenorganisation im Vergleich

Eine besondere Funktion hat die als Sync-Strobe bezeichnete Steuerleitung, die mit den Strobe-Registern für die horizontale Synchronisation verbunden ist. Da Denise diese Synchronisationsimpulse dringend benötigt, die zuständigen Zähler aber auf Fat Agnus liegen, mußte eine Verbindung zwischen beiden Chips geschaffen werden.

Die einfachste Lösung liegt in der Einspeisung des HSYNC-Signals, das man am entsprechenden Anschluß von Fat Agnus abgreifen kann. Allerdings ist dafür eine zusätzliche Leitung erforderlich, die nicht mehr in das 48-Pin-Schema von Denise paßt.

Wie so oft in platzkritischen Designs, sind Strobe-Register ein Ausweg zur Umgehung des vermeintlichen Dilemmas. Vier Register dieses Typs (STREQU, STRVBL, STRHOR, STRLONG ab Offset \$038), synchronisieren Denise intern auf die Zähler des Fat-Agnus-Chips. Zu Beginn jeder Rasterzeile legt eine Steuerlogik von Fat Agnus die Adressen der Strobe-Register auf den RGA-Bus, die Register-Adreßdecoder entschlüsseln die Adreßinformation, aktivieren die Strobe-Leitung und übertragen auf diese Weise den Synchronisationsimpuls.

Metamorphose – wie Bits zu Pixeln werden

Per DMA gelangen die Grafikdaten wortweise (entspricht 16 Pixeln) in die Datenregister der Bitplane-Hardware (Registerbezeichnung: BPL0..6DAT), wo sie von den Bitbenen-Sequenzern (realisiert als schnelle Schieberegister) in einen seriellen Datenstrom verwandelt werden. Von dort aus wandern sie als maximal 6 Bit breiter Datenpfad (je nach Anzahl der aktivierten Bitplanes [1...6]) in den komplexesten Funktionsblock von Denise – die Prioritäts-Kontrolllogik (= Priority Control Logic). Hier laufen auch die Pixelinformationen der momentan aktiven Sprites als Datenstrom mit einer Breite von 8 x 2 Bits ein (jeweils 2 Bits aus den beiden Sprite-Datenworten für maximal 8 darstellbare Sprites pro Rasterzeile). Die Priority-Control-Logic filtert nun anhand der aktuellen Grafikparameter (in den Bitplane-Kontrollregistern BPLCON0..2) und Darstellungsprioritäten die gültigen Pixeldaten heraus und erzeugt den Zeiger (bzw. Vektor) auf ein Farbregister.

Die Farbregister enthalten die digitalen RGB-Parameter der Pixelfarbe (als jeweils 4 Bit breite Rot-, Grün- und Blauanteile). Sie sind tabellenartig organisiert und bilden den sogenannten Color-Lookup-Table. (Lookup bezeichnet den Zuordnungsprozeß einer Farbe, das System "schlägt" die Farbinformationen eines Pixels im Color-Lookup-Table nach.)

Denise besitzt 32 Farbregister, die über einen 5-Bit-Offset ($2^5 = 32$, Vektor auf 32 Einträge der Farbtabelle) adressiert werden (entspricht fünf aktiven Bitplanes). Die Tiefe des Color-Lookup-Table begrenzt die Anzahl der gleichzeitig darstellbaren Far-

ben. Bei 32 Farbregistern können auch nur 32 verschiedene Farben auf einmal gezeigt werden – zuwenig für fließende Schattierungen. Um die begrenzte Kapazität des Color-Lookup-Table zu überwinden, den schaltungstechnischen Aufwand aber nicht durch einen überdimensionalen Ausbau der Farbtabelle in die Höhe zu treiben, implementierten die Entwickler zwei spezielle Modi, bei denen die Informationen aus den Farbregistern noch verändert werden, bevor sie zu den externen D/A-Wandlern gelangen. Während die meisten Bitplanes ihre normale Funktion behalten (korrespondierende Pixelpositionen ergänzen sich zu einem Zeiger auf die Farbtabelle), bestimmen die höchstwertigen Planes den Algorithmus zur Farbmodifikation. Schauen wir uns dieses Prinzip einmal genauer an!

Der Extra-Half-Bright-Modus beispielsweise (übrigens gab es diesen Modus noch nicht in den ersten NTSC-Versionen des Denise-Chip) verwendet alle sechs Bitplanes. Die Farbselektionslogik spaltet den 6 Bit breiten Datenstrom in einen 5-Bit-Vektor für die Registertabelle und eine 1-Bit-Steuerleitung zur Modifikationsschaltung auf. Hat das Steuerbit den logischen Wert 0, darf der Farbwert die Modifizierlogik passieren; hat es dagegen den Wert 1, wird jeder RGB-Farbanteil durch 2 dividiert, so daß die ursprüngliche Farbinformation mit halber Intensität erscheint (daher auch die Bezeichnung Half Bright = halbe Helligkeit). Mit relativ geringem Aufwand verdoppelt diese Methode den Umfang gleichzeitig darstellbarer Farben (64 anstatt der üblichen 32).

Auf der linken Seite des Blockdiagramms fällt eine weitere Kontrollsektion ins Auge – die Kollisionslogik. Sie ist (wie auch die Prioritäts-Kontrolllogik) mit den Datenströmen verbunden und registriert Sprite/Sprite-, Sprite/Bitebenen- bzw. Bitebenen/Bitebenen-Kollisionen. Die Schaltung ist voll programmierbar. Das CLXCON-Register (Collision X Control, Offset \$098) blendet die einzelnen Grafikobjekte in den Erkennungsprozeß ein. Gültige Zusammenstöße werden im CLXDAT-Register (Collision X Data, Offset \$00E) vermerkt.

Erläuterung zur Pinbelegung von Denise:

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
D0-D15	1-7 40-48	E/A	Bidirektionaler 16-Bit-Datenbus
M1H	8	E	Mouse 1 Horizontal (= horizontaler Quadraturimpuls der Maus). Die Zeilen- und Spaltenzähler von Denise registrieren die Mausbewegungen. Vier Signale pro Game-Port liefern die vollständige Bewegungsinformation (H, Q, HQ, VQ).

Pin Pin-Nr. Eing./Ausg. Funktionsbeschreibung

Da allerdings für beide Ports nur noch vier Pins von Denise frei waren, bediente man sich eines Multiplex-Verfahrens, bei dem die Game-Ports nacheinander abgefragt werden. Die Umschaltung geschieht synchron zum CCK-Takt. Sämtliche Mauseingänge sind nach diesem Schema geschaltet (detaillierte Beschreibung der Mausfunktionen im Schnittstellen-Kapitel).

M0H	9	E	Mouse 0 Horizontal (= horizontaler Richtungsimpuls der Mausebewegung).
RGA1-RGA8	10-17	E/A	Register-Adreßbus (siehe FAT AGNUS und Paula)
BST	18	A	Color-Burst-Signal (kennzeichnet die Phasenlage des Farbträgersignals).
VCC	19	E	Versorgungsspannung: +5V
R0-R3	20-23	A	Red-Bit 0..3 (= digitaler 4-Bit-Rotanteil der Farbinformation). Denise liefert noch keine analogen RGB-Signale, sondern gibt die Farbinformation als digitale Intensitätswerte für Rot, Grün und Blau aus; erst die D/A-Wandler der Videohybrid erzeugen das analoge Ausgangssignal. Bei einer Auflösung von 12 Bit, lassen sich maximal 4096 ($= 2^{12}$) unterschiedliche Farben darstellen.
B0-B3	24-27	A	Blue-Bit 0..3 (= digitaler 4-Bit-Blauanteil der RGB-Farbinformation).
G0-G3	28-31	A	Green-Bit 0..3 (= 4 Bit digitaler Grünanteil der RGB-Farbinformation).
N/C	32	–	Not Connected (nicht verbunden).
ZD*	33	A	Zero Detect (= Null-Kennung). Jedesmal, wenn ein Pixel in der Hintergrundfarbe dargestellt wird, geht ZD* auf Low.

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
			Das Signal dient als Steuerhilfe für ein Genlock-Interface, das die externe Videoquelle in den Hintergrund der Amiga-Grafik einblendet. Während der vertikalen Austastlücke reflektiert die ZD*-Leitung den Zustand des GAUD-Bits (= Genlock Audio Enable) im BPLCON0-Register (Offset \$100).
N/C	34	–	Not Connected.
7M	35	E	7MHz-CPU-Takt (PAL-Version = 7,09 MHz). Die hohe Auflösung (640 Spalten) arbeitet mit einer Pixelfrequenz von ca. 14,2 MHz (Verweildauer des Rasterstrahls an einer Position = 70 ns). Die Ausgabe der Pixeldaten läuft synchron zum CPU-Takt; mit jeder Flanke erhält der angeschlossene Monitor einen RGB-Impuls (das entspricht zwei Pixel pro Prozessor-Taktzyklus und vier pro DMA-Zyklus)
CCK	36	E	Color Clock (Farbträger, Taktfrequenz = 3,55 MHz).
Vss	37	E	Masse.
M0V	38	E	Mouse 0 Vertical (= vertikaler Richtungsimpuls der Mausbewegung).
M1V	39	E	Mouse 1 Vertical (= vertikaler Quadraturimpuls der Mausbewegung).

Paula – Tor zur Außenwelt

In den technischen Unterlagen der ersten Entwicklungssysteme lief dieser Customchip noch unter der lateinischen Bezeichnung PORTIA (= Tür, Eingang), da er den Prozessor von zeitaufwendigen E/A-Funktionen entlasten sollte.

In PAULAS Ressort fallen so komplexe E/A-Prozesse wie Tonausgabe, Datenübertragung zum Disketten-Laufwerk und zur RS232C-Schnittstelle oder Auswertung der Analog-Eingänge (Game-Port-Anschluß für Potentiometer).

Der hochintegrierte Chip enthält die komplette Audio-Hardware (bis auf den extern realisierten Tiefpaßfilter), den Floppy-Controller für die Steuerung des Datenflusses zum Diskettenlaufwerk und die UART-Schaltung (Universal Receiver Transmitter = Universeller Empfang/Sende-Baustein) zum Betrieb der seriellen Schnittstelle.

Die detaillierte Untersuchung dieser Funktionsblöcke ist weiteren Kapiteln vorbehalten.

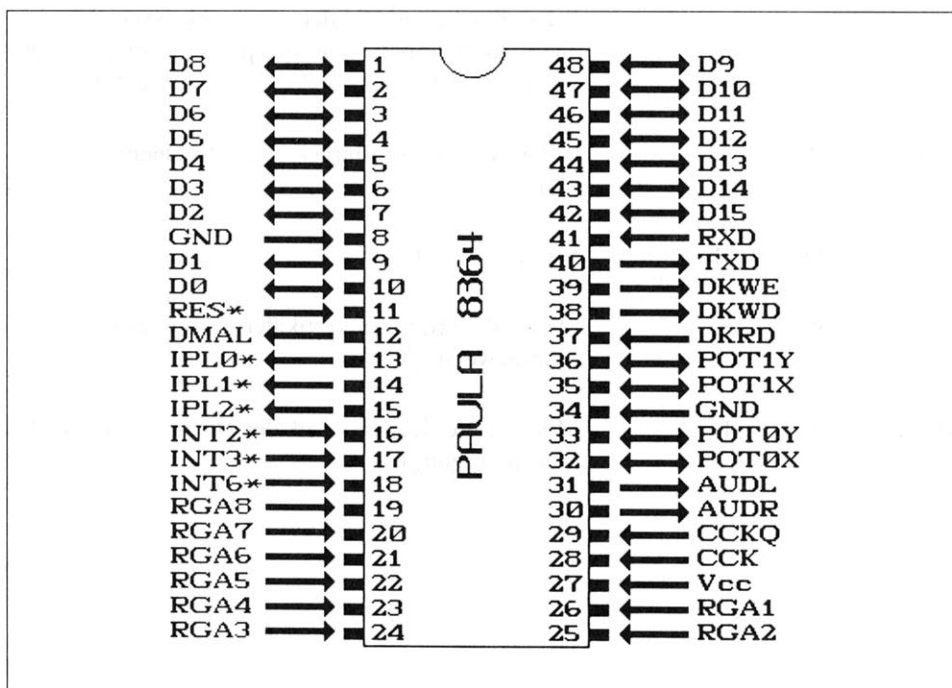


Abb. 3.10: Pinbelegung Paula

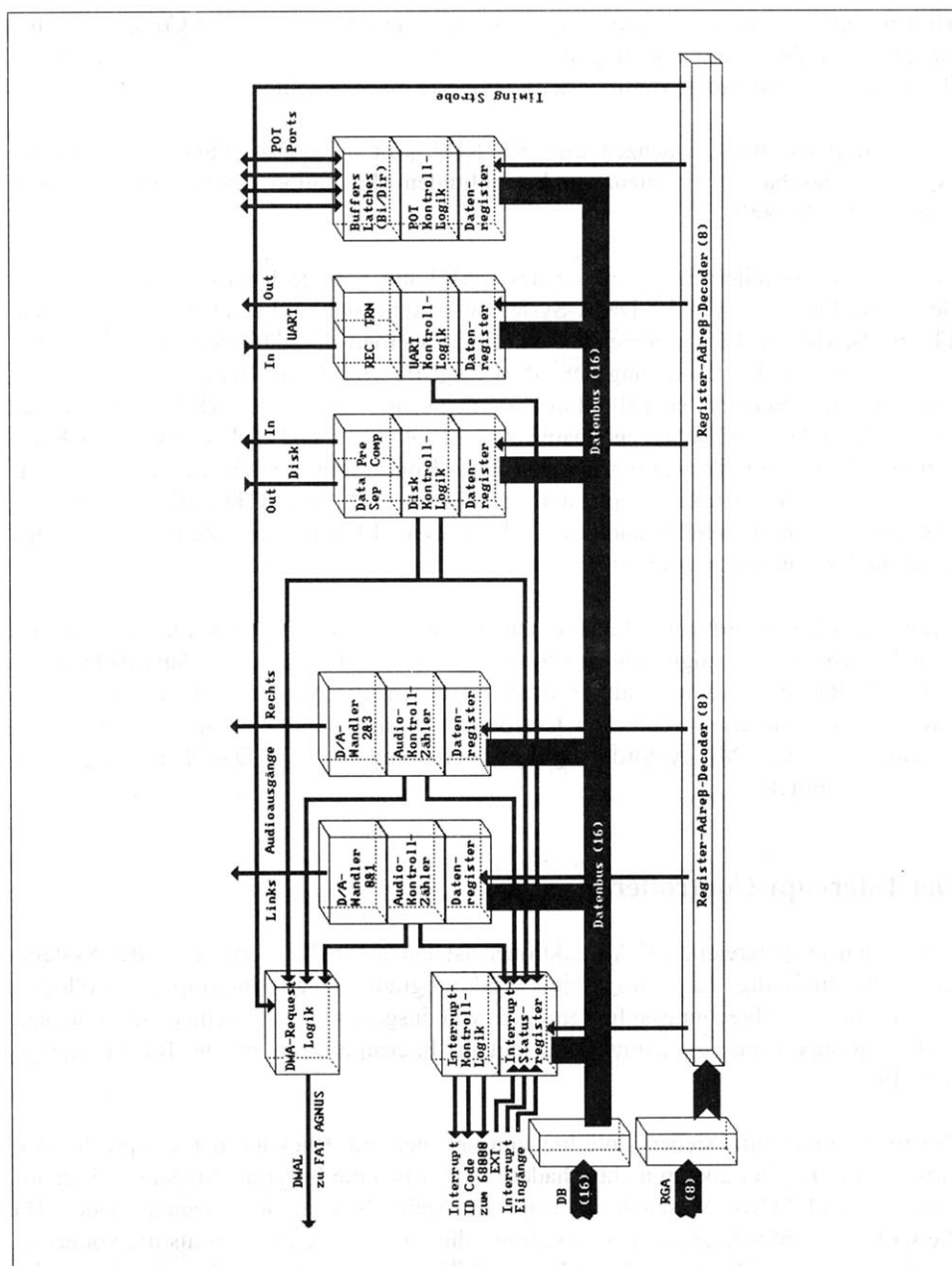


Abb. 3.11: Blockdiagramm Paula

Hier nur einige Stichwörter zur Soundhardware, die selbst notorische Computersound-Skeptiker (in ihrer Zurückhaltung durch das schwachbrüstige Gepiepse so mancher PC-Tongeneratoren bestätigt) zumindestens neugierig machen sollten:

8-Bit-Sampling mit Frequenzen bis 28,6 KHz, vier 8-Bit-D/A-Wandler (als Stereo-Ausgänge geschaltet), Frequenz- und Amplituden-Modulation, abschaltbarer Tiefpaßfilter (A500, B2000) ...

Während die seriellen E/A-Prozesse des UART nur geringe Übertragungsraten erfordern, eine Einbindung in das DMA-System also nicht sinnvoll ist, fallen bei Sound- und Floppy-Betrieb große Datenmengen an. Beide werden deshalb über spezielle DMA-Kanäle versorgt. Die Steuerung der DMA-Zugriffe von Audio-Hardware und Floppy-Controller unterscheidet sich allerdings von den bisher untersuchten DMA-Quellen. Der Grund dafür liegt im unberechenbaren Zugriffsverhalten und der eigenwilligen Registerverteilung. Normalerweise gibt es für jedes Kontrollregister DMA-unterstützter Customchip-Funktionen eine Kopie auf Fat Agnus (z.B. SPRxCTL, BPLCON0). Die DMA-Kontrolllogik behält ständig den Überblick und kann sowohl Zeitpunkt, als auch Umfang des Datenbedarfs exakt bestimmen.

Audio- und Disk-Hardware fallen hier aus dem Rahmen. Fat Agnus kann weder auf das ADKCON-Register zugreifen noch Länge (AUDxLEN) und Ausgabefrequenz (AUDxPER) eines Audiokanals ermitteln – die entsprechenden Signale müssen von Paula selbst geliefert werden. Der Customchip besitzt dafür eine spezielle Kontrollschaltung, die die DMA-Anforderungen auswertet und durch die DMAL-Leitung an Fat Agnus übermittelt.

Der Interrupt-Controller

Neben den umfangreichen E/A-Funktionen ist Paula für die Verwaltung der System-Interrupts zuständig. Eine ausgefeilte, voll programmierbare Interrupt-Kontrolllogik, sortiert die Unterbrechungsanforderungen von insgesamt 14 verschiedenen Quellen, faßt sie gruppenweise zusammen und legt ihr Interruptniveau auf die IPL*-Eingänge der CPU.

Der 68000 unterstützt sieben Unterbrechungsebenen (ein Blick auf den entsprechenden Abschnitt in Kapitel 2 kann nicht schaden), von denen die Interrupt-Niveaus 0 (Pseudo-Interrupt) und 7 (Non Maskable Interrupt) im Amiga-System nicht erzeugt werden. Da die CPU nur Anforderungen berücksichtigt, die eine höhere Priorität als die vorausgegangenen besitzen, laufen wichtige Interrupt-Prozesse ungestört, während andere – die es unter Umständen nicht ganz so eilig haben – geduldig warten müssen. Erkennt der Prozessor eine gültige Unterbrechungsanforderung, springt er in die entsprechende Po-

sition der Ausnahmevektortabelle (Exception-Table), wo er den Zeiger auf eine Interrupt-Routine findet.

Paula unterstützt allerdings nur die autovektorielle Unterbrechungsverarbeitung. Die erzeugten Interrupt-Vektoren liegen im Bereich von \$64 (Ebene 1) bis \$78 (Ebene 6).

Aufgaben und Funktion eines Interrupt-Controllers

An der Schnittstelle zwischen Systembausteinen und Interrupt-Eingängen der CPU beginnen die Aufgaben eines Interrupt-Controllers (kurz IC):

- Sortierung der Unterbrechungsanforderungen linear angeschlossener Peripherie-Chips (deren Zahl erheblich größer ist als die vorhandenen Interrupt-Eingänge der CPU).
- Codierung ihrer Priorität.
- Übermittlung des Unterbrechungsniveaus an die logischen Interrupt-Eingänge des Prozessors.

Erschöpften sich die Aufgaben einer derartigen Schaltung ausschließlich in der Verteilung der Interrupt-Quellen auf die Unterbrechungsniveaus der CPU, wäre ein einfacher Codierer-Baustein (z.B. 3 aus 8) vollkommen ausreichend. Allerdings könnten die Interrupt-Quellen den Prozessor dauernd mit Unterbrechungsanforderungen belästigen; zyklisch wiederkehrende Interrupts (z.B. Vertical-Blanking) wären in der Lage, das System allein durch den verursachten Bearbeitungsaufwand lahmzulegen.

Deshalb bieten sogenannte programmierbare Interrupt-Controller (PIC = Programmable Interrupt Controller) die Möglichkeit, gezielt einzelne Interrupt-Quellen zu sperren und vom Verarbeitungsprozeß auszugrenzen. Man spricht von einer maskierbaren Interrupt-Auswertung, für die Paula einen ebenso einfachen wie leistungsfähigen Mechanismus bereitstellt.

Die Programmierung des Interrupt-Systems

Während alle einlaufenden Unterbrechungsanforderungen im INTREQ-Register registriert werden (INTerrupt REquest = Unterbrechungsanforderung; Lesen an Offset \$01E, Schreiben bei \$09C), akzeptiert Paula die Anforderung nur dann, wenn auch das

korrespondierende Bit im INTENA-Maskenregister gesetzt ist (INTerrupt Enable = Unterbrechung erlauben; Lesezugriff an Offset \$01C, Schreibzugriff bei Offset \$09A).

Paula ermittelt die gültigen Unterbrechungsanforderung durch logische UND-Verknüpfung von Masken- und Anforderungsregister, wobei nicht zugelassene (gesperrte) Interrupt-Quellen ausgeblendet werden und gibt das Ergebnis an den Prozessor weiter.

Der Zugriff auf die Interrupt-Register ist wieder nach dem bewährten Set/Clear-Mechanismus realisiert (siehe DMACON).

Bit 14 (INTEN = Interrupt Enable) hat eine besondere Funktion. Es dient als Hauptschalter des Interrupt-Systems (vergleichbar mit DMAEN im DMACON-Register) – nur wenn dieses Bit logisch wahr ist (= 1), kann überhaupt ein Interrupt ausgelöst werden.

Um alle Interrupt-Quellen zu sperren, reicht es aus, das INTEN-Bit auf 0 zu setzen; ebenso einfach läßt sich der ursprüngliche Zustand durch Einblendung des Bits wiederherstellen. Da der Anwender bei der Veränderung kritischer Systemparameter (vor allen Dingen in einem Multitasking-Betriebssystem) häufiger auf eine vollständige Sperrung des Interrupt-Systems zurückgreifen muß, erweist sich das INTEN-Bit schnell als angenehme Arbeitserleichterung.

Nach diesem Prinzip arbeiten auch die System-Makros Enable() und Disable(); Kern beider Funktionen ist die Modifikation des INTEN-Bits (Bit 14 im INTENA-Register). Disable() sperrt sämtliche Interrupts durch Negierung von INTEN (move.w #4000,INTENA), Enable() blendet das Bit wieder ein (move.w #\$C000,INTENA).

Die begrenzte Anzahl der Interrupt-Ebenen und die daraus resultierende Zusammenfassung mehrerer Quellen unter einer Priorität hat zur Folge, daß sich der auslösende Baustein nicht einfach anhand seines Unterbrechungsniveaus bestimmen läßt.

Zur genauen Identifizierung muß das INTREQ(R)-Register ausgelesen werden (polling = nachfragen, siehe Interruptmechanismus der CIA-Chips im nächsten Kapitel).

Unterbrechungsanforderungen im INTREQ-Register bleiben auch nach der Verarbeitung erhalten, können also im folgenden Interrupt-Zyklus erneut eine Unterbrechung des Programmflusses bewirken. Deshalb sollte am Ende jeder Interrupt-Routine die auslösende Quelle wieder ausgeblendet werden; wer nicht daran denkt, wird von einem hartnäckigen Dauerbrenner überrascht.

INTENA \$09A w Paula
 (INTerrupt ENAbLe bits = Interrupt-Kontroll-Bits)

Zentrales Kontrollregister zum Ein- bzw. Ausblenden der einzelnen Interrupt-Quellen. Der Zugriff auf das Register erfolgt (wie bei DMACON) durch einen Set/Clear-Mechanismus. Ist das korrespondierende Bit einer Interrupt-Quelle gesetzt, wird ihre Anforderung akzeptiert und zum Prozessor weitergeleitet.

Bitbelegung: (Die eingeklammerten Zahlen stellen den zugewiesenen Prozessor-Interrupt-Level dar.)

<i>Nr.</i>	<i>Label</i>	<i>Funktion</i>
15	SET/CLR	Set/Clear-Bit. Schaltet zwischen Setz- (1) und Lösch-Modus um. Es werden nur die Bits beeinflusst, die an den Positionen der Einer-Bits im Maskenwort liegen.
14	INTEN	Interrupt-Hauptschalter (siehe Bit 9, DMACON). Unterbrechungsanforderungen werden erst dann bearbeitet, wenn dieses Bit gesetzt ist. Durch Löschen des Bits läßt sich das gesamte Interrupt-System sperren, ohne jede Quelle einzeln verbieten zu müssen.
13	EXTER(6)	Externer Interrupt von CIA-B oder Expansion-Port.
12	DSKSYN (5)	Disketten-Synchronisationsmarkierung gefunden.
11	RBF (5)	Empfangspuffer der seriellen Schnittstelle voll.
10-07	AUDx(4)	Audio-DMA-Zyklus beendet (AUDLEN heruntergezählt). (Bit 10 = Tonkanal 3, Bit 9 = Tonkanal 2 usw.)
06	BLIT (3)	Blitter-Operation beendet.
05	VERTB (3)	Beginn der vertikalen Austastlücke.
04	COPER (3)	Copper-Interrupt (durch Copper-Liste selbst ausgelöst!)
03	PORTS (2)	Interrupt von CIA-A.
02	SOFT (1)	für Software-Interrupt reserviert.

<i>Nr.</i>	<i>Label</i>	<i>Funktion</i>
01	DSKBLK (1)	Disk-DMA fertig.
00	TBE (1)	Sendepuffer der seriellen Schnittstelle leer.
INTENAR	\$01C	r Paula (INTerrupt ENable bits Read = Lesen der Interrupt-Kontroll-Bits)
Bitbelegung wie INTENA.		
INTREQ	\$09C	w Paula (INTerrupt REQuest bits = Interrupt-Statusbits (setzen und löschen))

Die Bitbelegung ist mit der des INTENA-Registers identisch. Sämtliche Interrupt-Anforderungen des Systems laufen in diesem Register zusammen. Der Vergleich mit INTENA entscheidet über die Annahme oder Ablehnung der Anforderung. Das INTREQ-Register ist für die CPU voll transparent, so daß sich System-Interrupts auch durch Setzen der korrespondierenden Status-Bits simulieren lassen (Bit 2 ist sogar speziell für Software-Interrupts reserviert).

Nach der Bearbeitung einer Unterbrechungsroutine muß der Programmierer das auslösende Interrupt-Bit löschen, da das Register nicht automatisch zurückgesetzt wird.

INTREQR	\$01E	r	Paula
(INTerrupt REQuest bits Read = Leseadresse des Interrupt-Anforderungs-Registers)			

Das Register reflektiert den aktuellen Status des Interrupt-Systems und erlaubt die exakte Lokalisierung der auslösenden Interrupt-Quelle. Dies ist notwendig, da die CPU lediglich über 7 Interrupt-Ebenen verfügt, das System aber 14 verschiedene Unterbrechungs-Quellen unterstützt. Die Handler des Betriebssystems werten das INTREQR-Register aus (polling) und springen in die korrespondierende Unterbrechungsroutine.

"Weiche Unterbrechungen" – die SOFT-Interrupts

Die weitgehende Transparenz des Interruptmechanismus erlaubt es dem Programmierer, auch softwaremäßig eine Unterbrechung auszulösen; sei es, um die Interrupt-Anforderung eines bestimmten System-Bausteins zu simulieren oder gezielt einen Software-Interrupt zu erzeugen (Bit 2 [SOFT] ist für Anwender-Unterbrechungen reserviert).

Bekanntlich enthält der Befehlssatz des 68000er (außer den TRAP-Instruktion zur Kommunikation mit dem Betriebssystem) keinen speziellen Befehl für die programmtechnische Erzeugung einer Unterbrechung – PAULAs SOFT-Interrupts erweitern den Reaktionsspielraum.

Erläuterungen zur Pinbelegung von Paula:

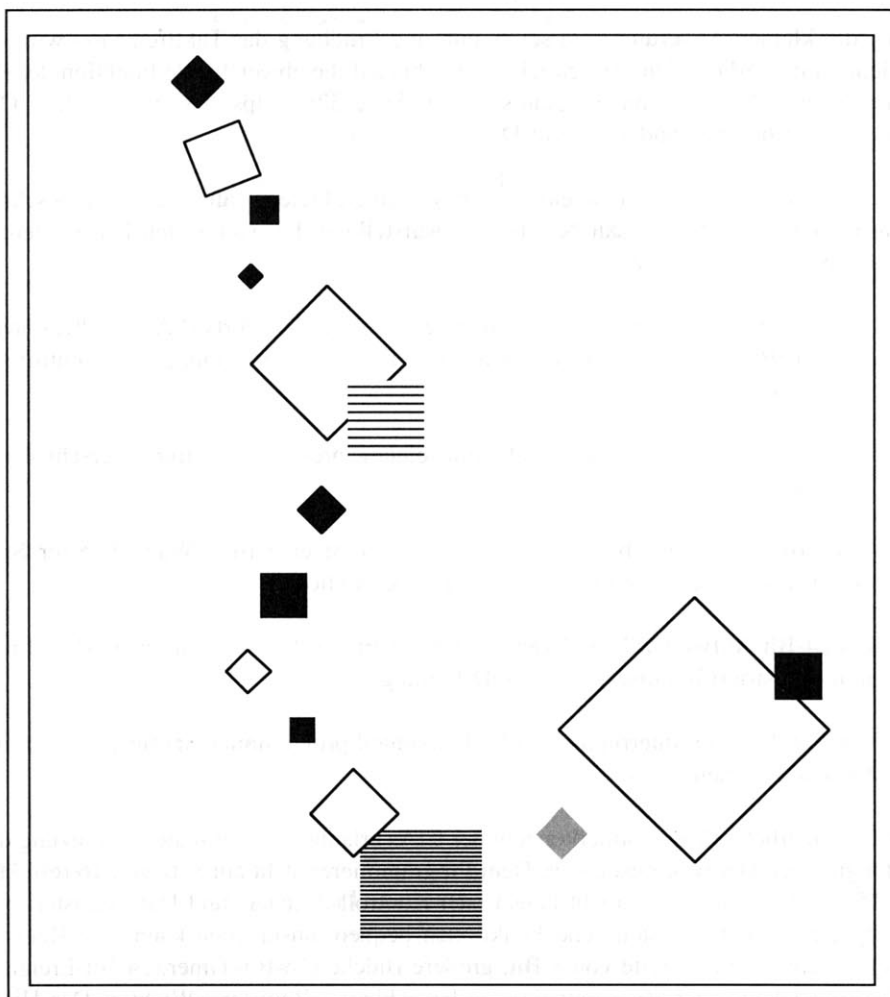
<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
D0-D15	1-7/ 9-10 40-48	E	16-Bit-Datenbus (intern gepuffert).
Vss	8	E	Masse.
RESET*	11	E	RESET-Leitung.
DMAL	12	E	Disk-/Audio-DMA-Anforderung (siehe gleichnamigen Eingang von Fat Agnus)
IPL0-2	13-15	A	Interrupt-Pending-Level der CPU. Als Interrupt-Controller verwaltet Paula sämtliche System-Interrupts. Er faßt die Unterbrechungsanforderungen gruppenweise zusammen, wertet sie anhand der aktuellen Maskierung aus und sendet das Unterbrechungsniveau über die IPL*-Leitungen an den Prozessor.
INT2, 3, INT6	16-18	E	Unterbrechungsanforderung der Systembausteine und INT6 Expansion-Ports (die Nummern entsprechen dem zugewiesenen Pegelniveau).
RGA1-8	19-26	A	Bidirektionaler 8-Bit-Register-Adreßbus (siehe Fat Agnus, Denise)
Vcc	27	E	Versorgungsspannung: +5V
CCK	28	A	Color CLoCK (Frequenz = 3,55 MHz).

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
CCKQ	29	A	Color CLock Quadrature (CCK-Takt, Phasenlage 90° verschoben).
AUDR	30	A	Audioausgang rechts (Kanäle 1 und 2 sind auf diese Leitung geschaltet).
AUDL	31	A	Audioausgang links (Kanäle 0 und 3).
POT0X	32	E	Analogeingang (X-Richtung, Game-Port 0). Jeder Game-Port verfügt über zwei dieser Analog-Ports (jeweils für X- und Y-Richtung). Paula bestimmt den Wert eines veränderlichen Widerstandes zwischen POTxx- und Masseleitung und schreibt das Meßergebnis in das korrespondierende POTxDAT-Register (POD0DAT an Offset \$00C, POT1DAT bei \$00E). Damit eignen sich die Game-Ports zum Anschluß von Paddles (Näheres über Funktion und Beschaltung im Kapitel 6, "Die Interface-Hardware Teil 1").
POT0Y	33	E	Analogeingang (Y-Richtung) Game-Port 0
VssANA	34	E	Masse für Analogleitungen (Funktion: siehe oben)
POT1X	35	E	Potentiometeranschluß X-Richtung (Game-Port 1)
POT1Y	36	E	Potentiometeranschluß Y-Richtung (Game-Port 1)
DKRD	37	E	DisK Read Data. Daten-Leseleitung des Floppy-Disk-Interface (serielle Datenübertragung).
DKWD	38	A	DisK Write Data. Schreibleitung zum Laufwerk.
DKWE	39	E	Disk Write Enable (= Schreibzugriff auf die Diskette erlauben); schaltet zwischen Schreib- und Lese-Modus um.
TXD	40	A	Transmit Data (= Daten-Sendeleitung des UART). Datenausgang der seriellen Schnittstelle.

<i>Pin</i>	<i>Pin-Nr.</i>	<i>Eing./Ausg.</i>	<i>Funktionsbeschreibung</i>
RXD	41	E	Receive Data (= Daten-Empfangsleitung des UART). Dateneingang der seriellen Schnittstelle.

Kapitel 4

Die CIA - Hardware



Das Amiga-System enthält zwei (von der Commodore-Tochter MOS-Technology entwickelte) CIA-Chips (Complex Interface Adapter = Komplexer Schnittstellenbaustein) mit der Typenbezeichnung 8520. Obwohl die Komplexität dieser Bausteine im Vergleich zu den hochintegrierten VLSI-Customchips eher bescheiden wirkt, vereinigen sie trotzdem eine Reihe leistungsfähiger E/A-Funktionen in einem bereits bestens bewährten Konzept. Die Serienbezeichnung 8520 klingt nach einer geheimnisvollen Spezialentwicklung, die sich aber schnell als Produkt einer typischen Commodore-Marotte entpuppt:

Bei jeder kleinen Änderung (und sei es nur eine Erhöhung der Taktfrequenz) wird die Seriennummer erhöht. Die Amiga-CIAs sind, bis auf die abweichende Funktion der Register 8 bis 11, vollkommen identisch mit den 6526-Chips des guten, alten C64 ("Greetings from the good, old 8-Bit-Days!").

Und genau wie diese haben sie eine ganze Menge zu bieten – allein die Kurzbeschreibung ihrer Fähigkeiten enthält bereits eine Aufstellung der wichtigsten Peripheriebausteine eines Computersystems:

- zwei frei programmierbare, bidirektionale 8-Bit-Parallel-Ports (PA und PB), davon einer (PB) mit Handshakemechanismus für die Verwendung als Centronics-Schnittstelle.
- ein serieller Port (bidirektional) mit einem breitem Spektrum verschiedener Baudraten (Bit/s).
- zwei 16-Bit-Timer (abwärtszählend); möglich ist eine freie Wahl diverser Synchronisationsquellen, Zählmodi und Ausgabefunktionen.
- ein 24-Bit-Aufwärtszähler (Event Counter = Ereigniszähler) mit einer Alarmfunktion (registriert Impulse auf der TOD-Leitung).
- eine maskierbare Interruptkontrolle (individuell programmierbar) für jede einzelne Funktionseinheit.

Das übersichtliche Programmierkonzept der CIAs erlaubt eine optimale Ausnutzung der umfangreichen Hardwareresourcen. Dem Programmierer steht ein Satz von 16 (effektiv nur 15, da das Register 11 nicht belegt ist!) Kontroll-, Steuer- und Datenregistern zur Verfügung, über die er sämtliche Funktionen bequem ansprechen kann. Die Register haben durchweg eine Breite von 8 Bit, größere Blöcke (16-Bit-Timer, 24-Bit-Ereigniszähler) entstehen durch Zusammenlegung benachbarter Register. (Wichtig! Das High-Byte einer kombinierten 16-Bit-Bank liegt an der höheren Byte-Adresse!)

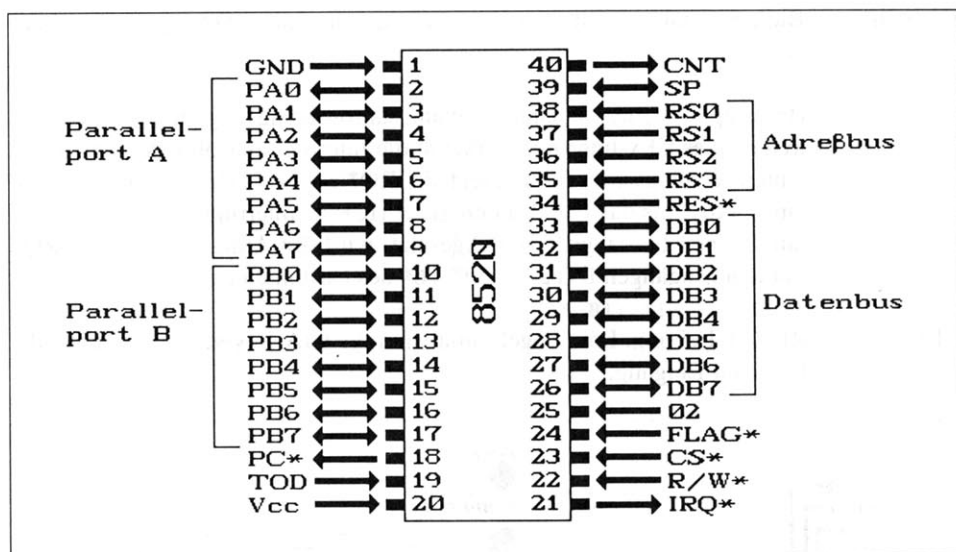


Abb. 4.1: Die Pinbelegung

Eine leistungsfähige Busschnittstelle (siehe Abbildung 4.2) verbindet die 8520-Chips mit dem Hauptprozessor. Die CPU spricht sie im synchronen Zugriffsmodus an.

Die Steuersignale der Busschnittstelle:

- 02: Takteingang (TTL-kompatibel). Hier liegt der E-Takt der CPU, der mit ca. 710 KHz (PAL-Amiga) einem Zehntel der Prozessor-Taktfrequenz entspricht.
- CS*: (Chip SELECT = Chipauswahl) steuert den Zugriff auf die Registerbank der 8520-Chips. Bei Low-Pegel wird die Adreßinformationen auf den RSx-Leitungen automatisch übernommen und decodiert.
- RS0-RS3: (Register SELECT = Registerauswahl) sind direkt mit den Adreßsignalen A8 bis A11 der CPU verbunden und dienen der Decodierlogik zur Ermittlung der adressierten Registerposition. Vier Leitungen erlauben die Unterscheidung von 16 (= 2⁴) verschiedenen Einträgen der Registermap.
- R/W*: (Read/Write = Schreiben/Lesen) wählt die Zugriffsrichtung aus (Low = Schreibmodus, High = Lesen).

DB0-DB7: Bidirektionaler 8-Bit-Datenbus (intern mit einem Datenpuffer verbunden).

IRQ*: (Interrupt ReQuest = Unterbrechungsanforderung) signalisiert das Auftreten eines CIA-Interrupts. Erkennt die Interruptkontrolllogik eine gültige Unterbrechungsanforderung, legt sie IRQ* auf Low. Die Leitung bleibt so lange aktiv, bis das ICR-Datenregister (ICR = Interrupt Control Register) durch einen Lesezugriff zurückgesetzt wird. Im Amiga-System ist IRQ* mit den Eingängen INT2* und INT6* des Customchips Paula verbunden.

RES*: RESET-Leitung. Low-Pegel initialisiert die CIA-Register und schaltet die Ports auf Eingang.

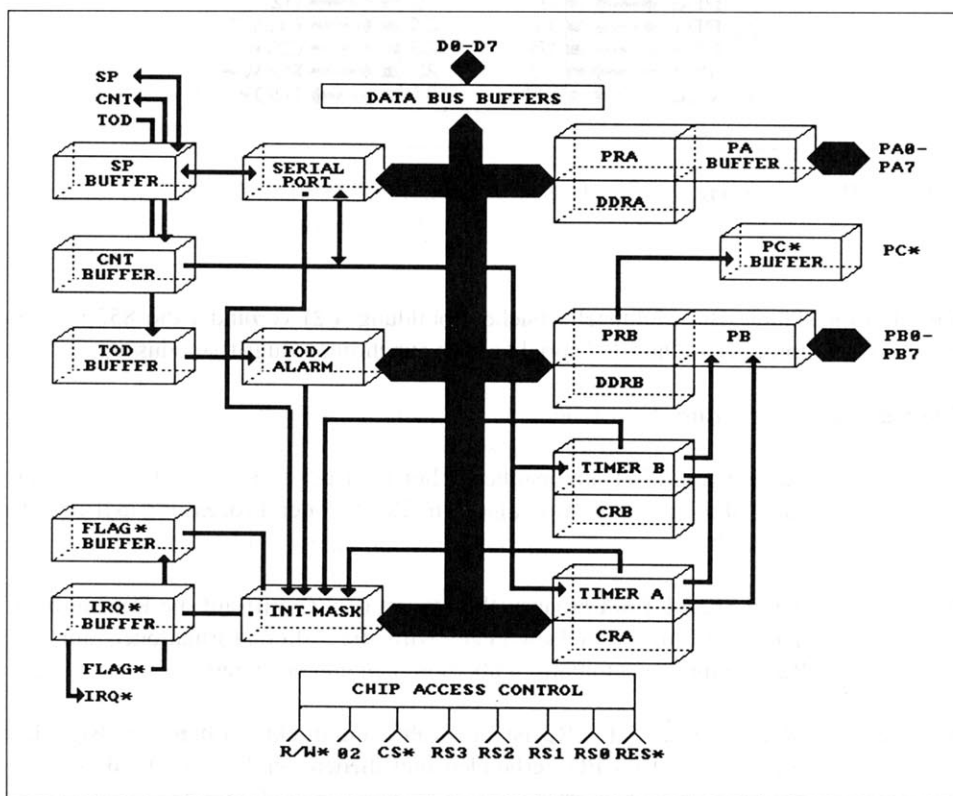


Abb. 4.2: Blockdiagramm des 8520

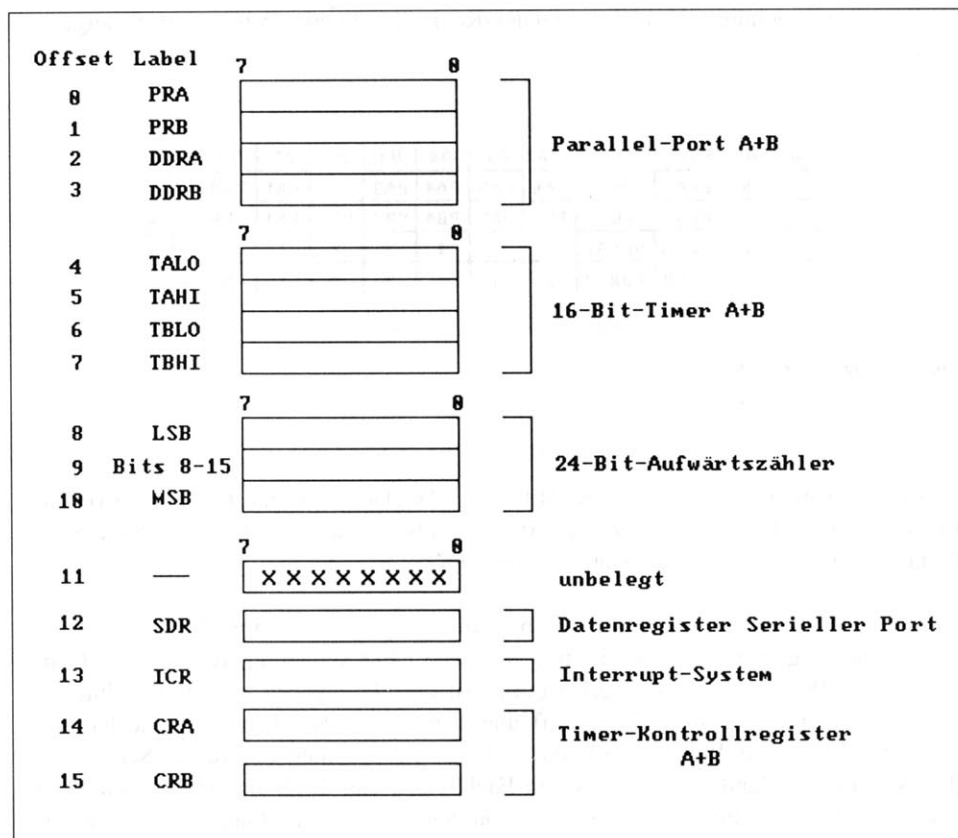


Abb. 4.3: Die Registerarchitektur der CIA-Chips

Der parallele Port

Der 8520-Baustein besitzt zwei frei programmierbare 8-Bit-Parallel-Ports (PA und PB, TTL-kompatibel).

Alle Datentransfers von und zu den Portleitungen (PA0...PA7, PB0...PB7) laufen über die Datenregister PRA bzw. PRB (PR = Peripheral-Data-Register [A und B]), während die zugehörigen Steuerregister DDRA und DDRB (DDR = Data-Direction-Register [A und B]) die Zugriffsrichtung festlegen. Damit kann jede einzelne Portleitung wahlweise

als Ein- (korrespondierendes Bit im DDRx-Register = 1) oder Ausgang (= 0) angesteuert werden.

Reg. Nr.	Bez.	D7	D6	D5	D4	D3	D2	D1	D0
8	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
2	DDRA	DDRA7	6	5	4	3	2	1	0
3	DDRB	DDRB7	6	5	4	3	2	1	0

Abb. 4.4: Der parallele Port

Im Empfangsmodus entsprechen die Bits des PRx-Datenregisters dem Pegelniveau der zugehörigen Portleitungen (Low = 0, High = 1); als Ausgang geschaltet, reflektieren die Portausgänge den Zustand des Datenregisters PRx.

Port A qualifiziert sich durch die kombinierte TTL/CMOS-Kompatibilität für die gezielte Steuerung systeminterner Leitungen (siehe LED-Schaltung, Abfrage der Gameports oder OVL = Memory-Overlay-Leitung). Port B dagegen eignet sich durch die hohe Belastbarkeit (zwei TTL-Lasten) und den integrierten Handshakemechanismus (über die Leitungen PC* und FLAG*) als Datenbus einer Centronics-Schnittstelle. Handshaking (= Händeschütteln) ist ein Fachbegriff aus der Schnittstellentechnik und beschreibt bildlich die Synchronisation von Sender und Empfänger – symbolisches Händeschütteln als Quittierung der empfangenen Daten.

Bei jedem Zugriff auf das parallele Datenregister geht der PC*-Ausgang (PC = Port Control) für einen E-Taktzyklus (= 1,4 Mikrosekunden) auf Low. Dieser negative Impuls wird von dem FLAG*-Eingang des Empfängers registriert und durch Setzen des FLG-Bit im ICR-Register markiert. Sofern der Parallel-Port als potentielle Interruptquelle zugelassen ist, kann das Auslesen und Abspeichern des PRB-Register von einer kleinen Unterbrechungsroutine übernommen werden.

Die Timer

Die 8520-Chips enthalten zwei 16-Bit-Abwärtszähler (zwei 8-Bit-Register ergänzen sich jeweils zu einem 16-Bit-Zählregister), die durch ihre umfangreichen Betriebsmodi ein breites Spektrum verschiedener Anwendungen abdecken. Das reicht von einfachen

Synchronisationsaufgaben bis hin zu komplexen Meßvorgängen (Impulslängen, Signalfrequenzen, Phasenlagen).

Register 4 bis 5 (TA-LO/HI, Timer A Low und High) enthalten den aktuellen Zählerstand von Timer A; die beiden folgenden Register 6 und 7 bilden das Zählregister für Timer B (TBLO/TBHI).

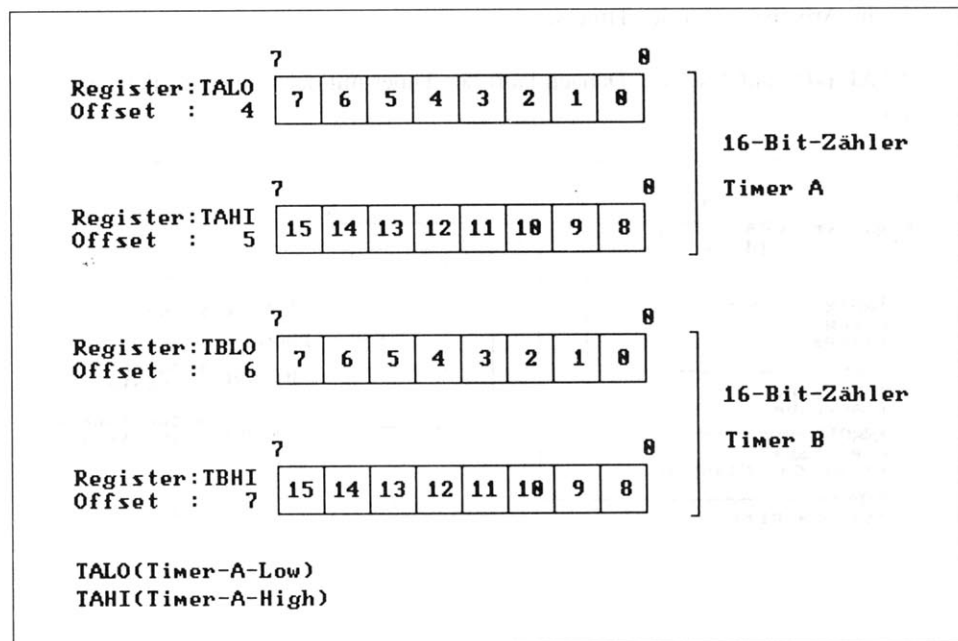


Abb. 4.5: Die Timer-Register

Das Zählintervall der Timer läßt sich innerhalb ihres 16-Bit-Wertebereichs ([0...65535]) beliebig verkleinern. Dazu wird die obere Intervallgrenze einfach in den sogenannten Prescaler (= Vorteiler) geschrieben (Unterlaufrate = Wert des Vorteilens / Zählfrequenz), der als vorgeschalteter Schlüsselspeicher Schreibzugriffe der CPU abfängt und in seinen Latchregistern puffert.

Normalerweise sieht der Initialisierungsvorgang etwa so aus:

1. Der Timer wird durch Negieren des START-Bits (Nr. 0) im CRx-Register angehalten (START = 1, STOP = 0).

- Übertragung des Low-Bytes (niederwertiger Teil des 16-Bit-Zählwertes) in das TxLO-Latchregister (Offset 4 bzw. 6).
- Übertragung des High-Bytes (höherwertiger Teil des 16-Bit-Zählwertes) in das TxHI-Latchregister (Offset 5 bzw. 7).
- Nach dem Zugriff auf das TxHI-Register wandert der Latchwert (Vorteiler) sofort in die Arbeitsregister des Timers.
- START-Bit auf 1 setzen. Danach läuft der Timer mit dem neuen Zählerstand weiter.

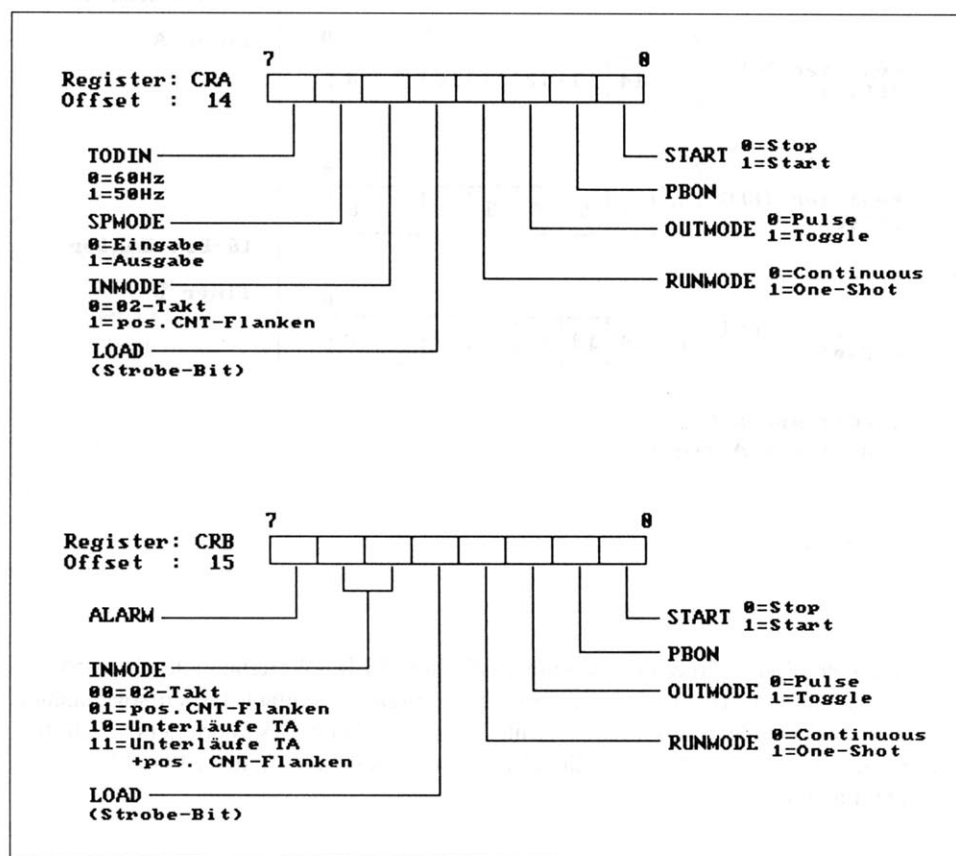


Abb. 4.6: Alles im Griff – die Kontrollregister CRA und CRB

Für einige Anwendungen ist es vorteilhaft, den Zählvorgang auch während des Schreibzugriffs nicht zu unterbrechen. In diesem Falle muß man allerdings auf die komfortable Initialisierungsautomatik (Latch wird nach Zugriff auf das High-Byte in die Timerregister kopiert) verzichten; erst nach dem nächsten Unterlauf werden die Register wieder auf den Wert des Vorteilers gesetzt. Wer nicht so lange warten will, kann die sofortige Übertragung durch das LOAD-Bit (Nr. 4, CRx) erzwingen (Bit 4 der Kontrollregister ist eine Strobleitung; sie speichert keine Parameter, sondern gibt den Steuerimpuls für den Kopiervorgang).

Ganz anders sieht es beim Lesezugriff aus. Um den Zählvorgang nicht ständig zu verfälschen, ist es sinnvoll, die Timer ungestört weiterlaufen zu lassen. Der Haken dabei (ein für derartige Bausteine durchaus typisches Problem), beim Lesen aus "voller Fahrt" drohen Übertragungsfehler.

Dazu ein Beispiel:

Der aktuelle Zählerstand sei \$0400. Lesen des High-Bytes ergibt den korrekten Wert \$04. Noch bevor der Prozessor auf das Low-Byte zugreifen kann, tritt ein erneuter Zählimpuls auf (Low-Byte jetzt: \$FF). Das Ergebnis wird verfälscht; ermittelter Zählerstand: \$04FF – Pech!

Troubleshooting – Übertragungsfehler sind vermeidbar

Durch erneuten Zugriff auf das High-Byte und anschließenden Vergleich mit dem zuvor gelesenen Wert (im Beispiel: \$04 zu \$03) kann man den Zählerstand leicht überprüfen. Stimmen die Werte nicht überein, muß der Lesevorgang eben wiederholt werden.

Die Chancen, daß es diesmal klappt, stehen nicht schlecht. Schließlich liest ein 68000 wesentlich schneller als die Timer zählen können.

Ein Timer für alle Fälle – die Programmierung

Die individuelle Programmierung der Timer erfolgt über die Steuerbits der Kontrollregister.

Die INMODE-Bits (INMODE = Eingabemodus) wählen diverse Quellen für den Zähltakt, das RUNMODE-Bit (RUNMODE = Betriebsmodus) regelt den Zählvorgang und das OUTMODE-Bit (OUTMODE = Ausgabemodus) steuert die Generierung eines (aus den Unterlaufimpulsen abgeleiteten) Taktimpulses.

Die Quellen für den Zähltakt:

- Taktimpulse auf der Phi2-Leitung (E-Takt der CPU), daraus resultiert eine Zählfrequenz von ungefähr 710 KHz. (Register CRA, Bit 5 (INMODE) = 0 / CRB, Bit 5 und 6 (INMODE) = 00)
- High-Impulse auf der CNT-Leitung; mit jeder steigenden Flanke erniedrigt sich der aktuelle Zählerstand um einen Schritt. (Register CRA, Bit 5 (INMODE) = 1 / CRB, Bit 5 und 6 (INMODE) = 01)

Timer B bietet zusätzlich noch:

- Unterläufe von Timer A (deshalb auch die Steuerleitung von Timer A zu B, siehe Abbildung 4.2). In diesem Modus bilden die Timer einen kombinierten 32-Bit-Zähler, so daß auch längere Vorgänge meßtechnisch erfaßt werden können (Register CRB, Bit 5 und 6 (INMODE) = 10).
- Unterläufe von Timer A bei gleichzeitigen High-Impulsen auf der CNT-Leitung. Die leistungsfähigste Kombination! Sie bietet die Möglichkeit, ein externes Signal mit dem frei programmierbaren Referenztakt der Timer-Unterläufe zu vergleichen (eignet sich zur Messung von Impulslängen; Register CRB, Bit 5 und 6 (INMODE) = 11).

Das RUNMODE-Bit unterscheidet zwei verschiedene Betriebsmodi:

- CONTINUOUS-Mode: So lange das START-Bit (Nr. 0) nicht verändert wird, läuft der Zählvorgang kontinuierlich weiter. Während eines Unterlaufes kopiert die Steuerlogik den gepufferten Wert des Prescalers in die Zählregister und bestätigt das Timer-Bit (TA bzw. TB = 1) im ICR. (CRA und CRB, Bit 3 (RUNMODE) = 0).
- ONE-SHOT: Der Zähler hält nach jedem Unterlauf an (START-Bit im CRx wechselt von 1 auf 0). (CRA, CRB, Bit 3 (RUNMODE) = 1)

"Timer an alle" – der Ausgabemodus

Die Fähigkeiten der CIA-Timer erschöpfen sich nicht nur in der Analyse externer Signale. Sie sind auch in der Lage, komplexe Impulse selbst zu erzeugen.

Ist das PBON-Bit (Bit 1 im CRx-Register) gesetzt, erscheinen die Timer-Unterläufe auf den Leitungen PB6 (für Timer A) und PB7 (für Timer B) des Parallel-Ports B. Da CIA-

B den Datenbus der Centronics-Schnittstelle betreibt, kann das so erzeugte Signal auch direkt an der parallelen Schnittstelle abgegriffen werden. Das OUTMODE-Bit im Kontrollregister steuert die Pegelumsetzung der Unterlaufimpulse. Kombiniert mit den umfangreichen Betriebsparametern entstehen komplexe Rechteckschwingungen mit fast beliebiger Phasenbeziehung. Potentielle Anwendungen reichen von Takt- und Synchronisationsaufgaben bis hin zur Generierung einfacher Toneffekte ("Wie wär's mit Wiener Walzer im Drei-Rechtecktakt?"). Im folgenden die Ausgangsmodi:

- Pulse-Mode: Jeder Unterlauf erzeugt einen positiven Impuls von einem Taktzyklus Länge (= 1,4 Mikrosekunden). (CRA, CRB, Bit 2 (OUTMODE) = 0)
- Toggle-Mode: Mit jedem Unterlauf ändert die zugehörige Portleitung ihr Pegelniveau (High->Low / Low->High); (CRA, CRB, Bit 2 (OUTMODE) = 1).

Die Unterschiede zwischen Pulse- und Toggle-Modus zeigen sich deutlich in der graphischen Umsetzung, wenn man gleiche Betriebsbedingungen zugrunde legt:

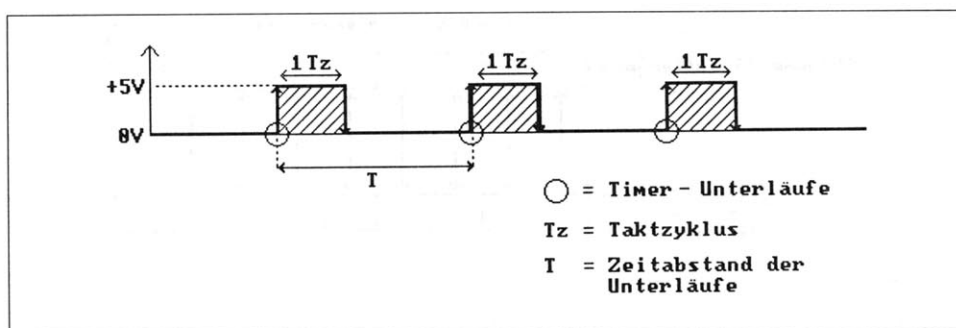


Abb. 4.7: Signalverlauf im Pulse-Mode (INMODE = 0, RUNMODE = 1, OUTMODE = 0)

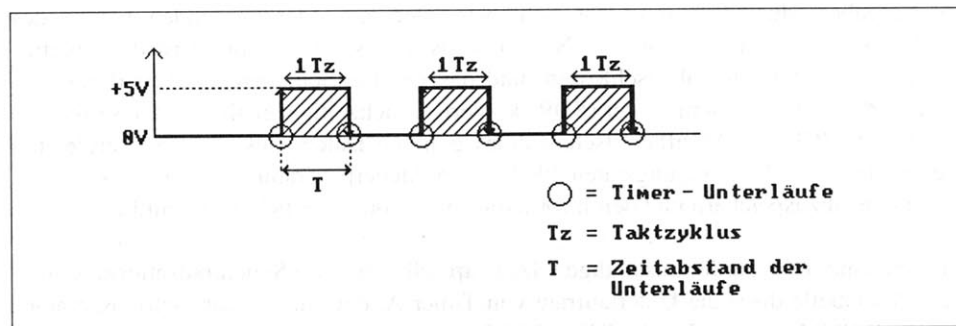


Abb. 4.8: Signalverlauf im Toggle-Mode (INMODE = 0, RUNMODE = 1, OUTMODE = 1)

Der serielle Port

Der CIA-Chip unterstützt einen seriellen Port für bitweise Datenübertragung mit breitem Spektrum unterschiedlicher Übertragungsfrequenzen. Auch dieser Port ist bidirektional; das SPMODE-Bit im Kontrollregister CRA schaltet zwischen Ein- (SPMODE = 0) und Ausgabemodus (SPMODE = 1) um.

Alle Datentransfers werden über die SP-Leitung (SP = Serial Port) abgewickelt. Das serielle Datenregister SDR (SDR = Serial Data Register, Offset 12) dient als Zwischenspeicher, während das 8-Bit-Schieberegister die bitweise übertragenen Daten zu byteorientierten Informationsblöcken zusammensetzt oder im umgekehrten Fall, byteweise gepufferte Daten in serielle Informationsströme aufspaltet. Ein TTL-kompatibles Signal auf der CNT-Leitung (siehe Abzweigung zum seriellen Port in Abbildung 4.2) dient zur Synchronisation von Sender und Empfänger.

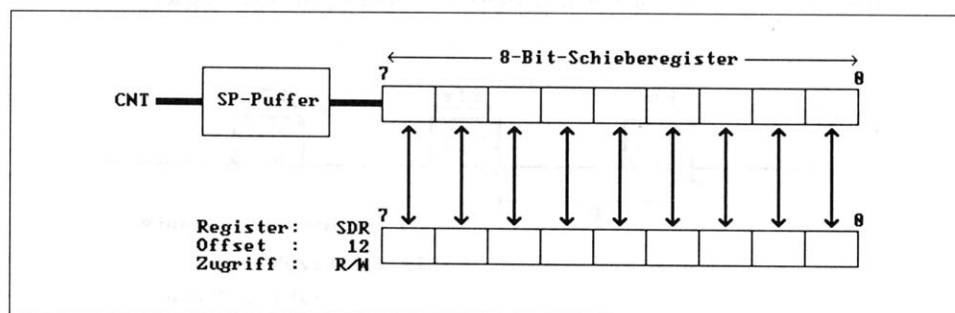


Abb. 4.9: Der serielle Port

Im Eingabemodus übernimmt das Schieberegister mit jeder steigenden Flanke des CNT-Taktes ein gültiges Port-Bit. Nach jeweils acht solcher Impulse ist die Übertragung eines Datenbytes abgeschlossen, und das gefüllte Schieberegister wird automatisch in das serielle Datenregister (SDR) kopiert. Gleichzeitig signalisiert das Setzen des SP-Bits im ICR die Abrufbereitschaft eines gültigen Datenbytes. Bei den vergleichsweise niedrigen Übertragungsraten bleibt dem Steuerprogramm genügend Zeit, das Datenbyte abzuspeichern und den Informationsfluß kontinuierlich weiterzuführen.

Im Sendemodus muß der zuständige CIA-Chip selbst für den Synchronisationstakt sorgen. Als Quelle dient die Unterlaufrate von Timer A, der unter festen Betriebsparametern läuft (INMODE = 02-Takt, RUNMODE = CONTINUOUS).

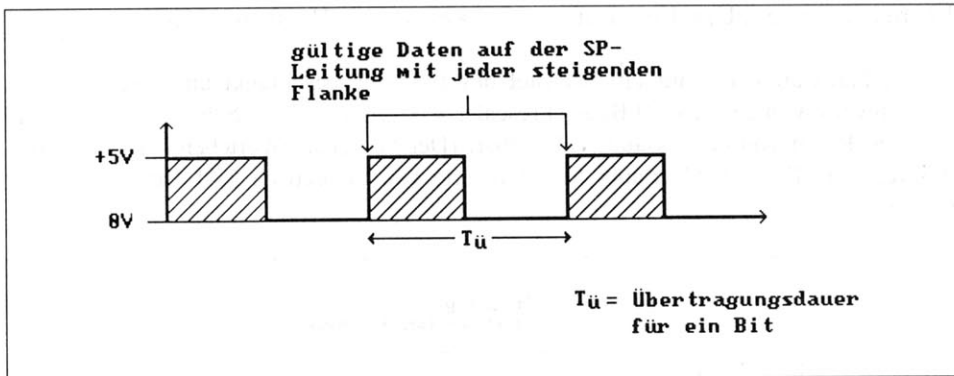


Abb. 4.10: Die Synchronisation im Empfangsmodus

Mit jedem Unterlaufimpuls wechselt die CNT-Leitung ihren Pegel. Synchron dazu bringt das Schieberegister bei fallender Taktflanke die Datenbits auf den SP-Ausgang. Nach acht Low-Impulsen ist das Schieberegister leer und das SP-Bit im ICR signalisiert die Aufnahmebereitschaft neuer Daten, so daß der Prozessor das Datenregister rechtzeitig nachladen kann. Ist die Datenübertragung abgeschlossen, wechselt die CNT-Leitung auf High (nach dem letzten Bit ist sie noch auf Low-Pegel!) und der SP-Port bleibt auf dem Pegelniveau des zuletzt ausgegebenen Datenbits.

Bits im Gleichschritt – der Synchronisationstakt

Bei vorgegebenen Betriebsparametern und einer Zählfrequenz von ca. 710 KHz (E-Takt) hängt die Übertragungsfrequenz lediglich von der Initialisierung des Timer-Prescalers ab.

Für die Häufigkeit der Timerunterläufe pro Sekunde gilt:

$$\text{Unterlauffrequenz (Fu)} = \text{Zählfrequenz} / \text{Prescalerwert} \rightarrow Fu = 710000/[0...65536]/s$$

Als zusätzlichen Faktor muß noch der Phasenabstand zweier gleichartiger Taktflanken berücksichtigt werden, da die Datenbits nur mit jedem negativen Taktimpuls (High->Low) auf der SP-Leitung erscheinen. Die Übertragung eines einzelnen Bits benötigt dementsprechend zwei Unterläufe, die effektive Übertragungsfrequenz (Baudrate = Bit pro Sekunde) entspricht also der halben Unterlaufrate von Timer A:

Übertragungsrate (FÜ) = Unterlastrate / 2 $\rightarrow$ FÜ = 710KHz/[0...65535]/2

Sie ist hardwaremäßig auf ein Sechstel des E-Taktes beschränkt und erreicht damit Spitzenwerte von ca. 118000 Baud (Prescaler = 3). Im extremen Schneckentempo sind minimal Raten von ca. 6 Baud realisierbar. (Der begrenzte Wertebereich des 16-Bit-Zählregisters [0...65535] erlaubt Minimalwerte von theoretisch 5,41 Bit/s = 710000/65535/2.)

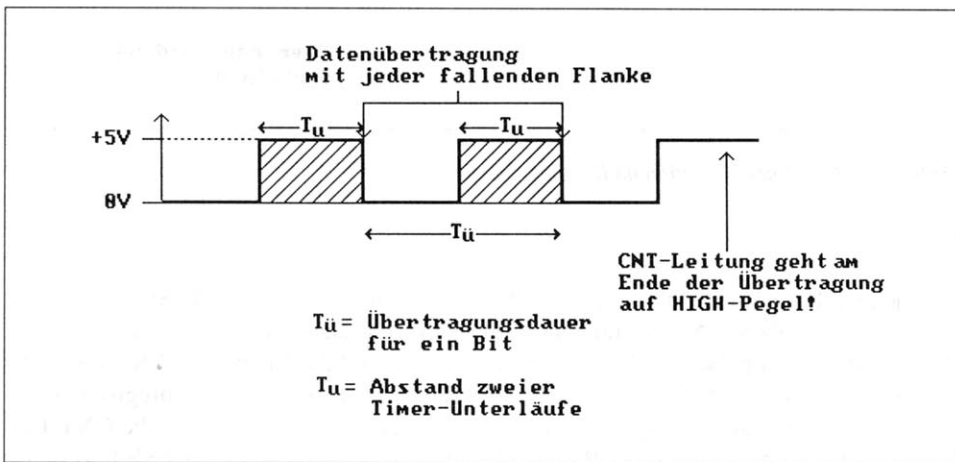


Abb. 4.11: Das Synchronisationssignal im Sendemodus

Der Event-Counter

Drei 8-Bit-Register (8 bis 10) bilden das Arbeitsregister eines 24-Bit-Aufwärtszählers, der positive Impulse (Low $\rightarrow$ High) auf dem TOD-Eingang registriert (daher auch die Bezeichnung Ereignis- bzw. Impulszähler).

Die Timer können zwar den Event-Count-Mode simulieren (INMODE = CNT, Signal auf der CNT-Leitung dient als Zähltakt), der größere Wertebereich (0...\$FFFFFF = 224) und die integrierte Alarmfunktion qualifizieren den Ereigniszähler allerdings für spezielle Synchronisationsaufgaben.

Im Gegensatz zu den extrem flexiblen Intervall-Timern kennt er keine verschiedenen Betriebsmodi (z.B. ONE-SHOT, CONTINUOUS); als Aufwärtszähler erhöht sich sein Zählerstand mit jeder steigenden Flanke des TOD-Signals. Gibt es einen Überlauf, so springen die Arbeitsregister auf 0 und der Zählvorgang geht kontinuierlich weiter. (Überläufe des Ereigniszählers werden nicht im Interrupt-Register vermerkt.)

Der Zugriff auf den Ereigniszähler ist elegant gelöst. Übertragungsfehler, wie sie beim Auslesen der Timerregister drohten, sind hier ausgeschlossen.

Dafür sorgt der folgende Mechanismus:

Die Register 10 (MSB) und 8 (LSB) (Most Significant Byte = Höchstwertiges Byte und Least Significant Byte = Niederwertiges Byte) spielen dabei eine Schlüsselrolle. Sie markieren Anfang und Ende einer Registeradressierung, so daß die Steuerlogik frühzeitig in den Initialisierungsvorgang eingreifen kann.

Beim Zugriff auf das MSB (Register 10) stoppt der Zähler automatisch und läuft erst nach Übertragung des niederwertigen Bytes (LSB, Register 8) mit dem neuen Wert weiter. Für Lesezugriffe existiert ein spezieller Zwischenspeicher (Latch), in den die komplette Registerpalette nach Adressierung des MSBs kopiert wird (ist das niederwertige Byte [Register 8] gelesen worden, wird das Latch wieder gelöscht und freigegeben).

Da der Prozessor nun nicht mehr direkt aus der Registerbank liest, sondern auf den gespeicherten Wert des Schlüsselspeichers zugreift, kann der Zählvorgang ungestört weiterlaufen.

Die Reihenfolge Register 10, dann 9 und schließlich 8 garantiert für beide Zugriffsrichtungen eine reibungslose Kommunikation von CPU und Ereigniszähler.

Der Ereigniszähler als Taktgeber

Der 24-Bit-Zähler besitzt eine integrierte Alarmfunktion, die beim Erreichen eines vom Programmierer festgelegten Zählerstands einen Interrupt auslöst.

Die Initialisierung unterscheidet sich vom normalen Schreibzugriff lediglich durch Setzen des ALARM-Bits im Kontrollregister CRA, das die Speicherung auf einen verdeckten Schlüsselspeicher umlenkt. Dieser Puffer kann nur beschrieben werden; Leseversuche sind zwecklos und landen in den Zählregistern des Timers. (Man sollte sich also den initialisierten Alarmwert gut merken.)

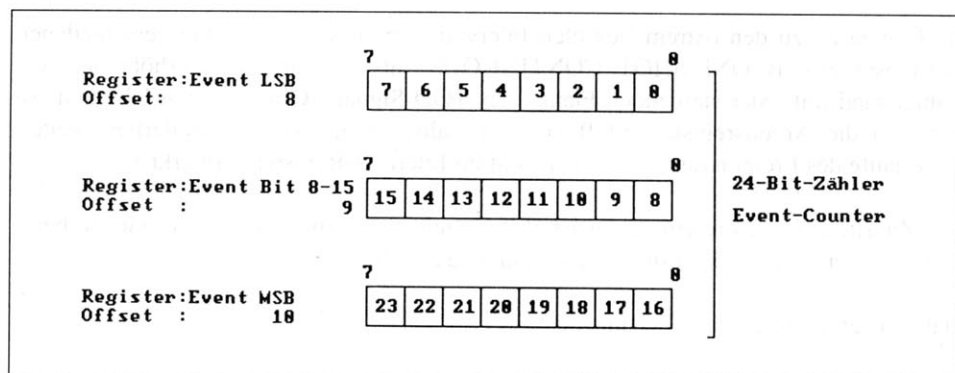


Abb. 4.12: Ereigniszähler

Während der Zähler im Alarmbetrieb läuft, wird er ständig mit dem voreingestellten Alarmwert verglichen. Bei Gleichstand setzt die Steuerlogik das korrespondierende ALARM-Bit im ICR-Register. Ist die Alarmfunktion im Interrupt-Kontrollregister entsprechend maskiert, entsteht ein Zeittakt, der in einem exakt festgelegten Verhältnis zum Referenzimpuls auf der TOD-Leitung steht. So nutzt der Amiga den Ereigniszähler, um aus den gleichmäßigen Impulsen der Netz und Zeilenfrequenz (PAL 15625Hz) Synchronisationstakte für das Betriebssystem abzuleiten.

"Break Dance" – die CIA-Interrupts

Das Interruptsystem unterstützt alle wichtigen Komponenten der CIA-Chips. Timer A und B, Serial-, Parallel-Port und der Event-Counter sind über fünf Kanäle (siehe Abbildung 4.2) direkt mit ICR-Datenregister verbunden. Es bildet die Schnittstelle zwischen Interruptlogik und CIA-Funktionen. Jede potentielle Interruptquelle (Timerunterläufe, SDR-Register gefüllt/leer, FLAG-Impulse, Ereigniszähler-Alarm) hat hier ein zugehöriges Bit, das ihren Interruptstatus reflektiert.

Die Kommunikation von Interruptsystem und Funktionsblöcken erfordert keine aufwendigen Anforderungsprotokolle; ein interruptfähiges Ereignis wird durch Setzen (= 1) des korrespondierenden Bits im Datenregister vermerkt. Die Entscheidung über die weitere Verarbeitung fällt nach dem Vergleich mit dem ICR-Maskenregister, dessen Bits (bis auf die unterschiedliche Funktion von Bit 7) exakt mit den Interruptbits des Datenregisters übereinstimmen.

Akzeptiert werden nur Anforderungen, deren Enable-Bit auch im Maskenregister logisch wahr ist (= 1). Auf diese Weise kann der Programmierer gezielt einzelne Interruptquellen in den Verarbeitungsprozeß ein- bzw. ausblenden.

In der Registermap fallen Daten- und Maskenregister als Interrupt-Kontrollregister (ICR) an Offset-Position 13 zusammen; einziges Unterscheidungskriterium ist die Zugriffsrichtung. Das Maskenregister kann nur beschrieben werden, während das Datenregister ein Nur-Lese-Register ist. Die Register-Decodierung der CIAs erkennt die Adressierungsrichtung und lenkt die CPU-Zugriffe auf das entsprechende Register um.

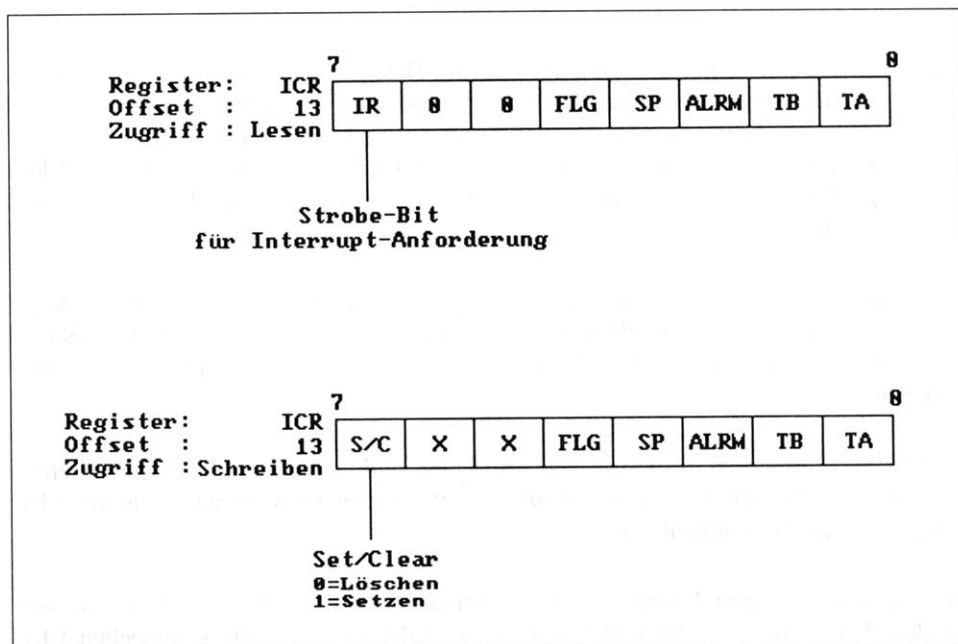


Abb. 4.13: Masken- und Datenregister des Interruptsystems

Für die gezielte Manipulation einzelner Bits unterstützt das Maskenregister den bewährten Set/Clear-Mechanismus. Bit 7 (S/C = Set/Clear) schaltet zwischen Löschen (= 0) und Setzen (= 1) um.

Nach bereits bekanntem Schema können nur die 1er-Bits im Maskenwort das Zielregister verändern, die Positionen der 0-Bits bleiben unberührt.

Ein Beispiel:

Es sollen nur Unterbrechungsanforderungen der beiden Timer von CIA-A akzeptiert werden. Im Maskenregister ist Bit 0 für Timer A zuständig, Bit 1 für Timer B.

```
lea $BFE001,a5           ;Basisadresse der
                           CIA-Register nach a5
move.b #%10000011,13(a0) ;Bits 0 und 1 im
                           Maskenregister
                           einblenden (Set-Bit = 1)!
```

Verändert ein Interruptkanal den Bitzustand im Datenregister, überprüft das Interruptsystem die Unterbrechungsanforderung durch logische Verknüpfung von Masken- und Datenregister; nicht zugelassene Quellen werden ausgeblendet. Bei gültiger Interruptanforderung setzt sie das Interrupt-Request-Bit (= IR, Bit 7) im Datenregister und legt den IRQ*-Ausgang auf Low, der so lange auf diesem Pegelniveau bleibt, bis das Datenregister zurückgesetzt wird.

Im Amiga-System ist IRQ* mit den Interruptleitungen INT6* (CIA-A) bzw. INT2* (CIA-B) des Customchips Paula verbunden, der die Unterbrechungsanforderung gemäß ihrer Eingangspriorität sortiert und zur weiteren Auswertung in das INTREQ-Register schreibt.

Der IRQ*-Ausgang ist ein Open-Drain-Anschluß und eignet sich durch den externen Pullup-Widerstand (hält die Leitung auf High-Pegel) für die Kopplung mehrerer CIA-Chips auf einer Interruptleitung.

Vor diesem Hintergrund wird auch die Funktion des IR-Bits deutlich. Ohne hardwaremäßige Unterscheidung der CIA-Interrupts muß der Prozessor die gekoppelten CIA-Chips in einer Art Umfrageaktion (Fachbegriff: polling) einzeln abfragen. Das gesetzte IR-Bit verrät den auslösenden Baustein.

Der Lesezugriff auf das ICR-Datenregister löscht sämtliche Einträge und bereitet es für weitere Interruptanforderungen vor. Durch diesen Mechanismus entfällt ein nachträgliches Ausblenden der Interruptquelle (siehe INTREQ-Register von Paula); dem Programmierer bleibt eine unnötige Fehlerquelle erspart.

Die Systemeinbindung der CIA-Hardware

Im Amiga-System kümmern sich gleich zwei dieser vielseitigen Allroundtalente um ein vielfältiges Aufgabenfeld. CIA-A ist mit den niederwertigen Teil des CPU-Datenbusses (D0...D7) verbunden und liegt deshalb an ungeraden Adressen, CIA-B ist an die höherwertige Hälfte (D8...D15) angeschlossen und spricht auf gerade Adressen an.

Eine genaue Analyse der Adressierungsmechanismen und Kommunikation zwischen Prozessor und CIAs folgt im nächsten Kapitel (Nr. 5).

Die CIAs – Mädchen für alles

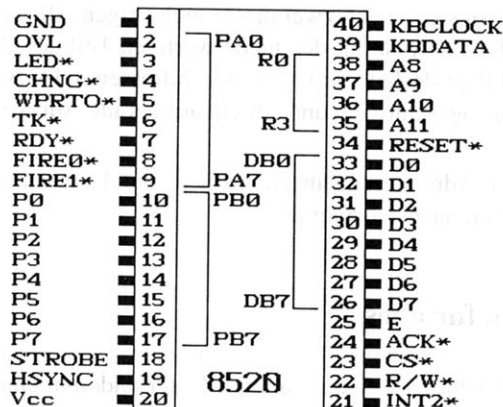
Der Terminplan der 8520-Chips ist dicht gedrängt. Unter anderem sind sie bei der Abfrage der Game-Ports beteiligt, übernehmen die Steuerung der Schnittstellen und Diskettenlaufwerke, regeln die Kommunikation von Rechner und Tastatur und synchronisieren diverse Systemfunktionen, ganz zu schweigen von solch banalen Aufgaben wie der Hell/Dunkel-Schaltung der Power-LED. Es gibt kaum eine I/O-Funktion, bei der sie nicht ihre "Transistoren im Spiel haben".

Eine derartig intensive Nutzung führt zwangsläufig dazu, daß nicht alle Funktionen für Anwendersoftware zu Verfügung stehen. Einige Ressourcen werden vom Betriebssystem nur unter bestimmten Bedingungen benötigt, können ansonsten aber frei programmiert werden, andere Funktionsblöcke sind ausschließlich für Systemsoftware reserviert.

Um Kollisionen zwischen Betriebssystem und eigenen Programmen zu vermeiden, ist es ratsam, in unklaren Fällen lieber über Bibliotheksfunktionen und Systemstrukturen auf die CIA-Hardware zuzugreifen. Für einfache Portabfragen (z.B. Maustate, Feuerknopf) oder zeitkritische Anwendungen (schnelle Datenübertragung von und zu den Schnittstellen) lohnt es sich, die 8520-Chips direkt anzusprechen.

Voraussetzung ist allerdings eine genaue Kenntnis ihrer Aufgaben und der Registerbelegung durch das Betriebssystem.

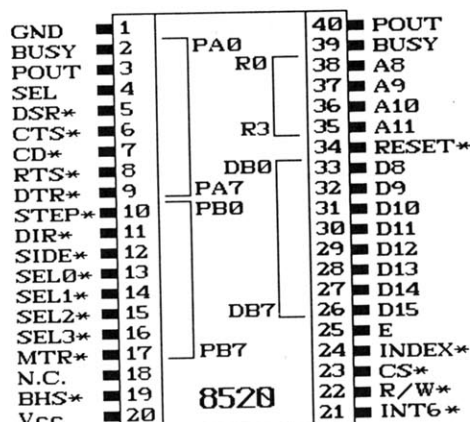
Beschaltung CIA-A



Adresse	Offset	Label	7	6	5	4	3	2	1	0	
SBFE001	8	PRA	FIRE0*	FIRE1*	RDY*	TK0*	WPRO*	CHNG*	LED*	OVL	Parallel Port A
SBFE201	2	DDRA	0	0	0	0	0	0	1	1	
			7	6	5	4	3	2	1	0	
SBFE101	1	PRB	Centronics								Parallel Port B
SBFE301	3	DDRB	Schnittstelle								
			7							0	
SBFE401	4	IALO	Länge des Handshake-Signals								Timer A
SBFE501	5	IAHI	für den Tastaturprozessors								
			7							0	
SBFE601	6	TBLO	Diverse								Timer B
SBFE701	7	TBHI	Synchronisationsaufgaben								
			7							0	
SBFE801	8	LSB	Schwingungsimpulse des Wechselstroms								Ereignis-zähler
SBFE901	9	8-15	(Ticks vom Netzteil)								
SBFEA01	10	MSB	Frequenz 50Hz								
			7							0	
SBFEB01	11	SDR	Tastencodes der Tastatur								Serieller Port
			7							0	
SBFEC01	12	ICR	X	X	X	0v1	1	1	1	1	
			7							0	
SBFEE01	14	CRA	entsprechend								
			7							0	
SBFEF01	15	CRB	Timer-Funktionen								

Abb. 4.14: Systemeinbindung und Registerbelegung CIA-A

Beschaltung CIA-B

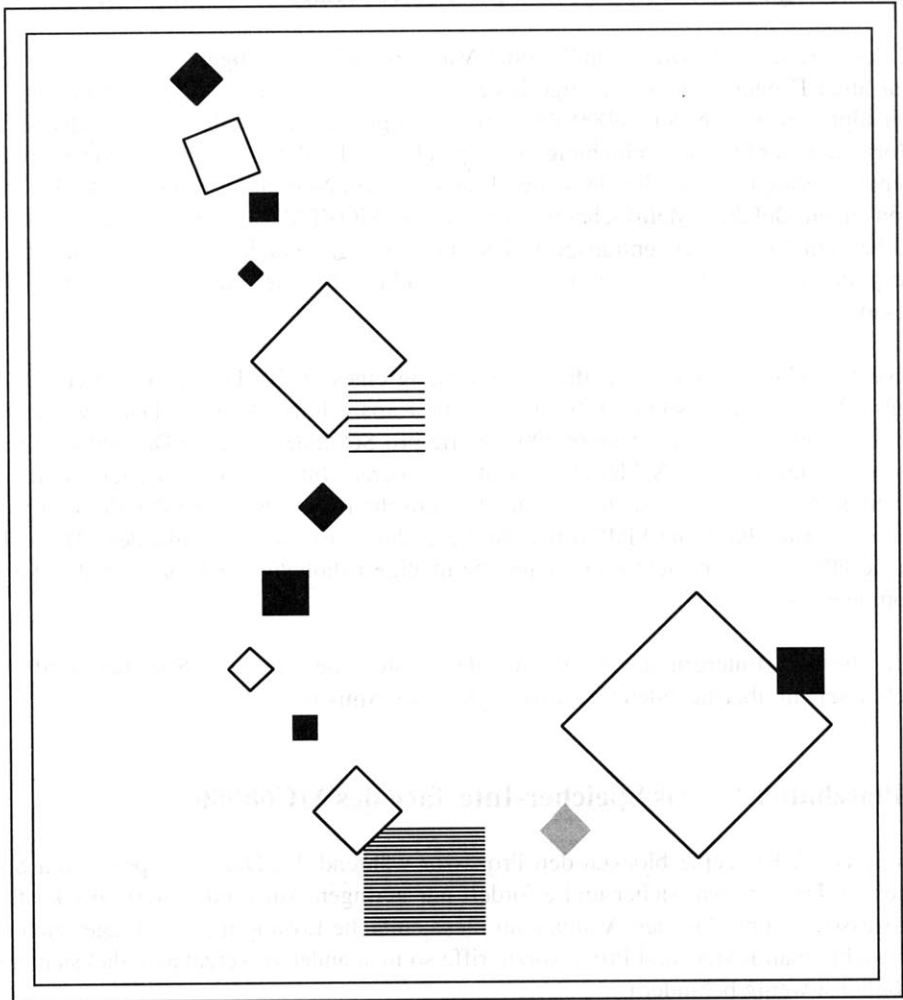


Adresse	Offset	Label	7	6	5	4	3	2	1	0		
SBFD000	0	PRA	DTR *	RTS *	CD *	CTS *	DSR *	SEL	POUT	BUSY	Parallel Port A	
SBFD200	2	DDRA	1	1	0	0	0	0	0	0		
SBFD100	1	PRB	MTR *	SEL3 *	SEL2 *	SEL1 *	SEL0 *	SIDE *	DIR	STEP *	Parallel Port B	
SBFD300	3	DDRB	1	1	1	1	1	1	1	1		
SBFD400	4	TALO	7 6 5 4 3 2 1 0 zeitweise vom									Timer A
SBFD500	5	TAHI	Betriebssystem belegt									
SBFD600	6	IBLO	7 6 5 4 3 2 1 0 Blitter									Timer B
SBFD700	7	IBHI	Synchronisationsaufgaben									
SBFD800	8	LSB	7 6 5 4 3 2 1 0 zählt									Ereignis - zähler
SBFD900	9	8-15	horizontale Synchronisationsimpulse									
SBFDA00	10	MSB	Frequenz 15625Hz									
SBFDB00	11	SDR	7 6 5 4 3 2 1 0 nicht benutzt / SP-Leitung totgeschaltet									Serieller Port
SBFDC00	12	ICR	X	X	X	0v1	1	1	1	1		
SBFDE00	14	CRA	7 6 5 4 3 2 1 0 entsprechend									
SBFDF00	15	CRB	7 6 5 4 3 2 1 0 Timer-Funktionen									

Abb. 4.15: Systemeinstellung und Registerbelegung CIA-B

Kapitel 5

Vom DMA - System zur Speicherorganisation



Das DMA-System unter der Lupe

Ein 7,1-MHz-68000er mit seinem 16-Bit-Datenbus und dem 4-Takt-Buszyklus hat eine Busbandbreite von maximal 3,5 MByte/s ($= 7,1\text{MHz} / 4\text{Tz} \times 2\text{ Bytes}$). Keine schlechten Werte; doch reicht das aus, um mit der Datenflut der Customchips fertig zu werden?

Der Amiga ohne DMA? – ein paar theoretische Überlegungen

Videoshifter, Audio-D/A-Wandler oder Massenspeicher verlangen eine kontinuierliche, vor allen Dingen aber rechtzeitige Datenversorgung. Für diese Aufgabe muß ein Universalprozessor wie der 68000 erst einmal programmiert werden. Der relativ simple Vorgang einer Datenverschiebung vom Speicher in die Pufferregister der Systemkomponenten verlangt entsprechende Steuersequenzen – im Normalfall pro bewegter Information einen Befehl. (Mehrfachanweisungen wie MOVEM seien hier ausgeklammert!) Außerdem läuft der Datentransport nicht direkt von der Quelle zum Ziel, sondern, aufgespalten in einen Lese- und einen Schreibzyklus, über die internen Register des Prozessors.

Eine typische Instruktion für die Verschiebung eines 16-Bit-Datenwortes benötigt bereits 12 Taktzyklen (MOVE (a0)+,(a1)+); die Kapazität mit der reine Daten verschoben werden sinkt auf bescheidene 600000 Worte pro Sekunde. Größere Datenblöcke erfordern zusätzlich eine Schleifenkonstruktion, deren Interpretation weitere Zeit verschlingt. Maximale Busbandbreite (als theoretische Höchstbelastung der Busse) und effektiver Datendurchsatz klaffen bei einer Standard-CPU weit auseinander. (Diese Tatsache läßt sich auch nicht durch eine übermäßige Erhöhung der Prozessortaktfrequenz kompensieren.)

Vor diesem Hintergrund erweist sich das System des direkten Speicherzugriffs als Schlüssel zur überragenden Geschwindigkeit des Amigas.

"Platzhalter" – das Speicher-Interface des MC68000

Viele DMA-Konzepte blocken den Prozessor während des Datentransports vom Speicher ab. Das ist zwar sicher und erfordert nur geringen Aufwand, opfert aber kostbare Prozessorleistung. Für den Amiga kam diese einfache Lösung nicht in Frage; vielmehr versuchte man DMA- und Prozessorzugriffe so ineinander zu verzahnen, daß sich beide möglichst wenig behindern.

Voraussetzung dafür ist die genaue Analyse des Zugriffsverhaltens einer 68000er-CPU, die den Speicher zwar wesentlich häufiger anspricht als 8-Bit-Prozessoren, von einer 100%igen Busauslastung aber noch weit entfernt ist. Speicherzyklen finden beim Lesen von Befehlen, Operanden und deren Adressen statt, die restliche Zeit verbringt die CPU mit der internen Datenmanipulation.

Je komplexer eine Instruktion desto größer der Anteil dieser internen Arbeitsphasen an der Gesamtdauer eines Befehlszyklus. (Der Divisionsbefehl DIVS benötigt allein für die arithmetische Ausführung 154 Taktzyklen – Zeit, in der das Speicher-Interface inaktiv bleibt.)

Trotz Befehls-Prefetch (Vorhol-Mechanismus zur Steigerung des Durchsatzes), intensivem Pipelining (dreistufige Warteschlange für die Befehlsverarbeitung), verdeckter Adreßberechnung (Address-Unit wird parallel betrieben) und optimiertem Businterface (BIU = Bus Interface Unit) bremsst die mikrocodierte Befehlsausführung so stark, daß der Prozessor effektiv nur jeden zweiten Buszyklus für den Zugriff auf das Speichersystem nutzen kann (Durchschnittswert für sämtliche Instruktionen des Befehlssatzes).

Einige Befehle (Datentransport-Instruktionen) kommen jedoch dem Ideal einer vollständigen Busauslastung sehr nahe.

Der MOVE-Befehl beispielsweise, die mit Abstand häufigste Instruktion des 68000-Code, hat entscheidenden Einfluß auf die Ablaufgeschwindigkeit von Programmen. Seine einfache Struktur erlaubt die lückenlose Überlappung von Befehlsdecodierung, Adreßberechnung (die Ermittlung der effektiven Adresse schmilzt auf den reinen Lesevorgang zusammen) und Speicherzugriffen. Das bringt die Busse kurzfristig auf Spitzendurchsatz, so daß die internen Taktphasen in der Summe der benötigten Takte für die Befehlsausführung nicht auftauchen.

Befehlszyklus einer MOVE-Instruktion:

- Einlesen des Befehls : 4 Tz
- Ermittlung der Quelladresse (EAq) : Lesen des Adreßwertes (max. 8 Tz)
- Ermittlung der Zieladresse (EAz) : Lesen des Adreßwertes (max. 8 Tz)
- Lesen des Datenwertes (Quelle) : 4 bis 8 Taktzyklen (Byte, Langwort)
- Schreiben des Datenwertes (Ziel) : 4 bis 8 Taktzyklen (Byte, Langwort)

Execution-Time/Ausführungszeit = $4 \text{ Tz} + \text{EAq} + \text{EAz} + \text{Dq} + \text{Dz}$

Doch selbst bei extrem hoher Busauslastung (MOVE) benötigt ein 68000er bei 7,1 MHz Takt mindestens 560 ns ($4 \times 140 \text{ ns}$), um ein Wort aus dem Speicher zu holen

bzw. zurückzuschreiben. Der DMA-Controller ist doppelt so schnell und schafft im gleichen Zeitraum zwei komplette 16-Bit-Zugriffe.

Folgerichtig orientierte man sich nicht mehr an dem vergleichsweise großzügigen Bustiming der CPU, sondern den schnellen 280-ns-Speicherzyklen des DMA-Systems. Um die Freiräume im Speicherzugriffsschema des 68000-Prozessors auszunutzen, entwarfen die Entwickler ein komplexes Zeitmuster zur wechselseitigen Verteilung von CHIP-RAM-Speicherzyklen auf Zentralprozessor und DMA-Controller. Damit erreicht das System – quasi aus dem Stand – eine Busbandbreite von beachtlichen 3,5 MWords pro Sekunde.

"Slotmachine" – das Timing des DMA-Systems

Grundlage des Bustimings bildet der Verlauf einer PAL-Rasterzeile, die inklusive Strahlrücklaufzeit (horizontale Austastlücke) ungefähr 63,5 Mikrosekunden dauert.

Aufgelöst in die schnellen 1-Takt-Buszyklen des DMA-Controllers, zerfällt sie rein rechnerisch in 227,5 Speicherzyklen, von denen hardwarebedingt nur die ersten 225 Zyklen für Speicherzugriffe zur Verfügung stehen. Kontinuierlich durchnummeriert repräsentieren sie eindeutig festgelegte Timeslots, die von der DMA-Kontrolllogik anhand aktueller Systemzustände und Zugriffsprioritäten verplant werden.

Die Bussteuerung unterscheidet gerade bzw. ungerade Timeslots. Während letztere ausschließlich für den DMA-Controller reserviert sind, darf der Prozessor die geraden Zyklen nutzen (sofern diese nicht bereits von Blitter-, Copper- oder Bitplane-DMA belegt sind). Die Privilegierung der ungeraden Timeslots erlaubt eine reibungslose Datenversorgung von DMA-Quellen hoher Priorität. Hier liegen Audio-, Disk-, Sprite- und Bitplane-DMA – Funktionen, die aufgrund ihrer herausragenden Stellung als Mensch/Maschine- bzw. Massenspeicher-Schnittstelle auf ungestörte Speicherzugriffe angewiesen sind.

Sound- und Video-Hardware reagieren extrem empfindlich auf das Abreißen eines gleichmäßigen Datenstroms. Um knacksende Geräusche oder diffuse Bildschirmdarstellungen zu verhindern, haben diese DMA-Quellen fest zugewiesene Timeslots, die ihnen im aktiven Zustand uneingeschränkt zur Verfügung stehen; Konflikte mit der CPU sind ausgeschlossen.

Mit den geraden Timeslots erhält der 68000er zwar nur jeden zweiten Speicherzyklus (280 ns), liegt damit aber genau im Rahmen seiner minimalen Buszykluszeit ($4T_z = 560\text{ns}$). Bei exakter Synchronisation sind also keine Zugriffsverzögerungen zu erwarten.

Allerdings korrespondiert nicht jeder Befehlszyklus mit diesem Schema. Instruktionen, die sich nicht in ganzzahlige Vielfache des 4-Takt-Buszyklus auflösen lassen (deren Ausführungszeit also nicht glatt durch 4 teilbar ist), landen zwangsläufig auf den ungeraden Timeslots des DMA-Controllers. Derartig "entgleiste" Zugriffe bringen die filigrane Zyklusverzahnung schnell zum Einsturz, so daß eine spezielle Kontrollogik für die genaue Einhaltung der zugewiesenen Timeslots sorgen muß.

"Weichenstellung" – wenn der Prozessor auf die "schiefe Bahn" gerät

Ständig überwacht sie das Pegelniveau des Bussignals AS* (= Address Strobe) und vergleicht es mit der Phasenlage des C1- bzw. C3-Taktes (Die C-Takte sind für das DRAM-Timing verantwortlich). Liegen alle Impulse auf Low, läuft der Buszyklus nach gewohntem Schema ab (IN SYNC), ein Eingriff ist nicht erforderlich, alle anderen Kombinationen signalisieren ein Abdriften aus der Synchronisation (OUT SYNC).

In diesem Fall hält die Speichersteuerung das Quittungssignal DTACK* zurück (zur Erinnerung: es bestätigt die Bereitschaft zur Datenübergabe) und lenkt den Speicherzyklus auf den nächsten freien Timeslot um. Nur wenige Befehle haben jedoch eine kosmetische Korrektur nötig, so daß die resultierenden Leistungseinbußen der CPU-Performance entsprechend gering sind.

"Rush Hour" – Verkehrsplanung auf dem CHIP-RAM-Bus

Kritisch wird es bei aktivierter Copper- bzw. Blitter-DMA, denn für deren Datenversorgung ist auf den ungeraden Timeslots einfach kein Platz mehr vorhanden. Hier stößt das System wechselseitiger Speicherzugriffe an seine Grenzen – zu viele DMA-Quellen müssen innerhalb einer Rasterzeile bedient werden.

Copper- und Blitter brechen damit zwangsläufig in die Domäne der CPU ein und konkurrieren mit dem Prozessor um die geraden Speicherzyklen. Da die CPU auf der Prioritätsskala der Zugriffsverteilung ganz unten rangiert, steht der Verlierer des Konkurrenzkampfes bereits fest.

Zugriffspriorität: CPU < Blitter < Copper

Während sich der Copper vergleichsweise bescheiden gibt und mit zwei bis drei Slots pro Befehlszyklus zufrieden ist, kennt der Datenhunger des Blitters theoretisch keine Grenzen. Ohne eingebaute Notbremse würde er im aktiven Zustand sämtliche geraden Speicherzyklen an sich reißen. Versucht der 68000 einen bereits belegten Speicherzy-

klus zu nutzen, blockt die Buszugriffssteuerung ab und quittiert DTACK* erst nach Beendigung des DMA-Zugriffs (gaukelt der CPU quasi einen extrem langsamen Speicher vor).

Glücklicherweise verhindert ein Schutzmechanismus (Erniedrigen der Blitterpriorität, BLTPRI = 0), daß der Prozessor total vom CHIP-RAM-Bus abgeschnitten wird. Hat die CPU bereits drei Buszyklen verpaßt, zwingt die Bussteuerung den Blitter durch Aktivierung der BLS*-Leitung (Blitter Slow Down), die aktuelle Operation unverzüglich zu unterbrechen und den folgenden Timeslot freizugeben.

"Wanderdüne" – die Bitplane-DMA

Noch enger wird es für den Prozessor bei hochauflösender und/oder besonders vielfarbiger Grafikdarstellung, da nun auch die Bitplane-DMA teilweise auf gerade Buszyklen ausweichen muß.

Das liegt an dem unterschiedlichen Datenbedarf der diversen Videomodi. Die Bitplane-DMA arbeitet wort-orientiert (16 Bit), liest also mit jedem Speicherzugriff gleich 16 Grafikpunkte (Pixels) auf einmal.

Zur Ermittlung des Farbwertes benötigt sie Informationen aus allen aktiven Bitplanes, denn erst die Kombination der korrespondierenden Pixels liefert den Zeiger auf das Farbregister des Color-Lookup-Table. Wie häufig die Bitplanes dazu ausgelesen werden müssen, hängt von der Pixelfrequenz ab.

In der hohen Zeilenauflösung (> 384 Spalten) werden die einzelnen Pixel mit einer Frequenz von 14,2 MHz ausgegeben (also pro Buszyklus vier Punkte), die niedrige Auflösung arbeitet mit 7,1 MHz und zwei Punkten pro DMA-Zyklus.

Da bleibt der Bitplane-DMA nicht viel Zeit; für einen kompletten Lesezyklus stehen lediglich vier (16 Punkte im Puffer / 4 Punkte pro Speicherzyklus) bzw. acht Buszyklen zur Verfügung (16-Punkte-Puffer / 2 Pixel pro 280ns).

Ohne Teile der Bitplane-DMA auf gerade Timeslots verlagern zu müssen, verkraftet das System im Low-Resolution-Mode nicht mehr als 16 (= 24, 4 aktive Bitplanes), im High-Resolution-Mode sogar nicht mehr als 4 Farben.

Alles was darüber hinausgeht (32 Farben, HAM = Hold And Modify, Half Bright) stiehlt gerade Speicherzyklen des Prozessors – im Extremfall bis zu 50%.

Die PAL-Rasterzeile als Timing-Grundlage

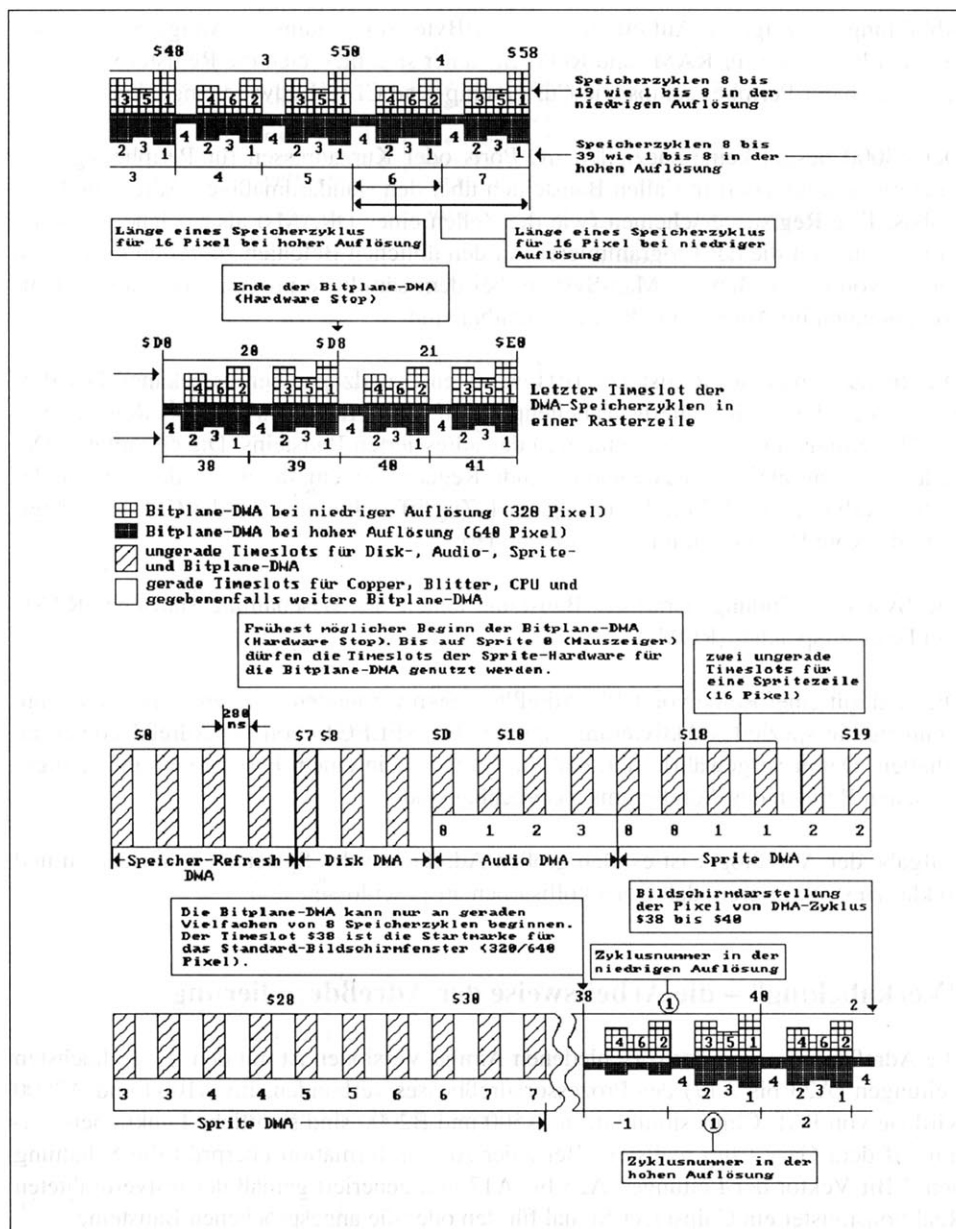


Abb. 5.1: Das Timeslot-Allocation-System

Die Speicherbelegung

Abbildung 5.2 zeigt die Aufteilung des 16-MByte-Adreßraums im Amiga-System. Neben den Bereichen für RAM- und ROM-Speicher sind hier auch die Registerbänke programmierbarer Peripheriechips wie Customchip- und CIA-Hardware eingeblendet.

Der 68000 besitzt keine speziellen I/O-Ports oder Kurzadressen für Peripheriegeräte, sondern kommuniziert mit allen Bausteinen über den standardmäßigen Adreß- und Datenbus. Ihre Register erscheinen (wie die Zellen eines DRAMs) als Speicherplätze im Adreßraum, auf die der Programmierer mit den üblichen Befehlen zugreifen kann. Man spricht von einem Memory-Map-System, bei dem sämtliche programmierbare Systemkomponenten im Adreßraum der CPU sichtbar sind.

Die Realisierung eines Registerzugriffs ist in den Grundzügen bereits bekannt. Der Prozessor legt die binär-codierte Tabellenposition eines Registers auf die Register-SELECT-Leitungen (= Auswahlleitungen) des adressierten Bausteins. Die chipinterne Decoderschaltung aktiviert das entsprechende Register, indem sie es mit dem Systemdatenbus verbindet, so daß die Daten – je nach Zugriffsrichtung – von der CPU zum Register oder vom Register zum Prozessor wandern.

Die Systemeinbindung derartiger Bausteine ähnelt der Beschaltung statischer RAMs und Festwertspeicher (ROMs):

Die sind mit einer Reihe von CPU-Adreßleitungen verbunden, reagieren aber erst dann, wenn sie ein spezielles Aktivierungssignal (Chip SELECT) von der Adreßdecodierung erhalten. Damit ist gewährleistet, daß die Chips nur in einem bestimmten, ihnen zugewiesenen Bereich des Adreßraums ansprechbar sind.

Aufgabe der Adreßlogik ist es, den großen Adreßraum des Systems zu entflechten und so klar zu strukturieren, daß Datenkollisionen ausgeschlossen sind.

"Verkabelung" – die Arbeitsweise der Adreßdecodierung

Die Adreßdecodierung der verschiedenen Amiga-Versionen ist mit den sieben höchsten Leitungen (A23 bis A17) des Prozessoradreßbusses verbunden. Im A1000 und A2000 wird sie von PAL-Chips simuliert, im A500 und B2000 sind sämtliche Funktionen zentral auf dem Gary-Chip realisiert. Bei jeder Adreßinformation überprüft die Schaltung den 7-Bit-Vektor der Leitungen A23 bis A17 und generiert gemäß der festverdrahteten Reaktionsmuster ein Chipselect-Signal für den oder die angesprochenen Bausteine.

Mit sieben Leitungen läßt sich der Adreßraum theoretisch in 128 ($= 2^7$) Blöcke zu je 128 KByte aufteilen; praktisch ist diese hohe Auflösung jedoch nicht erforderlich und wird auch nicht annähernd erreicht.

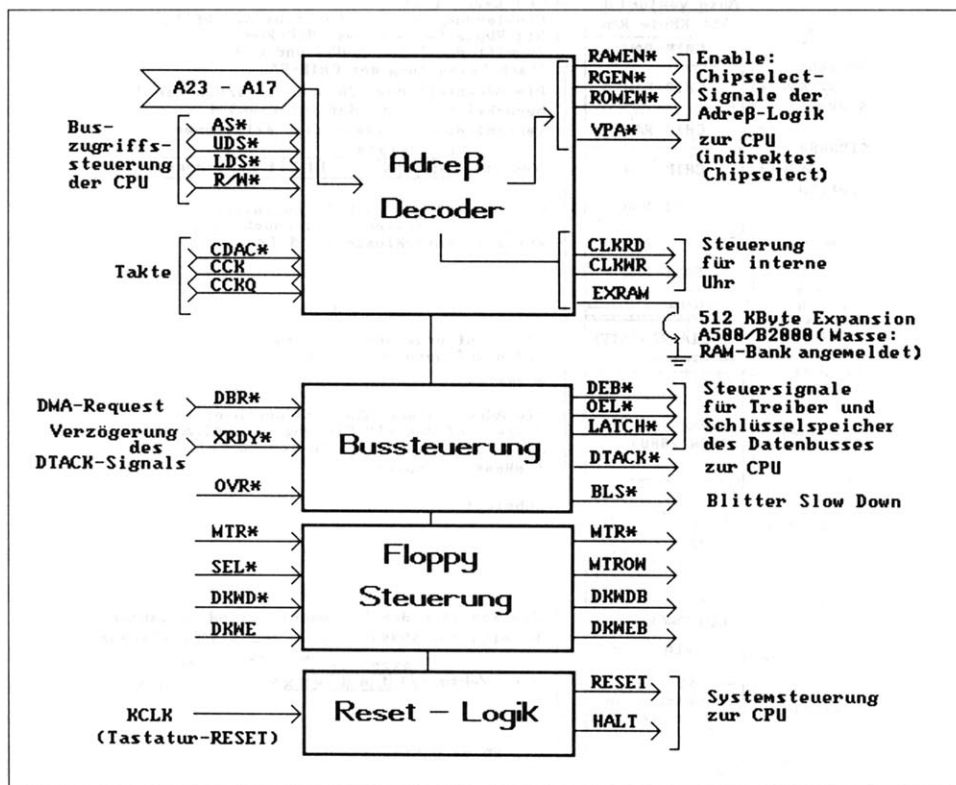


Abb. 5.3: Blockschaltbild Gary

Bei der relativ kleinen Zahl unabhängiger, für die CPU unbedingt sichtbarer Funktionsgruppen (CHIP-RAM, FAST-RAM, ROM, Customchips, CIAs, Expansion-Slots, Echtzeituhr) genügen der Adreßlogik bereits in vielen Fällen die höchsten vier Adreßleitungen (A23 bis A20), so daß die meisten Bausteine effektiv auf MByte-Grenzen liegen (Basisadresse = $A23...A20 * 16\,5 \text{ mit } 16\,5 = 2^{20} = 1 \text{ MByte}$).

In der schematischen Darstellung der Speicherbelegung (Abbildung 5.2) stößt man häufig auf Spiegelungen eines Speicherbereichs (so taucht das CHIP-RAM gleich vier mal

auf) – ein Effekt, der sich zwangsläufig aus der begrenzten Anzahl von Adreßleitungen ergibt, die vom Decoder untersucht werden.

Der Adreßbus der 68000-CPU hat eine Breite von 24 Bit (Besonderheit: Aufspaltung von A0 in UDS* und LDS*), die zentrale Adreßdecodierung ist aber nur mit sieben Leitungen verbunden. In dieser Konfiguration sind Mehrfachbelegungen unvermeidbar, denn je weniger Adreßsignale für die Selektion der chipinternen Register benötigt werden, desto mehr Leitungen können während des Adressierungsvorgangs einen beliebigen Zustand annehmen. Aus diesem Grund liegen die meisten Basisadressen nicht an festen Positionen, sondern innerhalb mehr oder weniger weit gefaßter Intervallgrenzen.

Um die Adressierung möglichst übersichtlich und durchschaubar zu halten, ist man bemüht, die Register-SELECT-Leitungen mit aufeinanderfolgenden Adreßsignalen der CPU zu verbinden. (Man stelle sich die chaotischen Verhältnisse in zufällig geschalteten Systemen vor!) Das heißt allerdings nicht, daß zwangsläufig mit der niedrigsten Adreßleitung (A1) begonnen werden muß. So sind die SELECT-Eingänge der CIA-Chips an die Leitungen A11 bis A8 angeschlossen, woraus ein Registerabstand von 256 Byte ($2^8 = 256$) resultiert.

Allgemein gilt:

Je höherwertiger die logisch niedrigste Adreßleitung ist, desto größer sind die Sprünge und Abstände zwischen den einzelnen Registereinträgen im Adreßraum.

Fallstudie 1 – Adreßeinbindung der Customchip-Hardware

Die Register der Customchip-Hardware beispielsweise werden über die Adreßsignale A8 bis A1 selektiert; die Adreßdecodier-Logik akzeptiert alle Adreßinformationen, deren höchstes Nibble (= 4 Bit) dem Bitmuster 1101 (hexadezimal \$D) entspricht ($A_{23}...A_{20} = 1101$). Die Zustände der Leitungen A19 bis A9 sind dabei unerheblich, sie spielen im Adressierungsmechanismus der Customchips keine Rolle.

Gültige Registeradressen bewegen sich deshalb im gesamten Bereich von \$D00000 bis \$E00000. Das ist rund ein MByte mit über 2000 verschiedenen Basisadressen (elf Leitungen $[A_{19}...A_9]$ ergeben $2^{11} = 2048$ Kombinationen).

Bei einer derartigen Fülle potentieller Basisadressen mag es überraschen, daß Commodore in sämtlichen Unterlagen beharrlich den Hex-Wert \$DFF000 als Tabellenanfang der Customchip-Register empfiehlt.

Der Grund dafür liegt in der Kompatibilität zu späteren Amiga-Modellen. Für die Entwickler stellt es kein Problem dar, den Adreßraum durch Änderung der Chipmaske von Gary neu zu strukturieren bzw. wesentlich dichter zu belegen (schließlich bietet der 7-Bit-Adreßvektor noch eine Menge Spielraum). Außerdem liegt die Echtzeituhr der 500er- und 2000er-Modelle an Adresse \$DC0000, überschneidet sich in diesem Bereich also ausgerechnet mit einer der vielen Customchipadressen. Da ist die Überraschung groß, wenn (bei nicht vorhandener Echtzeituhr) der vermeintliche Zugriff auf ein Uhren-Register in der Registerbank der Customchip-Hardware landen.

Wer daher Kompatibilitätsprobleme mit folgenden Amiga-Generationen vermeiden will, sollte sich an die empfohlenen Basisadressen halten und seine Experimentierfreudigkeit zügeln.

Fallstudie 2 – Adreßeinbindung der CIA-Hardware

Die Adreßleitungen A11 bis A8 wählen eins der 16 internen 8-Bit-Register (4 Signale adressieren $2^4 = 16$ verschiedene Registereinträge). CIA-A ist an den unteren Teil des Datenbusses (D7 bis D0) angeschlossen, liegt also zwangsläufig auf ungeraden Adressen, CIA-B ist mit den oberen acht Datenleitungen verbunden und spricht auf gerade Adressen an – eine unproblematische Schaltung, da der 68000 dynamische Datenbusbreiten mit 8 oder 16 Bit unterstützt. Als 8-Bit-Peripheriechips müssen die 8520er im synchronen Busmodus adressiert werden, ein Vorgang, der folgendermaßen abläuft:

Nachdem der Prozessor das VPA-Signal empfangen hat und die interne Synchronisation auf den E-Takt abgeschlossen ist, quittiert die CPU mit VMA* (Valid Memory Address) und signalisiert gleichzeitig das Anliegen einer gültigen Adresse.

Der Schlüssel zur Adressierung der CIA-Hardware liegt in der Generierung von VPA*. Das Signal "Gültige Peripheriegerätadresse" wird von der Adreßdecodierung erzeugt und ist die Reaktion auf einen, für den angesprochenen Peripheriebaustein gültigen Adreßbereich. Die CPU-Antwort VMA* dient dann üblicherweise als Chip-SELECT-Signal; eine spezielle Auswahlleitung kann entfallen.

Die Adreßlogik akzeptiert für die CIA-Hardware sämtliche Adreßkombinationen, deren höchste drei Bits (A23, A22, A21) den Wert 101 führen, so daß die Registeradressen im Bereich von \$A00000 bis \$BFFFFFF liegen. Mit 2-MByte Abstand umfaßt das Intervall erlaubter Basisadressen ein Achtel (!!) des gesamten Adreßraums – bei 16 MByte Adreßkapazität kann man eben etwas großzügiger sein.

Ungelöst ist allerdings noch das Problem der Unterscheidung von CIA-A und CIA-B. Da beide Peripheriechips auf verschiedenen Hälften des Datenbusses liegen, bieten sich dazu die Bussignale UDS* (Upper Data Strobe) und LDS* (Lower Data Strobe) an. Das

Chip-SELECT-Signal ergibt sich dann als Verknüpfung der entsprechenden Strobe-Leitung mit der VMA*-Quittung des Prozessors. Obwohl diese Lösung naheliegt, gingen die Entwickler des Amigas einen anderen Weg. Sie verzichteten auf die Verwendung der Bussteuersignale und nutzten stattdessen zwei weitere Adreßleitungen (A12 für CIA-A und A13 für CIA-B), die zusammen mit der VMA*-Leitung jeweils über ein ODER-Gatter an die low-aktiven SELECT-Eingänge der CIA-Bausteine geführt werden. Liegen sowohl die korrespondierende Adreßleitung als auch VMA* auf Low-Pegel, übernimmt der angesprochene CIA-Chip die Registeradresse auf den Leitungen A11 bis A8.

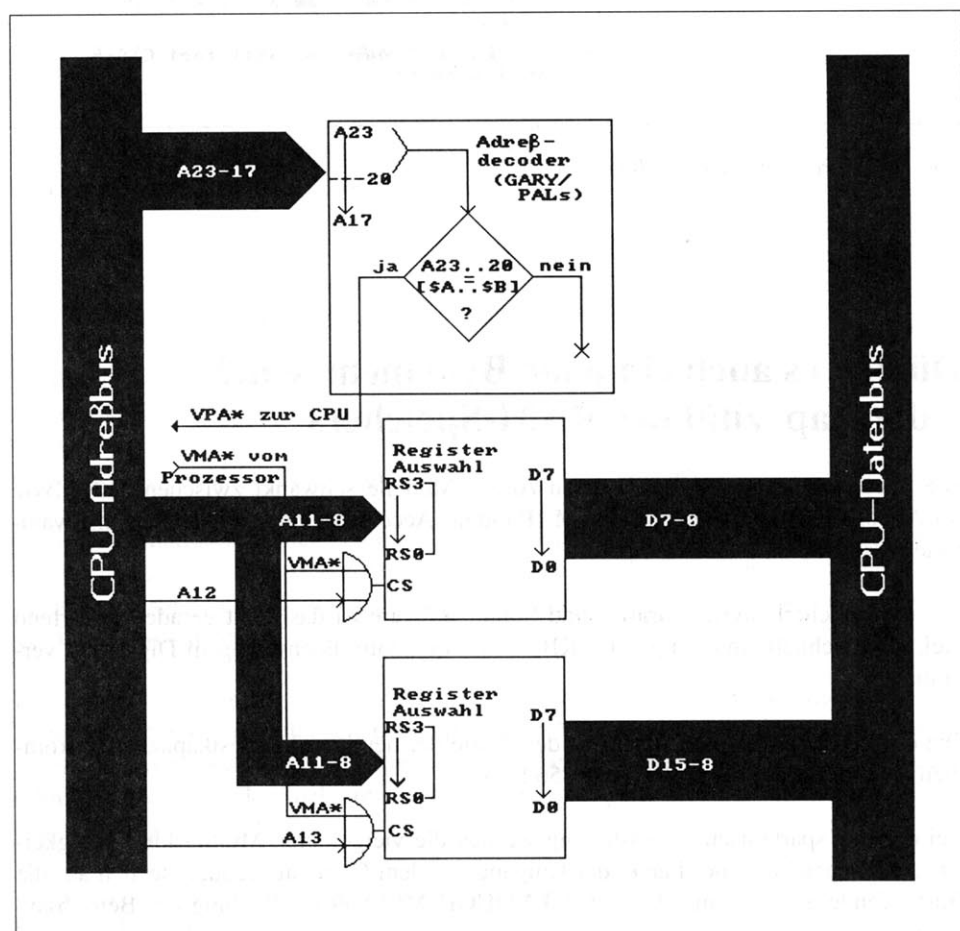


Abb. 5.4: Adressierungsmechanismus der CIA-Chips

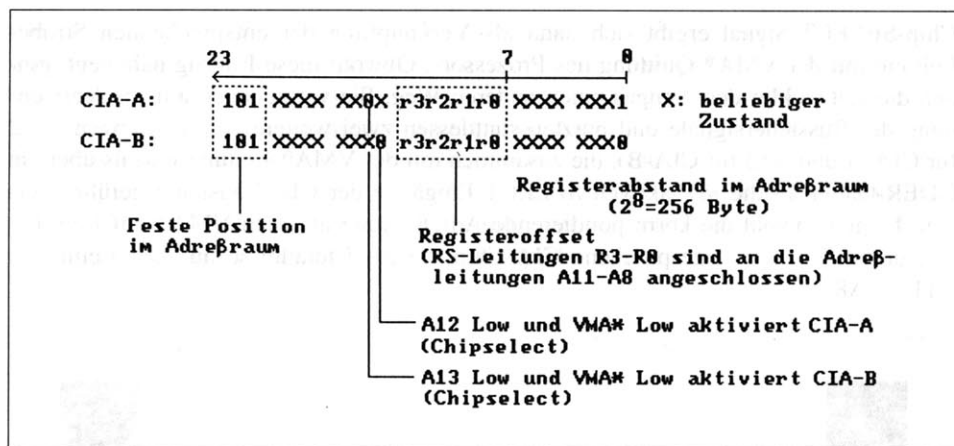


Abb. 5.5: Adreßvektor der CIA-Bausteine

Dürfen es auch ein paar Byte mehr sein? – die Kapazität des RAM-Speichers

Die Speicherausstattung der diversen Amiga-Modelle schwankt zwischen 512 KByte im A500/A1000 und 1 MByte RAM (Random Access Memory = Speicher mit wahlfreiem Zugriff) des A2000/B2000.

Für die speicherhungrige Grafik- und Soundhardware ist das nicht gerade umwerfend viel. (Wie schnell sind einige 100 KByte von qualitativ hochwertigem Digisound verbraucht!)

Die ersten 1000er mußten sogar mit der Hälfte der heutigen Mindestkapazität auskommen – da gab es standardmäßig nur 256 KByte.

Bei derartig spartanischer Ausstattung wurden die vielgelobten Multitasking-Fähigkeiten des "Neuen" zu einer Farce; der Umgang mit dem Computer reduzierte sich auf die frustrierende Schlacht mit der "NOT ENOUGH MEMORY"-Meldung des Betriebssystems.

Hochauflösende Grafiken konnten auf Messeständen und in Produktberichten bestaunt werden; am heimischen System paßten sie nicht zusammen mit Workbench oder Grafikprogramm in den Speicher!

Wer seinen Amiga sinnvoll einsetzen wollte, kam um die Investition einer Speichererweiterung nicht herum. Das wußten auch die Marketing-Strategen von Commodore, die mit der zusätzlich angebotenen 256 KByte-Aufrüstung noch einmal kräftig abkassierten.

Allerdings ließ sich dieses Konzept auf dem deutschen Markt nicht lange durchhalten.

Die aggressive Konkurrenz sorgte dafür, daß man dem Amiga wenigstens 512KByte-RAM als Standardkapazität spendierte.

Doch nach wie vor gibt die Speicherausstattung sämtlicher Modelle, insbesondere der Profigeräte vom Typ 2000, Anlaß zur Kritik.

1 MByte gehört mittlerweile sogar im Heimbereich zum guten Ton (vom Profi-Sektor ganz zu schweigen) – und das, obwohl es durch den Strukturwandel in der DRAM-Produktion erhebliche Engpässe bei den 256 KBit-Chips gibt.

Konkurrent ATARI preist schon in der Produktbezeichnung die vergleichsweise üppige Speicherkapazität seiner Systeme an (Mega-ST-Serie), und viele Hersteller von PC-Kompatiblen steigern die Attraktivität ihrer – technisch dem Original oft gleichwertigen – Geräte durch einen großzügig dimensionierten RAM-Speicher.

In diesem Punkt könnte sich Commodore von manch' anderem Hersteller eine Scheibe abschneiden, zumal der Amiga mit seinen vielfältigen Qualitäten eigentlich nie genug Speicher haben kann; jede Stufe der Aufstockung öffnet neue Türen und erschließt ungeahnte Anwendungen.

Das reicht von schnellen Zwischenspeichern (RAM-Cache, Shadow-RAM, Buffer) und komplexen Benutzeroberflächen über effizientes Multitasking bis hin zu ausgefeilten Animationssystemen mit getrennten Arbeits- und Darstellungsbereichen (Doublebuffering) oder mehreren, gleichzeitig im Speicher gehaltenen Grafikseiten (Pageflipping).

Die Liste läßt sich endlos fortsetzen; viele innovative Ideen stoßen aber mit monotoner Gleichmäßigkeit an die Grenzen der Speicherkapazität – eigentlich schade für ein so leistungsfähiges System.

Ein Ausflug in die Technologie dynamischer RAM-Chips

Der englische Begriff RAM (Random Access Memory = Speicher mit wahlfreiem Zugriff) läßt sich nicht sofort einordnen; "griffiger" ist die Bezeichnung "Schreib-/Lesespeicher", um den wesentlichen Unterschied zu einem ROM-Baustein (Read Only Memory = Nur Lesespeicher) deutlich zu machen, der aufgrund seiner Architektur nur eine Zugriffsrichtung erlaubt.

"Innenansichten" – die Architektur eines RAM-Bausteins

Je nach Technik der Informationsspeicherung differenziert man zwischen dynamischen und statischen RAM-Chips.

Beim letzteren Typ werden die einzelnen Speicherelemente durch bistabile Kippstufen gebildet (siehe Abbildung 5.6).

Aus Gründen der Platzersparnis ersetzt man die Lastwiderstände durch zwei Transistoren (hier T3 und T4) mit fester Vorspannung.

Die Transistoren T5 und T6 dienen zur Adressierung der Speicherzelle.

Sobald sie durch die Zeilenauswahl leitend werden, kann die gespeicherte Information als Potentialsprung auf der Spaltenleitung abgegriffen werden.

Da über einen Zweig der Kippstufe ständig Strom fließt, ist der Verbrauch eines statischen RAMs recht hoch; dafür benötigen diese Bausteine aber keine aufwendigen Refresh-Schaltungen (Refresh = Auffrischung) – sinkt die anliegende Spannung nicht unter einen kritischen Wert, bleiben alle gespeicherten Informationen erhalten.

Durch die rasante Entwicklung der Herstellungstechnologien gelang es die Schaltverzögerung von Flip-Flops (FF) so weit zu minimieren, daß statische Bausteine ihre "dynamischen Kollegen" in puncto Zugriffsgeschwindigkeit längst überholt und weit hinter sich gelassen haben. (Hochgeschwindigkeitssysteme arbeiten schon mit Zugriffszeiten < 10 ns.)

Damit eignen sich statische RAM-Bausteine vor allen Dingen für schnelle Cache-Speicher mit extrem kurzen Zugriffszeiten, ohne die moderne Speicherkonzepte nicht mehr auskommen.

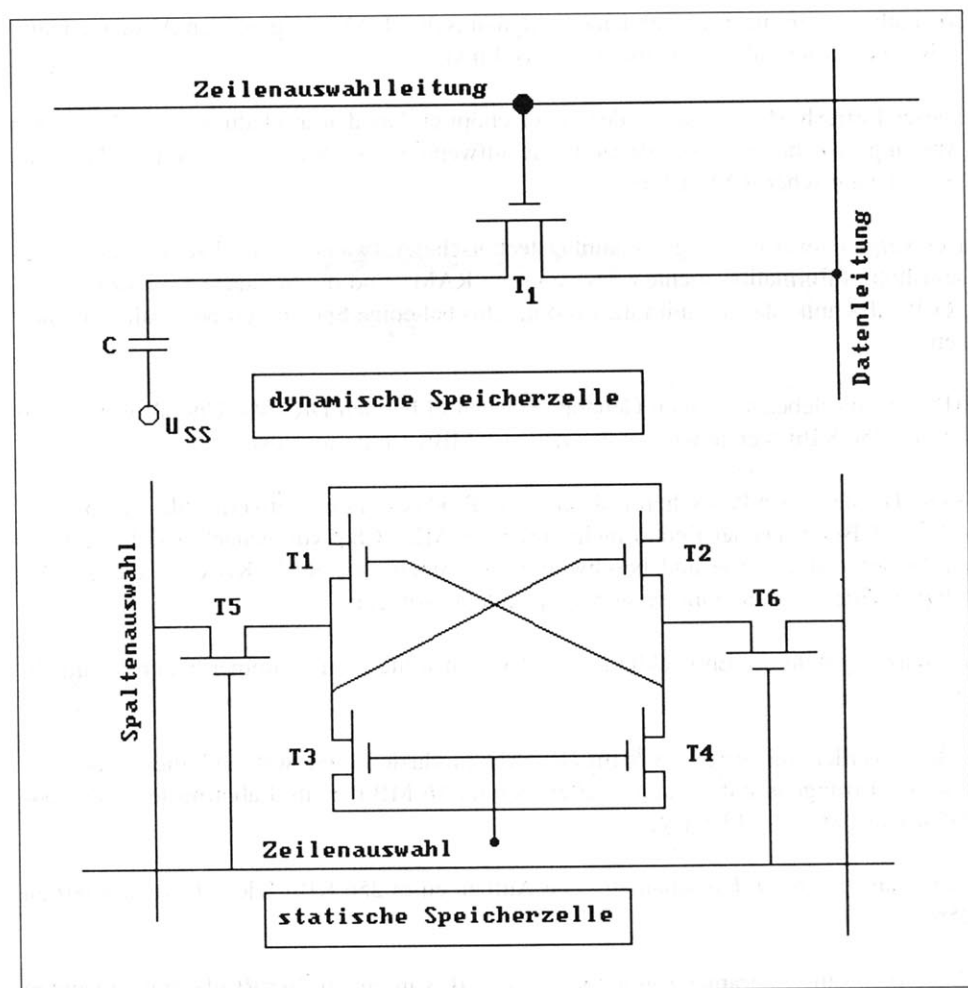


Abb. 5.6: Zellenarchitektur dynamischer und statischer Bausteine

Die Speicherelemente dynamischer RAM-Chips (DRAMs) kommen mit nur zwei Bauelementen aus, einem Kondensator als Ladungsspeicher und einem Transistor zur Selektion der Bitzelle.

Das physikalische Phänomen der Diffusion führt zu einer schrittweisen Entladung des Kondensators (Leckströme) – es droht der Verlust sämtlicher Informationen.

Deshalb müssen die Speicherelemente dynamischer RAMs in gewissen Abständen aufgefrischt werden (üblicherweise alle 2 bis 4 ms).

Dieser Refresh-Mechanismus, der die erschöpften Kondensatorladungen wieder in den Ausgangszustand versetzt, erfordert eine aufwendige Logik und erschwert die Handhabung dynamischer RAM-Chips.

Der vergleichsweise geringe schaltungstechnische Aufwand einer Bitzelle erlaubt aber sehr hohe Informationsdichten. Dynamische RAMs sind die Standard-Speicherchips in der PC-Technik, da sich mit ihnen problemlos beliebige Speichergrößen realisieren lassen.

Momentan erleben wir einen Generationswechsel bei den DRAMs. Überall werden die älteren 256 KBit-Versionen von den neuen 1-MBit-Chips abgelöst.

Viele Hersteller stellten schon frühzeitig die Produktion um und vernachlässigten dabei die 256 KBitter. Da der Bedarf nicht durch den MBit-Chip voll gedeckt werden konnte, explodierten die Preise und beschworen eine regelrechte RAM-Krise herauf. (RAM-Chips werden zu einem interessanten Spekulationsobjekt!)

Gleichzeitig geht die Entwicklung unaufhaltsam weiter, hin zu immer höheren Bitdichten.

Schon werden die ersten 4-MBit-DRAMs produziert und während diese noch als höchste Errungenschaft gelten, warten bereits 16-MBitter als Labormuster – der 64-MBit-Chip ist in der Planung.

Abbildung 5.7 zeigt den schematischen Aufbau eines 256 KBit-DRAMs, organisiert zu 256 K x 1 Bit.

Der Organisationsparameter gibt an, wieviele Bits in einem Zugriff übertragen werden und bestimmt dadurch den Adressierungsmechanismus.

Bei einem 256 K x 1 DRAM läßt sich jede, der insgesamt 262144 (= 218, 256 K) Bitzellen über eine 18-Bit-Adresse einzeln ansprechen.

Für Speicherchips mit einer Kapazität von 32 KByte findet man auch Organisationsformen wie 64K x 4 (Amiga 1000!). Hier hat jede Speicherzelle eine Auflösung von 4 Bit und wird über einen 16-Bit-Vektor adressiert. (Entsprechend weicht auch die Pinbelegung von der des 256 K x 1 Typs ab.)

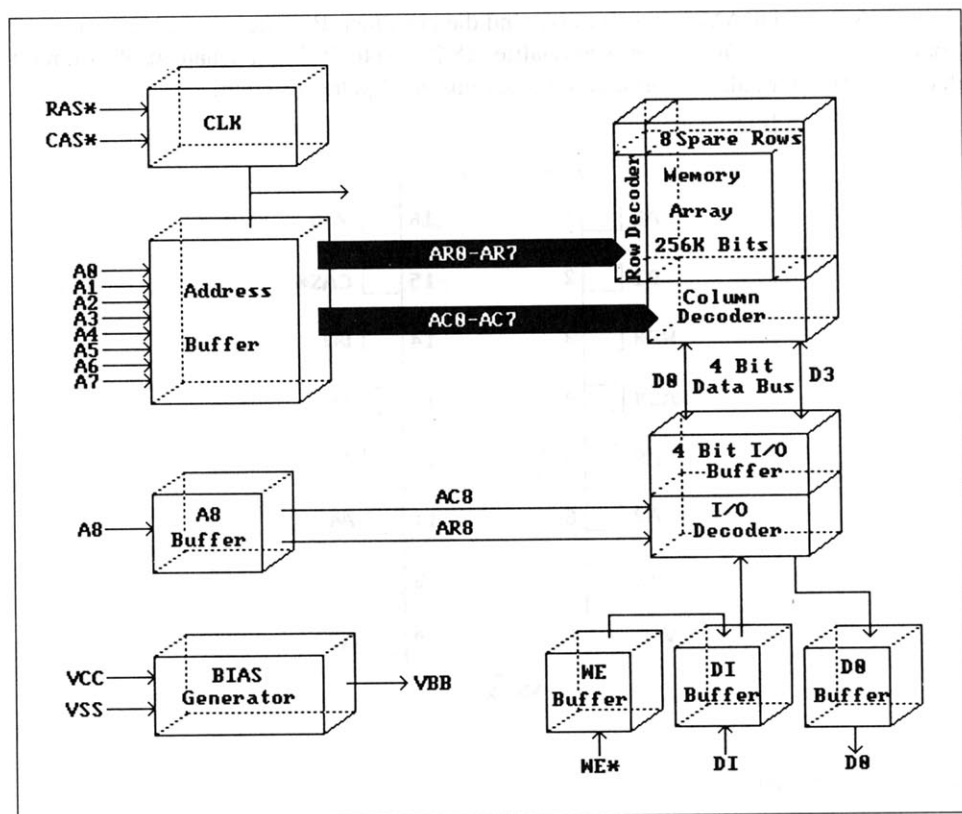


Abb. 5.7: Interner Aufbau und Pinbelegung eines RAM-Chips

Um die Gehäusegröße dynamischer RAM-Chips möglichst klein zu halten, entschied man sich bei der Adreßzuführung für ein Multiplex-Verfahren – die komplette Adresse zerfällt in zwei Hälften, die zeitlich nacheinander auf die Adreßeingänge des RAMs gelegt werden.

Die Methode erfordert zwar zusätzliche Bausteine (Multiplexer), spart aber 50% der normalerweise erforderlichen Adreßanschlüsse (9 anstatt 18 für den 256 K x 1 Typ).

Der Adressierungsmechanismus spiegelt sich denn auch in der matrixartigen Anordnung dynamischer Speicherelemente wider.

Beim 256 K x 1 DRAM beispielsweise sind die einzelnen Bitzellen zu 512 Zeilen x 512 Spalten organisiert. Die untere Adreßhälfte ($18/2 = 9$) trifft die Zeilenauswahl, während der obere Teil der Adreßinformation die gewünschte Spalte ansteuert.

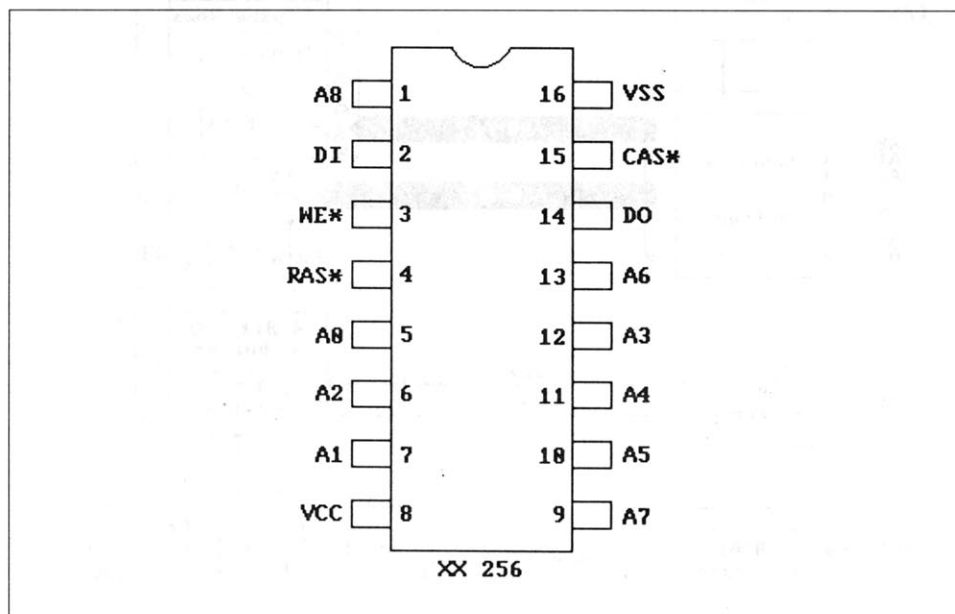


Abb. 5.8: Pinbelegung eines DRAMs vom Typ 41256

Von Zugriffszeiten, Zyklusdauer und Refreshmechanismen

Die Strobesignale RAS (Row Address Strobe = Zeilen-Adreßimpuls) und CAS (Column Address Strobe = Spalten-Adreßimpuls) markieren die anliegende Adreßhälfte.

Ihr Zusammenspiel bestimmt das Timing dynamischer RAM-Chips.

Sobald die Zeilenadresse stabil ist, aktiviert die Speichersteuerung des Systems das RAS-Signal und signalisiert dadurch das Anliegen einer gültigen Zeilenadresse. Gleichzeitig wird die Zugriffsrichtung durch den Zustand der WE*-Leitung (Write Enable) festgelegt.

Daraufhin überträgt das DRAM sämtliche 512 Bitzellen der angesprochenen Zeile in einen Ausgabepuffer (DOL – Data Output Latch). Die Spaltenadresse greift nun nicht mehr direkt auf die Speichermatrix zu, sondern holt die entsprechende Bitzelle aus dem DOL-Speicher. Der Wechsel von der Zeilen- zur Spaltenadresse wird durch das CAS-Signal markiert.

Bevor die Daten jedoch abrufbereit sind, müssen sie noch verstärkt und stabilisiert werden (Leseverstärker, Ausgabe-Flip-Flops), so daß von der Aktivierung der RAS-Leitung bis zum Erscheinen des gültigen Datums ein gewisser Zeitraum vergeht, den man als Zugriffszeit des RAMs bezeichnet.

Typische Zugriffszeiten moderner Bausteine bewegen sich zwischen 100 und 200 ns.

Nach jedem Lesezugriff sind die gespeicherten Ladungen einer Zeile erschöpft; das RAM benötigt weitere Zeit, um die gelesenen Informationen aufzubereiten (Erholungszeit, RAS Precharge Time entspricht üblicherweise zwischen 60% und 100% der RAS-Zugriffszeit)

Diese Aufgabe übernimmt eine integrierte Refreshautomatik; nach Beendigung des Datenzugriffs durch die CPU (CAS inaktiv) wird der komplette Inhalt des Ausgabepuffers in die Speichermatrix zurückgeschrieben, wobei die Bitzellen wieder ihr ursprüngliches Ladungsniveau erreichen.

Addiert man Zugriffs- und Erholungszeit erhält man die Zykluszeit des verwendeten DRAM-Typs.

Sie bildet den zeitlichen Rahmen für den Mindestabstand aufeinanderfolgender Speicherzyklen.

Zugriffs- und Zykluszeit zusammen definieren eindeutig die Maximalgeschwindigkeit mit der auf die gespeicherten Informationen eines dynamischen RAM-Bausteins zugegriffen werden kann.

Die Speichersteuerung sorgt für die reibungslose Abwicklung eines Speicherzugriffs, indem sie das dynamische Speicher-Interface des Prozessors an die Geschwindigkeit der verwendeten RAM-Chips anpaßt.

Dazu dient das Quittungssignal DTACK*; kommt es nicht rechtzeitig (vor Ende des Halbtaktes S4), legt der Prozessor Wartezyklen ein.

Erst nach Bestätigung von DTACK* setzt er den Zugriff fort und beendet ordnungsgemäß den Buszyklus.

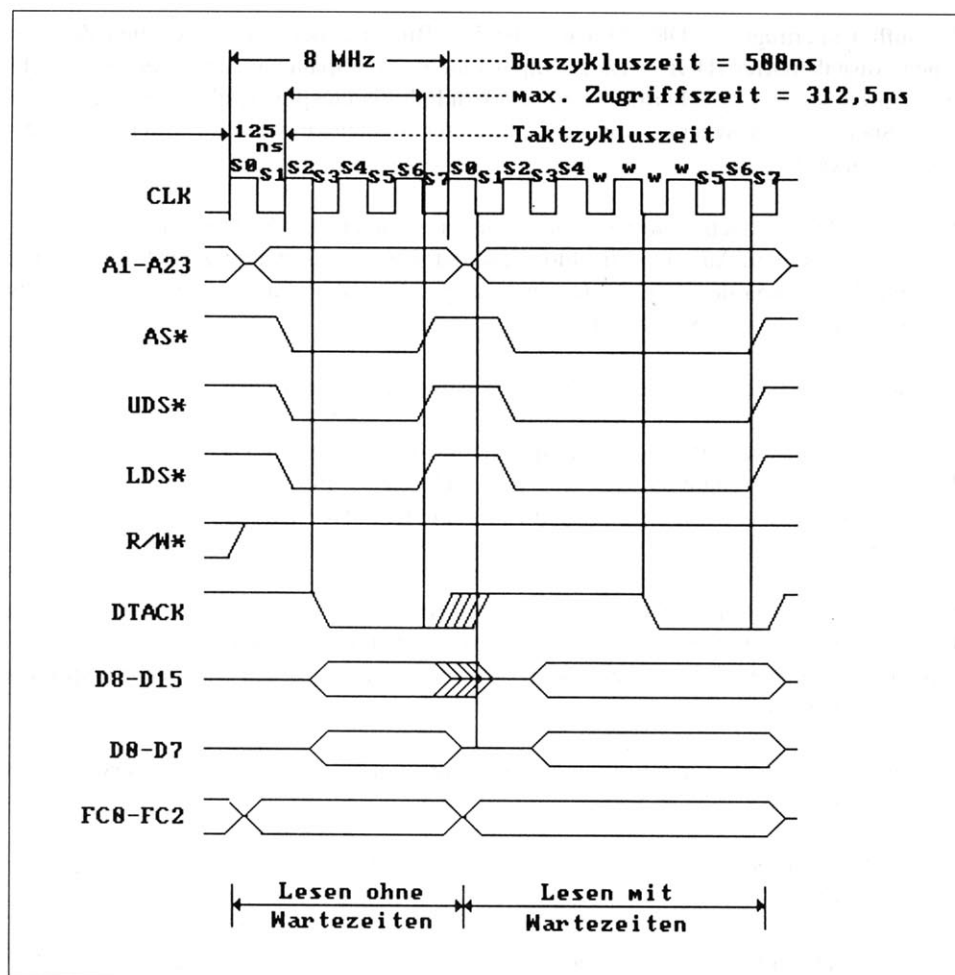


Abb. 5.9: Standard-Lesezyklus des 68000-Prozessors

Für ein verzögerungsfreies Zusammenspiel von CPU und Speicher sollten deshalb Zugriffs- und Zykluszeit der verwendeten RAM-Chips mit dem Bustiming des Prozessor korrespondieren.

Das 68000er-Speicher-Interface stellt allerdings keine besonders hohen Anforderungen:

Von der Aktivierung des AS*-Signals bis hin zur endgültigen Datenübernahme vergehen bei der 7,1-MHz-Version des Amigas volle 350 ns (siehe Abbildung 5.9).

Rechnet man die erforderliche Datenstabilisierungsphase (DST = Data Setup Time) von mindestens 17 ns ab, bleiben immerhin noch satte 333 ns von der Adressierung bis zur Bereitstellung der Daten.

Nun hängt die erforderliche Reaktionsgeschwindigkeit geeigneter RAM-Chips aber nicht nur vom Zugriffsmechanismus des verwendeten CPU-Typs ab, sondern auch von der Signalverzögerung durch Bausteine für Speichersteuerung und Datenaufbereitung.

Die Adreßdecodierung kann erst nach Aktivierung der AS*-Leitung mit ihrer Arbeit beginnen und schränkt damit zwangsläufig den Reaktionsspielraum der RAM-Chips ein. Der Zeitaufwand für die Signalverstärkung durch Datenbustreiber wirkt dagegen zwar eher bescheiden, muß aber ebenfalls mit einkalkuliert werden.

Deshalb finden standardmäßig in den diversen Amiga-Modellen DRAMs mit einer Zugriffszeit von 150 ns Verwendung.

Diese Bausteine liegen auch mit ihrer Zykluszeit (250ns..300ns) im sicheren Rahmen und garantieren, zumindestens bei der jetzigen Systemkonfiguration, einen wartetakt-freien Speicherzugriff.

Die Speicherorganisation

Um die Produktionskosten durch einen möglichst geringen Aufwand an Bauteilen zu senken, faßten die Entwickler auf dem neuen Customchip Fat-Agnus alle wesentlichen Funktionen eines DRAM-Controllers zusammen.

Der Chip verfügt über die vollständige Signpalette zur Steuerung von insgesamt 1 MByte Speicher und ein bereits gemultiplextes Speicherinterface (MA0 bis MA8) für den direkten Anschluß dynamischer RAM-Chips vom Typ xx256.

Der integrierte Refreshmechanismus nutzt die Fähigkeit der Speicherchips, sich zeilenweise selbst zu regenerieren. Bekanntlich wird nach jedem Zugriff auf ein Element der Speichermatrix, die zugehörige Zeile automatisch aufgefrischt. Um einen Datenverlust sicher auszuschließen, müssen lediglich sämtliche Zeilen der Speichermatrix innerhalb eines bestimmten Intervalls adressiert werden.

Diese Aufgabe erfüllt ein 9-Bit-Refreshzähler (REFPTR, Offset \$028) mit dem Fat-Agnus jede der 512 Zeilen eines 256 Kx1-Chips einzeln ansprechen kann ($2^9=512$). Zu Beginn einer Rasterzeile legt der Customchip 4 9-Bit-Refreshadressen auf den Memory-Address-Bus und zählt den Refreshzeiger automatisch hoch.

Ein speziell für diesen Zweck reservierter DMA-Kanal sorgt für einen reibungslosen Ablauf. Während bei vielen anderen Systemen der zyklische Refreshmechanismus dem Prozessor zusätzlich Zeit stiehlt, läuft dieser Vorgang beim Amiga vollkommen verdeckt ab.

Das Speichersystem des A500 und B2000 unterscheidet sich lediglich durch die unterschiedliche Speicherkapazität. Während der 500er von Hause aus nur 512Kbyte-RAM mitbringt, verfügt der B2000 standardmäßig über den vollen, von Fat Agnus unterstützten Speicherausbau von 1 MByte. Eine kleine Erweiterungsplatine für den Einschubschacht am Gehäuseboden bringt den A500 ebenfalls auf runde 1 MByte. Da die Mutterplatine bereits alle Signale für den Speicherausbau bereitstellt, kann diese Platine vergleichsweise preiswert vertrieben werden.

Da funktionell zwischen der Speicherorganisations des A500 und B2000 kein Unterschied besteht, soll letzterer exemplarisch für beide Modelle stehen.

Die 1-MByte dynamischer RAM-Speicher werden von insgesamt 32 RAM-Chips vom Typ xx256-15 (der Anhang -15 weist auf die Zugriffsgeschwindigkeit von 150 ns hin) gebildet.

Die RAMs sind zu vier Speicherbänken à 8 x 256 K organisiert, jeweils acht parallel geschaltete Chips bilden das Low- bzw. Highbyte des 68000-Datenbus. (Dabei werden die Datenleitungen Di(n) und Do(ut) zu einem Anschluß zusammengefaßt. Diese Dual-Ported Datenschnittstelle ist ein Zugeständnis an die Großrechnerarchitektur, die häufig mit getrennten Bussystemen arbeitet, wird hier aber nicht benötigt!)

Die gemultiplexten Adreßleitungen von Fat-Agnus sind kontinuierlich von Chip zu Chip durchführt, so daß jeder Bausteine die selbe 18-Bit-Adreßinformation in zwei Schüben à 9-Bit erhält. Durch die Parallelschaltung verhalten sich die Speicherbänke wie eine 8 x 256 K-Speichermatrix, erlauben also den byteweisen Zugriff auf insgesamt 256 KByte Speicher.

Für die Verwaltung von vier RAM-Banken mit insgesamt 1MByte Speicherkapazität bedarf es eines Auswahlmechanismus, der den gezielten Zugriff auf eine bestimmte Speicherbank erlaubt. Um den Schaltungsaufwand nicht durch zusätzliche Chipselect-Leitungen in die Höhe zu treiben, bedient man sich dazu der Adreßstrobes (RAS/CAS). Ohne ihr exaktes Zusammenspiel reagiert der adressierte RAM-Chip ja überhaupt nicht.

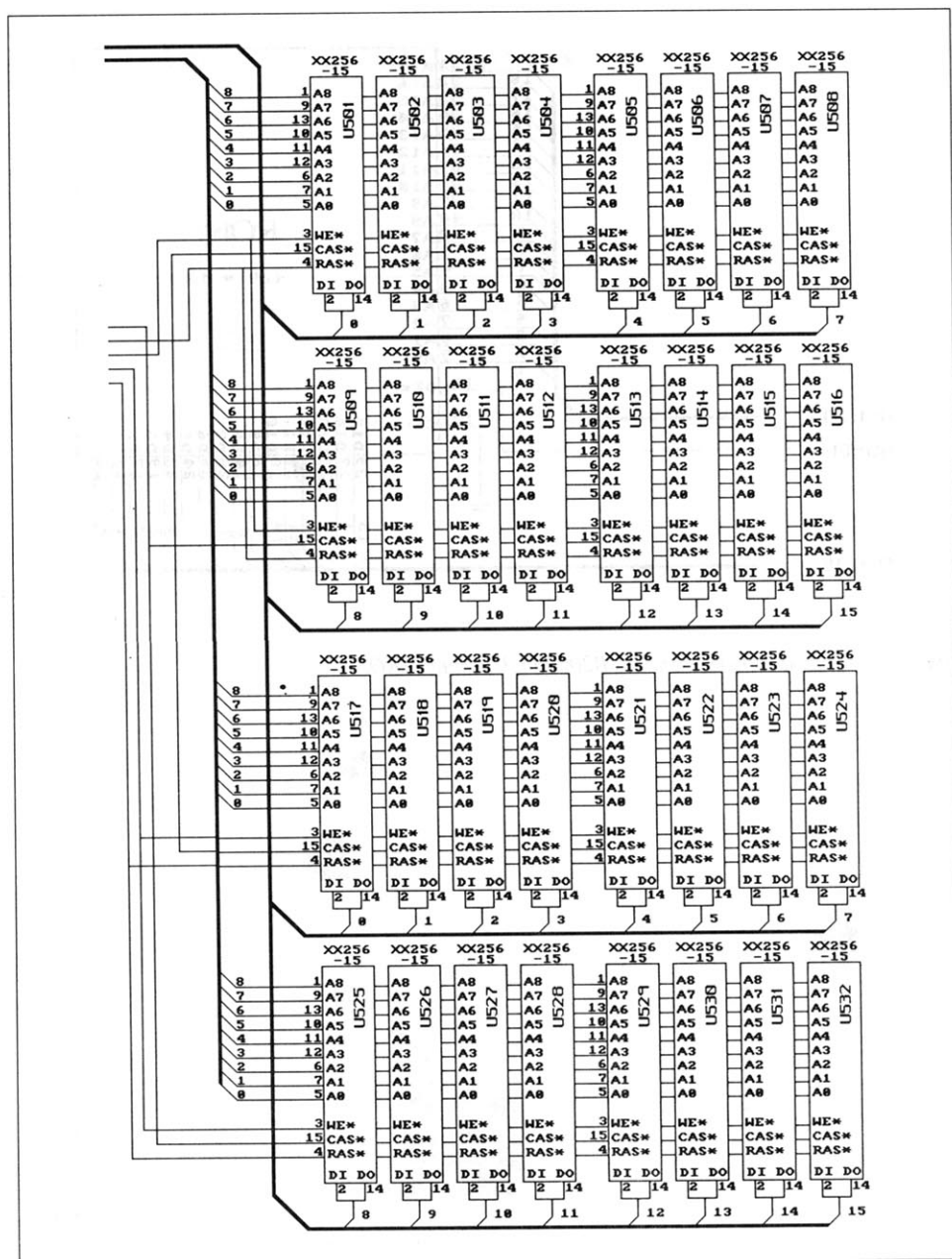


Abb. 5.10: Das Speichersystem des B2000 im Überblick (RAM)

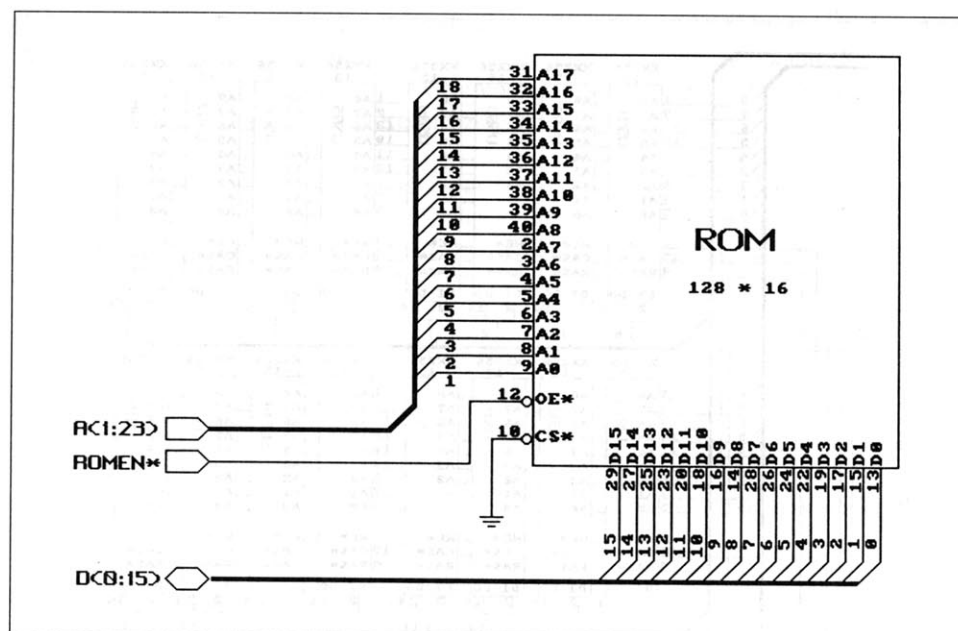
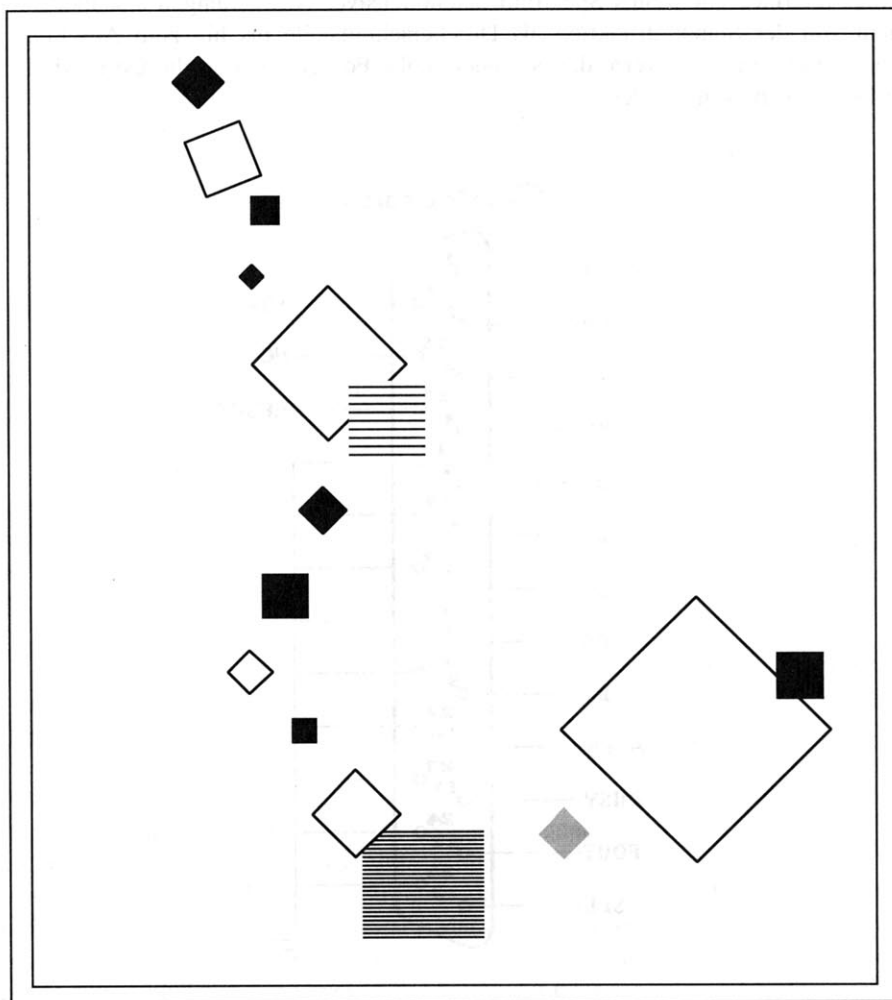


Abb. 5.11: Das Speichersystem des B2000 im Überblick (ROM)

Kapitel 6

Die Interface - Hardware Teil 1



Die Centronics-Schnittstelle

Der Amiga besitzt eine Centronics-Schnittstelle für den Anschluß schneller Peripheriegeräte. Bei diesem Interface-Typ handelt es sich um eine Parallelschnittstelle mit 8-Bit-Datenbus und TTL-kompatiblen Pegelniveaus [-5V bis +5V].

Dank der breiten Busarchitektur werden hohe Übertragungsraten erzielt, die dem Centronics-Interface ein weites Spektrum datenintensiver Anwendungen erschließen. Das reicht von der Standardfunktion als Druckerschnittstelle bis hin zum Anschluß von Sound- und Videodigitizern, die besonders hohe Forderungen an die Geschwindigkeit der Dateneinspeisung stellen.

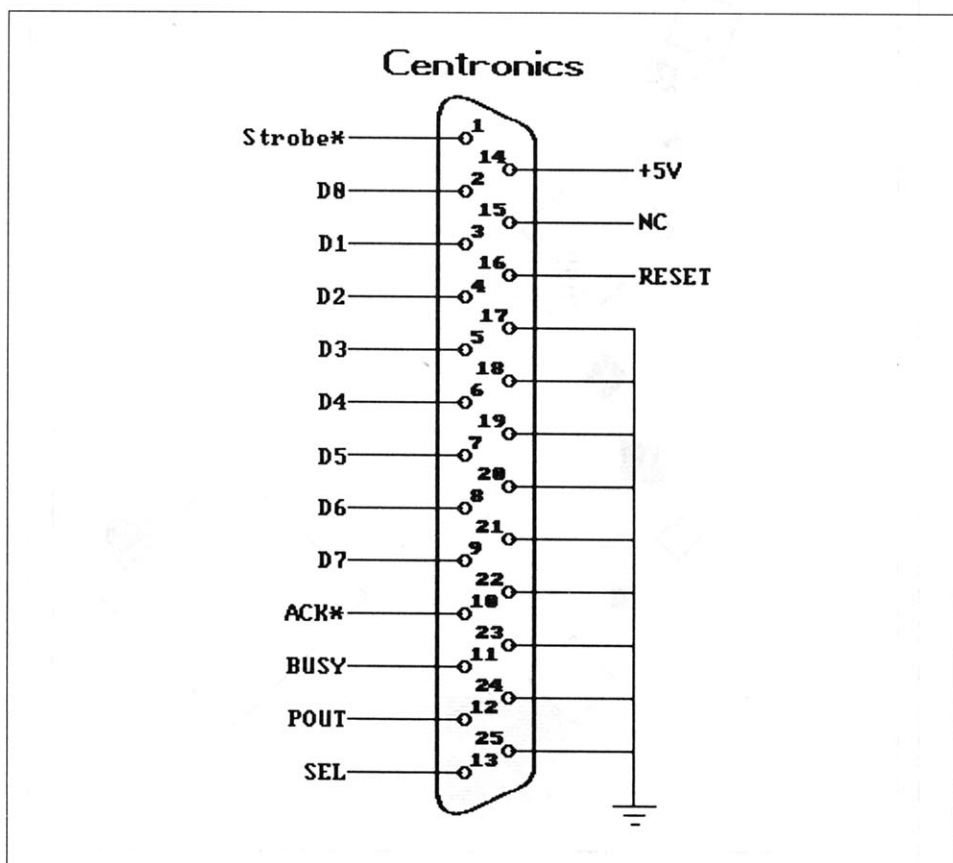


Abb. 6.1: Die Pinbelegung der Centronics-Schnittstelle

Pinbelegung und Signalanordnung entsprechen – zumindestens bei den Amiga-Modellen A500 und B2000 – weitgehend dem IBM-Standard für Drucker-Schnittstellen.

<i>Pin</i>	<i>A1000</i>	<i>A500/B2000</i>	<i>IBM-Standard</i>	<i>Übertragungsrichtung</i>
1	DRDY*	STROBE*	STROBE*	Ausgang
2	Data 0	Data 0	Data 0	bidirektional
3	Data 1	Data 1	Data 1	bidirektional
4	Data 2	Data 2	Data 2	bidirektional
5	Data 3	Data 3	Data 3	bidirektional
6	Data 4	Data 4	Data 4	bidirektional
7	Data 5	Data 5	Data 5	bidirektional
8	Data 6	Data 6	Data 6	bidirektional
9	Data 7	Data 7	Data 7	bidirektional
10	ACK*	ACK*	ACK*	Eingang
11	BUSY*	BUSY*	BUSY*	Eingang
12	POUT	POUT	POUT	Eingang
13	SEL	SEL	SEL	Eingang
– 14	GND	+5V	(pullup)	AUTOFDXT*
– 15	GND	NC	ERROR*	
– 16	GND	RESET*	INIT*	
– 17	GND	GND	SLCTIN*	
– 18-22	GND	GND	GND	
– 23	+5V	GND	GND	
– 24	NC	GND	GND	
– 25	RESET*	GND	GND	

Dank der engen Anlehnung an die IBM-Norm läßt sich jeder kompatible Drucker (und das ist heutzutage fast jeder) über ein handelsübliches Interface-Kabel mit dem A500/B2000 verbinden.

Der A1000 fällt hier mit seinem Steckeranschluß (üblich ist eine weibliche Buchsen-Verbindung) und der eigenwilligen Signalbelegung aus dem Rahmen. Das "Geschlechtsproblem" kann relativ einfach durch einen Adapter gelöst werden. Viel gravierender ist die Anordnung von Stromversorgung (Pin 23) und RESET-Leitung (Pin 25). Normalerweise liegen diese Pins auf Masse (siehe Tabelle), so daß sich die Verwendung eines Standardkabels erübrigt (oder interessiert sich jemand dafür, ob der entstehende Kurzschluß nun den Drucker, den Computer oder gar beide zerstört?).

Die Signale der Parallel-Schnittstelle

Die Signalkürzel deuten es bereits an – das häufigste Einsatzgebiet der Centronics-Schnittstelle ist die Kommunikation mit Druckerperipherie.

Die Datenleitungen Data 0 bis Data 7 (Pins 2 bis 9) ergänzen sich zu einem 8-Bit-Datenbus. STROBE* und ACK* signalisieren Sende- bzw. Empfangsbereitschaft und realisieren auf diese Weise einen einfachen Handshake-Mechanismus.

Mit jedem gültigen Datenbyte geht STROBE* für einen gewissen Zeitraum (beim Amiga für einen E-Taktzyklus = 1,4 Mikrosekunden) auf LOW-Pegel. Sobald das Peripheriegerät bereit ist, die Daten zu übernehmen, quittiert es durch Negierung von ACK* (ebenfalls LOW-aktiv). Erst nach dieser Bestätigung wird ein neues Byte auf den Datenbus gelegt, so daß ein kontinuierlicher Datenfluß mit hoher Fehlertoleranz entsteht.

Ist der Drucker nicht mehr in der Lage, weitere Daten aufzunehmen (beispielsweise wegen Überlastung des internen Datenpuffers), so kann er den Übertragungsprozeß mit einem HIGH-Pegel auf der BUSY-Leitung ("beschäftigt") unterbrechen. Daraufhin wartet der Sender, bis BUSY wieder auf LOW-Pegel wechselt und setzt erst dann die Übertragung weiter fort.

Die Signale POUT (steht für "PAPER OUT" = Papiervorrat zuende) und SEL (SELECT/On LINE) haben mit der eigentlichen Datenübertragung nichts zu tun, sondern dienen zur Übermittlung wichtiger Zustandsinformationen vom Peripheriegerät. Mit POUT (Kürzel für "Paper out") meldet der Drucker, daß sein Papiervorrat erschöpft ist. Die SEL-Leitung reflektiert den Bereitschaftsstatus durch die Zustände ON-LINE (High-Pegel) und OFF-LINE (Low-Pegel).

Aufbau und technische Realisierung der Centronics-Schnittstelle

Der Betrieb des Centronics-Interfaces ist Gemeinschaftsarbeit der beiden CIA-Bausteine.

Parallel-Port B von Chip A bildet den 8-Bit-Datenbus der Schnittstelle. Ein direkter Anschluß der Datenleitungen ist wegen seiner hohen Belastbarkeit und der TTL-Kompatibilität kein Problem. Die Signale STROBE* und ACK* sind mit den korrespondierenden Handshakeleitungen PC* bzw. FLAG* verbunden.

Port A des zweiten CIA-Bausteins wertet die Information der Zustandssignale BUSY, POUT und SEL aus. Sie liegen (in der aufgezählten Reihenfolge) an den untersten drei Bits des Ports (PA0 bis PA2), die in der normalen Konfiguration als Eingang program-

miert sind (entsprechende Bits im DDRA-Register auf 1 gesetzt). Das Pegelniveau der Signale lässt sich jederzeit durch Auslesen des PRA-Registers ermitteln.

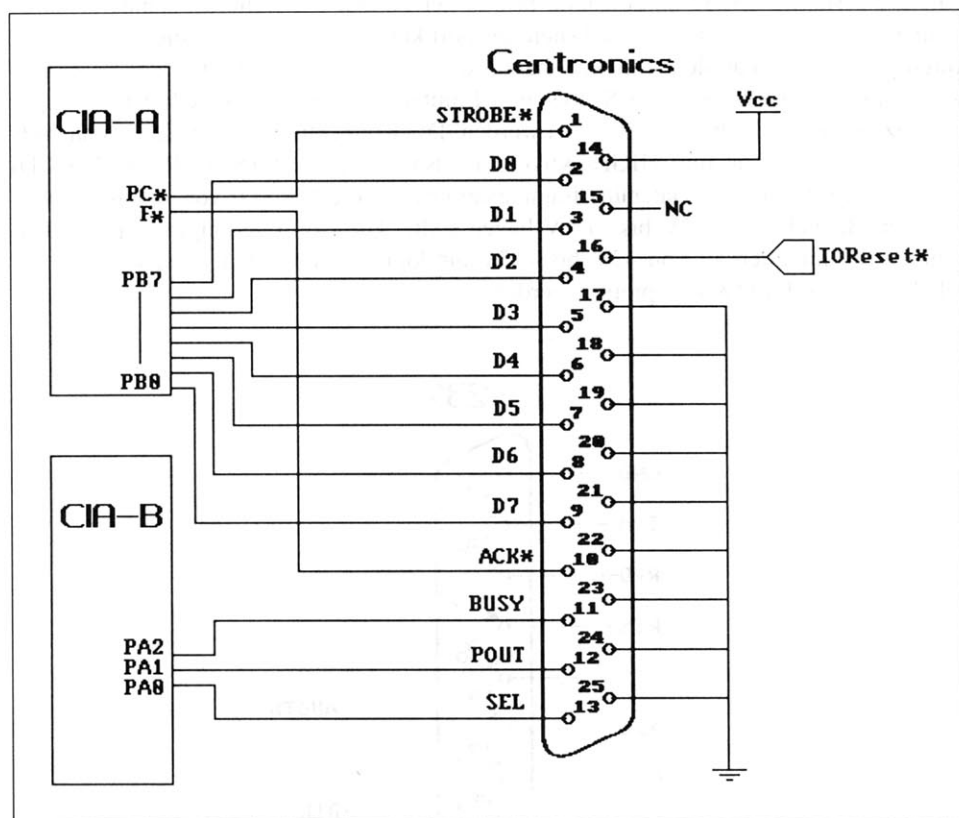


Abb. 6.2: Die Schaltung der Parallel-Schnittstelle

Die Steuerung der Interface-Hardware ist normalerweise Sache des Betriebssystems. Das Parallel-Device bildet die unterste Ebene zur Kommunikation mit der Centronics-Schnittstelle. Es liefert die grundlegenden Funktionen für das Senden und Empfangen von Daten und verwaltet die Interrupts der Handshakeleitung (FLAG*). Für besonders schnelle Reaktionen und extrem datenintensive Anwendungen kann man das Interface auch direkt über die CIA-Hardware programmieren.

Die theoretische Grundlage dazu liefert Kapitel 4 "Die CIA-Hardware".

Die serielle Schnittstelle

Mit der seriellen RS232-Schnittstelle besitzt der Amiga ein weiteres, standardisiertes Universal-Interface. Der bitweise Datentransport kommt zwar nicht an die hohen Übertragungsraten der Parallelarchitektur heran, dafür reichen aber bereits zwei Leitungen zur Kommunikation zwischen Sende- und Empfangsteil aus – bei Bedarf kann sogar gleichzeitig gesendet bzw. empfangen werden. Die Konventionen des RS232-Standards sehen einen Satz von fünf HIGH-aktiven Handshake- (RTS, CTS, DSR, DTR, DCD) und zwei LOW-aktiven Datenübertragungsleitungen (TXD, RXD) vor, deren Pegelniveau im Bereich von +12V bis -12 V liegen sollte. Die Norm verlangt invertierte Datenbits, die im Intervall von -3V bis -15V als logisch HIGH und zwischen +3V bis +15V als logisch LOW interpretiert werden.

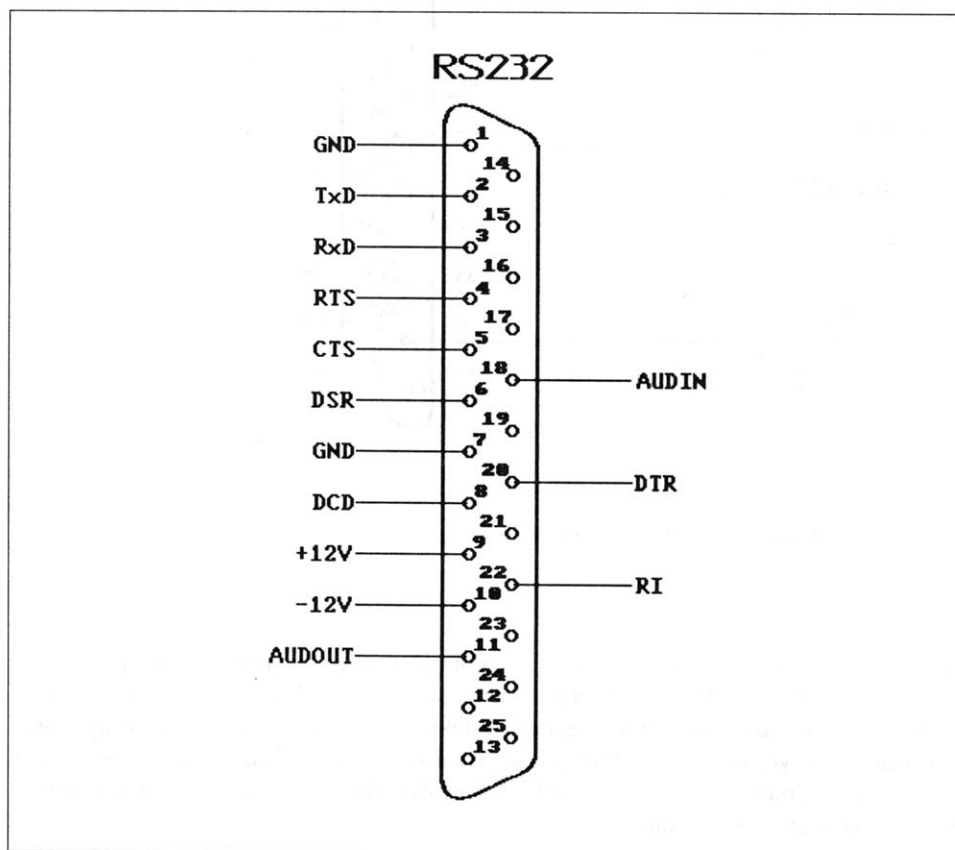


Abb. 6.3: Pinbelegung des RS232-Interfaces

Folgende Tabelle zeigt die Pinbelegung des RS232-Ports bei den verschiedenen Modellen der Amiga-Familie im Vergleich zu den Standard-Spezifikationen; auch hier geht der A1000 seine eigenen Wege.

Pin RS232-Standard A1000 A500/B2000 Eingang/Ausgang

1	GND	GND	GND	–
2	TXD	TXD	TXD	Ausgang
3	RXD	RXD	RXD	Eingang
4	RTS	RTS	RTS	Ausgang
5	CTS	CTS	CTS	Eingang
6	DSR	DSR	DSR	Ausgang
7	GND	GND	GND	–
8	DCD	DCD	DCD	Eingang
9	–	–	+12V	–
10	–	–	–12V	–
11	–	–	AUDOUT	–
12	S.SD	–	–	–
13	S.CTS	–	–	–
14	S.TXD	–5V	–	–
15	TXC	AUDOUT–	–	–
16	S.RXD	AUDIN	–	–
17	RXC	EB	–	–
18	–	INT2*	AUDIN	–
19	S.RTS	–	–	–
20	DTR	DTR	DTR	Ausgang
21	SQD	+5V	–	–
22	RI	–	RI	–
23	SS	+12V	–	–
24	TXC1	C2*	–	–
25	–	RESET*	–	–

Neben den Standardsignalen liegen beim A500 und B2000 auch zwei Leitungen des Audiosystems (AUDIN und AUDOUT) am seriellen Schnittstellenausgang.

AUDIN dient zur Einspeisung eines externen Tonsignals, das zusammen mit dem rechten Audioausgang über einen 47-Ohm-Widerstand durch den Tiefpaßfilter läuft und an der rechten Audiobuchse wieder abgegriffen werden kann.

Die AUDOUT-Leitung führt lediglich das Signal des linken Stereokanals (1-kOhm-Längswiderstand).

Das RI-Signal (Ring Indicator = Klingel-Signal) hat schon etwas mehr mit der seriellen Schnittstelle zu tun. Es handelt sich dabei um eine Art Rufzeichen zur Aktivierung des Sende-/Empfangsbetriebs, das aber nur für direktgekoppelte Modems interessant ist.

Die Schnittstelle des A1000 führt zusätzlich noch den E-Takt des Prozessors (als gepufferte Leitung), einen ebenfalls gepufferten RESET-Ausgang, die INT2-Leitung von Paula und einen 3,54-MHz-Takt mit ungewöhnlicher Phasenlage (C2*).

Die Signale der seriellen Schnittstelle

Das RS232-Interface arbeitet mit getrennten Datenleitungen für Sende- und Empfangsbetrieb. Die TXD-Leitung (Transfer Data = Sendedaten) schickt den seriellen Bitstrom zum Peripheriegerät, über die RXD-Leitung (Receive Data = Empfangsdaten) empfängt der Computer Daten eines externen Senders.

Alle anderen Leitungen dienen ausschließlich der Initiierung und Überwachung des Übertragungsprozesses. Es beginnt mit der gegenseitigen Anmeldung der Kommunikationsteilnehmer. Computerseitig geschieht dies durch Aktivierung der DTR-Leitung (Data Terminal Ready = Datenterminal betriebsbereit), das Peripheriegerät antwortet mit DSR (Data Set Ready). Ist die Verbindung ordnungsgemäß aufgebaut, signalisiert der Sender seine Kommunikationsbereitschaft durch HIGH-Pegel auf RTS (Request to Send = Sende-anforderung). Das Peripheriegerät prüft die Fähigkeit zur Datenübernahme und quittiert bei positivem Ergebnis mit CTS (Clear to Send, High-Pegel). Während der gesamten Übertragungsphase bleiben die Signale RTS und CTS aktiv und wechseln erst dann wieder ihr Pegelniveau, wenn die Verbindung abgebrochen wird.

Das Handshakesignal DCD (Carrier Detect = Trägerfrequenz erkannt) dient im Modembetrieb als Bestätigung für die Sendestation, daß ihre Signale mit ausreichender Stärke empfangen werden.

Aufbau und technische Realisierung des RS232-Interfaces

Die Datenübertragungsleitungen TXD und RXD sind mit den gleichnamigen Anschlüssen der UART-Schaltung auf dem Customchip Paula verbunden. Die englische Auflösung des Kürzels UART (Universal Asynchronous Receiver Transmitter) ist ziemlich unhandlich, in etwas erweiterter Form kann man das Begriffsmonster etwa als "universell einsetzbaren Sende- und Empfangsbaustein für den asynchronen Übertragungsmodus" übersetzen.

DSR, CTS, DCD, RTS und DTR liegen in der aufgezählten Reihenfolge an den Portleitungen PA3 bis PA7 von CIA B.

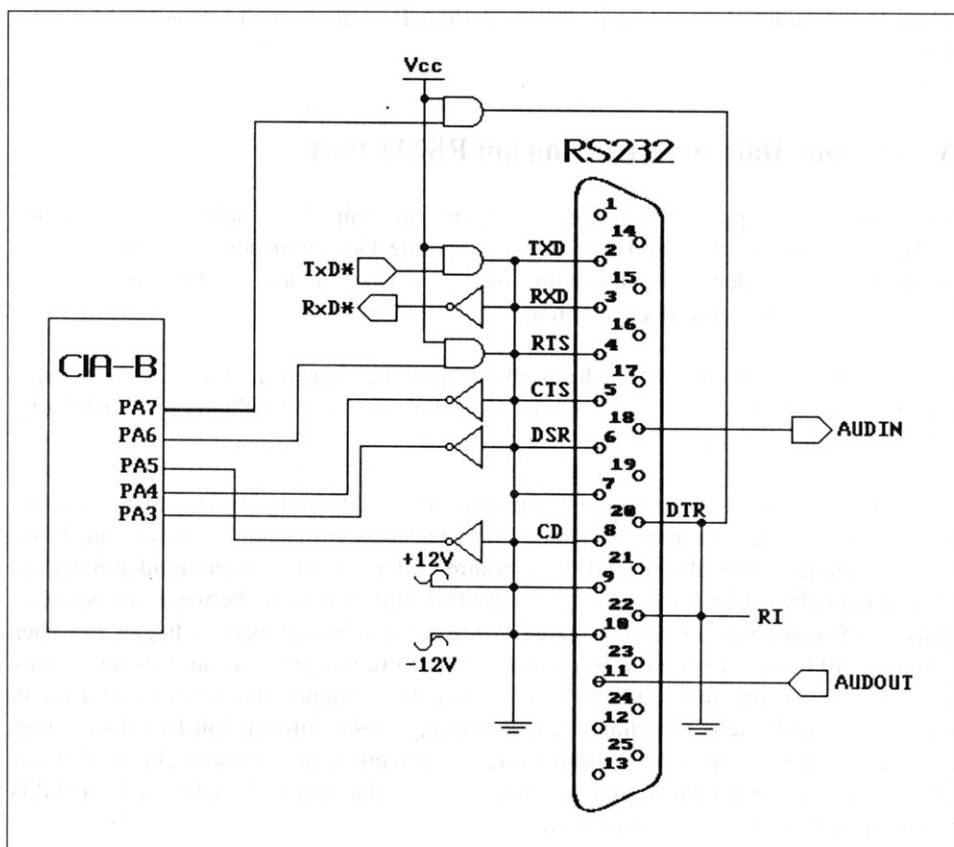


Abb. 6.4: Die Schaltung der seriellen Schnittstelle

Sowohl Datenübertragungs- als auch Handshakeleitungen sind allerdings nicht direkt mit den entsprechenden Bausteinen verbunden, sondern laufen über invertierende RS232-Treiber vom Typ MC1488 (am Ausgang) bzw. MC1489 (für die einlaufenden Signale) mit einer Versorgungsspannung von $\pm 12V$. Je nach Übertragungsrichtung setzen die Pegelwandler TTL-Signale $[-5V$ bis $+5V]$ in RS232-kompatible Spannungspegel von $\pm 12V$ um oder umgekehrt, einlaufende RS232-Pegel in TTL-Niveaus. Da die Treiberbausteine invertieren, haben die Ausgangspegel natürlich immer das umgekehrte Vorzeichen der Eingangsspannung.

Die RS232-Norm arbeitet mit Low-aktiven Datenleitungen und High-aktiven Handshakesignalen, beide werden aber durch die Pegelwandler negiert. Das gilt es bei einer direkten Programmierung der Steuerleitungen zu beachten. Zur Aktivierung von RTS (High-Pegel) müßte beispielsweise Bit 6 im PRA-Register von CIA-B gelöscht werden (Low).

Asynchrone Datenübertragung am RS232-Port

Im Gegensatz zur parallelen Datenübertragung, die mit einem sehr einfachen Handshakemechanismus auskommt, verlangt der serielle Datentransport – zumindestens in der heiklen Phase der Aufspaltung und bitweisen Übertragung des Datenbytes – eine exakte Synchronisation zwischen Sender und Empfänger.

Einen Weg zur Lösung des Synchronisationsproblems zeigen die CIA-Chips. Ihr serieller Port verschickt die Bits im Takt der CNT-Leitung, der Empfänger muß sich lediglich an den Impulsen des CNT-Signals orientieren.

Das RS232-Interface verfügt über keine derartige Taktleitung. Hier sind Sende- und Empfangsstation gezwungen, vor Beginn des Datenaustausches eine bestimmte Übertragungsfrequenz festzulegen und diese während der gesamten Kommunikationsphase exakt einzuhalten. Die Ausgabefrequenz der Datenbits wird üblicherweise als Baudrate (Bits pro Sekunde) bezeichnet, gängige Werte für den Modembetrieb liegen zwischen 300 und 2400 Baud. Durch die Baudrate weiß der Empfänger zwar, in welchen Zeitabständen der Bitstrom abzutasten ist; wo ein Datenbyte beginnt und wo es endet, kann er daraus aber noch nicht entnehmen. Zur Lieferung dieser Information haben sich zwei Übertragungstechniken herauskristallisiert, der synchrone und der asynchrone Betrieb, wobei letzterer die am häufigsten genutzte Methode darstellt und auch von der UART-Schaltung des Amigas verwendet wird.

Das synchrone System hat nämlich einen entscheidenden Nachteil. Der Empfänger darf in den Ruhephasen (wo keine gültigen Daten anliegen) nicht aus der Synchronisation driften, so daß er auch in der übertragungsfreien Zeit durch einen Strom von Pseudodaten "bei der Stange" gehalten werden muß.

Dieses Problem kennt die asynchrone Betriebstechnik nicht. Die Datenübertragung beginnt immer mit einem Null-Bit (High-Pegel) als Startmarkierung und endet nach acht Zyklen mit ein oder zwei Einser-Bits als Stoppmarke. Der Empfänger erkennt den Anfang eines neuen Datenbytes am Pegelwechsel zwischen Start- und Stopp-Bit (negative Flanke) und beginnt sofort mit der Synchronisation auf den eintreffenden Bitstrom.

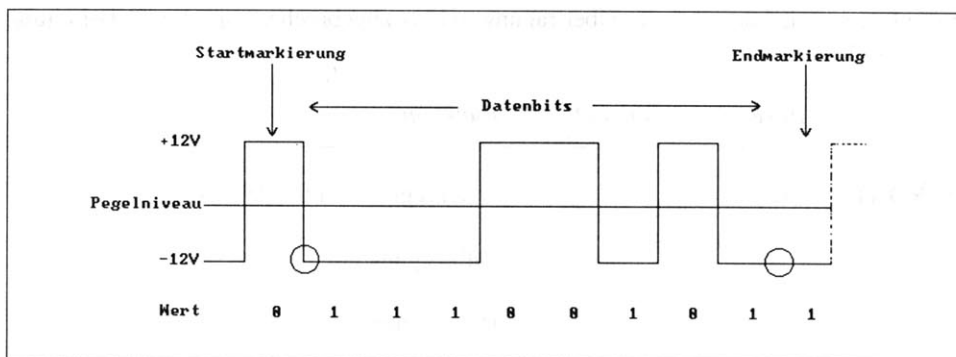


Abb. 6.5: Übertragungszyklus der seriellen Schnittstelle

Die Programmierung der UART-Schaltung

Ein serieller Port besteht im Prinzip (siehe CIA-Chips) aus drei Komponenten, einem Puffer zur kurzzeitigen Datenspeicherung, einem Schieberegister für die Erzeugung des seriellen Bitstroms und schließlich einem programmierbaren Taktgenerator zur Festlegung der Ausgabefrequenz. Da das RS232-Interface simultan senden und empfangen kann, gibt es Schieberegister und Datenpuffer in zweifacher Ausführung.

Bei den relativ niedrigen Übertragungsraten der seriellen Schnittstelle lohnt sich keine DMA-Unterstützung; für einen kontinuierlichen Datentransport von und zu den Datenregistern muß der Mikroprozessor sorgen. Ein einfacher Interruptmechanismus (Sendepuffer leer, Empfangspuffer voll) unterstützt ihn dabei.

Die Programmierschnittstelle zwischen Anwender und UART kommt mit nur einem Steuerregister und zwei Datenregistern aus. SERPER (Serial Period) legt die Baudrate als Zeitabstand zweier Ausgabezyklen fest, SERDAT bildet den Datenpuffer im Sendebetrieb und SERDATR puffert die Daten im Empfangsmodus. Das SERDATR-Register weist eine interessante Besonderheit auf. Die Flag-Bits RBF (Receive Buffer Full = Empfangspuffer voll) und TBE (Transmit Buffer Empty = Sendepuffer leer) sind mit den gleichnamigen Bits im INTREQ-Register (Bit 11 und 0) gekoppelt, werden also immer parallel mit diesen gesetzt bzw. gelöscht.

Die Abbruchsteuerung ist als einzige UART-Funktion nicht in den drei SERxxx-Registern untergebracht, sondern auf Bit 11 des ADKCON-Registers ausgelagert. Setzt man

es auf 1, so wird der aktuelle Übertragungszyklus abgebrochen und die TXD-Leitung geht auf 0.

<i>Register</i>	<i>Adresse</i>	<i>Zugriff</i>	<i>Funktion</i>
SERDAT	\$DFF030	w	Datenregister im Sendebetrieb
Bitbelegung:			
15			unbenutzt
14			–
13			–
12			–
11			–
10			–
9			Stoppbit
8			Stoppbit oder DB8 (9. Datenbit)
7			DB7
6			DB6
5			DB5
4			DB4
3			DB3
2			DB2
1			DB1
0			DB0
SERDATR	\$DFF018	r	Datenregister im Empfangsmodus
Bitbelegung:			
15	OVRUN		Überlauf des Schieberegisters
14	RBF		Empfangsdatenpuffer voll
13	TBE		Sendedatenpuffer leer
12	TSRE		Sendeschieberegister leer
11	RXD		Pegel der RXD-Leitung
10	–		
9	STP		Stoppbit
8	STP		Stoppbit oder Datenbit
	o. DB8		(LONG)
7	DB7		Datenbit 7
6	DB6		Datenbit 6

Register	Adresse	Zugriff	Funktion
		5	DB5 Datenbit 5
		4	DB4 Datenbit 4
		3	DB3 Datenbit 3
		2	DB2 Datenbit 2
		1	DB1 Datenbit 1
		0	DB0 Datenbit 0
SERPER	\$DFF032	w	Periodenregister für die Ausgabefrequenz

Bitbelegung:

Bit 15 LONG Länge der empfangenen Daten
 1 = 9 Bit pro Byte
 0 = normal mit 8 Bits pro Byte
 Bit 14...0 Zeitabstand zwischen Ausgabe bzw. Übernahme zweier Datenbits in Buszyklen

Die Baudrate wird indirekt programmiert, da für die Hardware nur der Zeitabstand zwischen zwei Datenbits interessant ist (Kehrwert der Baudrate). Das Ergebnis muß in Buszyklen umgerechnet (dividiert durch 282 ns) und um einen Zyklus vermindert werden (der UART zählt immer einen Buszyklus zu viel).

Ausgabeperiode = $(3,54 \text{ MHz} / \text{Baudrate}) - 1$

ADKCON	\$DFF09E	w	Audio-Disk-Kontrollregister
--------	----------	---	-----------------------------

Bit 11 UARTBRK

unterbricht die Ausgabe und setzt TXD auf 0.

Im Sendemodus bildet das SERDAT-Register einen Datenpuffer zwischen Speicherblock und Schieberegister des UART. Der Prozessor beginnt nun damit, die Datenbytes in das SERDAT-Register zu schreiben, je nach Format als überlanges Byte mit 9 Bit und einem Stopp-Bit oder als normales Byte mit 1 bzw. 2 Stopp-Bits davor. Sobald das Datum im SERDAT-Register ankommt, wird der Puffer in das Schieberegister kopiert und die Steuerlogik setzt das TBE-Bit (Sendepuffer leer).

Während das Schieberegister die Datenbits (LSB zuerst) im Takt der Baudrate auf die TXD-Leitung schiebt, hat der Prozessor genügend Zeit, SERDAT wieder mit neuen Daten zu versorgen. Wenn dieses nicht bis zum Zyklusende geschieht, ruft die Steuerlogik den Datennotstand aus und flaggt TSRE (Transmit Shift Register Empty = Sendeschieberegister leer; intern mit dem TBE-Bit des Interrupt-Systems gekoppelt).

Um die CPU möglichst wenig zu belasten, bietet es sich an, den TBE-Interrupt zu nutzen und die Datenversorgung des SERDAT-Registers in einer Unterbrechungsroutine abzuwickeln. Natürlich muß auch hier (wie bei jeder Interruptprogrammierung auf dem Amiga) am Ende der Routine das auslösende Bit im INTREQ-Register ausgeblendet werden:

```
move #$0001,$DFF09c ;INTREQ, Bit 0 (TBE) löschen
```

Hier noch einmal die unterstützten Datenformate der UART-Schaltung:

Bit-Nr.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8 Bit, 2 Stoppbits:	x	x	x	x	x	x	1	1	D7	D6	D5	D4	D3	D2	D1	D0
8 Bit, 1 Stoppbit :	x	x	x	x	x	x	0	1	D7	D6	D5	D4	D3	D2	D1	D0
9 Bit, 1 Stoppbit :	x	x	x	x	x	x	1	D8	D7	D6	D5	D4	D3	D2	D1	D0

Im Empfangsbetrieb setzt das Schieberegister die seriellen Datenströme zu Datenbytes zusammen. Nach Empfang des letzten Stopp-Bits (die Interpretation der Datenbits bestimmt Bit 15 [LONG] des SERPER-Registers) wird sein Inhalt in den Datenpuffer kopiert und durch Setzen des RBF-Bits (Receive Buffer full = Empfangspuffer voll) als abholbereit gekennzeichnet.

Da der Sender im allgemeinen nicht darauf wartet, daß der Empfangspuffer auch geleert wird, kommt der Sicherung des SERDATR-Registers höchste Priorität zu. Der RBF-Interrupt hat das Unterbrechungsniveau 5 und damit Vorrang gegenüber den meisten anderen Interruptanforderungen. Ist der Datenpuffer dennoch nicht rechtzeitig geleert worden und das Schieberegister wieder voll, droht ein Verlust der Daten. Die Hardware setzt das OVRUN-Bit (Überlauf des Schieberegisters) und löst erneut einen Level-5-Interrupt aus, der diesmal unverzüglich bedient werden sollte.

Nicht nur zum Spielen geeignet – die Game-Ports

Mit den beiden Game-Ports erschließt sich der Amiga ein großes Feld preiswerter Eingabegeräte.

Obwohl es keinen offiziellen Standard für diese Anschlüsse gibt, haben die Spielkonsolen von Atari und der C64 eine heimliche Norm etabliert. Erfreulicherweise ist der Game-Port des Amigas so konzipiert, daß es keinerlei Kompatibilitätsprobleme gibt; von der Maus zum Joystick, vom Paddle zum Trackball – es paßt fast alles.

Aufbau und Schaltung der Game-Ports

Game-Port-0 (links):

<i>Pin</i>	<i>Bezeichnung</i>	<i>Anschluß</i>	<i>Chip</i>	<i>Verbindung</i>
1	Forward*	M0V	Denise	2 zu 1 Multiplexer
2	Back*	M0H	Denise	2 zu 1 Multiplexer
3	Left*	M1V	Denise	2 zu 1 Multiplexer
4	Right*	M1H	Denise	2 zu 1 Multiplexer
5	POTX	POT0X	Paula	gepuffert
6	Fire*	PA6/LPEN	CIA-A	gepuffert/Jumper*
7	+5V	+5V	+5V	Strombegrenzung
8	GND	GND	GND	–
9	POTY	POT0Y	Paula	gepuffert

* im B2000 kann der Light-Pen-Eingang wahlweise auf Port 0 oder Port 1 gelegt werden

Game-Port-1 (rechts):

<i>Pin</i>	<i>Bezeichnung</i>	<i>Anschluß</i>	<i>Chip</i>	<i>Verbindung</i>
1	Forward*	M0V	Denise	2 zu 1 Multiplexer
2	Back*	M0H	Denise	2 zu 1 Multiplexer
3	Left*	M1V	Denise	2 zu 1 Multiplexer
4	Right*	M1H	Denise	2 zu 1 Multiplexer
5	POTX	POT1X	Paula	gepuffert
6	Fire*	PA7/LPEN	CIA-A	gepuffert /Jumper

Pin	Bezeichnung	Anschluß	Chip	Verbindung
7	+5V	+5V	+5V	Strombegrenzung
8	GND	GND	GND	—
9	POTY	POT1Y	Paula	gepuffert

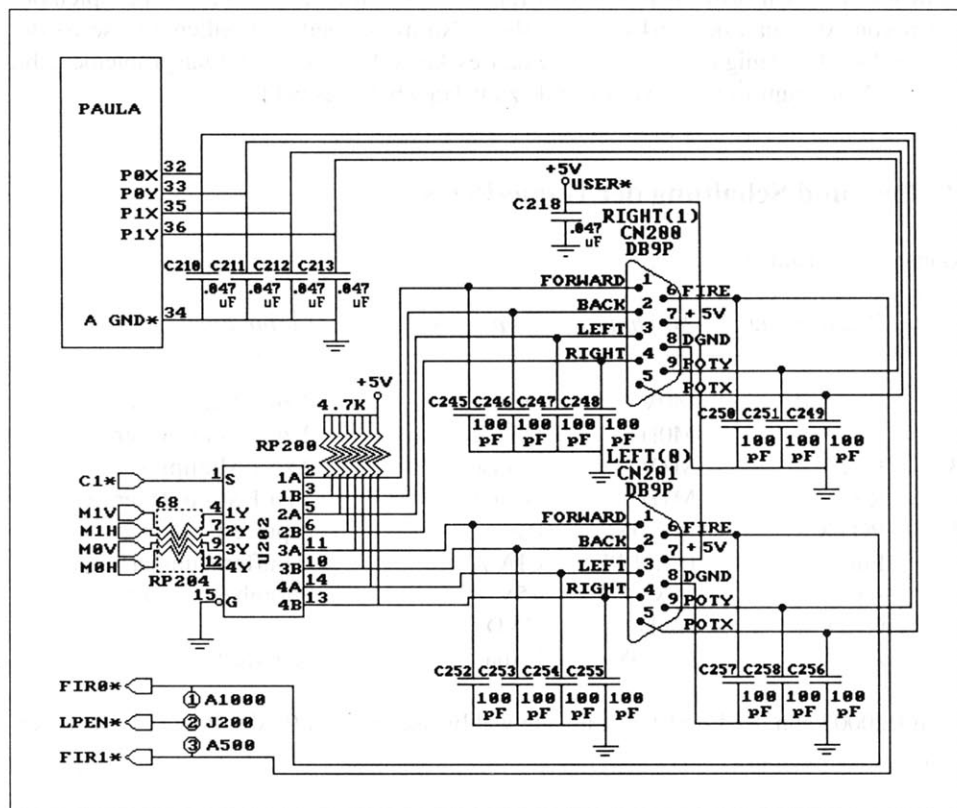


Abb. 6.6: Schaltung der Game-Ports beim B2000

Forward*, Back*, Left* und Right* dienen, je nach Verwendung des Ports, zur Übertragung der Joystickstellung (siehe Bezeichnung) oder der Bewegungsimpulse einer angeschlossenen Maus. Ein 2 zu 1 Multiplexer vom Typ 74LS157 faßt die ersten vier Leitungen beider Game-Ports zusammen und legt sie abwechselnd (im Systemtakt der Customchips) auf die Mauseingänge von Denise. Sämtliche Abfrageleitungen für die Knöpfe der Eingabegeräte sind Low-aktiv. Ihre Schalter erzeugen im offenen Zustand

einen High-Pegel, im geschlossenen dagegen Low-Pegel (Leitung wird mit Masse verbunden). Wenn die Potentiometeranschlüsse von Paula (POTxx) auf Eingang programmiert sind, kann jeder Game-Port bis zu drei Knöpfe verwalten (Pins 5, 6 und 9).

Flinkes Nagetier am Game-Port – die Maus

<i>Pin</i>	<i>Leitung</i>	<i>Mausanschluß</i>
1	M0V	V-Impulse
2	M0H	H-Impulse
3	M1V	VQ-Impulse
4	M1H	HQ-Impulse
5	PxY	–
6	PA6 (Port 0)	linke Maustaste PA7 (Port 1) "
7	+5V	+5V
8	GND	GND
9	P0X (Port 0)	rechte Maustaste P1X (Port 1) "

Die Maus hat sich wohl in allen Bereichen des Computeralltags als sinnvolle Ergänzung der Tastatur etabliert. Scheinbar mühelos huscht der vertraute Pfeil über den Bildschirm, verschiebt Symbole, räumt einen Fensterstapel auf und klickt sich durch unzählige Menüebenen – wie ein Spiegelbild folgt er dabei exakt dem Pfad, den die Maus auf dem Schreibtisch zieht. Bei derart intensiver Nutzung und dem "hautnahen" Kontakt ist es nicht weiter erstaunlich, daß auch das Funktionsprinzip dieses vielseitigen Eingabegerätes die Neugier jedes frischgebackenen Amiga-Besitzers erregt.

Wer schon etwas länger mit seinem Computer arbeitet, hat sicherlich bereits einen Blick hinter die Kulissen geworfen und die Kugel auf der Unterseite des Gehäuses herausgenommen. Man erkennt einen schalenförmigen Hohlraum mit zwei zueinander senkrecht stehenden Achsen, die die Bewegungen der Kugel in eine horizontale und eine vertikale Komponente zerlegen. Das mechanische Funktionsprinzip dürfte soweit klar sein. Um allerdings eine für den Computer verwertbare Information zu erhalten, muß die analoge Achsendrehung in ein elektrisches Signal umgesetzt werden. Dazu ist auf jeder Achse eine Lochscheibe befestigt, die von jeweils zwei (im Winkel von 180 Grad angeordneten) Lichtschranken abgetastet wird. Sobald sich eine Achse dreht, bewegt sich auch die zugehörige Lochscheibe und die Lichtschranke registriert einen ständigen Wechsel von Unterbrechung bzw. Durchlaß des Lichtstrahls.

Das resultierende Rechtecksignal wird verstärkt und als horizontaler (H) bzw. vertikaler (V) Bewegungsimpuls zum Computer geschickt. Der kann zwar jetzt die Geschwindig-

keit der Bewegung ermitteln (aus der Frequenz des Rechteckimpulses), ihre Orientierung (hoch, runter, rechts, links) bleibt aber nach wie vor unbekannt.

Hier kommt die zweite Lichtschranke zum Zuge. Durch die versetzte Anordnung liefert sie ein phasenverschobenes Signal – den sogenannten Quadraturimpuls (HQ und VQ). Je nach Bewegungsrichtung eilt dieser dem anderen Rechteckpuls leicht "voraus" oder "hinkt" etwas hinter. Alle vier Signale zusammen enthalten dann den vollständigen, richtungsorientierten Bewegungsimpuls für die horizontalen und vertikalen Mausezähler auf dem Denise-Chip (insgesamt vier Stück, zwei pro Game-Port). Der Zähltakt wird aus der logischen EOR-Verknüpfung zwischen Achsen- und Quadraturimpuls und der Invertierung der Quadratursignale gewonnen. Pro Achse entstehen also zwei neue Signale, die einen 6-Bit-Counter mit einer Schrittweite von 4 Impulsen takten und zwei niederwertige Bits direkt steuern. Das hochfrequente Verknüpfungssignal bestimmt dabei den Pegel des untersten Zählbits (x0 für horizontale Bewegungen, y0 für vertikale), das achsengespiegelte Quadratursignal beeinflusst den Pegel des nächsthöheren Bits (x1 oder y1). Counter und Zählbits ergänzen sich schließlich zu einem 8-Bit-Zählregister mit einem Wertebereich von 0 bis 255.

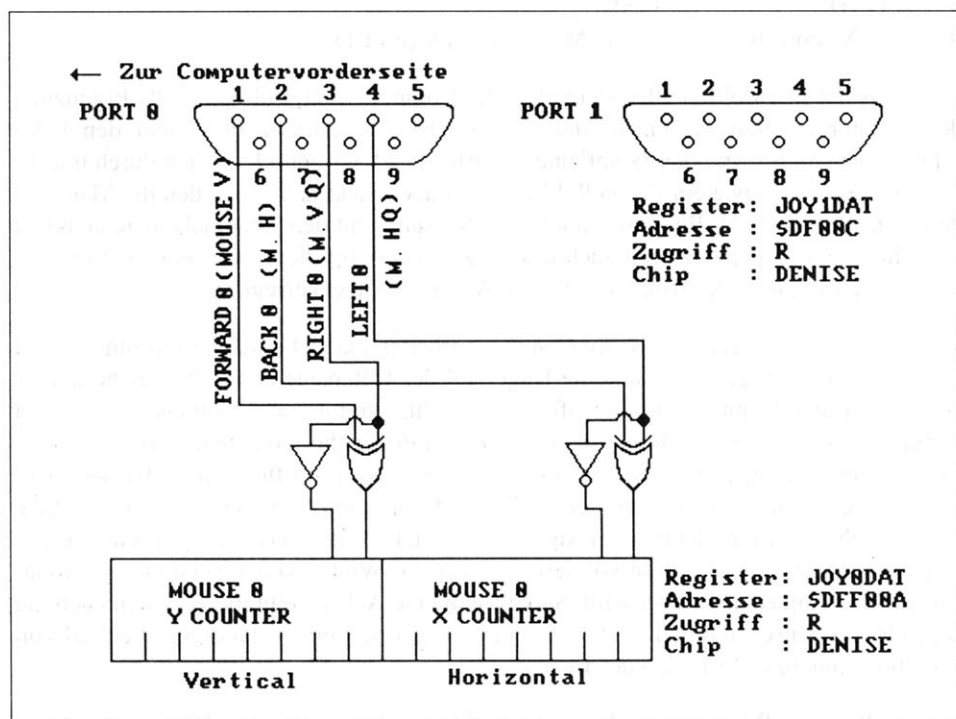


Abb. 6.7: Die Mauseingänge der Game-Ports

Horizontales und vertikales Zählregister des Game-Ports sind in einem 16-Bit-Register mit der Bezeichnung JOYxDAT zusammengefaßt, daß für Port 0 an Offset \$00A, für Port 1 an Offset \$00C in der Registertabelle der Customchips liegt.

Register	Adresse	Zugriff	Funktion																																
JOY0DAT	\$DFF00A	r	<p>horizontale und vertikale Mausposition am Game-Port-0</p> <p>Registerbelegung:</p> <table><tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td></tr><tr><td>y7</td><td>y6</td><td>y5</td><td>y4</td><td>y3</td><td>y2</td><td>y1</td><td>y0</td><td>x7</td><td>x6</td><td>x5</td></tr></table> <table><tr><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>x4</td><td>x3</td><td>x2</td><td>x1</td><td>x0</td></tr></table> <p>höherwertiges Byte (y7-y0): vertikaler Zähler niederwertiges Byte (x7-x0): horizontaler Zähler</p>	15	14	13	12	11	10	9	8	7	6	5	y7	y6	y5	y4	y3	y2	y1	y0	x7	x6	x5	4	3	2	1	0	x4	x3	x2	x1	x0
15	14	13	12	11	10	9	8	7	6	5																									
y7	y6	y5	y4	y3	y2	y1	y0	x7	x6	x5																									
4	3	2	1	0																															
x4	x3	x2	x1	x0																															
JOY0DAT	\$DFF00C	r	<p>horizontale und vertikale Mausposition am Game-Port-1</p> <p>Registerbelegung:</p> <table><tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td></tr><tr><td>y7</td><td>y6</td><td>y5</td><td>y4</td><td>y3</td><td>y2</td><td>y1</td><td>y0</td><td>x7</td><td>x6</td><td>x5</td></tr></table> <table><tr><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>x4</td><td>x3</td><td>x2</td><td>x1</td><td>x0</td></tr></table> <p>höherwertiges Byte (y7-y0): vertikaler Zähler niederwertiges Byte (x7-x0): horizontaler Zähler</p>	15	14	13	12	11	10	9	8	7	6	5	y7	y6	y5	y4	y3	y2	y1	y0	x7	x6	x5	4	3	2	1	0	x4	x3	x2	x1	x0
15	14	13	12	11	10	9	8	7	6	5																									
y7	y6	y5	y4	y3	y2	y1	y0	x7	x6	x5																									
4	3	2	1	0																															
x4	x3	x2	x1	x0																															
JOYTEST	\$DFF036	r	<p>schreibt zu Testzwecken in die Mauszähler Game-Ports</p> <p>Registerbelegung:</p> <table><tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td></tr><tr><td>y7</td><td>y6</td><td>y5</td><td>y4</td><td>y3</td><td>y2</td><td>xx</td><td>xx</td><td>x7</td><td>x6</td><td>x5</td></tr></table> <table><tr><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr><tr><td>x4</td><td>x3</td><td>x2</td><td>xx</td><td>xx</td></tr></table>	15	14	13	12	11	10	9	8	7	6	5	y7	y6	y5	y4	y3	y2	xx	xx	x7	x6	x5	4	3	2	1	0	x4	x3	x2	xx	xx
15	14	13	12	11	10	9	8	7	6	5																									
y7	y6	y5	y4	y3	y2	xx	xx	x7	x6	x5																									
4	3	2	1	0																															
x4	x3	x2	xx	xx																															

Es können nur die 6-Bit-Zähler für horizontale und vertikale Mausposition gesetzt werden. Die unteren zwei Bits gewinnt man direkt aus dem Pegelniveau der Zählakte.

Die Auswertung der Mauszählers obliegt normalerweise dem Betriebssystem.

Dazu wird das JOY0DAT-Register von Game-Port-0 jede 1/50 Sekunde (Vertical-Blanking-Interrupt, PAL-Frequenz 50 Hz) ausgelesen und der neue Zählerstand mit dem zuletzt ermittelten verglichen. Die Differenz zwischen altem und neuem Wert entspricht genau der relativen Mausbewegung in diesem Zeitintervall.

Auf Basis der Tatsache, daß die Maus pro Zentimeter ungefähr 80 Zählimpulse liefert, wird die Differenz in eine Bildschirmbewegung umgerechnet und zu den aktuellen Koordinaten des Mauspeils addiert. Sobald die neuen Werte im Listenkopf der Spritedaten eingetragen sind, führt die Sprite-Hardware den Mauspeil automatisch nach. Der Pfeil befindet sich in einem Koordinatensystem mit Nullpunkt in der linken, oberen Bildschirmcke, so daß die Erhöhung des horizontalen Mauszählers einer Bewegung nach rechts, das gleiche beim vertikalen Zähler dagegen einer Verschiebung nach unten entspricht. Wie aber läßt sich diese Orientierung feststellen, wenn die 8-Bit-Zähler bereits alle 4 cm einen Über- bzw. Unterlauf produzieren (255 / 80 Impulse pro Zentimeter)? Ein kleiner Trick löst auch dieses Problem. Man geht einfach davon aus, daß innerhalb des Abfrageintervalls von 1/50 Sekunde kaum mehr als 127 Zählimpulse erzeugt werden (es sei denn, man jagt die Maus mit 0,8 m/s über den Schreibtisch). Dann aber müssen alle Zählerdifferenzen, deren Betrag über 127 liegt, durch Über- bzw. Unterschreitung des Wertebereiches entstanden sein.

Ist die ermittelte Zählerdifferenz größer als +127, folgt die effektive Mausbewegung aus dem Term:

$$-255 + (\text{Zählerstand.neu} - \text{Zählerstand.alt})$$

Ist sie kleiner als -127, führt die Summe aus Zählerdifferenz und oberer Grenze des Wertebereiches zum richtigen Ergebnis:

$$255 + (\text{Zählerstand.neu} - \text{Zählerstand.alt})$$

Sehen wir uns dazu ein paar Beispiele an:

Zählerstand		Differenz neu - alt	effektive Mausbewegung	Überlauf/ Unterlauf	Bewegungs- richtung
alt	neu				
70	150	+80	+80	–	rechts (unten)
180	60	–120	–120	–	links (oben)
30	100	+70	+70	–	rechts (unten)
10	140	+130	–125	Unterlauf	links (oben)
200	50	–150	+105	Überlauf	rechts (unten)
140	5	–135	+120	Überlauf	rechts (unten)

Nach diesem Exkurs in die Technik der Mausabfrage läßt sich das Phänomen des unberechenbaren Mausfeilverhaltens bei extrem schnellen Bewegungen sofort erklären. Die Systemsoftware geht davon aus, daß pro Abtastintervall nicht mehr als 127 Zählimpulse erzeugt werden. Sobald die Geschwindigkeit über 80cm/s hinausgeht, bricht diese Eselsbrücke kläglich zusammen und der sonst so folgsame Pfeil irrt nervös durch die Gegend.

Der Joystick am Game-Port

Pin	Leitung	Joystickanschluß
1	M0V (V)	Oben
2	M0H (H)	Unten
3	M1V (VQ)	Links
4	M1H (HQ)	Rechts
5	PxY	—
6	PA6/	Feuerknopf PA7 "
7	+5V	+5V
8	GND	GND
9	PxX	—

Die vier Schalter für die Hebelstellung des Joysticks liegen an denselben Pins wie die Achsen- und Quadraturimpulse der Maus, folglich gelangen sie auch auf die Mauseingänge von Denise und werden als Bewegungsimpulse ausgewertet.

Wie wir wissen, generiert Denise die Zählakte sowohl aus der logischen EOR-Verknüpfung von Achsen- und Quadraturimpuls (H EOR HQ bzw. V EOR VQ), als auch durch die Spiegelung (Invertierung) der phasenverschobenen Quadratursignale (NOT HQ, NOT VQ). Während letztere den Pegel der Einser-Bits in den Zählregistern für horizontale und vertikale Mauspositionen takten, steuern die Verknüpfungssignale das niederwertigste Bit (0).

Genau darin liegt der Schlüssel zur Joystickabfrage, die acht Stellungen des Hebels müssen aus den vier Bits 1 und 0 bzw. 8 und 9 des JOYDAT-Registers hervorgehen.

Das läßt sich am leichtesten für die Positionen "Links" (VQ) und "Rechts" (HQ) nachvollziehen. Bei geschlossenem Schalter werden die entsprechenden Eingänge auf Masse gezogen und liegen damit auf LOW-Pegel. Da Denise die Quadratursignale invertiert, haben die Bits 1 (x1) und 9 (y1) des JOY0DAT-Registers immer das umgekehrte Pe-

gelniveau der Eingangsimpulse. Für die Stellungen "Links" und "Rechts" sind sie also logisch HIGH (Wert 1).

Komplizierter wird es bei den Joystickpositionen "Oben" bzw. "Unten". Ihre Pegel können nicht direkt aus den Bits 0 (x0) und 8 (y1) abgelesen werden, sondern sind in der logischen Verknüpfung der beiden unteren Bits eines Zählers verschlüsselt ($x0 \text{ EOR } x1$, $y0 \text{ EOR } y1$). Nach den Regeln der formalen Logik kann man die Eingangspegel aber durch erneute Verknüpfung rekonstruieren. Der Schalter für die Hebelstellung "Oben" ist also genau dann geschlossen, wenn der Ausdruck $y0 \text{ EOR } (\text{not } y1)$ logisch unwahr ($= 0$) ist (an die invertierten Eingänge HQ und VQ denken!).

Das Werkzeug zur Abfrage sämtlicher Hebelstellungen sind demnach vier logische Operationen, deren Wahrheitsgehalt geprüft werden muß:

$x1 = 1$	wenn wahr, dann	Hebel nach rechts
$y1 = 1$	wenn wahr, dann	Hebel nach links $x0 \text{ EOR}$
$(\text{not } x1) = 0$	wenn wahr, dann	Hebel nach unten $\Leftrightarrow x0 \text{ EOR}$
$x1 = 1$	wenn wahr, dann	Hebel nach unten $\Leftrightarrow y0 \text{ EOR}$
$(\text{not } y1) = 0$	wenn wahr, dann	Hebel nach oben $\Leftrightarrow y0 \text{ EOR}$
$y1 = 1$	wenn wahr, dann	Hebel nach oben $\Leftrightarrow y0 \text{ EOR}$

Zur Ermittlung der Joystick-Stellung bieten sich zwei Wege an:

Der eine ist die schrittweise Auswertung der logischen Ausdrücke durch ein kleines Maschinenprogramm. Der andere geht davon aus, daß nach Ausblendung sämtlicher nicht relevanter Bits im JOYDAT-Register, jeder Hebelstellung ein bestimmter Wert zugeordnet ist. In der Programmiersprache C kann dieser Wert dann mittels einer Switch()-Anweisung systematisch abgefragt werden. Das geht besonders komfortabel, wenn die Wertzuweisung und Initialisierung bereits in Form einer Include-Datei vorliegt:

Include-Datei "joyquest":

```
#define OST 0x0003
#define WEST 0x0300
#define NORD 0x0100
#define SUED 0x0001
#define NORDOST 0x0103
#define SUEDOST 0x0002
#define NORDWEST 0x0200
#define SUEDWEST 0x0301
```

```

#define JOY0DAT (*( (WORD *) 0xDFF00a) & 0x303)
                                ;alle Bits bis auf 0,
                                1, 8 und 9
#define JOY1DAT (*( (WORD *) 0xDFF00c) & 0x303)
                                ;ausblenden und
                                Register holen
#define FIRE0 !(*( (BYTE *) 0xBFE001) & 64*1)
                                ;CIA-A, PA6 auf LOW-
                                Pegel?
#define FIRE1 !(*( (BYTE *) 0xBFE001) & 64*2)
                                ;CIA-A, PA7 LOW ?

```

Abfrage in C:

```

#include <joyquest>
.....
.....
switch(JOY0DAT)
{
    case OST: rechts();
              break;
    case WEST: links();
              break;
    .....
}
if(FIRE0) fire();

```

Paddles, XY-Joysticks, Potentiometer

Die Leitungen POT0X, POT0Y, POT1X und POT1Y des Paula-Chips (jeweils zwei pro Game-Port) eignen sich zum Anschluß veränderlicher Widerstände (sogenannter Potentiometer), die zwischen +5V und dem entsprechenden Analogeingang liegen.

Das eröffnet dem Amiga eine Reihe von interessanten Analoggeräten, wie das Paddle oder der XY-Joysticks, bei dem die Hebelstellung von zwei veränderlichen Widerständen (einer auf der X-Achse, der andere auf der Y-Achse) dargestellt wird.

Analog-Joysticks bieten eine extrem feine Auflösung, die lediglich durch die digitale Umsetzung der Widerstandswerte begrenzt ist – dagegen wirkt die Schalterversion mit ihren acht Hebelstellungen geradezu plump.

Die Register des Analog-Ports:

<i>Register</i>	<i>Adresse</i>	<i>Zugriff</i>	<i>Funktion</i>
POT0DAT	\$DFF012	r	horizontaler und vertikaler 8-Bit-Zähler für Analoggeräte am Game-Port 0 (enthalten die gemessenen Widerstandswerte der X- und Y-Potentiometer)
Registerbelegung:			
15 14 13 12 11 10 9 8 7 6 5 y7 y6 y5 y4 y3 y2 y1 y0 x7 x6 x5 4 3 2 1 0 x4 x3 x2 x1 x0			
höherwertiges Byte (y7-y0): vertikaler Zähler niederwertiges Byte (x7-x0): horiz. Zähler			
POT1DAT	\$DFF014	r	horizontaler und vertikaler 8-Bit-Zähler für analoge Eingabegeräte am Game-Port 1 (siehe oben)
Registerbelegung:			
15 14 13 12 11 10 9 8 7 6 5 y7 y6 y5 y4 y3 y2 y1 y0 x7 x6 x5 4 3 2 1 0 x4 x3 x2 x1 x0			
POTGO	\$DFF034	w	Steueregister der Analog-Ports (beide Game-Ports)

Register	Adresse	Zugriff	Funktion
----------	---------	---------	----------

Registerbelegung:

Bit	Label	Funktion
15	OUTRY	POT0Y-Leitung (Port 0) von Eingang (Bit = 0) auf Ausgang (Bit = 1) umschalten
14	DATRY	Datenbit (Eingangs- /Ausgangspegel) der POT0Y-Leitung
13	OUTRY	POT0X Eingang/Ausgang
12	DATRY	Datenbit POT0X
11	OUTLY	POT1Y (Port 1) Eingang/Ausgang
10	DATLY	Datenbit von POT1Y
9	OUTLY	POT1X Eingang/Ausgang
8	DATLY	Datenbit (Ausgangs/Ein- gangspegel der POT1X- leitung
7 bis 1	–	unbenutzt
0	START	initialisiert die Meßschal- tung (Kondensatoren werden entladen, Zähler gelöscht) und startet Wider- standsmessung.

POTGOR	\$DFF016	r	POTGO-Leseadresse (Belegung siehe oben)
--------	----------	---	---

Die Meßschaltung auf dem Paula-Chip arbeitet im Prinzip wie ein integrierender Analog-/Digitalwandler (mehr dazu im Sound-Kapitel), allerdings mit einer bescheidenen Auflösung von nur einem Bit.

Intern liegt zwischen den Analogeingängen und Masse jeweils ein Kondensator mit einer Kapazität von 47 nF. Die Messung beginnt mit dem Setzen des START-Bits im POTGO-Register. Zur Initialisierung löscht Paula die Zähler in den beiden POTDAT-Registern und legt die Kondensatoren kurz auf Masse, so daß sie ihre Restladung abgeben und sich vollständig entladen. Während sie nun von den, über die Potentiometer fließenden Strömen aufgeladen werden, zählt der Customchip die entsprechenden X- und Y-Zähler im Takt der Zeilenfrequenz (PAL-System 15625 Hz) hoch. Mit jeder dar-

gestellten Bildschirmzeile (Dauer: 63,5 Mikrosekunden) erhöht sich der Zählerstand um einen weiteren Schritt.

Am Ausgang der Kondensatoren vergleicht ein Komparator die anliegende Spannung mit einem festgelegten Referenzwert. Ist dieser nach einiger Zeit erreicht, werden die Zähler sofort gestoppt. Je größer der vorgeschaltete Widerstand, desto höher auch der Stand im Zählregister.

In den technischen Unterlagen wird ein maximaler Widerstandswert von 470 kOhm (Toleranz: $\pm 10\%$) empfohlen. Dieser Wert entspricht einem Zählerstand von 255, noch größere Widerstände bewirken einen Zählerüberlauf und verfälschen das Meßergebnis. Um den gesamten Wertebereich der Zähler auszunutzen, empfiehlt es sich, die Messung in der vertikalen Austastlücke zu starten.

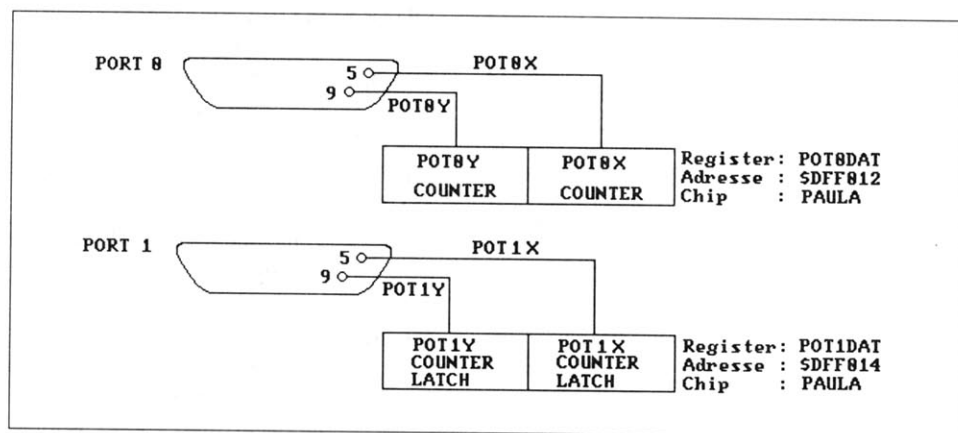


Abb. 6.8: Die Analogeingänge der Game-Ports

Bei PAULAs Analogeingängen handelt es sich um Multifunktions-Ports, die nicht nur zur Widerstandsmessung, sondern auch als digitale Datenleitungen zur Abfrage logischer Zustände geeignet sind. Wie die Portleitungen der CIA-Chips, können sie einzeln als Eingang oder Ausgang geschaltet werden.

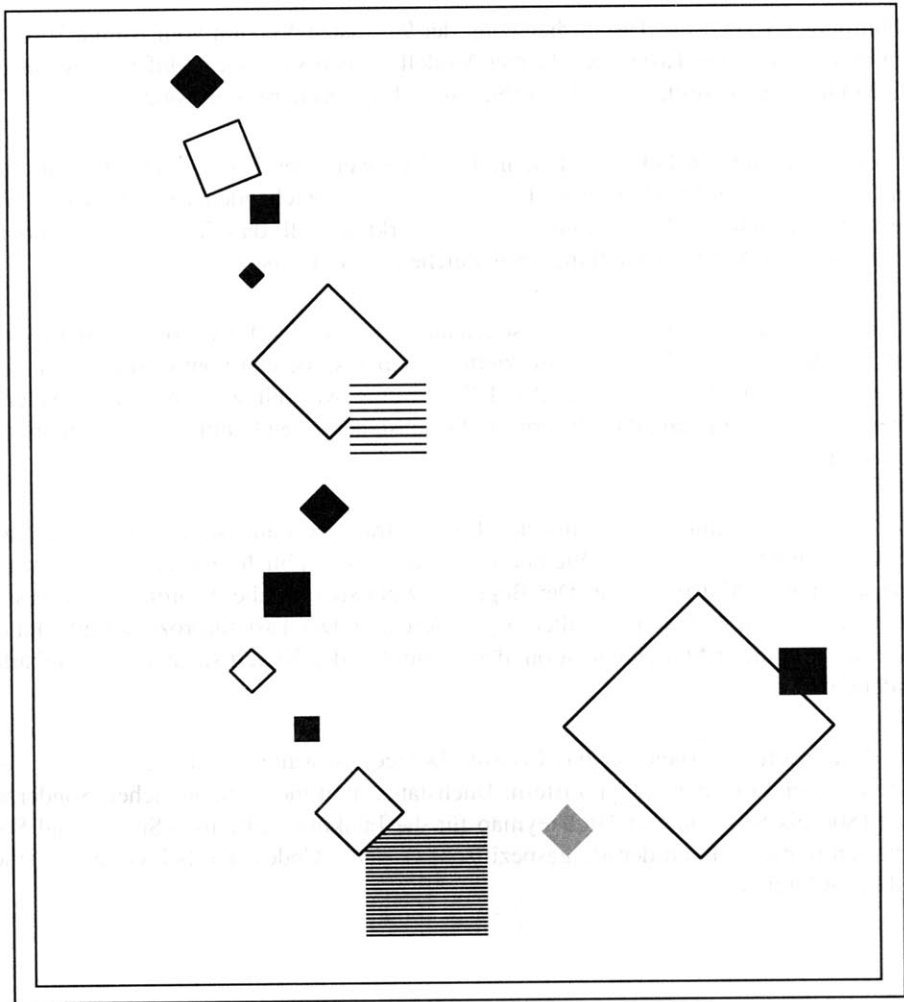
Zur Steuerung dienen die OUTRY/OUTLY-Bits (Ausgabe rechter und linker Port) im POTGO-Register. Jeder Leitung ist ein solches Bit zugeordnet, sein Zustand bestimmt die Datenrichtung (0 = Eingang, 1 = Ausgang). Zusätzlich gibt es jeweils noch ein Datenbit DATRY/DATLY, dessen Wert immer das Pegelniveau der Portleitung repräsentiert.

tiert. Im Eingabemodus kann es über das POTGOR-Register (Adresse \$DFF012) gelesen, im Ausgabemodus durch die Schreibadresse (POTGO, \$DFF034) gesetzt werden.

Sobald eine Leitung als Ausgang geschaltet ist, koppelt Paula die Bewertungsschaltung vom Kondensator ab und gibt den Pegel des Datenbits aus. Die logischen TTL-Pegel werden allerdings erst nach einer erheblichen Verzögerung von bis zu 300 Mikrosekunden erreicht (von logisch Low auf logisch High), da sich der Kondensator entsprechend auf- bzw. entladen muß.

Kapitel 7

Die Interface - Hardware Teil 2



Die Tastatur

Seit der Markteinführung der ersten Amiga-Systeme hat nicht nur die Grundplatine mehrere Generationen durchlaufen, sondern auch die Tastatur kann eine bewegte (oft äußerlich gar nicht sichtbare) Entwicklungsgeschichte vorweisen. Im Gegensatz zu vielen anderen Herstellern ist es Commodore aber immer wieder gelungen, die Softwarekompatibilität weitgehend zu erhalten.

Abbildung 7.1 zeigt die Tastaturbelegung der 94-Tasten-Version vom Amiga 2000. Zu den ursprünglich 89 Tasten des 1000er-Modells haben sich noch fünf weitere für den PC-Modus hinzugesellt, die in der Abbildung schraffiert dargestellt sind.

In der rechten oberen Ecke der Tastenfelder ist jeweils der Tastencode aufgeführt, den der Computer beim Drücken (bzw. Loslassen) der entsprechenden Taste empfängt. Wer die ASCII-Tabelle noch etwas im Kopf hat, merkt schnell, daß diese Codes nichts mit der gewohnten ASCII-Darstellung eines Zeichens zu tun haben.

Vielmehr handelt es sich dabei um sogenannte Raw-Keycodes ("Rohe Tastencodes"), also unbehandelte, noch nicht modifizierte Codebytes, die erst vom Computer (anhand einer Tabelle) in die vertrauten ASCII-Werte umgewandelt werden. (Bei genauerem Hinschauen erkennt man, daß Ziffern- und Buchstabentasten kontinuierlich durchnummeriert sind.)

Wer schon etwas mit der Technik der Tastaturabfrage vertraut ist, wird über die Raw-Keycodes nicht erstaunt sein. Wie heutzutage allgemein üblich, besitzt auch der Amiga eine sogenannte Matrixtastatur. Der Begriff bezieht sich auf die Anordnung der Tasten, die matrixartig in Zeilen und Spalten organisiert sind. Der Tastaturprozessor erkennt die jeweilige – an ihrer Matrizenposition, der Computer identifiziert sie an der vereinbarten Codenummer.

Die Systemsoftware (genauer das Console-Device) verwaltet die Tastencodes in zwei Tabellen, der Lowkeymap für Ziffern, Buchstaben und die gebräuchlichen Sonderzeichen (\$00 bis \$37) und der Highkeymap für die Funktions-, Pfeil-, <Shift>- und Sondertasten (einschließlich der amigaspezifischen), deren Codes sich bei \$40 anschließen und bis \$6F reichen.

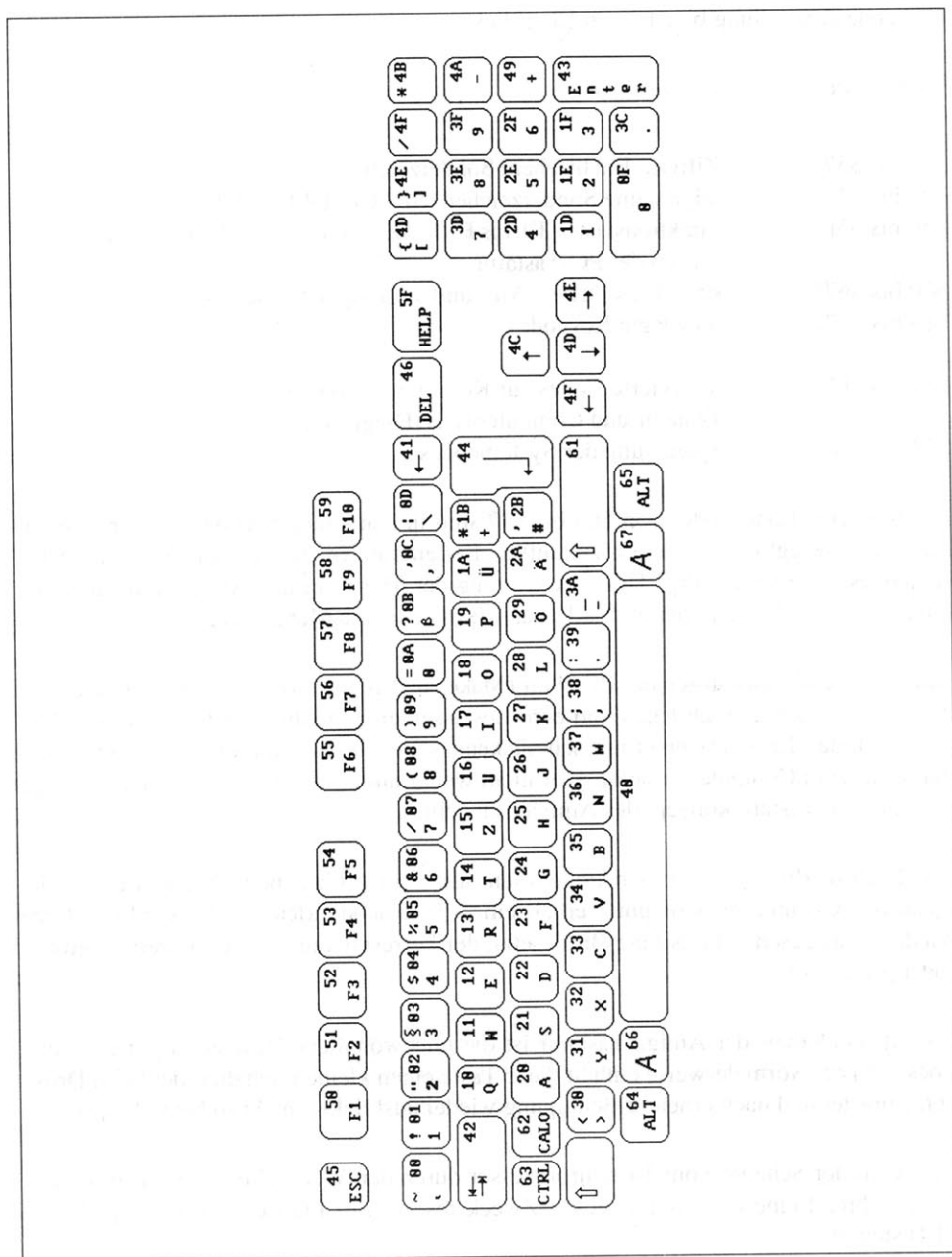


Abb. 7.1: Tastaturbelegung des Amiga 2000

Folgende Aufstellung bietet einen Überblick:

<i>Tastencodes</i>	<i>Beschreibung</i>
\$00 bis \$37:	Ziffern, Buchstaben, Sonderzeichen
\$40 bis \$47:	allgemeine Sonderzeichen (SPACE, DEL, TAB etc.)
\$50 bis \$5F:	Funktionstasten F1 bis F10, HELP und die fünf zusätzlichen Tasten der PC-Tastatur
\$60 bis \$67:	<Shift>, <Ctrl>, <Alt> und amigaspezifische Sondertasten
\$68 bis \$77:	unbelegte Keycodes
\$F9 bis \$FF:	reservierte Codes zur Kommunikation zwischen Tastatur und Computer (z.B. Diagnose)
\$78 bis \$7F:	Spiegelung der Systemcodes

Die fünfzehn Tastencodes von \$68 bis \$77 sind in den jetzigen Versionen der Tastatur noch nicht belegt und stehen für zukünftige Erweiterungen zur Verfügung. Da diese Tastencodes ohne aufwendige Umstrukturierung der Matrix in den Abfrageprozeß eingebunden werden können, bleibt den Tastaturdesignern eine Menge Spielraum.

Ähnlich wie die Tastaturen der PC-Welt (Make- und Breakcodes) differenziert auch der Amiga zwischen dem Drücken und dem Loslassen einer Taste; allerdings mit dem Unterschied, daß Letzteres nicht nur schmückendes Beiwerk ist (die PC-Breakcodes werden zwar zum Computer gesandt, aber nicht weiter ausgewertet), sondern eine wichtige Funktion im Tastaturkonzept des Amigas darstellt.

Das höchste Bit (Nr. 7) im Codebyte dient als KeyUp-Flag, beim Niederdrücken der Taste ist es 0 und der Computer erhält einen Tastencode kleiner 127; wird die Taste wieder losgelassen, ist das Flag-Bit gesetzt, der korrespondierende Codewert ist größer (oder gleich) 127.

Eine Besonderheit der Amiga-Tastatur ist die ungewöhnliche Realisierung der <Caps-Lock>-Taste. Normalerweise enthält diese Taste einen kleinen Schalter, der beim Drücken einrastet und nach erneuter Betätigung wieder ausklinkt – nicht so beim Amiga.

Hier wird der Schalter vom Tastaturprozessor durch den Wechsel eines Statusbits emuliert, während eine (eigens für diesen Zweck reservierte) I/O-Leitung die Caps-Lock-LED steuert.

Tastatur mit "Köpfchen" – der Tastaturprozessor

Der Amiga verfügt über eine sogenannte "intelligente" Tastatur, da das Innere nicht nur aus einem Netz von Mikroschaltern für die Tastenabfrage besteht, sondern auch einen (durchaus eigenständigen) 8-Bit-Mikroprozessor (bzw. Mikrocontroller) beherbergt.

Natürlich hat sich hier im Laufe der Zeit einiges getan. So verläßt sich die Tastatur des 1000er auf einen 6500/1, der A500 schwört auf die Dienste des bewährten 6502 (eingefleischte C64-Freaks mögen feuchte Augen bekommen) und beim B2000 werkelt sogar ein Intel-Design, der IO-Chip 8049 (es sind aber auch schon 6570er gesichtet worden).

Egal, welcher Chip nun Verwendung findet, die grundlegenden Funktionen unterscheiden sich kaum und die Technik der Matrixabfrage ist von Modell zu Modell gleich geblieben. Im folgenden soll deshalb Aufbau, Schaltung und Arbeitsweise des 6500 (A1000) als exemplarische Grundlage dienen und stellvertretend für die nachfolgenden Tastaturgenerationen genauer untersucht werden.

Beim 6500 handelt es sich um einen Ein-Chip-Mikrocomputer, da sämtliche Komponenten eines funktionstüchtigen Computersystems, wie RAM, ROM, E/A-Ports und Interruptlogik, bereits auf dem Chip integriert sind. Die Basis dieser MOS-Technology-Entwicklung bildet ein alter Bekannter, der "8-Bit-Haudegen" 6502, den man um 64 Byte On-Chip-RAM, 2 KByte ROM-Speicher (organisiert zu 2048 x 8), 4 bidirektionale 8-Bit-Ports und einen Zähler erweitert hat. Im A1000 arbeitet eine 1-MHz-Version, die mit "leichtem Hochdruck" auf ca. 1,5 MHz betrieben wird (3-MHz-Quartz-Takt, intern durch 2 geteilt).

Ein derartiger Ein-Chip-Mikrocomputer ist die preiswerteste Lösung, denn im Prinzip benötigt er keinerlei Zusatzbausteine und kann (nachdem die ROM-Maske mit dem individuell zugeschnittenen Steuerprogramm erstellt ist) in großen Stückzahlen produziert werden. Auf der Tastaturplatine des Amigas steht ihm aber trotzdem ein Helfer zur Seite. Es handelt sich dabei um einen Timerbaustein (Typ 556), der als Watchdog (Wachhund) die Abfrage der TastaturmatriX überwacht. Droht ein Endloszyklus (beispielsweise durch einen Matrixfehler), greift er klärend ein und generiert ein RESET-Signal für den Tastaturprozessor.

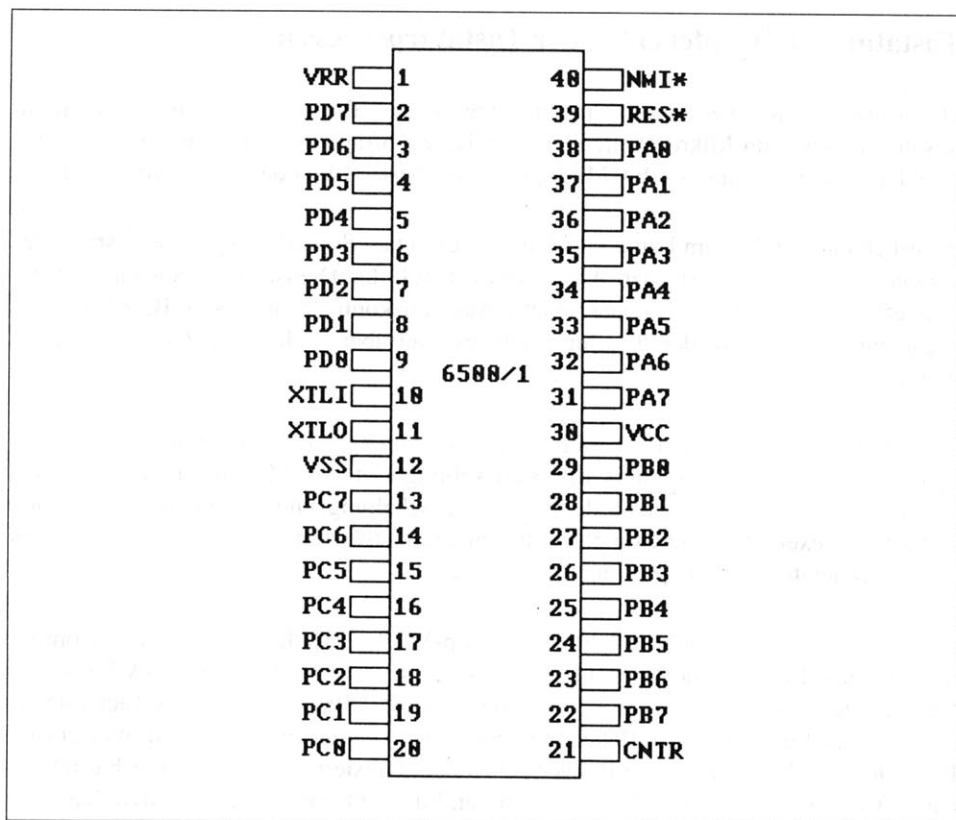


Abb. 7.2: Die Pinbelegung des Tastaturprozessors

"Tastsinn" – die Schaltung der Tastaturplatine

Bisher war zwar immer von der Abfrage der Tastaturmatrix die Rede. Wie sieht diese aber im Detail aus und welche Rolle spielt der Tastaturprozessor? Sehen wir uns dazu die Schaltung der Tastaturplatine am Beispiel des A1000 an:

Die sieben zentralen Steuertasten <Shift>, <Alt> und <Amiga> (jeweils links und rechts), sowie <Ctrl> sind direkt mit den Eingängen 0 bis 6 von I/O-Port B verbunden und werden gesondert ausgewertet. Sie liegen zwischen Masseanschluß und ihrer Portleitung, so daß sie im gedrückten Zustand einen LOW-Pegel am Eingang erzeugen.

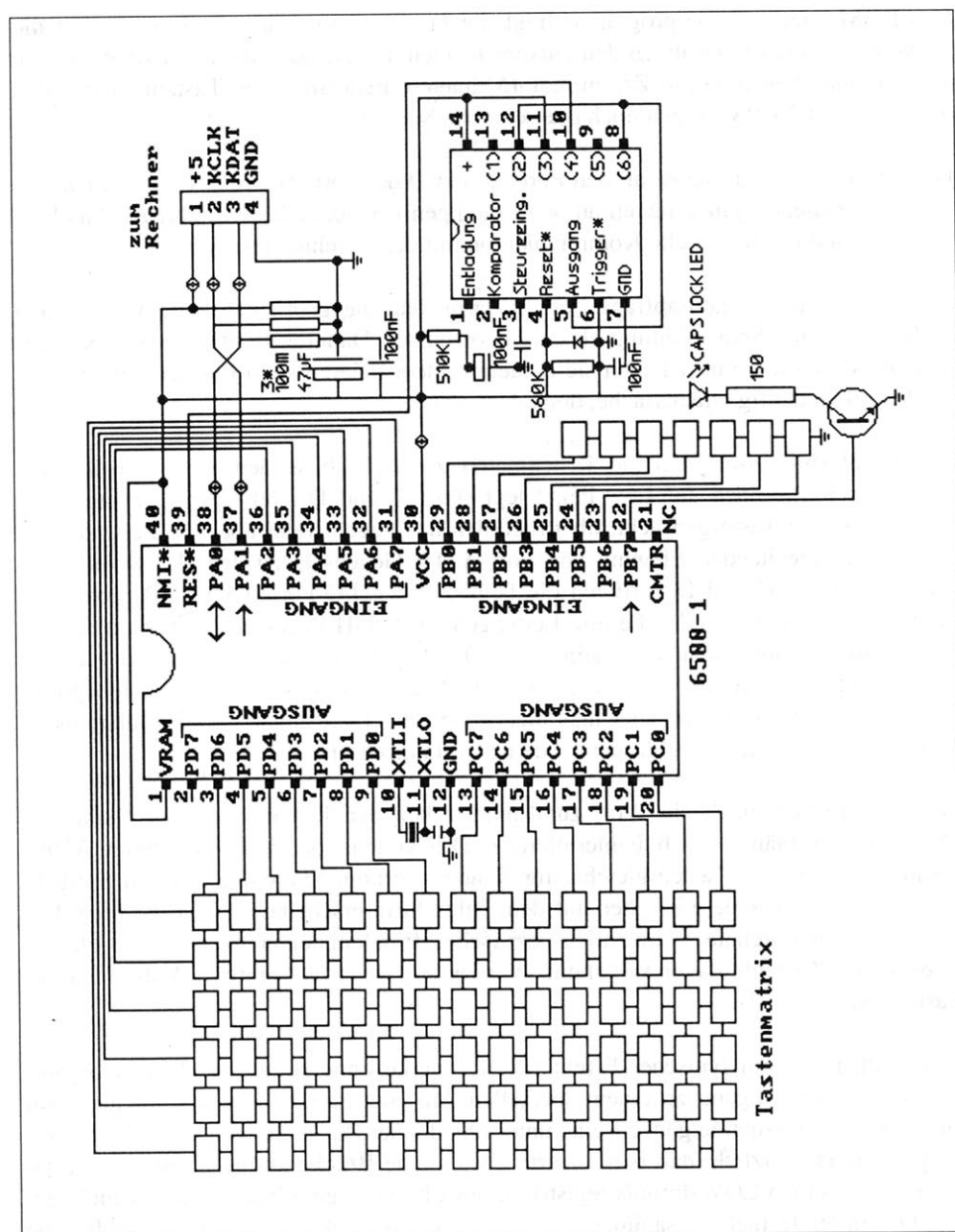


Abb. 7.3: Die Schaltung der Tastaturplatine beim A1000

Das ROM-interne Steuerprogramm fragt die Porteingänge zyklisch ab, registriert die LOW-Pegel und ordnet ihnen den entsprechenden Tastencode zu. Alle anderen Tasten sind in einer Matrix von 6 Zeilen mal 15 Spalten organisiert (die Tastaturen der Modelle 500 und 2000 verfügen noch über eine 16. Spalte).

Die Spaltenleitungen liegen an den Ports C und D des 6500 (Leitung PD7 ist nicht belegt), die Zeileneingänge liegen an den Leitungen PA2 bis PA7 von Port A; Pins PA1 und PA2 sind für die serielle Kommunikation mit dem Rechner reserviert.

Das Prinzip der Tastaturabfrage besteht darin, daß die gedrückte Taste jeweils eine Zeilen- und eine Spaltenleitung elektrisch verbindet. Dadurch besitzt sie eine Koordinate im Matrixnetz mit der man sie eindeutig identifizieren kann (tatsächlich ist diese Eindeutigkeit häufig Glückssache, doch dazu später!).

Der Tastaturprozessor fragt die Tastenmatrix zyklisch ab, in dem er die Spaltenausgänge der Reihe nach auf Low-Pegel legt (Port C und D sind Open-Collector-Ausgänge). Ist eine Taste gedrückt, überträgt sich das Pegelniveau über die Zeilenleitung auf den entsprechenden Eingang von Port A. Im Gegensatz zu den als Ausgang geschalteten Ports C und D verfügen die beiden ersten I/O-Ports (A und B) über integrierte Pullup-Widerstände, die ihre Leitungen auf HIGH-Pegel halten. Sobald der Tastaturprozessor auf einem Zeileneingang LOW-Pegel registriert, ermittelt er die momentan aktive Spaltenleitung (die einzige auf LOW) und bestimmt die Tastenkoordinate. Das Steuerprogramm des Chips identifiziert die Taste anhand ihrer Matrizenposition und speichert den zugehörigen Code im Tastaturpuffer.

Wie versprochen, noch ein Wort zur Eindeutigkeit der Tastenerkennung. Nicht jede Matrixtastatur glänzt durch Fehlertoleranz. Drückt man beispielsweise beim A1000 mehrere benachbarte Tasten gleichzeitig, kann es vorkommen, daß außer den gedrückten noch ein paar andere Zeichen auf dem Bildschirm erscheinen. Derartige Mehrdeutigkeiten sind durchaus kein amigaspezifisches Problem, erstaunlicherweise bleiben selbst teure PCs nicht davon verschont. Was ist der Grund für derartige Willkürakte der Tastaturen?

Eigentlich ist es ein typischer Effekt für Netzwerke ohne strenge Richtungsvorgabe. Um von einem Eingang zu einem speziellen Ausgang zu kommen, gibt es nicht nur einen Weg, sondern eine ganze Reihe unterschiedlicher Pfade, die alle zum Ziel führen. So passiert es plötzlich, daß benachbarte Tasten eine Brücke aufbauen und der Tastaturprozessor einen LOW-Impuls registriert, obwohl die zugeordnete Taste gar nicht gedrückt wurde. Je mehr zusammenliegende Tasten gedrückt werden, desto größer die Verwirrung.

Glücklicherweise kann man nicht alle Tastaturen so leicht austricksen, denn Dioden an den Kreuzungspunkten der Matrix verhindern die Brückenbildung und schließen jede Mehrdeutigkeit sicher aus. Die Lösung des Problems erscheint derart banal, daß man sich ernsthaft fragt, warum Tastaturen in den Zeiten von WYSIWYG (What you see is what you get) eine Zwei-Klassen-Gesellschaft bilden, aus solchen mit und solchen ohne – ein Riß, der sich sogar mitten durch die Amiga-Familie zieht. Beim B2000 hat sich Commodore nämlich nicht mehr "lumpen lassen" und der Tastatur Dioden spendiert.

"Small-Talk" – die Kommunikation zwischen Rechner und Tastatur

System:	A1000	A500	A2000/B2000
Verbindung:	Flachstecker	Kabelleiste (intern)	DIN-Buchse
1	+5V	KCLK-Takt	KCLK-Takt
2	KCKL-Takt	KDAT-Daten	KDAT-Daten
3	KDAT-Daten	+5V	unbelegt
4	GND	unbelegt	GND
5		GND	+5V
6		RESET*	Abschirmung
7		Power-LED	
8		Drive-LED	

Die Datenübertragung vom Tastaturprozessor zum Amiga läuft seriell.

Während die KCKL-Leitung (Keyboard-Clock) den Takt liefert, werden die einzelnen Datenbits synchron dazu auf den KDAT-Ausgang geschoben. Beide Leitungen liegen an den untersten Pins (PA0 und PA1) von Port A des Tastaturprozessors und sind intern sowohl mit dem Tastenpuffer (KDAT), als auch mit dem Taktzähler (KCLK) verbunden. Amigaseitig werden die Daten vom seriellen Port des CIA-Chips-A empfangen, wobei das Taktsignal am CNT-Eingang des Ports liegt. Im A500/B2000 läuft es über einen Nebenzweig auch noch zum KCLK-Anschluß des Gary-Chips ("Hard-wired-RESET").

Der synchrone Übertragungsmodus arbeitet nach gewohntem Schema. Mit jeder steigenden Flanke des KCKL-Signals werden die Datenbits der KDAT-Leitung in das serielle Schieberegister hineingetaktet (SP-Anschluß). Nach acht Taktphasen ist das Schieberegister voll, die Steuerlogik (des CIA-Chips) kopiert es in das SDR-Datenregister und setzt das SP-Bit im Interrupt-Port (ICR). Normalerweise folgt dann eine Unterbrechung der Ebene 2, in der die Systemsoftware das SDR-Register ausliest und den Tastencode zur weiteren Verarbeitung an das Keyboard-Device übergibt.

Im Anschluß an den Übertragungszyklus eines Datenbytes verlangt der Tastaturprozessor eine Bestätigung für die erfolgreiche Übernahme. Dazu tauschen Tastatur und Computer ihre Rollen. Die KDAT-Leitung wird kurzzeitig auf Eingang geschaltet und der Amiga legt den SP-Port für mindestens 75 Mikrosekunden auf LOW-Pegel. Während das Schieberegister ein Null-Byte ausgibt, kontrolliert Timer A die Länge des Handshakesignals. Ist der Tastaturprozessor zufrieden, geht er wieder in den normalen Übertragungsmodus und die nächste Sendesequenz kann beginnen.

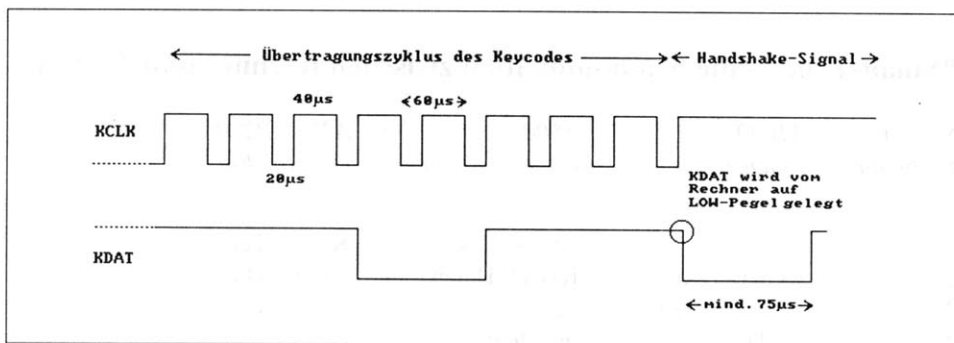


Abb. 7.4: Die Kommunikation zwischen Rechner und Tastaturprozessor

Abbildung 7.4 zeigt den Verlauf des Taktsignals und die Pegelniveaus der KDAT-Leitung bei Übertragung des Buchstabens "G" (gedrückt).

Das KCLK-Signal ist ein Rechteckimpuls mit einer LOW-Phase von 20 Mikrosekunden und einer HIGH-Phase von 40 Mikrosekunden. Die Übertragungszeit für ein Datenbit entspricht dem Abstand zweier steigender Flanken (60 Mikrosekunden). Bei 8 Bits pro Byte und einer mittleren Handshakedauer von 110 Mikrosekunden ($(145+75)/2$), erreicht die Tastaturschnittstelle eine Spitzenleistung von ca 13,56 Kbaud (Bits/s).

Wie man im Impulsdiagramm (7.4) gut erkennt, arbeitet die KDAT-Leitung LOW-aktiv und überträgt die logischen Zustände der Datenbits mit invertiertem Pegelniveau – Einer-Bits werden als LOW-Pegel, Null-Bits dagegen als HIGH-Pegel dargestellt. Das Codebyte im SDR-Register des CIA-Chips ist also immer das invertierte Abbild des ursprünglichen Tastencodes. Hinzukommt eine weitere Eigenart der Datenübertragung:

Bevor der Tastaturprozessor einen Tastencode losschickt, wird das gesamte Datenbyte um eine Stelle nach links rotiert. Aus der Bitfolge 7-6-5-4-3-2-1-0 entsteht die Sequenz 6-5-4-3-2-1-0-7, so daß das KeyUp-Flag immer zuletzt ausgegeben wird (Übertragung des MSBs an erster Stelle steigert die Fehlertoleranz bei Synchronisationsverlust).

Um die Schritte zur Rekonstruktion des Tastencodes nachzuvollziehen, kann man eine kleine Assembler-Routine in den Level-2-Interrupt einklinken:

```
_getkeycode
move.b $BFEC01,d0      ;SDR-Register von CIA-A auslesen und
                        ;in d0 speichern
ror.b #1,d0            ;1 Stelle nach rechts rotieren
not.b d0               ;und invertieren
jsr _print             ;hier könnte eine Unteroutine zur
                        ;Ausgabe des Tastencodes in d0
                        ;angesprungen werden
```

Für das Betriebssystem ist die Rückwandlung des Tastencodes nur ein kleiner Schritt in einer langen Kette verschiedener Arbeitsgänge. So durchläuft das Codebyte zahllose Ebenen – vom Keyboard-Device über das Input-Device gelangt es schließlich zum Console-Device und wird erst hier vollständig ausgewertet. Da muß man sich nicht wundern, daß viele Editoren träge wirken und nur langsam reagieren – nun, die Tastatur bietet ein weites Experimentierfeld für schnelle Treiber und trickreiche Lösungen.

Informationen bitte – die Zustandcodes

Das besondere Kennzeichen einer "intelligenten" Tastatur ist nicht nur die weitgehend selbstständige Auswertung der Tasteneingaben, sondern auch die Fähigkeit, Störungen des Normalbetriebs zu erkennen und den Rechner entsprechend zu informieren. Dafür sind einige Bytes oberhalb der verwendeten Tastencodes reserviert, in denen der Tastaturprozessor wichtige Informationen verschlüsselt und auf dem normalen Übertragungsweg an den Computer sendet. Diese Zustands- bzw. Sondercodes liefern beispielsweise das Ergebnis der Selbstdiagnose nach dem Hochfahren des Systems, warnen bei Verlust der Synchronisation oder markieren den Überlauf des Tastaturpuffers:

<i>Codebyte</i>	<i>Funktion</i>
\$F9	markiert den Synchronisationsverlust und warnt vor der Fehlerhaftigkeit des letzten Tastencodes
\$FA	Tastenpuffer voll
\$FC	Fehler beim Selbsttest bemerkt
\$FD	Startmarkierung für die Übertragung der beim Hochfahren gedrückten Tasten
\$FE	Endmarkierung dieser Tastensequenz

Beim Einschalten des Systems verzweigt der Tastaturprozessor in eine Selbsttest-Routine (kurzzeitiges Aufblinken der Caps-Lock-LED). Dabei kontrolliert er seinen RAM- und ROM-Speicher, testet die Tastaturmatrix und prüft den Watchdog-Timer. Verläuft alles glatt, springt er in den normalen Betriebsmodus und überträgt danach sofort die Keycodes, der in der Startphase gedrückten Tasten (eingerahmt durch die \$FD- und \$FE-Markierung). Sollte während der Bootphase ein Fehler aufgetreten sein, wird der Rechner durch das Codebyte \$FC informiert. Kurze Blinkzeichen der Caps-Lock-LED geben eine grobe Fehlerbeschreibung (Wartungshilfe!), bevor sich der Tastaturprozessor in einer Endlosschleife verabschiedet:

Blinkzeichen Diagnose

1x	Fehler im On-Chip-ROM
2x	Fehler im On-Chip-RAM
3x	Watchdog-Timer defekt
4x	Tastaturmatrix defekt

Für die korrekte Übermittlung der Tastencodes sind Rechner und Tastaturchip auf eine exakte Synchronisation angewiesen. Kurzzeitiges Abreißen der Kabelverbindung zum Computer, das Sperren des Level-2-Interrupts und der Bootvorgang nach einem RESET sind Phasen, in denen die Synchronisation verloren geht bzw. aufgebaut werden muß.

Um jetzt möglichst schnell eine stabile Verbindung herzustellen, geht der Tastaturprozessor in den RESYNC-Modus. Dazu gibt die KDAT-Leitung nur noch Einsen aus (LOW-Impulse), während der Tastaturprozessor (maximal 145 Mikrosekunden) auf das Handshakesignal wartet. Sobald der Rechner wieder an der "Strippe" hängt und sich ordnungsgemäß gemeldet hat, signalisiert der Tastaturprozessor mit dem Sondercode \$F9, daß der zuletzt gesendete Tastencode verunglückt ist und deshalb nochmal übertragen wird. Ist dies geschehen, kann der Normalbetrieb wieder aufgenommen werden – die Verbindung steht, als wäre nichts geschehen.

Vor Mithörern wird gewarnt – RESET über die Tastatur

Beim ersten Kontakt mit dem Amiga (und ohne Vorwarnung) hat schon so mancher verzeifelt nach einem RESET-Schalter gesucht, um dann verblüfft festzustellen, daß das RESET-Signal über eine spezielle Tastenkombination ausgelöst wird: <Ctrl> + <Amiga> + <Amiga>

Drückt man alle drei Tasten gleichzeitig, legt der Tastaturprozessor die KCLK-Leitung für 500 ms auf LOW-Pegel. Bekanntlich ist diese Leitung mit dem CNT-Eingang von

CIA-A verbunden, doch Vorsicht – beim A500 und B2000 gibt es einen heimlichen "Lauscher", der die KCLK-Leitung anzapft und überwacht. Der Customchip Gary hat einen speziellen Eingang mit der Bezeichnung KBRESET* (Keyboard RESET), an dem das KCLK-Signal liegt. Sobald der Takt auf LOW-Pegel geht, läuft ein interner Zähler an. Wechselt die KCLK-Leitung vor Ablauf der 500-Millisekunden-Frist wieder auf HIGH-Pegel, wird der Zähler gelöscht, ansonsten löst Gary einen Prozessor-RESET aus. Nach dem Loslassen einer Taste geht auch die Tastatur in eine spezielle RESET-Prozedur.

Die Amiga-Modelle 1000 und 2000 verfügen über keine festverdrahtete RESET-Logik, sondern eine Software-Lösung. Deshalb liest man in den Datenblättern häufig die Bezeichnung Soft-RESET, im Gegensatz zu dem Hard-wired-RESET der neuen Amiga-Generation.

Das Floppy-Disk-Interface

Die Floppy-Schnittstelle des Amigas entspricht den Spezifikationen des Shugart-Busses und unterstützt bis zu vier 3,5- bzw. 5,25-Zoll-Laufwerke. Die meisten Standard-Laufwerke (z.B. NEC FD1035, FD1036A) sind zu diesem Standard kompatibel und können deshalb problemlos angeschlossen werden. Interne und externe Floppy-Schnittstelle unterscheiden sich bei den diversen Amiga-Modellen nur geringfügig in der Verteilung der SELECT-Leitungen zur Laufwerks-Aktivierung. A1000 und A500 können intern nur ein Laufwerk aufnehmen und führen die restlichen drei Auswahlleitungen zum externen Floppy-Anschluß. Das Gehäuse der 2000er-Systeme verkraftet zwei Amiga-Floppies, so daß auf dem externen Bus lediglich die beiden höchsten SELECT-Leitungen liegen.

Die nun folgende Tabelle zeigt Pinbelegung und Beschaltung der externen Floppy-Schnittstelle, die als 23-polige D-SUB-Buchse ausgeführt ist:

<i>Pin Nr.</i>	<i>Leitung (Label)</i>	<i>intern verbunden mit</i>	<i>Signalrichtung Eingang/Ausgang</i>	<i>ausführliches Label</i>
1	RDY*	PA5 von CIA-A	E	"Ready"
2	DKRD*	DKRD* von Paula	E	"Disk-Read-Data"
3..7	GND	GND	—	—
8	MTRXD*	PB7 von CIA-B	A	"Motor-on-x-Data"
9	SEL2* (A500)	PB5 von CIA-B	A	"SELECT-2"
	/SEL3* (B2000)	PB6 " "	A	"SELECT-3"

<i>Pin Nr.</i>	<i>Leitung (Label)</i>	<i>intern verbunden mit</i>	<i>Signalrichtung Eingang/Ausgang</i>	<i>ausführliches Label</i>
10	DRES*	System-RESET	A	"Disk-RESET"
11	CHNG	PA2 von CIA-A	E	"Change"
12	+5V	+5V	—	—
13	SIDE*	PB2 von CIA-B	A	"Side"
14	WPRO*	PA3 von CIA-A	E	"Write-Protect"
15	TK0*	PA4 von CIA-A	E	"Track-0"
16	DKWE*	DKWE von Paula	A	"Disk-Write-enable"
17	DKWD*	DKWD von Paula	A	"Disk-Write-Data"
18	STEP*	PB0 von CIA-B	A	"Step"
19	DIR	PB1 von CIA-B	A	"Direction"
20	SEL3* (A500)	PB6 von CIA-B	A	"SELECT-3"
	/N.C. (B2000)	—	—	—
21	SEL1* (A500)	PB4 von CIA-B	A	"SELECT-1"
	/SEL2* (B2000)	PB5 von CIA-B	A	"SELECT-2"
22	INDEX*	FLAG-Eingang, CIA-B	E	"Index-Signal"
23	+12V	+12V	—	—

Bei der internen Floppy-Schnittstelle handelt es sich um eine 34-polige-Steckleiste. Zur Unterdrückung von Störimpulsen liegen sämtliche ungeraden Pins auf Masse. Interne Beschaltung und Signpalette sind – bis auf die zusätzlichen INUSEx-Leitungen – mit dem externen Interface identisch. Die Signalverteilung sieht wie folgt aus (ungerade Masse-Pins werden aus Platzgründen nicht aufgeführt):

<i>Pin</i>	<i>Leitung</i>	<i>Besonderheit</i>	<i>Signalrichtung</i>
2	CHNG*	—	E
4	INUSE1*	verbunden mit MTR/SEL1*	A
6	INUSE0*	verbunden mit MTRON	A
8	INDEX*	—	E
10	SEL0*	—	A
12	SEL1*	—	A
14	unbelegt	verbunden mit INUSE1*	—
16	MTRON*	—	A
18	DIR	—	A
20	STEP*	—	A
22	DKWD*	—	A
24	DKWE*	—	A
26	TK0*	—	E

<i>Pin</i>	<i>Leitung</i>	<i>Besonderheit</i>	<i>Signalrichtung</i>
28	WPRO*	—	E
30	DKRD*	—	E
32	SIDE*	—	A
34	RDY*	—	E

Die Signale der Floppy-Schnittstelle

In Anlehnung an den Shugart-Bus arbeitet das Floppy-Disk-Interface des Amigas ausschließlich mit LOW-aktiven Steuerleitungen.

Sämtliche Ausgänge laufen über Open-Collector-Treiber (im B2000 vom Typ 74LS38) und sind laufwerkseitig mit 1KOhm-Pullup-Widerständen belastet. Um bei den verwendeten Treiberbausteinen eine einwandfreie Signalübertragung zu gewährleisten, sollte die Kabelverbindung (zum externen Laufwerk) nicht länger als 1 m sein.

Die Stromversorgung der 23-poligen-D-SUB-Buchse ist für den Betrieb eines externen Laufwerks ausgelegt, weitere Floppies müssen über ein eigenes Netzteil verfügen.

Neben den Anschlüssen für Masse und Stromversorgung besitzt die Schnittstelle neun Steuer-, zwei Daten- und vier Informationsleitungen, die sich – ihren Funktionen entsprechend – in vier Gruppen einteilen lassen:

Die Signale des Floppy-Controllers: DKWD*, DKRD*, DKWE*

Die Datenübertragung von und zum Diskettenlaufwerk läuft seriell. DKWD* ist die Schreibleitung, DKRD* die Leseleitung des Floppy-Controllers (Paula). Wie üblich beim Shugart-Bus sind auch diese Leitungen LOW-aktiv und liefern invertierte Datenpegel.

Mit DKWE* schaltet der Controller zwischen Lese- (HIGH) und Schreibbetrieb (LOW) um.

Selektion, Motorsteuerung und Initialisierung: SELX*, MTRX*, DRES*, RDY*

Die SELECT-Signale (SEL0* bis SEL3*) dienen zur gezielten Auswahl eines Floppy-Drives und bauen die Verbindung zwischen Interface und Laufwerkselektronik auf. So-

bald die selektierte Floppy einen LOW-Impuls auf "ihrer" SELECT-Leitung erkennt, ist sie für Steuerimpulse des Gastrechners bereit.

Bevor die Datenübertragung beginnen kann, muß der Floppy-Motor aktiviert werden und seine Solldrehzahl von ca. 300 Umdrehungen/min erreicht haben. Gesteuert wird das über die MTRX-Leitung ("Motor on"). Um Leitungen einzusparen, ist das Signal nicht für jedes Laufwerk einzeln herausgeführt, sondern liegt nur einmal am Floppy-Interface. Zur gezielten Ansteuerung eines bestimmten Laufwerkmotors, muß MTRX* mit der zugehörigen SELECT-Leitung verknüpft werden. Dazu bietet sich ein Flip-Flop an, da es die Eingangspegel in einem stabilen Zustand am Ausgang speichert.

Dieser Baustein wird so geschaltet, daß SELX* und MTRX* am Eingang liegen und die Motorsteuerung mit dem Ausgang verbunden ist. Sobald beide Signale gleichzeitig LOW sind, läuft der Motor der selektierten Floppy an. Diese Technik benötigt zwar für jedes angeschlossene Laufwerk ein zusätzliches Flip-Flop, spart dafür aber drei Leitungen ein. Bei den Modellen A500 und A1000 befindet sich das Motor-Flip-Flop für die interne Floppy bereits auf der Grundplatine, die 2000er-Serie besitzt ein weiteres für das zweite interne Laufwerk. In den neueren Amiga-Modellen ist die Flip-Flop-Schaltung auf dem Customchip Gary integriert.

Das DRES-Signal ist mit der internen RESET-Leitung des Systems verbunden und setzt die Motor-Flip-Flops zurück.

Die RDY-Leitung signalisiert das Erreichen der Solldrehzahl des Floppy-Motors und markiert damit die Bereitschaft zur Datenübertragung.

Spurtrue – die Positionierung des Kopfes: SIDE*, STEP*, DIR*, TK0*

SIDE* aktiviert den Schreib-/Lesekopf auf der Unter- (Seite 0, HIGH-Pegel) bzw. Oberseite (Seite 1, LOW-Pegel). Das STEP-Signal liefert die Bewegungsimpulse für den Schrittmotor; mit jeder steigenden Flanke (Wechsel von LOW nach HIGH) des STEP-Taktes führt der Schrittmotor den Schreib-/Lesekopf eine Spur weiter.

Die Richtung dieser Bewegung wird durch die DIR-Leitung festgelegt. Ist sie LOW, wandert der Kopf zu den höheren Spuren in Richtung Diskettenmitte, bei HIGH-Pegel geht die Bewegung nach außen zu den niedrigeren Spuren.

Wird Spur 0 (am äußersten Rand) erreicht, springt das TK0-Signal ("Track 0") auf LOW.

Die STEP-Leitung sollte nach Ausgabe eines Schrittimulses auf HIGH-Pegel bleiben, da sonst die Diskettenwechsel-Logik ins "Schleudern" gerät.

Die Zustandsinformationen: WPRO*, CHNG*, INUSE*, INDEX*

Eine Lichtschranke im Laufwerk überprüft das Schreibschutzfenster (offen, verdeckt) und sendet den Zustand über die WPRO-Leitung zum Rechner. Bei HIGH-Pegel handelt es sich um eine ungeschützte Diskette, LOW-Pegel signalisiert "Schreibschutz".

Die CHNG-Leitung informiert den Computer über einen Diskettenwechsel. Da eine interaktive Benutzeroberfläche (wie die Workbench) immer auf dem aktuellsten Stand sein sollte, ist diese Information nicht unwichtig. Nach dem Herausnehmen einer Diskette, springt CHNG* sofort auf LOW-Pegel. Das Track-Disk-Device versucht nun durch regelmäßige STEP-Impulse eine neu eingelegte Diskette zu erkennen. Bleibt das Laufwerk jedoch leer, behält die CHNG-Leitung ihr Pegelniveau und der Schrittmotor muß weiter "kratzen". Erst beim Einlegen einer Diskette haben die Versuche Erfolg, CHNG* wechselt auf HIGH-Pegel und das Track-Disk-Device beginnt damit, den "Neuankömmling" zu identifizieren.

INUSE* liegt lediglich an der internen Floppy-Leiste und dient zur Steuerung der Laufwerk-LED (aktiv/inaktiv). Eigentlich ist diese Leitung überflüssig, da sie mit der Motor-An/Aus-Schaltung verbunden ist und somit keine eigenständige Steuerung darstellt.

Das INDEX-Signal geht mit jeder vollständigen Diskettenumdrehung kurz auf LOW-Pegel. Es markiert Anfang und Ende einer Spur und ist intern mit dem FLAG-Eingang von Parallel-Port-B (CIA-B) verbunden. Bei entsprechender Programmierung der Interrupt-Logik, kann der INDEX-Impuls einen Level-6-Interrupt auslösen.

Gesichtskontrolle am Floppy-Port – der Identifikationsmodus

Da das Disk-Operating-System wissen muß, mit welchem Laufwerkstyp es zu tun hat, gibt es einen speziellen Abfragemodus, bei dem sich das angeschlossene Laufwerk selbst zu erkennen gibt.

Eingeleitet wird das Ganze durch kurzzeitige Aktivierung der MTRX-Leitung, wobei das serielle Schieberegister der Floppy-Disk zurückgesetzt und mit einem 32-Bit-Identifikationscode initialisiert wird.

Zur Auswertung legt man das zugehörige SELECT-Signal auf LOW-Pegel und liest das erste Datenbit über die RDY-Leitung. Den Ausgabetakt liefern negative Flanken (von HIGH nach LOW) der SELECT-Leitung, deshalb muß das Signal nach jedem Datenbit wieder auf HIGH gesetzt werden, bevor das nächste Bit mit der LOW-Flanke geholt werden kann.

Hat man diese Prozedur 32 mal wiederholt, ist das Identifikationsmuster komplett und muß lediglich invertiert werden:

\$0000 0000 kein Laufwerk vorhanden
 \$FFFF FFFF 3,5-Zoll-Standard-Floppy
 \$5555 5555 5,25-Zoll-Laufwerk (80 Spuren, Amiga-kompatibel)

Floppy-Funktionen von CIA-A:

Adresse	Offset	Label	7	6	5	4	3	2	1	0
SBFE001	0	PRA	FIRE0	FIRE1	RDY	TK0	MPRO	CHNG	LED	OVL
SBFE201	2	DDRA	8	8	8	8	8	8	1	1

Parallel Port A

Floppy-Funktionen von CIA-B:

Adresse	Offset	Label	7	6	5	4	3	2	1	0
SBFD100	1	PRB	MTR	SEL3	SEL2	SEL1	SEL0	SIDE	DIR	STEP
SBFD300	3	DDRB	1	1	1	1	1	1	1	1

Parallel Port B

Abb. 7.5: Die CIA-Register zur Steuerung der Floppy-Schnittstelle

Die Aufzeichnungsformate

Das Prinzip der physikalischen Aufzeichnung beruht auf den Gesetzen des Elektromagnetismus.

Das Magnetfeld des Schreib-/Lesekopfes richtet die Elementarmagneten der Diskettenoberfläche in eine bestimmte Richtung aus. Null-Bits werden als konstante Flußrichtung, Einser-Bits dagegen als Richtungsänderung des Magnetfeldes aufgezeichnet. Beim Rekonstruktionsprozeß induzieren die Magnetflußänderungen in der Spule des Lesekopfes einen Strom, der die gesetzten Bits deutlich hervorhebt.

Im Prinzip liefert diese Tatsache das zentrale Entscheidungskriterium für die Aufzeichnung von Binärcode.

Trotzdem ist es damit noch nicht getan, denn für die sichere Unterscheidung längerer Nullfolgen müßte die einzelnen Bitzellen exakt gleich lang sein. Das ist aber wiederum nur möglich, wenn die Diskette mit absolut konstanter Geschwindigkeit rotiert – in der Praxis ein Ding der Unmöglichkeit. Gleichlaufschwankungen des Antriebsmotors (und damit hat schließlich jeder Motor zu kämpfen) bringen das rein physikalische Aufzeichnungsverfahren sofort aus der Synchronisation.

Man benötigt einen Referenztakt, der der Lese-Elektronik jedesmal mitteilt, wo eine Bitzelle beginnt und wo sie endet. Auf diese Weise lassen sich auch ganze Ansammlungen von Nullen oder Einsen dekodieren. Aus ökonomischen Gründen wird dieser Referenztakt nicht extern eingespeist, sondern als eine Art Wegmarkierung in den seriellen Datenstrom gemischt.

Eine sehr einfache Lösung stellt das FM-Verfahren dar (Frequency-Modulation). Jedes Datenbit wird mit einem Taktbit versehen und gemeinsam aufgezeichnet, Einsen bestehen dann aus zwei aufeinanderfolgenden Flußwechseln, ein Null-Bit wird von einem Taktflußwechsel eingeleitet. Der Nachteil dieses Verfahrens liegt in der vergleichsweise geringen Speicherkapazität, da jeder zusätzliche Flußwechsel den Raum für die reine Informationsspeicherung verringert.

MFM

Hier setzt das MFM-Verfahren (Modified Frequency Modulation) an und erzeugt nur dann zusätzliche Flußwechsel, wenn sie zur Dekodierung unbedingt erforderlich sind. Jedes Datenbit stellt eine Zelle mit konstanter Länge dar. Einsen entsprechen einem Flußwechsel in der Mitte ihrer Zelle, Nullfolgen werden durch Taktbits am Zellenanfang aufgelockert. Der wesentliche Vorteil gegenüber der einfachen FM-Aufzeichnung liegt aber in der Darstellung von 0-1- bzw. 1-0-Übergängen. MFM zeichnet hier keine unnötigen Taktbits auf, sondern nutzt die Flußwechsel der Übergänge als Synchronisationshilfe. Die erheblich verringerte Flußwechseldichte führt zu einer besseren Raumnutzung und höheren Speicherkapazitäten.

GCR

Einen ganz anderen Weg beschreitet das GCR-Verfahren (Group Coded Recording = gruppenweise codierte Aufzeichnung) der Firma Apple, indem es jeweils 4 Bit breite Daten-Nibble zu einem 5-Bit-Code zusammenfaßt. (Um auf Byte-Grenzen zu bleiben, sollte der ursprüngliche Datenblock glatt durch 4 teilbar sein.)

Die Codewerte sind so gewählt, daß nicht mehr als zwei Null-Bits aber maximal acht Einser-Bits aufeinanderfolgen. Die hohe Flußwechseldichte erlaubt keine so hohen Übertragungsraten wie bei MFM; üblicherweise arbeitet GCR mit 4 Mikrosekunden pro Bit, während MFM doppelt so schnell ist (2 Mikrosekunden/Bit).

Die GCR-Codierung:

4-Bit-Nibble:

5-Bit-Codewert:

0000	01010
0001	01011
0010	10010
0011	10011
0100	01110
0101	01111
0110	10110
0111	10111
1000	01001
1001	11001
1010	11010
1011	11011
1100	01101
1101	11101
1110	11110
1111	10101

Die Programmierung des Floppy-Controllers

Registerübersicht:

<i>Register</i>	<i>Adresse</i>	<i>Zugriff</i>	<i>Funktion</i>
DSKPTH	\$DFF020	w	Disk-DMA-Adreßregister (HIGH, Bits 18 bis 16)
DSKPTL	\$DFF022	w	Disk-DMA-Adreßregister (LOW, Bits 15 bis 0) zeigen auf einen Speicherbereich im CHIP-RAM (19-Bit-Adresse)
DSKDAT	\$DFF026	w	DMA-Schreibregister als Puffer zwischen Datenblock im Speicher und Schieberegister

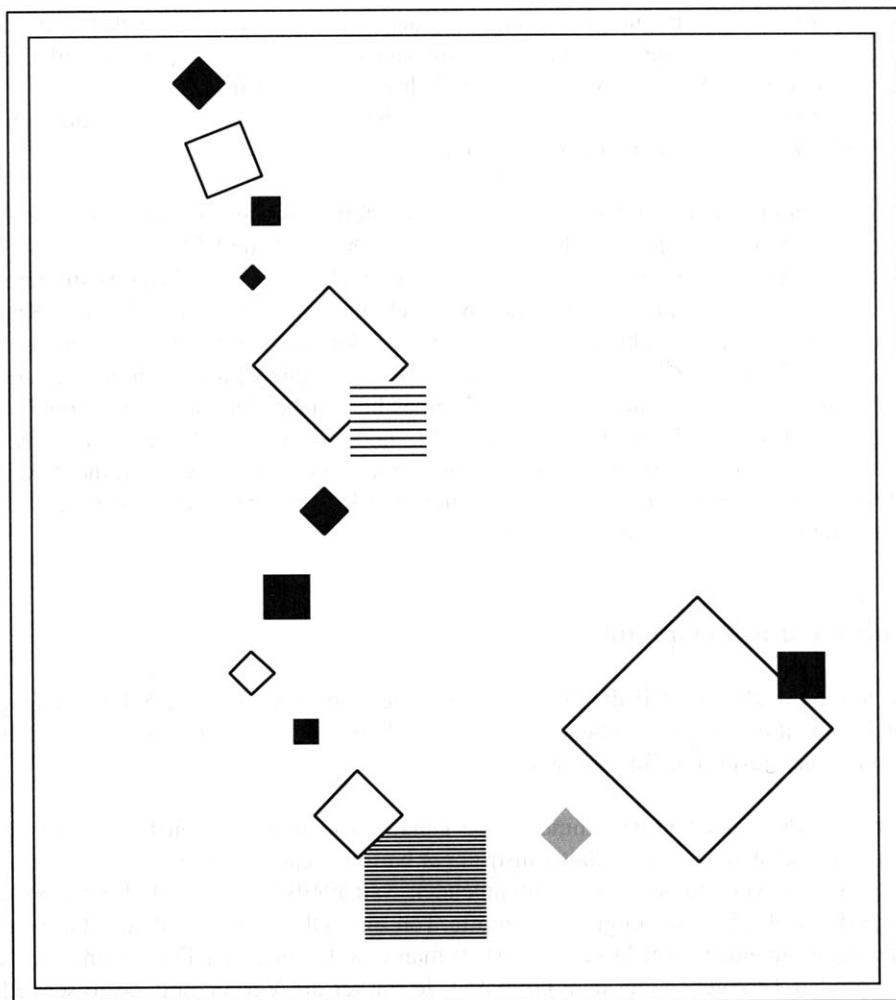
Register	Adresse	Zugriff	Funktion												
			des Floppy-Laufwerks (wird normalerweise vom DMA-Controller beschrieben)												
DSKDATR	\$DFF008	r	DMA-Leseregister für die Lesedaten des Floppy-Controllers. Da es sich um ein Early-Read-Register handelt (wie BLTDAT, siehe Registertabelle), kann es nur vom DMA-Controller gelesen werden.												
DSKLEN	\$DFF024	w	Speicherblock-Länge, Zugriffsrichtung, DMA-Aktivierung												
			Bitbelegung:												
			<table><tr><th>Nr.</th><th>Label</th><th>Funktion</th></tr><tr><td>15</td><td>DMAEN</td><td>Disk-DMA einschalten (= 1).</td></tr><tr><td>14</td><td>WRITE</td><td>1 = schreiben / 0 = lesen</td></tr><tr><td>13 – 0</td><td>LENGTH</td><td>Wortlänge des Datenpuffers im Speicher ($2^{13} = 16 \text{ KByte}$)</td></tr></table>	Nr.	Label	Funktion	15	DMAEN	Disk-DMA einschalten (= 1).	14	WRITE	1 = schreiben / 0 = lesen	13 – 0	LENGTH	Wortlänge des Datenpuffers im Speicher ($2^{13} = 16 \text{ KByte}$)
Nr.	Label	Funktion													
15	DMAEN	Disk-DMA einschalten (= 1).													
14	WRITE	1 = schreiben / 0 = lesen													
13 – 0	LENGTH	Wortlänge des Datenpuffers im Speicher ($2^{13} = 16 \text{ KByte}$)													
			Zur Verhinderung eines unbeabsichtigten Schreibzugriffs, muß das DMAEN-Bit zweimal hintereinander gesetzt werden. Das erreicht man am einfachsten durch doppelte Initialisierung des DSKLEN-Registers.												
DSKBYTR	\$DFF01A	r	Status- und Datenregister des Floppy-Controllers												
			Bitbelegung:												
			<table><tr><th>Nr.</th><th>Label</th><th>Funktion</th></tr><tr><td>15</td><td>BYTEREADY</td><td>signalisiert ein gültiges Lesebyte von der Diskette (bleibt nur bis zum nächsten DMA-</td></tr></table>	Nr.	Label	Funktion	15	BYTEREADY	signalisiert ein gültiges Lesebyte von der Diskette (bleibt nur bis zum nächsten DMA-						
Nr.	Label	Funktion													
15	BYTEREADY	signalisiert ein gültiges Lesebyte von der Diskette (bleibt nur bis zum nächsten DMA-													

Register	Adresse	Zugriff	Funktion
			Zugriff gültig, 2 oder 4 Mikrosek.)
		14	DMAON
			zeigt aktuellen DMA-Status (1 = an / 0 = aus)
		13	DSKWRITE
			reflektiert WRITE-Bit im DSKLEN-Register
		12	WORDEQUAL
			Sync-Wort gefunden
		11 – 8 –	–
		7 – 0	DATA
			niederwertiges Byte der Disk-Lesedaten
DSKSYNC	\$DFF07E	w	Synchronisationswert für den Floppy-Controller. Die eigentliche Datenübertragung startet erst dann, wenn die DMA-Daten von der Diskette dem Synchronisationswort im DSKSYNC-Register entsprechen. Das Setzen diverser Sync-Marken erlaubt den gezielten Zugriff auf Teile einer Spur.
ADKCON	\$DFF09E (\$DFF010	w r)	Audio-Disk-Kontrollregister wählt zwischen den beiden Aufzeichnungsformaten (MFM oder GCR), steuert die Präkompensation und schaltet die Sync-Marke ein.
Bitbelegung (nur für Floppy-Steuerung):			
Nr.	Label	Funktion	
15	SET/CLR	(1 = setzen / 0 = löschen)	
14 – 13	PRECOMP	Präkompensationszeit:	
		00 = Null	
		01 = 140 Nanosek.	
		10 = 280 Nanosek.	
		11 = 560 Nanosek.	

<i>Register</i>	<i>Adresse</i>	<i>Zugriff</i>	<i>Funktion</i>
		12	MFMPREC Aufzeichnungs- format: 1 = MFM 0 = GCR
		10	WORDSYNC Synchronisations- Erkennung aktivieren (= 1)
		9	MSBSYNC GCR-Sync-Marke (\$FF) ein (= 1)
		8	FAST Zugriffsgeschwin- digkeit des Controllers: 0 = GCR-Format mit 4 Mikrosek. pro Bit 1 = MFM-Format mit 2 Mikrosek. pro Bit

Kapitel 8

Das Audio - System



Einführung in die Akustik

Schallwellen sind Druckschwankungen, die sich in einem homogenen (hier: nur aus einem Stoff bestehenden) Medium mit konstanter Geschwindigkeit fortpflanzen. Da diese Ausbreitung parallel zur Amplitudenauslenkung läuft, spricht man von einer Longitudinalschwingung. (Im Gegensatz zur Transversalwelle, bei der Auslenk- bzw. Ausbreitungsrichtung zueinander senkrecht stehen.)

Die Ausbreitungsgeschwindigkeit hängt von der Elastizität und Dichte des Fortpflanzungsmediums ab – Faktoren, die wiederum von Größen wie Temperatur und Druck beeinflusst werden. So leitet Wasser wesentlich besser als Luft, kalte (und damit auch dichtere) Luft besser als warme. Der Weltraum ist schalltot, da es im Vakuum keinen Stoff gibt, der den Druckimpuls übertragen könnte.

Schon in der ersten Hälfte des 17. Jahrhunderts lieferte das französische Allroundtalent Marin Mersennes – seines Zeichens Geistlicher, Philosoph und Mathematiker – eine brauchbare Abschätzung für die Schallgeschwindigkeit in der Luft. Dazu bestimmte er die zeitliche Verzögerung zwischen der Wahrnehmung von Mündungsblitz und Knallgeräusch eines Kanonenschusses und setzte das Meßergebnis mit seiner Entfernung zur schießenden Kanone in Beziehung. Mersennes Meßverfahren findet sich noch heute in der Faustregel zur Bestimmungsbestimmung eines herannahenden Gewitters, wobei man einfach die gezählten Sekunden zwischen Blitz und Donner als Abstandsmaß in Kilometern nimmt. Das Ergebnis untertreibt etwas; tatsächlich ist der Gewitterherd weiter entfernt, denn der Schall bewegt sich schneller als 1 Kilometer pro Sekunde ($= 278 \text{ m/s}$) und erreicht bei Raumtemperatur 344 m/s bzw. $1,238 \text{ km/s}$.

Tonhöhe und Frequenz

Fast intuitiv verbinden wir die Höhenempfindung eines Tons mit der Schwingungsgeschwindigkeit eines periodischen Vorgangs, wie beispielsweise der gleichförmigen Bewegung einer gezupften Gitarrensaite.

Ohne die physikalischen Zusammenhänge genau zu kennen, wußten bereits die alten Griechen, daß sich auf einem Saiteninstrument harmonische Tonintervalle aus sehr einfachen Längenverhältnissen des schwingenden Saitenteils ergeben. Halbiert man die Länge der Vollsaiten, so klingt der gezupfte Ton eine Oktave höher als der Grundton. Schwingen nur noch zwei Drittel, so erhält man eine Quinte; vier Fünftel entsprechen einer großen Terz gegenüber dem Grundton. Je stärker die Verkürzung, desto schneller die Schwingung und desto höher der erzeugte Ton.

Was die Griechen nicht wußten, ist der ebenso einfache Zusammenhang zwischen Frequenz und Saitenlänge – sie sind umgekehrt proportional, die eine Größe also immer der Kehrwert der anderen. Bei Halbierung der Vollsaiten verdoppelt sich die Frequenz und der korrespondierende Ton liegt eine Oktave höher, der Quintabstand entspricht einer Frequenzzunahme von 50%.

Die Beziehungen zwischen Schwingungsdauer, Frequenz und Wellenlänge werden sehr schön am Modell einer Sinusfunktion deutlich, dem Standardbeispiel für periodische Schwingungsverläufe.

Unter der Schwingungsdauer (T) versteht man den Zeitabstand zweier Punkte mit gleicher Amplitudenhöhe und Auslenkrichtung. Das können zwei Nulldurchgänge mit positivem Vorzeichen, zwei Wellenberge oder zwei Täler sein....einzige Bedingung, zwischen beiden Punkten liegt ein kompletter Wellenzug. Der räumliche Abstand dieser Punkte wird durch die Wellenlänge (w) beschrieben. Die Frequenz (f) schließlich gibt die Anzahl der Schwingungsperioden pro Zeiteinheit wider (die Maßeinheit Hertz [Hz] bezieht sich auf eine Sekunde) und entspricht dem Kehrwert der Schwingungsdauer:

$f = 1 / T$ bzw. $f = w / v$ mit v = Schallgeschwindigkeit.

Die Periodizität der Sinuswelle liegt in der ständigen Wiederkehr eines Wellenzuges. Während der Periodendauer T durchläuft jeder beliebige Punkt sämtliche Auslenkphasen des Schwingungssystems; das $s(t)$ -Diagramm, also die Wellendarstellung als Funktion der Zeit, zeigt eine Kette von gleichförmigen Amplitudenverläufen. Dieses zyklische Verhalten ist letztendlich für die Höhenempfindung eines Tons verantwortlich – aperiodische Schallwellen hinterlassen keinen Toneindruck, sondern eine diffuse Geräuschwahrnehmung ohne klare Höhen- bzw. Tiefenempfindung.

In der Musik spielt der Kammerton A eine besondere Rolle. Mit einer festgelegten Frequenz von 440 Hz bildet er das Fundament unseres Tonleitersystems. Da die Frequenzverhältnisse sämtlicher Intervalle bekannt sind, kann man alle andere Töne rekonstruieren. Das eingestrichene A¹ beispielsweise liegt eine Oktave höher und hat eine Frequenz von 880 Hz (doppelte Frequenz des Grundtons).

Schalleistung und Lautstärkeempfindung

Die Leistung einer Schallquelle stellt einen absoluten Wert dar, der sich in Watt ausdrücken läßt. Für den individuellen Lautstärkeindruck ist aber weniger die Gesamtleistung ausschlaggebend (zumal ein erheblicher Teil ohnehin im Raum durch Absorption verloren geht), als vielmehr der Bruchteil, der unser Ohr direkt trifft und eine Laut/Leise-Empfindung auslöst.

Die Maßeinheit Dezibel (dB) beschreibt daher den Schalldruck (Intensität einer Schallquelle) pro Quadratzentimeter, bezogen auf einen Referenzpegel (P_r) von einem billionstel Watt/Quadratzentimeter ($= 10^{-12}$). Dieser Wert entspricht in etwa der menschlichen Hörschwelle, also der Schallintensität, bei der ein Ton, Klang oder Geräusch gerade noch wahrgenommen wird. Selbst angenehme Lautstärkeempfindungen schießen schnell ins Zehntausendfache dieses Referenzpegels. Ein logarithmisches Maß sorgt deshalb für handliche Zahlen:

Schallpegel in dB = $10 \times \log(P/P_r)$ mit

P = absolut gemessener Schalldruck

P_r = Referenzpegel 10^{-12} Watt/cm²

Löst man den Term nach dem Quotienten P/P_r auf, läßt sich der absolute Druckanstieg für jede beliebige Pegelerhöhung nachvollziehen. Bei einem Anstieg von 60 auf 80 dB beispielsweise, schnellst die Intensität um das 100fache in die Höhe:

$$P/P_r = 10^{(x \text{ dB}/10)}$$

$$60 \text{ dB} = 1\,000\,000 \times P_r, \quad 80 \text{ dB} = 100\,000\,000 \times P_r$$

Die Frequenzbandbreite des menschlichen Gehörs ist naturgemäß begrenzt. Gut trainiert und noch nicht durch übermäßigen "Acid"-Konsum geschädigt, reicht der wahrnehmbare Bereich ungefähr von 16 bis 16.000 Hz. Die Lautstärkeempfindlichkeit ist aber nicht über die gesamte Bandbreite konstant, sondern nimmt zu den Rändern hin extrem (tatsächlich fast exponentiell) ab.

Am besten hören wir Frequenzen zwischen 1000 und 3000 Hz. Interessanterweise liegt die menschliche Sprache genau in diesem Frequenzbereich. Die Frequenzabhängigkeit der Lautstärkeempfindung spiegelt sich in einer Reihe von Bewertungskurven, in denen die absoluten Schalleistungen diverser Frequenzkomponenten unterschiedlich gewichtet werden.

Kurve A kommt der Wahrnehmung des menschlichen Ohrs am nächsten, der bewertete Schallpegel als Funktion der Frequenz erhält den Zusatz A [dBA].

Ein ähnliches Bewertungsschema stellt die Maßeinheit Phon dar. Ausgehend von einer Sinusschwingung mit 1000 Hz (deren Maßzahl in Phon dem Schalldruckpegel in dB entspricht), tastet man die gesamte Bandbreite des menschlichen Gehörs ab. Parallel dazu wird der Schalldruckpegel jeweils so weit erhöht, daß Referenzschwingung und eingestellte Frequenz gleich laut klingen. Das Ergebnis ist eine Isophone (Kurve gleich lauter Frequenzen), deren Ränder erwartungsgemäß stark gekrümmt sind.

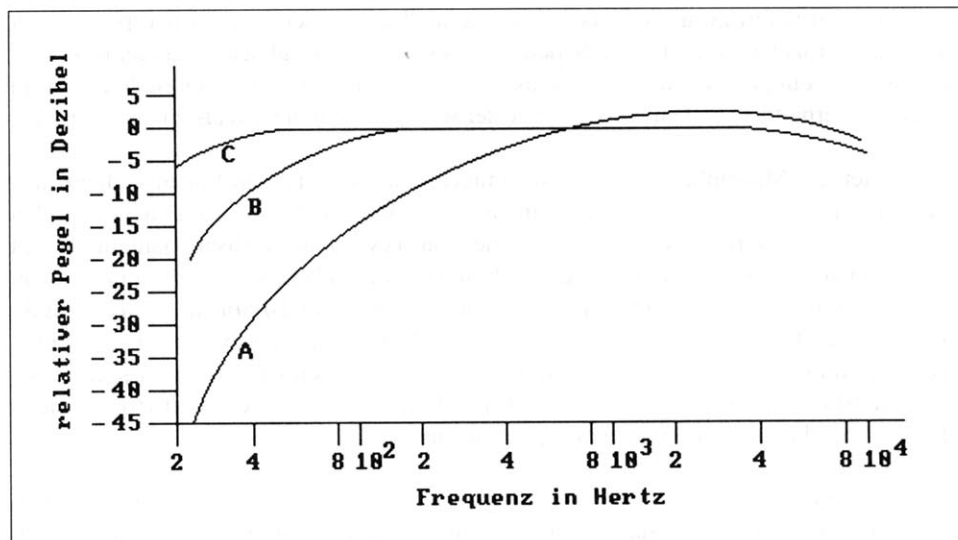


Abb. 8.1: Bewertungskurven des Schallpegels

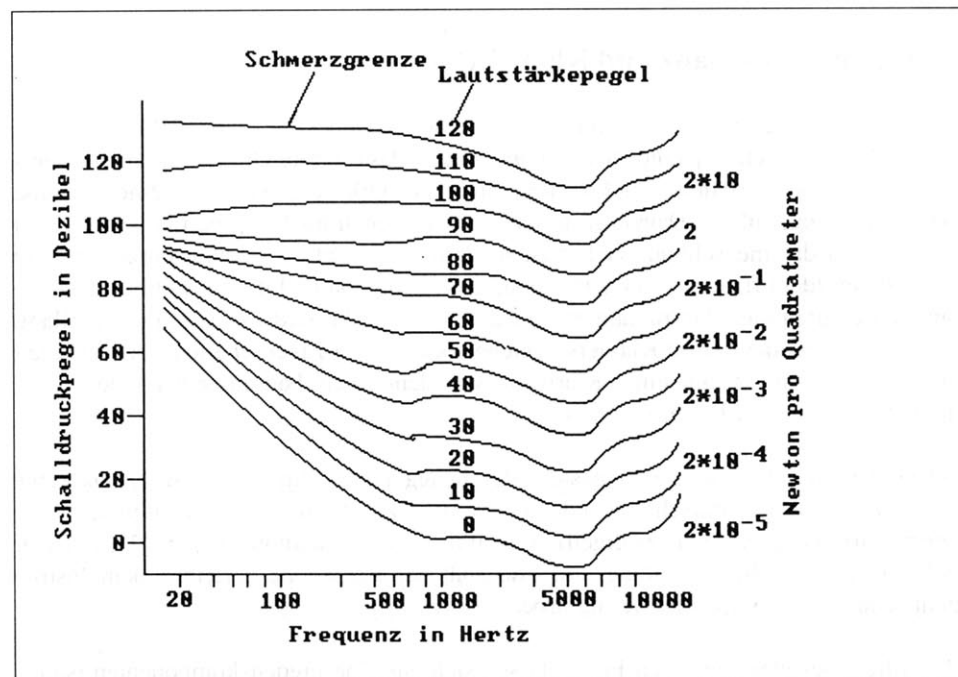


Abb. 8.2: Frequenzabhängigkeit der Lautstärkeempfindung

Die Phonzahl liefert zwar die Entscheidungsgrundlage für den Vergleich verschiedener Frequenzen (nach dem einfachen Schema: lauter, leiser oder gleich). Was fehlt ist aber nach wie vor ein gleichmäßig geordnetes System, in dem man sofort feststellen kann, in welchem Verhältnis ein Ton lauter oder leiser wahrgenommen wird als ein anderer.

Das leistet die Maßeinheit Sonen. Zur Umrechnung von Phon in Sonen bedient man sich wieder einer Bewertungskurve (in diesem Falle einer Linearfunktion), die den subjektiven Lautstärkeindruck einer Reihe von Testpersonen zusammenfaßt. Damit gibt es ein absolutes Maß für den Vergleich mehrerer Schallerzeuger. Ein Geräusch von 20 Sonen wird doppelt so laut empfunden wie eines von nur 10 Sonen, ein Ton mit 30 Sonen Lautheit ist subjektiv dreimal lauter. Ein Klavier erreicht mit seinem enormen Dynamikumfang Werte zwischen 1 und 40 Sonen. Der Vergleich mit den sechs üblichen Lautstärkestufen der Musik, von pp (pianissimo = sehr leise) bis ff (fortissimo = sehr laut), deckt einen interessanten Zusammenhang auf.

Jede Stufe entspricht etwa einer Verdoppelung der Lautstärkewahrnehmung, zwischen den Stufen pp und ff insgesamt sechs mal. Im Bereich bis 40 Sonen verdoppelt sich aber auch die subjektiv empfundene Lautheit über fünf mal ($2^5 = 32$). Sie entspricht damit weitgehend der musikalischen Lautstärkequantelung.

Partialtöne, Resonanz und Klangfarbe

Jeder, der schon einmal reine Sinustöne auf einem Synthesizer erzeugt hat, kennt das schrille Pfeifen hochfrequenter Schwingungen und das dumpfe Grollen im unteren Frequenzband. Die Variation von Frequenz und Lautstärke erlaubt zwar eine einfache Melodieführung, auf die Dauer ist dieser Zustand jedoch höchst unbefriedigend. Doch was befähigt das menschliche Gehör zu solch subtilen Unterscheidungsprozessen, wie der sicheren Identifizierung eines Bekannten am Telefon und das, obwohl die Signalbandbreite durch die Übertragung erheblich eingeschränkt wird? Was trennt den Klang einer Geige von dem eines Klaviers, welche spezifischen Eigenschaften unterscheiden das eingestrichene C¹ auf einer Klarinette von demselben Ton aus einer Oboe?... sicherlich nicht nur der Dynamikumfang.

Neben der Grundfrequenz (f_0) besteht der Klang eines Musikinstruments aus einer Vielzahl gleichzeitig mitschwingender Obertöne (bzw. Partialtöne = Teiltöne), die in einem harmonischen (ganzzahligen) Verhältnis zum Grundton stehen ($2 \times f_0$, $3 \times f_0$, $4 \times f_0$, ...). Individuelle Verteilung und Intensität der Partialtöne verleihen dem Instrument seine unverwechselbare Klangfarbe.

Mit Hilfe eines elektronischen Filters lassen sich einzelne Frequenzkomponenten isolieren. Trägt man ihre Energiedichte (Druckpegel) als Funktion der Frequenz auf, entsteht

ein charakteristisches Linienspektrum, eine Art "Fingerabdruck", der seine Schallquelle eindeutig verrät. Maxima im Linienspektrum stellen Obertöne dar, die besonders stark hervortreten und die Klangeigenschaften entscheidend prägen. Mit einem Equalizer kann man den Klang einer Aufnahme nachbearbeiten und individuell verändern, indem einzelne Frequenzen künstlich verstärkt, andere gedämpft oder ganz ausblendet werden. Was hier elektronisch geschieht, leistet auf mechanischem Wege ein sogenannter Resonator (resonare = wiederklingen). Bestimmte Frequenzen regen den Resonator zur Eigenschwingung an, erfahren dadurch eine enorme Verstärkung und dominieren das Linienspektrum. Gleichzeitig werden schwache Obertöne von den Spitzenpegeln maskiert (überlagert), so daß sie kaum noch hörbar sind.

Der Klangkörper einer Geige wirkt als Resonator und verleiht dem Instrument seine typische Klangfülle. Die gestrichene Saite allein klingt flach und bis ins Unkenntliche verzerrt. Auch der Sprachapparat arbeitet nach diesem Prinzip. Er verstärkt die Schwingungen der Stimmbänder, hohe Resonanzfrequenzen (die sogenannten Formanten) treten besonders deutlich hervor.

Nun ist ein Linienspektrum immer nur eine Momentaufnahme, eine Art Stichprobe an einem zufällig gewählten Zeitpunkt. Der Klang einer Gitarre zeichnet sich aber durch einen spezifischen Lautstärkeverlauf aus. Mit dem Zupfen der Saite erreicht er schnell den größten Gesamtdruckpegel, verharrt dort für einen gewissen Zeitraum, um dann langsam abzuklingen und schließlich ganz unter die Hörschwelle zu fallen. (Sofern er nicht schon längst von Umgebungsgeräusche überlagert worden ist.)

Dieser als Hüllkurve bezeichnete Verlauf unterscheidet grob die drei Phasen Anstieg (Attack), Halten (Delay) und Abklingen (Decay) und ist für den Klang eines Instruments ebenso typisch wie Verteilung und Intensität der Obertöne. Da nicht alle Partialtöne gleichmäßig auf ihren Spitzendruckpegel einschwngen und im selben Verhältnis abklingen, müßte die Hüllkurve für jede Frequenzkomponente einzeln analysiert werden. Eine möglichst genaue Tonrekonstruktion umfaßt Linienspektren über den gesamten Hüllkurvenverlauf. Es entsteht ein dreidimensionales Spektrogramm mit den Achsen Zeit, Frequenz und Schalldruckpegel.

Wellensalat – von Rausch- und Knacksgeräuschen

Das typische Spektrum eines harmonischen Klangs enthält steile Spitzen und tiefe Täler, die immer wieder den Null-Pegel erreichen. Es wirkt wie eine grotesk zerklüftete Berglandschaft, entstanden durch das Fehlen nichtganzzahliger Vielfacher der Grundfrequenz.

Geräusche haben keine harmonische Obertonreihe. Ihre Partialtöne stehen nicht in einem festen Frequenzverhältnis, sondern verteilen sich kontinuierlich über eine gewisse Bandbreite. Die dicht gepackten Frequenzen bilden im Pegeldiagramm eine geschlossene Hügelkette. Aus einem derartigen Wirrwarr von Frequenzen kann das menschliche Ohr keine eindeutige Höhenempfindung isolieren, man hört ein diffuses Rauschgeräusch.

Entscheidend ist der Kurvenverlauf des Linienspektrums. Ein schmalbandiges Rauschen mit einem steilen Pegelanstieg zum Maximum klingt wie ein Sinuston mit der Frequenz des stärksten Partialtons im Spektrum. Verteilt sich die Energiedichte dagegen fast gleichmäßig auf die Frequenzkomponenten, zeigt das Spektrogramm also einen sehr seichten Hügelverlauf ohne deutliches Maximum, erhält man ein extrem unruhiges Geräusch, das wohl jeder schon einmal (mehr oder weniger bewußt) gehört hat, das sogenannte Weiße Rauschen.

Es ist ein alter Bekannter aus der Nachrichtentechnik und taucht immer dann auf, wenn man bei der Senderwahl zwischen zwei Kanälen landet. Übrigens ist das Phänomen des Farbrauschens nicht nur auf den Akustikbereich beschränkt – ein Fernseher ohne Empfang zeigt das visuelle Äquivalent.

Ein interessanter Effekt entsteht, wenn sich in einem Rauschspektrum die Phasen sämtlicher Frequenzkomponenten so überlagern, daß sie in einem gemeinsamen Punkt ihr Maximum erreichen. Anstatt eines langgezogenen Rauscheindrucks hört man nun ein kurzes knacksendes Geräusch. Der Puls wird um so kürzer, je mehr Frequenzkomponenten im Linienspektrum enthalten sind; die Dauer (D) des Geräusches ist umgekehrt proportional zu seiner Bandbreite (B): $D = 1 / B$.

Theoretische Grundlagen der Analog-/Digitalwandlung

Die Temperatur eines Metalls, der Schweredruck einer Flüssigkeit, die Fallbeschleunigung an einem beliebigen Punkt der Erde und die Druckschwankungen eines Knallgeräusches haben eines gemeinsam – sie sind analoge Größen mit fließenden Übergängen und einer unendlich feinen Auflösung.

Um sie zu erfassen, entwickelte man eine Vielzahl analoger Meßinstrumente, deren Funktion auf natürlichen Wechselwirkungen zwischen Energie und Materie beruht. So nutzt ein Fieberthermometer die temperaturabhängige Ausdehnung von Alkohol oder Quecksilber zur Messung der analogen Größe Temperatur. Die Umsetzung des analo-

gen Ausdehnungsprozesses in ein digitales (= zählbares) Äquivalent übernimmt der Mensch durch das Ablesen der Meßskala, das Thermometer liefert lediglich den Vergleichsmaßstab zwischen Flüssigkeitssäule und geeichter Skala. Das digitale Meßergebnis stellt einen rationalen, konkret faßbaren Zahlenwert dar und kann beliebig weiterverarbeitet werden, beispielsweise als Eintrag in einer Fieberkurve.

Die Entwicklung geeigneter Verfahren zur direkten Analog-/Digitalwandlung ersetzte den Menschen als Mittler zwischen beiden Welten. Es gehört zu den grundlegenden Gesetzmäßigkeiten des Digitalisierungsvorgangs, daß bei der Umsetzung einer theoretisch unvorstellbar fein gequantelten (gestückelten) Größe in ein Zahlensystem mit festen Abständen Informationsverluste entstehen.

Im Audibereich hat sich für die Digitalisierung der Begriff Sampling eingebürgert. Dabei werden aus einem beliebigen Audiosignal in regelmäßigen Zeitabständen Stichproben (Samples) entnommen, die ermittelte Amplitudenhöhe digitalisiert und als Zahlencode im Speicher abgelegt. Ein D/A-Baustein kehrt den Prozeß wieder um und verwandelt die Wellendaten in analoge Spannungsimpulse, die über Verstärker und Lautsprecher als Klangeindruck wahrgenommen werden.

Auflösung und Quantisierungsfehler

Die Qualität des Samples hängt vor allen Dingen davon ab, wie häufig die Stichproben aus dem zeitkontinuierlichen Audiosignal entnommen werden (der sogenannten Samplingfrequenz) und mit welcher Auflösung der verwendete A/D-Wandler arbeitet.

Ein 16-Bit-Baustein, wie er beispielsweise in CD-Playern Verwendung findet, ist in der Lage 65536 ($= 2^{16}$) verschiedene Spannungspegel zu unterscheiden. Der maximale Rundungsfehler, der sich zwangsläufig aus der Diskrepanz zwischen endlichem Auflösungsvermögen des Wandlers und fließenden Übergängen der analogen Amplitudenauslenkung ergibt, entspricht dem kleinsten Quantisierungsschritt von einem 65536stel ($1/65536 = 1/2^{16}$). Bezogen auf den binären Charakter des Wandlungscodes, liest man auch häufig die Angabe 1 LSB (Least Significant Bit = niederwertigstes Bit).

Allgemein folgt der maximale Quantisierungsfehler (Q_m) eines A/D-Bausteins aus dem Term:

$$Q_m = 1/2^n \text{ [n = verarbeitete Wortbreite in Bit]}$$

Er ist ein theoretisches Maß für die größtmögliche Verfälschung des Ursprungssignals, im Durchschnitt liegt der Quantisierungsfehler wesentlich niedriger und bewegt sich im gesamten Intervall von 0 bis 1 LSB $[0 \dots 1/2^n]$.

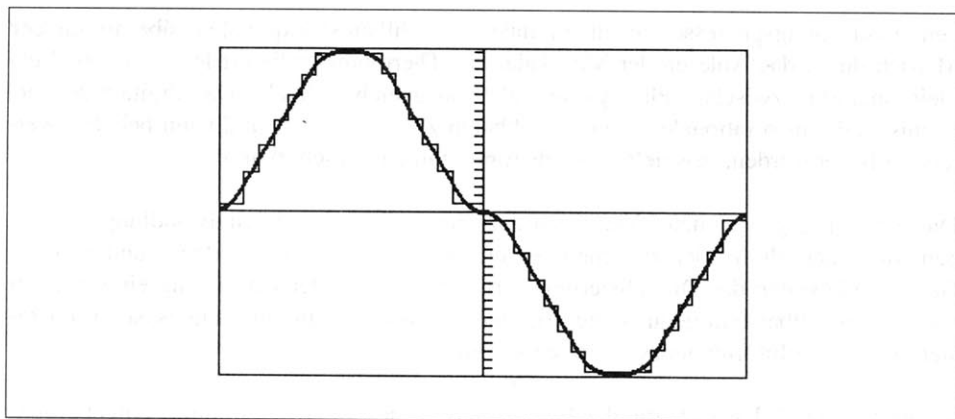


Abb. 8.3: Quantisierungsprozeß und Fehlerfunktion

Abbildung 8.3 zeigt die Schwankungen des Rundungsfehlers in einer Samplingperiode von 16 Abtastpunkten und beschreibt ihn als Funktion der Zeit. Der sägezahnförmige Verlauf des Quantisierungsfehlers ist kein Zufall, sondern absolut typisch für die Analog-/Digitalwandlung. Das wird am besten deutlich, wenn man das Verhalten der Fehlerfunktion zwischen zwei aufeinanderfolgenden Abtastpunkten T_n und T_{n+1} analysiert.

Nehmen wir einmal an, diese Punkte liegen auf dem steigenden Ast des Analogsignals, dann entsteht mit der Amplitudenquantisierung zum Zeitpunkt T_n ein gewisser Rundungsfehler, der durch den weiteren Verlauf des Analogsignals ausgeglichen wird (steil fallende Flanke des Sägezahn-Pulses) und schließlich beim Durchstoßen der Quantisierungsstufe (Berührung der x-Achse) gegen Null strebt. Während sich die ursprüngliche Wellenfunktion wieder von der Abtaststufe entfernt, steigt auch der resultierende Quantisierungsfehler (steigende Flanke des Sägezahns) und erreicht ein erneutes Maximum mit der Abtastung bei T_{n+1} . Dieses Wachstumsverhalten wiederholt sich zyklisch für jeden weiteren Samplepunkt und weicht nur bei Überschreitung des Wellenberges bzw. Tals von der strengen Sägezahnstruktur ab.

Bei der Digitalisierung eines Audiosignals entsteht also zwangsläufig ein sägezahnförmiges Quantisierungsrauschen, das den Rekonstruktionsprozeß des Samples gleichmäßig überlagert. Wegen des zyklischen Verhaltens zwischen zwei Abtastpunkten liegen seine Frequenzkomponenten durchweg oberhalb der Samplingfrequenz. Die Intensität des Quantisierungsrauschens hängt vom Auflösungsvermögen des Analog-/Digitalwandlers ab. Ein gutes Maß für die Stärke der Überlagerungen ist der Rausch-

/Signalpegelabstand, der aus dem effektiven Verhältnis zwischen Signalpegel und Quantisierungsfehler resultiert. Bei einer Auflösung von 8 Bit ist das Verhältnis wie 1 zu 256, der Pegelabstand beträgt 48 dB. Mit jedem weiteren Bit verdoppelt sich dieser Abstand, steigt also um 6 dB an. Die 16-Bit-Quantisierung eines CD-Players bringt es immerhin auf beachtliche 96 dB Rauschabstand.

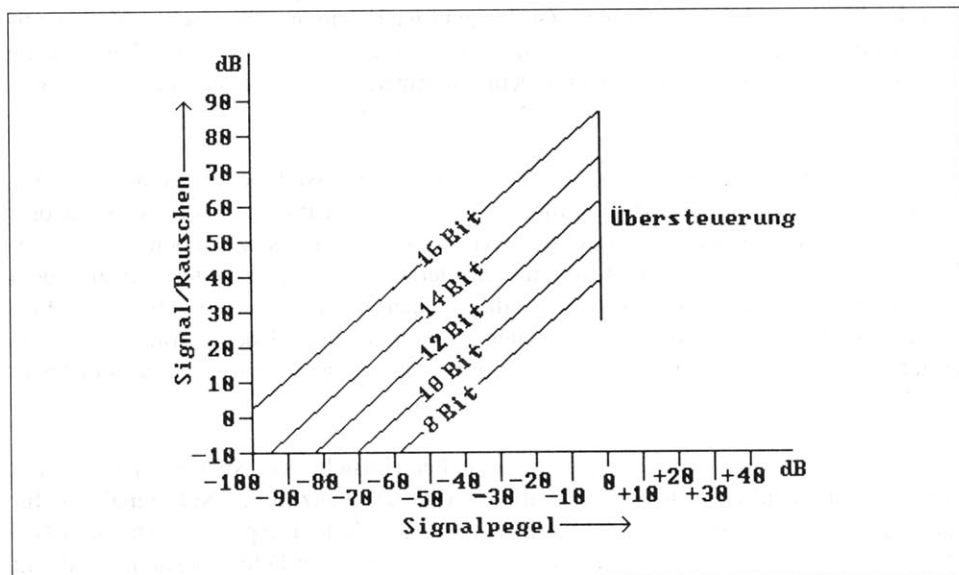


Abb. 8.4: Rausch-/Signalabstand bei verschiedenen Auflösungen

Die Abtastfrequenz

Neben dem Auflösungsvermögen eines A/D-Wandler spielt auch die Abtastrate eine entscheidende Rolle für eine möglichst verlustarme Wiedergabe des Analogsignals. Da der Wellenverlauf zwischen zwei Abtastpunkten nicht mehr rekonstruierbar ist, scheint der Zusammenhang recht einfach zu sein:

Je höher die Samplingfrequenz, desto besser die Wiedergabequalität.

Das ist sicherlich richtig, einer beliebigen Erhöhung der Abtastrate setzt die Hardware allerdings Grenzen; selbst die schnellsten Flash-Wandler benötigen für den Quantisierungsprozeß zweistellige Werte im Nanosekundenbereich – Leistung, die selbstver-

ständig auch ihren Preis hat. Im Audibereich sind diese "Wandler-Crays" fehl am Platze, hier gilt es, einen möglichst wirtschaftlichen Kompromiß zwischen Bauteilkosten und wahrnehmbaren Qualitätsverlusten zu finden.

Zum Verhältnis zwischen Abtastfrequenz und Genauigkeit der Rekonstruktion gibt es eine Reihe von theoretischen Untersuchungen. Die vielleicht wichtigste Erkenntnis liefert das Shannonsche-Abtasttheorem (so benannt nach dem amerikanischen Mathematiker Shannon). Es besagt, daß eine Schwingung mit der höchsten Frequenzkomponente B (Hz) vollständig determiniert ist (eindeutig definiert), sofern sie durch eine Serie von Abtastpunkten mit den Abständen $1/2 \times B^{-1}$ beschrieben wird.

Auf die Akustik übertragen liefert dieser Satz ein leistungsfähiges Instrument zur Abschätzung der erforderlichen Samplingfrequenz – sie sollte mindestens um den Faktor 2 größer sein, als die höchste im Audiosignal vertretene Frequenzkomponente. Ausgehend von dieser Erkenntnis, läßt sich die erforderliche Abtastrate weiter senken, indem man die Bandbreite des Eingangssignals durch einen Tiefpaßfilter begrenzt (tiefe Frequenzen passieren den Filter, hohe Frequenzen werden ausgeblendet). Auf diese Weise erzielen selbst preiswertere A/D-Wandler mit relativ langen Umsetzzeiten brauchbare Ergebnisse.

Ein derartiger Tiefpaßfilter ist sogar in doppelter Hinsicht von Nutzen. Er verhindert nämlich, daß hochfrequente Obertöne an der Abtastfrequenz in das Nutzsignal (das für die akustische Wahrnehmung wichtige Frequenzband) hineingespiegelt werden. Dieses Phänomen ist im angelsächsischen Sprachraum als Aliasing-Fehler bekannt und tritt immer dann auf, wenn aus einem periodischen Vorgang in gleichmäßigen Abständen Proben entnommen werden. Der Effekt beruht auf dem Verlust der Eindeutigkeit, sobald es Schwingungen geben kann, die sich aufgrund ihres Phasen- und Frequenzverhältnisses in den Abtastpunkten überschneiden.

Für jede Frequenz (F_x) des Eingangssignals entsteht eine Folge von Spiegelfrequenzen, die sich links (in den NF-Bereich) und rechts (analog in den HF-Bereich) von der Abtastrate (F_s) und ihren geradzahlgigen Vielfachen (nF_s) als Spiegelachse ansiedeln:

$F_x, (F_s - F_x), (F_s + F_x), (2F_s - F_x), (2F_s + F_x), \dots, (nF_s - F_x), (nF_s + F_x)$

Werden die Frequenzen zwischen Nutzsignalband und Samplingfrequenz nicht unterdrückt, reicht das Störspektrum weit in das Nutzsignal hinein und verzerrt die Wiedergabe. Schneidet der Tiefpaßfilter dagegen sämtliche Frequenzen oberhalb der halben Abtastrate ab (auch Faltungsfrequenz oder Nyquist-Frequenz genannt), so werden lediglich die Frequenzen des Nutzbandes gespiegelt. Das ist nicht weiter tragisch, da das Störspektrum jetzt sogar mit seiner tiefsten Frequenz (Linksspiegelung der höchsten Nutzfrequenz) nicht mehr in das Nutzband eindringen kann.

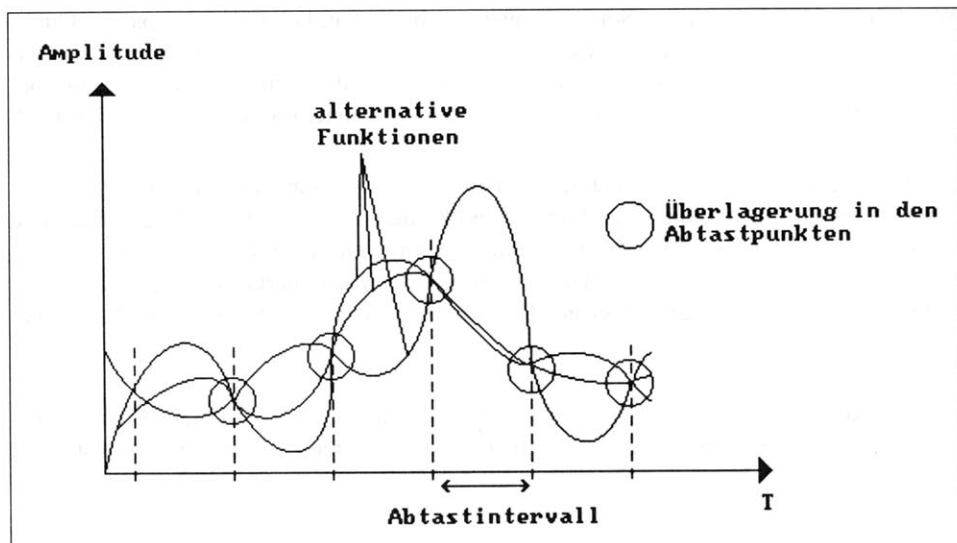


Abb. 8.5: Der Aliasing-Fehler

Dazu ein konkretes Beispiel:

- Nutzband: [0...4000 Hz]
- höchste Frequenz des Nutzbandes: $N_h = 4000 \text{ Hz}$
- Samplingfrequenz gemäß Abtasttheorem: $F_s = 8000 \text{ Hz}$
- tiefste Frequenz des Störspektrums: $F_t = F_s - N_h = 8000 - 4000 = 4000 \text{ Hz}$
- Störband: [4000... $nF_s + N_h$]

Ein weiterer Tiefpaßfilter am Ausgang des D/A-Wandlers schneidet schließlich auch das Störspektrum jenseits des Nutzbandes ab und sorgt für eine tadellose Wiedergabequalität.

Die Aussteuerung

Systemspezifische Störquellen, wie das thermische Eigenrauschen der Digitalisierungsschaltung lassen sich nicht beseitigen, wohl aber unnötige Rauscheinflüsse durch Über- oder Untersteuerung des Eingangssignals.

Wer schon einmal mit einem Soundsampler gearbeitet hat, kennt den frappanten Qualitätsverlust durch falsche Austeuerung. Viele Samplingprogramme weisen den Anwender deshalb mit akustischen oder visuellen Signalen auf den richtigen Aussteuerungspegel hin. Warum machen sich diese Fehler bei der Digitalisierung so negativ bemerkbar?

Wird ein Eingangssignal übersteuert, so driftet die Amplitude aus dem Quantisierungsbereich des A/D-Wandlers. Der Sampler wirkt nun wie ein Dynamikbegrenzer und schneidet die Informationen oberhalb seines maximalen Abtastwertes einfach ab. Bei Untersteuerung fällt der Signalverlauf in den unteren Quantisierungsbereich, die Auflösung des Samples wird grob und der Pegelabstand zwischen Rausch- und Nutzsignal nimmt stark ab.

Der Effekt läßt sich am Schicksal einer Sinusschwingung nachvollziehen, die in den Bereich von 2 LSB driftet...sie wird gnadenlos in ein rechteckförmiges Abtastsignal verwandelt.

Die Audio-Hardware des Amigas

Der Amiga besitzt vier unabhängige Kanäle, die als Stereoausgänge geschaltet sind (Kanal 0 und 1 auf der linken Audiobuchse, Kanal 2 und 3 auf der rechten Audiobuchse). Jeder Kanal verfügt über einen Digital-/Analogwandler, der vom DMA-System mit Daten versorgt wird.

Der D/A-Wandler

Die Audio-D/A-Wandler haben ein Auflösungsvermögen von 8 Bit und erzeugen Spannungspegel von $-0,4$ bis $+0,4$ V.

Die minimale Spannungsdifferenz am Ausgang liegt bei $0,003$ V ($= 0,8 / 256$ Quantisierungsstufen), der Rausch-/Signalabstand verhält sich wie 1 zu 256 und beträgt damit 48 dB.

Die Wandler verarbeiten vorzeichenbehafteten Binärcode in Zweierkomplementdarstellung (Wertebereich: -127 bis $+128$), so daß die Sampledaten problemlos als Byte-Feld eingebunden werden können.

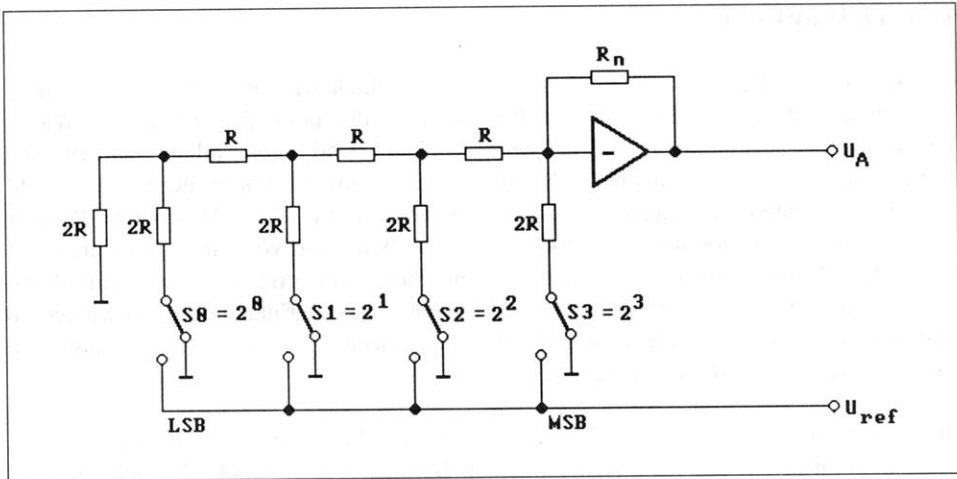


Abb. 8.6: Schaltung eines 4-Bit-D/A-Wandlers

Abbildung 8.8 zeigt den typischen Aufbau eines D/A-Wandlers, bestehend aus einem R-2R-Widerstandsnetzwerk mit nachgeschaltetem Operationsverstärker. Das Rezept ist ebenso einfach, wie trickreich und beruht auf den fundamentalen Gesetzmäßigkeiten der Reihenschaltung:

Man knüpfe eine Kette aus Widerständen mit der Maßzahl R und schlage an den Knotenpunkten Verbindungen zu einer Referenzspannung U_{ref} . Die Brücken sind mit jeweils $2R$ belastet und werden durch Schalter wahlweise mit der Referenzschiene verbunden oder von ihr abgetrennt. Auf diese Weise können die Bitzustände 1 und 0 simuliert werden. Mit jeder Schalterstufe wächst der Ersatzwiderstand des Netzes um den Faktor zwei, die erzeugten Ströme nehmen von rechts nach links proportional ab. Direkt vor dem Operationsverstärker ist der Ersatzwiderstand am geringsten und der Stromfluß am größten. Der hier liegende Schalter symbolisiert deshalb die höchste Binärstelle (MSB = 2^3) des Digitalcodes, der am weitesten links entfernte steht für die niederwertigste Bitposition (LSB = 2^0).

Damit die kleinen Ströme nicht von der Ungenauigkeit der großen überlagert werden, nimmt die Anforderung an die Fehlertoleranz der Widerstände von links nach rechts stetig zu (direkt vor dem Operationsverstärker muß die Genauigkeit am größten sein).

Im Amiga liegt hinter dem Ausgang des D/A-Wandlers noch eine Schaltung zur Umsetzung der Stromschwankungen in analoge Spannungsimpulse.

Der Tiefpaßfilter

Bevor das Signal aus den D/A-Wandlern an die Cinch-Ausgänge der Audiobuchsen gelangt, durchläuft es einen Tiefpaßfilter zur Unterdrückung der (bei jedem Wandlungsprozeß) auftretenden Störfrequenzen. Das sind vor allen Dingen die "unangenehmen" Spiegelungen des Nutzbandes rund um die Abtastrate und das hochfrequente Quantisierungsrauschen (Signalpegelabstand für 8-Bit-Auflösung: 48 dB). Ein guter Filter ist möglichst steilflankig, läßt das Nutzband vollständig passieren und schneidet alle Frequenzen jenseits des Paßbereiches schlagartig ab. Ideal – eine dynamische Unterdrückung, die sich nach dem Pegelniveau des Eingangssignals richtet, so daß trotz optimaler Unterdrückung der Störfrequenzen ein großer Teil des hörbaren Frequenzspektrums erhalten bleibt.

Der Tiefpaßfilter im Amiga ist allerdings von diesem Ideal weit entfernt; die Filterwirkung beginnt schon tief im niederfrequenten Bereich (bei ca. 4000 Hz), wird schnell stärker und erreicht schließlich bei 7 KHz das Sperrniveau.

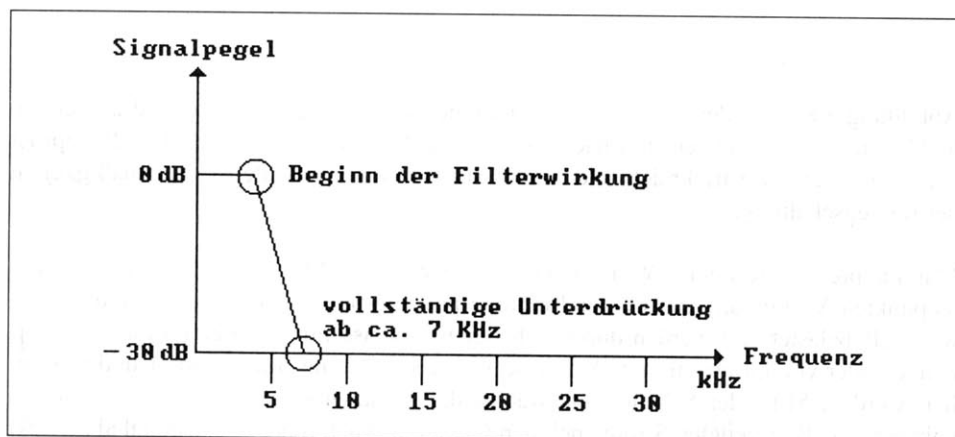


Abb. 8.7: Übereifrig – Filterwirkung des Tiefpasses im Amiga

Hier hat man sprichwörtlich "den Teufel mit dem Bezebub ausgetrieben". Viele Partiatöne, die für einen vollen Klangeindruck sorgen, fallen der frühen Unterdrückung zum Opfer. Da muß man sich nicht wundern, wenn aus einem hervorragend digitalisierten Geigenklang mit seiner sägezahnförmigen Wellenform jenseits des Filters ein in der Grundfrequenz pfeifender Sinus wird. Dieser Übereifer des Tiefpaßfilter ist um so ärgerlicher, da der Amiga bei einer maximalen Samplingrate von 28,6 KHz sogar Frequenzen von über 14 KHz einwandfrei ausgeben könnte (Shannonsches Abtasttheorem,

Faltungsfrequenz) – mit Programmiertricks (siehe Abschnitt Soft-Tuning) sind sogar noch höhere Frequenzen machbar.

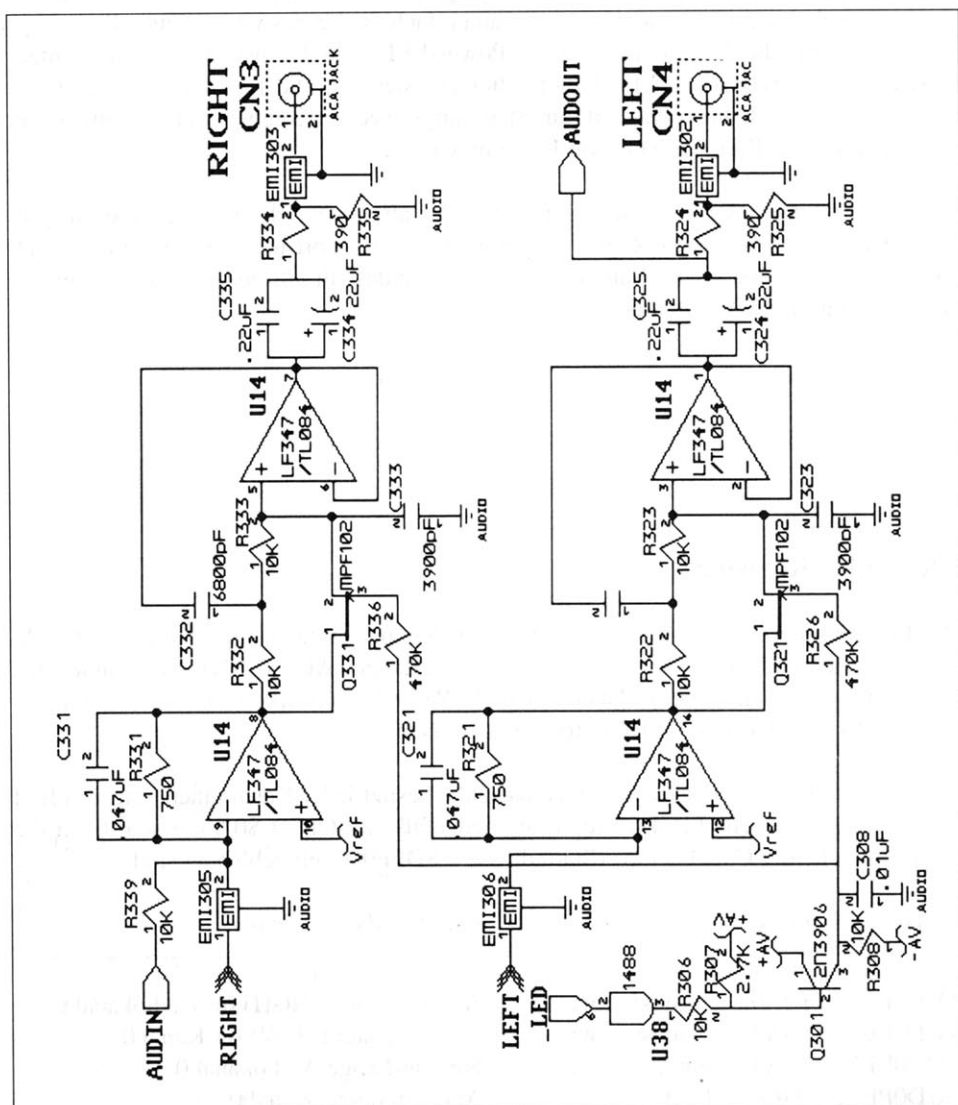


Abb. 8.8: Die technische Realisierung des Tiefpaßfilters

Glücklicherweise haben die Entwickler bei den Nachfolgemodellen A500 und B2000 das Problem entschärft und dem Programmierer ein "Hintertürchen" für die softwaremäßige Abschaltung des Tiefpaßfilters offengelassen. Um ein aufwendiges Redesign der Systemplatine zu vermeiden, nutzte man einfach die bereits vorhandenen Ressourcen und verlängerte die An/Aus-Leitung der Power-LED zum Tiefpaß, wo sie eine Umgehungsschaltung (HIGH-Pegel schaltet Filter ab) steuert. (Die Parallel-Ports der CIAs sind ja bekanntlich hervorragend für Steuerungszwecke geeignet und der einzige in Frage kommende Port-Ausgang war PA1 von CIA-A).

Zur Abschaltung des Filters muß der Anwender also lediglich eine 1 an die entsprechende Bitposition im PRA-Register schreiben. Da das Port-Register neben der LED-Kontrolle eine Reihe vitaler Funktion steuert, sollte das Bit nur gezielt (etwa mit einem Bitmanipulationsbefehl) verändert werden:

```
bset #1,$BFE001    ; schaltet Tiefpaß ab, Power-LED erlischt

bclr #1,$BFE001    ; schaltet NF-Filter wieder ein, LED
                    leuchtet
```

Die Audio-Register

Die Programmierung der Audio-Hardware ist denkbar einfach; man muß dem DMA-Controller lediglich Position (AUDxLCH/L) und Länge (AUDxLEN) der Sampledaten mitteilen, sowie Schwingungsdauer (AUDxPER) und Lautstärke (AUDxVOL) initialisieren – der Rest läuft im DMA-Betrieb automatisch.

Zur Steuerung und Abwicklung der Tonausgabe besitzt jeder Kanal einen Satz von fünf Registern, die (jeweils blockweise zusammengefaßt) ab Offset \$0A0 (Kanal 0) in die Registertabelle der Customchips (Startadresse = \$DFF000) eingeblendet sind.

Register	Offset	Chip	Zugriff	Funktionsbeschreibung
AUD0LCH:	\$0A0:	Agnus:	w:	Adreßregister (HIGH) für Audiokanal 0
AUD0LCL:	\$0A2:	Agnus:	w:	Adreßregister (LOW) für Kanal 0
AUD0LEN:	\$0A4:	Paula:	w:	Sample-Länge Audiokanal 0
AUD0PER:	\$0A6:	Paula:	w:	Abtastfrequenz Kanal 0
AUD0VOL:	\$0A8:	Paula:	w:	Lautstärkeregelung
AUD0DAT:	\$0AA:	Paula:	w:	DMA-Datenregister für Audiokanal 0

Register	Offset	Chip	Zugriff	Funktionsbeschreibung
xxxx:	\$0AC:	—:	—:	nicht belegt
xxxx:	\$0AE:	—:	—:	nicht belegt
AUD1LCH:	\$0B0:	Agnus:	w:	siehe Audiokanal 0
AUD1LCL:	\$0B2:	Agnus:	w:	dito
AUD1LEN:	\$0B4:	Paula:	w:	dito
AUD1PER:	\$0B6:	Paula:	w:	dito
AUD1VOL:	\$0B8:	Paula:	w:	dito
AUD1DAT:	\$0BA:	Paula:	w:	dito
xxxx:	\$0BC:	—:	—:	nicht belegt
xxxx:	\$0BE:	—:	—:	nicht belegt
AUD2LCH:	\$0C0:	Agnus:	w:	siehe Audiokanal 0
AUD2LCL:	\$0C2:	Agnus:	w:	dito
AUD2LEN:	\$0C4:	Paula:	w:	dito
AUD2PER:	\$0C6:	Paula:	w:	dito
AUD2VOL:	\$0C8:	Paula:	w:	dito
AUD2DAT:	\$0CA:	Paula:	w:	dito
xxxx:	\$0CC:	—:	—:	nicht belegt
xxxx:	\$0CE:	—:	—:	nicht belegt
AUD3LCH:	\$0D0:	Agnus:	w:	siehe Audiokanal 0
AUD3LCL:	\$0D2:	Agnus:	w:	dito
AUD3LEN:	\$0D4:	Paula:	w:	dito
AUD3PER:	\$0D6:	Paula:	w:	dito
AUD3VOL:	\$0D8:	Paula:	w:	dito
AUD3DAT:	\$0DA:	Paula:	w:	dito

Die ersten beiden Register der Tabelle (AUDxLCH und AUDxLCL) ergänzen sich zu einem 19-Bit-Adreßregister, das dem DMA-Controller als Zeiger auf die Sampledaten des zugehörigen Tonkanals dient. AUDxLCH (höherwertiger Teil des Adreßregisters) enthält die Bits 18 bis 16, AUDxLCL (niederwertiger Teil) nimmt die restlichen Bits 15 bis 0 der 19-Bit-CHIP-RAM-Adresse auf. Wie alle Adreßzeiger des DMA-Systems liegen auch diese Register als Arbeitsregister des Adreßgenerators auf dem Fat-Agnus-Chip.

Das AUDxLEN-Register gibt die Länge der Sampledaten an. Da sämtliche DMA-Prozesse wortweise ablaufen, bezieht sich diese Größenangabe auch auf Worte. Die Byte-

länge des Samples muß also bei der Initialisierung des AUDxLEN-Register durch den Faktor 2 dividiert werden. Leider hat das Register nur eine Breite von 16 Bit, wodurch die maximale Länge eines Samples auf 65536 Worte bzw. 128 KByte begrenzt wird.

Mit dem AUDxPER-Register legt der Programmierer die Dauer einer Ausgabeperiode fest, also den Zeitraum, der zwischen der Ausgabe zweier Datenbytes (Abtastwerte) liegt. Um von einer vorgegebenen Samplingrate zur Abtastperiode zu kommen, bildet man einfach den Kehrwert der Samplingfrequenz ($f = 1/T$). Die Audio-Hardware kann jedoch mit einer absoluten Zeitangabe in Mikro- oder Nanosekunden nichts anfangen, sondern benötigt die Anzahl von Buszyklen, die während dieses Zeitraums verstreichen. Der endgültige Wert ergibt sich dann aus dem Quotienten von ermittelter Ausgabeperiode und Buszykluszeit (282 ns):

$$\text{AUDxPER} = 1 / (\text{Samplingrate} * 282 \text{ ns})$$

Obwohl das AUDxPER-Register theoretisch alle Werte von 0 bis 65536 annehmen kann, ist nicht jede Initialisierung auch sinnvoll. Im DMA-Betrieb wird die maximale Ausgaberate durch die statische Verteilung der DMA-Slots beschränkt. Pro Rasterzeile und Kanal steht nur ein Slot zur Verfügung; der DMA-Controller liest dann ein 16-Bit-Wort ein, das für zwei Ausgabezyklen reicht. Bei einer Zeilendauer von 228 (lange Zeile) Buszyklen, darf die Ausgabeperiode nicht unter 114 ($= 228/2$) sinken, da die Daten sonst nicht mehr rechtzeitig bereitgestellt werden können. Tritt dieser Fall auf, wiederholt der Audiokanal einfach die letzten beiden Datenbytes, bis neue Daten eintreffen.

Schließlich benötigt auch der D/A-Wandler für den Umsetzungsprozeß und die Stabilisierung der Ausgangsspannung (Einschwingzeit) weitere 10 Zyklen, so daß sich der Mindestwert auf 124 Buszyklen erhöht. Die maximale Samplingrate im DMA-Betrieb ist dadurch auf ca. 28,6 KHz ($1/[124*280\text{ns}]$) begrenzt.

Die Lautstärke der Tonausgabe läßt sich mit dem AUDxVOL-Register in 65 Stufen [1 ... 64] einstellen (dafür werden die untersten 6 Bit des Registers verwendet). Die Pegelniveaus orientieren sich am Austeuerungsbereich für Musikaufnahmen (optimales Niveau bei ± 0 dB) und korrespondieren mit Schalldruckpegeln von ca. -36 dB (Stufe 1) bis 0 dB (Stufe 64).

Zum Ausgleich für das logarithmische Wachstumsverhalten der Maßeinheit dB ist die Auflösung mit 1/10 dB im oberen Lautstärkebereich besonders fein und wird nach unten hin zunehmend gröber. Das erlaubt eine optimale Ausnutzung des 6-Bit-Intervalls, selbst geringe Abweichungen (wie beispielsweise zwischen Stufe 25 und 28) können noch wahrgenommen werden.

Einige Lautstärkestufen und der korrespondierende Druckpegel in Dezibel:

Stufe	Schalldruckpegel [dB]
64	0.0
52	-1.8
40	-4.1
28	-7.2
16	-12.0
8	-18.1
4	-24.1
2	-30.1
1	-36.1

Der Ausgabezyklus

Nachdem alle Register eines Audiokanals korrekt initialisiert sind, beginnt die Ausgabe mit der Aktivierung des zugehörigen DMA-Kanals. Das geschieht durch Setzen des entsprechenden AUDxEN-Bit (Audio X Enable) im DMA-Kontrollregister DMACON (Bits 0 bis 3 für die Kanäle 0...3). Beim Zugriff auf das Register muß das Set-Bit (Nr. 15) und der DMA-Hauptschalter (Bit 9) gesetzt sein.

Folgende Sequenz schaltet Kanal 1 kurzzeitig ein und wieder aus:

```
move #$8402,$DFF09a    ;DMA aktiv
move #$0002,$DFF09a    ;schaltet beide Kanäle ab
```

Ablauf und Organisation der Audio-DMA unterscheidet sich deutlich von anderen DMA-unterstützten Systemkomponenten wie Bitplane-, Blitter- oder Sprite-Hardware. Während bei diesen Quellen die Registerinhalte im DMA-Betrieb nicht verändert werden können, muß der Soundprogrammierer für die Generierung spezieller Effekte auch im DMA-Zyklus vollen Zugriff behalten. Aus diesem Grund nutzt das DMA-System die Audioregister nicht direkt als Arbeitspuffer, sondern kopiert sie vor Beginn des DMA-Zugriffs in dafür vorgesehene Register um. Die Adreßregister bleiben unverändert, obwohl sie normalerweise als Datenzeiger mit jedem eingelesenen Wort inkrementiert werden (siehe Bitplane- oder Sprite-Hardware).

Aber auch in der organisatorischen Abwicklung des DMA-Betriebs fällt die Audio-Hardware aus dem Rahmen. Durch die Möglichkeit, Registerinhalte während des DMA-Zyklus zu verändern, verhält sich die Audio-Hardware für den DMA-Controller

unberechenbar (beispielsweise schwankt der Datenbedarf bei Frequenzmodulation erheblich). Um dennoch eine reibungslose Datenversorgung zu garantieren, arbeiten Paula und Fat Agnus eng zusammen. Eine Steuerlogik auf Paula fordert immer rechtzeitig neue Daten für die D/A-Wandler an, der Rest bleibt dem DMA-Controller überlassen.

Zu Beginn des Audio-DMA-Zyklus erledigt das System eine Reihe wichtiger Initialisierungsarbeiten. Das AUDxLCH/L-Register wird in den Sampledatenzeiger vom Adreßgenerator kopiert, AUDxLEN und AUDxPER in die Kontrollzähler des Audiokanals geschoben. Sobald der Kopiervorgang abgeschlossen ist, holt der DMA-Controller das erste Datenwort aus dem Speicher und schreibt es in das AUDxDAT-Register des zugehörigen D/A-Wandlers. Mit jedem Speicherzugriff wird der Datenzeiger inkrementiert, der AUDxLEN-Kontrollzähler parallel dazu heruntergezählt.

Bei Ausgabe des niederwertigen bzw. zweiten Datenbytes des AUDxDAT-Registers erhält der DMA-Controller sofort eine erneute Datenanforderung. Den korrekten Zeitabstand zwischen zwei Ausgabezyklen ermittelt die Audiohardware durch Dekrementierung des Periodenzählers (Kopie von AUDxPER), nach jedem Unterlauf wird er wieder mit dem Wert des AUDxPER-Registers initialisiert. Kann der DMA-Controller die Sampledaten nicht schnell genug heranschaffen, wird die Wartezeit durch erneute Ausgabe der letzten Datenbytes (im AUDxDAT-Register) überbrückt.

Sobald der AUDxLEN-Kontrollzähler bei Null angelangt ist, löst das System einen Level-4-Interrupt aus. (Bits 7 bis 10 von INTENA/INTREQ sind in der aufgezählten Reihenfolge für die Kanäle 0 bis 3 des Audiosystems reserviert.) Während für die Veränderung der aktuellen Soundparameter nun bis zur nächsten Unterbrechung Zeit bleibt, beginnt der DMA-Zyklus erneut mit der Initialisierung der Arbeitsregister. Die Audio-Interrupts stecken den Zeitrahmen ab, in dem die Registerinhalte problemlos modifiziert werden können und bilden damit eine wichtige Kommunikationsschnittstelle zwischen Sound-Hardware und Programmierer.

Da das AUDxDAT-Register auch für den Anwender transparent ist, muß die Tonausgabe nicht zwangsläufig über das DMA-System laufen. Alternativ kann der Prozessor die Sampledaten lesen und zum D/A-Wandler transportieren. Kaum der Rede wert, daß diese Methode extrem viel Prozessorzeit kostet und deshalb nur in Ausnahmefällen (beispielsweise zur Realisierung höherer Samplingraten) sinnvoll erscheint.

Beim direkten Zugriff auf das AUDxDAT-Register ändert sich allerdings das Interrupt-Timing; die Hardware unterbricht jetzt bereits nach Ausgabe eines Datenwortes (also jeden zweiten Ausgabezyklus). Anders verhält sich das System auch bei verspätetem Nachladen des AUDxDAT-Registers. Im Gegensatz zum DMA-Betrieb werden nun

nicht mehr die letzten beiden Datenbytes wiederholt, sondern der D/A-Ausgang schwingt auf sein aktuelles Niveau ein (letztes Datenbyte).

Die Register AUDxLCH/L und AUDxLEN haben im CPU-Betrieb keine Funktion mehr, so daß sich ihre Initialisierung erübrigt.

Frequenz- und Amplitudenmodulation

Um Frequenz und Auslenkpegel (Amplitudenhöhe) einer Schwingung während des Ausgabezyklus gezielt zu modulieren (hier: zu verändern), ist es jederzeit möglich, Lautstärkeregister AUDxVOL (AM-Modulation) bzw. Periodenregister AUDxPER (FM-Modulation) mit dem Prozessor entsprechend zu modifizieren.

Der Amiga bietet dem Programmierer allerdings eine wesentlich komfortablere Alternative – gezielte Modulationseffekte mit DMA-Unterstützung. Dabei arbeitet ein Audiokanal weiterhin als D/A-Wandler, setzt seine Sampledaten also ganz normal in akustische Signale um. Der andere verwendet seine Daten zur Amplituden- bzw. Frequenzmodulation; wahlweise lassen sich auch beide Modulationsarten koppeln.

Die Kombinationsmöglichkeiten zwischen aktivem und modulierendem Kanal sind jedoch beschränkt und durch die Bitfunktion im ADKCON-Register (Offset \$09E) festgelegt. So kann ein Audiokanal nur seinen unmittelbaren Nachfolger modulieren. Kanal 3 hat definitionsgemäß keinen Nachfolger und wird nicht zur Modulation verwendet.

<i>Register</i>	<i>Adresse</i>	<i>Zugriff</i>	<i>Funktion</i>
ADKCON	\$DFF09E	w	Audio-Disk-Kontrollregister
			Bitbelegung (nur Funktionen der Audio-Hardware):
			Nr. Label Erläuterung
Frequenzmodulation:			
		7	USE3PN Audiokanal 3 moduliert nichts
		6	USE2P3 Kanal 2 moduliert AUDPER von Kanal 3
		5	USE1P2 1 moduliert AUDPER von 2
		4	USE0P1 0 moduliert AUDPER von 1

Register	Adresse	Zugriff	Funktion
Amplitudenmodulation:			
		3	USE3VN Kanal 3 moduliert nichts
		2	USE2V3 Kanal 2 moduliert AUDVOL von 3
		1	USE1V2 1 moduliert AUDVOL von 2
		0	USE0V1 0 moduliert AUDVOL von 1

Bit 7 und 3 sind zwar zur Steuerung von Kanal 3 aufgeführt, haben aber keine Funktion.

Sobald ein Audiokanal durch Setzen des entsprechenden Bits im ADKCON-Register als Modulator gekennzeichnet ist, wird sein DMA-Kanal umfunktioniert. Der DMA-Controller liest zwar wie immer die Daten aus dem CHIP-RAM-Speicher, schreibt sie aber nicht mehr in das AUDxDAT-Pufferregister des D/A-Wandlers, sondern in das Lautstärke- und/oder Periodenregister des nächsthöheren Kanals. Je nach Modulationsart ist die Interpretation der Datenworte unterschiedlich; während der kombinierten FM/AM-Modulation schreibt der DMA-Controller (beginnend mit dem Lautstärkeregister) abwechselnd in das AUDxVOL- und AUDxPER-Register.

Verwendung der Datenworte bei unterschiedlichen Modulationsarten:

AM-Modulation:	[15 ... 7]	unbenutzt
	[6 ... 0]	6-Bit-Lautstärkepegel für AUDxVOL
FM-Modulation:	[15 ... 0]	16-Bit-Frequenzwert nach AUDxPER

Soft-Tuning oder: wie man der Hardware auf die Sprünge hilft

Nicht immer sind die Grenzen der Hardware unüberwindbar; oft genügen ein paar programmtechnische Tricks, um ungeahnte Fähigkeiten zu entfesseln.

Longplay – Samples mit mehr als 128 KByte

Eine ärgerliche Beschränkung für die Länge eines Samples ergibt sich aus dem begrenzten Wertebereich des AUDxLEN-Registers. Wer größere Musikstücke mit mehr als 128 KByte abspielen möchte, muß aber deshalb nicht gleich resignieren; mit etwas mehr Programmieraufwand lassen sich auch extra lange Samples problemlos verarbeiten.

Der Trick dabei – man überlistet die Audio-Hardware und spaltet das überlange Samplefile in leicht "verdauliche" Portionen auf (kleiner 128 KByte, Wert \$FFFF im AUDxLEN-Register). Die Umschaltung zwischen den einzelnen Teilen läuft verdeckt innerhalb des Audio-Interrupts. Dazu muß lediglich eine kleine Interrupt-Routine in den Level-4-Interrupt eingeklinkt werden, die zu Beginn jedes DMA-Zyklus aufgerufen wird und die Register AUDxLCH/L und AUDxLEN entsprechend modifiziert.

Soweit die Theorie – es folgt eine Beispielroutine für die Verwaltung eines überlangen Soundstückes mit bis zu 256 KByte. Beim Aufruf übergibt der Interrupt-Handler des Betriebssystems in a0 die Basisadresse der Customchips und in a1 den Zeiger auf einen lokalen Speicherbereich (iv data), in dem das Hauptprogramm folgende Parameter abgelegt hat:

Offset	Parameter
0	Zustand: 1 = auf zweiten Teil umschalten / 0 = ersten Teil spielen (beim Start mit 0 initialisiert)
4	Länge des zweiten Teils (Wert für AUDxLEN = Gesamtlänge – \$FFFF)
8	Backup-Adresse, Anfang des Samples

#asm

```

        public _audint
_audint:
        tst.l (a1)                ;Zustand testen
        bne teil2                ;wenn 1, dann verzweige zu teil2

teil1:
        move.l 8(a1), $0a0(a0)    ;Sampleanfang nach AUD0LCH
        move.w #$FFFF, $0a4(a0)  ;AUD0LEN auf maximale Länge
                                   ;setzen
        addq.l #1, (a1)           ;Zustand 1 erzeugen
        bra end                  ;überspringe teil2

teil2:
        move.l 8(a1), a2          ;Sampleanfang in a2 puffern
        add.l #$20000, a2         ;Zeiger um 128K erhöhen
        move.l a2, $0a0(a0)       ;und in AUD0LCH abspeichern
        move 4(a1), $0a4(a0)      ;Länge des zweiten Teils nach
                                   ;AUD0LEN
        subq.l #1, (a1)           ;Zustand 0 erzeugen

```

```

end:
    move #$80,$09c(a0)      ;Interrupt-Request löschen!!
                             (Bit 7, INTREQ)
    rts                     ;Rückkehr ins Hauptprogramm

#endasm

```

Die Routine ist aus dem Programm "LongPlay" auf der Demodiskette entnommen. Mit erweiterter Zustandsabfrage kann sie auch als Basis für Samples mit über 256 K-Länge dienen.

Höhere Ausgabefrequenz, größere Auflösung

Was auf den ersten Blick unmöglich erscheint, entpuppt sich als vielleicht gar nicht so abwegig. Obwohl die Samplingrate im DMA-Betrieb bekanntlich auf 28,6 KHz begrenzt ist, kann der Prozessor wesentlich höhere Ausgabefrequenzen erreichen; vorausgesetzt, die Audio-Interrupts kommen ihm nicht in die Quere. Durch Ausblendung der Bits 7 bis 10 im INTENA-Register werden sie gesperrt, das Experiment kann beginnen:

Nehmen wir an, die Adresse der Sampledaten befindet sich in dem Zeiger "samplestart" und die Länge des Stücks (zur Vereinfachung: < 128K) ist in der Variablen "länge" gespeichert, dann könnte die Ausgaberroutine für Kanal 0 etwa so aussehen:

```

    lea $DFF0aa,a5           ;AUD0DAT nach a5
    move.l _samplestart,a4    ;Startadresse nach a4
    move.w _länge,d0          ;Länge nach d0
    subq.w #1,d0              ;dbra zählt bis -1 !!

loop:
    move.w (a4)+,(a5)         ;CHIP-RAM -> AUD0DAT,
                             Befehlszykl.: 12 Tz
    dbra d0,loop              ;zurück zum Schleifen-
                             kopf, 10 Tz

```

Die Schleife ist extrem schnell. Sie benötigt pro bewegter Information lediglich 22 Prozessortaktzyklen (à 140 ns) bzw. 11 Buszyklen (à 282 ns), so daß sich theoretisch Samplingraten von über 640 KHz ($[3.545 \text{ MHz} / 11] * 2$) realisieren lassen.

Das Problem dieser Technik liegt in der fehlenden Abstimmung zwischen Periodenzähler (AUDPER) und Datenversorgungsroutine, alles hängt von einer exakten Syn-

chronisation ab. Die Ausgabeperiode für die 22-Taktschleife müßte eigentlich 5,5 Buszyklen ($= 11 / 2$) betragen, da aber nur ganzzahlige Werte akzeptiert werden, bleibt nichts anderes übrig, als die Routine durch zusätzliche Befehle zu dehnen und an eine ganzzahlige Ausgabeperiode anzupassen (benötigte Prozessortakte müssen glatt durch den Faktor 4 teilbar sein).

Sampeln wie ein CD-Player

Dieses Dilemma kennt der DMA-Betrieb nicht und auch er kann – entsprechende Programmierung vorausgesetzt – noch kräftig zulegen. Jedem Tonkanal steht zwar nur ein DMA-Slot pro Rasterzeile zu, nichts spricht jedoch dagegen, sich die Ausgabearbeit mit einem weiteren Kanal zu teilen. Dazu wird der (mit einer hohen Samplingrate digitalisierte) Datenblock in zwei Puffer für gerade bzw. ungerade Bytes umkopiert (ein Tonkanal ist für die geraden, ein anderer für die ungeraden Sample-Bytes zuständig) und die Ausgabe der D/A-Wandler zeitlich ineinander verschachtelt. Während der eine Audiokanal sich nun mitten in seinem Ausgabezyklus befindet (Periodenzähler ist bis zur Hälfte heruntergezählt), setzt der andere D/A-Wandler bereits ein neues Datenbyte um – die effektive Ausgaberate steigt auf das Doppelte.

Schon mit zwei Tonkanälen (die Technik funktioniert auch mit 3 oder sogar 4 Kanälen) erreicht man Samplingraten (max. $57,2 \text{ KHz} = 28,6 \text{ KHz} * 2$), die jenseits der üblichen Werte eines CD-Players liegen!

Sehen wir uns dazu ein Beispiel an:

Der CD-Player arbeitet mit einer Ausgabefrequenz von ca. 44,1 KHz, was auf dem Amiga einer Periodendauer von 80,5 Buszyklen ($= 3,55 \text{ MHz} / 44,1 \text{ KHz}$) entspricht. Da der DMA-Betrieb aber pro Kanal nicht weniger als 124 Systemtakte zwischen zwei Ausgabezyklen verkraftet, muß die Belastung auf zwei Audiokanäle verteilt werden, denen dadurch die doppelte Zeitspanne von ca. 160 Buszyklen (abgerundet!) zur Verfügung steht.

Nehmen wir einmal an, die Datenpuffer seien bereits erstellt und die Audioregister korrekt initialisiert (Adreßregister zeigen auf den geraden bzw. ungeraden Bytepuffer, Länge = Puffergröße, Periode = 160, Lautstärke beliebig). Die Wiedergabequalität steht und fällt mit der Synchronisation der Ausgabezyklen, der DMA-Start des zweiten Kanals muß exakt um eine halbe Ausgabeperiode (hier: 80 Buszyklen) verzögert werden. Da es für derartig kurze Zeiträume keine Systemfunktionen gibt und auch kein Rasterstrahl-Interrupt in Frage kommt, bleibt nichts anderes übrig, als die Verzögerungsroutine selbst zu basteln – beispielsweise so:

```

move %10000010000000001,$DFF09a
                                ;Audio-DMA von Kanal 0
                                einschalten

```

Beschäftigungstherapie für den Prozessor:

```

move.l $c00000,$c00000        ;nicht besonders sinnvoll,
                                "frißt" aber 36 Tz
move.l $c00000,$c00000        ;+ 36 Tz
move.l $c00000,$c00000        ;+ 36 Tz
move.w $c00000,$c00000        ;+ 28 Tz
nop                             ;"nichts tun"
                                kostet auch Zeit (4 Tz)

move %10000010000000010,$DFF09a
                                ;Audio-DMA Kanal 1
                                einschalten (20 Tz).

```

Der MOVE-Befehl zum Einschalten des zweiten Tonkanals benötigt zur Ausführung weitere 20 Taktzyklen, so daß sich insgesamt eine Startverzögerung von 160 CPU-Takten oder 80 Buszyklen ergibt!

Nach dem Experiment zur Steigerung der Ausgabefrequenz liegt es nahe, die Verschachtelungstechnik auch auf den Quantisierungsbereich auszudehnen. Ein 16-Bit-Sample läßt sich in Byte-Hälften aufspalten und auf zwei benachbarten Tonkanälen ausgeben.

Zur Datenaufbereitung muß der Zweierkomplementcode des Samples in eine Dualzahl umgerechnet und sein Vorzeichenbit (Nr. 15, zeigt die Auslenkrichtung an) in Bit 7 des unteren Datenbytes kopiert werden. Anschließend werden beide Hälften wieder in die übliche Darstellung zurückverwandelt.

Da die DMA-Slots des Audio-Systems dicht zusammenliegen, beträgt die Verzögerung zwischen den Ausgabezyklen lediglich 560 Nanosekunden (2 Buszyklen), eine Zeitspanne also, die weit unterhalb der Wahrnehmungsgrenze liegt.

Damit beide Kanäle zu einem 16-Bit-System verschmelzen, muß die Basisamplitude des höheren Datenbytes (Bits 8 bis 15) durch entsprechende Initialisierung des AUDVOL-Registers nach oben verschoben werden.

Der Amiga als Synthesizer

Rekonstruieren statt Digitalisieren – Wellenformen selbst entworfen

Die D/A-Wandler des Audio-Hardware eignen sich natürlich nicht nur zur Reproduktion digitalisierter Aufnahmen, sondern auch zur Synthese beliebiger Klänge und Soundeffekte, der Erzeugung melodischer Tonfolgen oder gar einer täuschend echten Simulation von Musikinstrumenten.

Um einen Audiokanal als Tongenerator zu programmieren, bedarf es keiner aufwendigen Digitalisierungsarbeit – einfache periodische Wellenformen kann man auch mit Papier und Bleistift konstruieren. Man muß lediglich den Amplitudenverlauf einer Schwingungsperiode durch eine Reihe gleichmäßig gewählter Abtastpunkte beschreiben und die ermittelten Werte in einer Datenliste für den D/A-Wandler zusammenfassen.

Ist der Amplitudenverlauf sehr komplex, kann es hilfreich sein, die Schwingung in ein Koordinatensystem zu übertragen, das die Abtastpunkte als X-Achse und den Quantisierungsbereich $[+127, -128]$ als Y-Achse enthält.

Besonders einfach geht es dagegen, wenn die analysierte Wellenform bereits von einer mathematischen Funktion exakt beschrieben wird, wie beispielsweise die Sinuskurve. Da sich sämtliche periodische Schwingungen aus Sinuskomponenten mit festen Phasen- und Frequenzbeziehungen zusammensetzen (Fourier-Reihe), ist diese Wellenform für die Generierung künstlicher Klänge besonders interessant.

Im Bogenmaß hat die Sinuskurve eine Schwingungsperiode von 2π . Zur ihrer Beschreibung nimmt man beispielsweise eine Folge von acht Amplitudenwerten im Abstand von $\pi/4$. Die Sinuswelle soll den gesamten Quantisierungsbereich ausnutzen und oszilliert deshalb zwischen den Werten $+127$ und -127 . Der Amplitudenverlauf folgt dann aus dem Term $127 \cdot \sin(x)$ und es entsteht die Datenliste:

sample:

```
dc.b 0,+91  
dc.b +127,+91  
dc.b 0,-91  
dc.b -127,-91
```

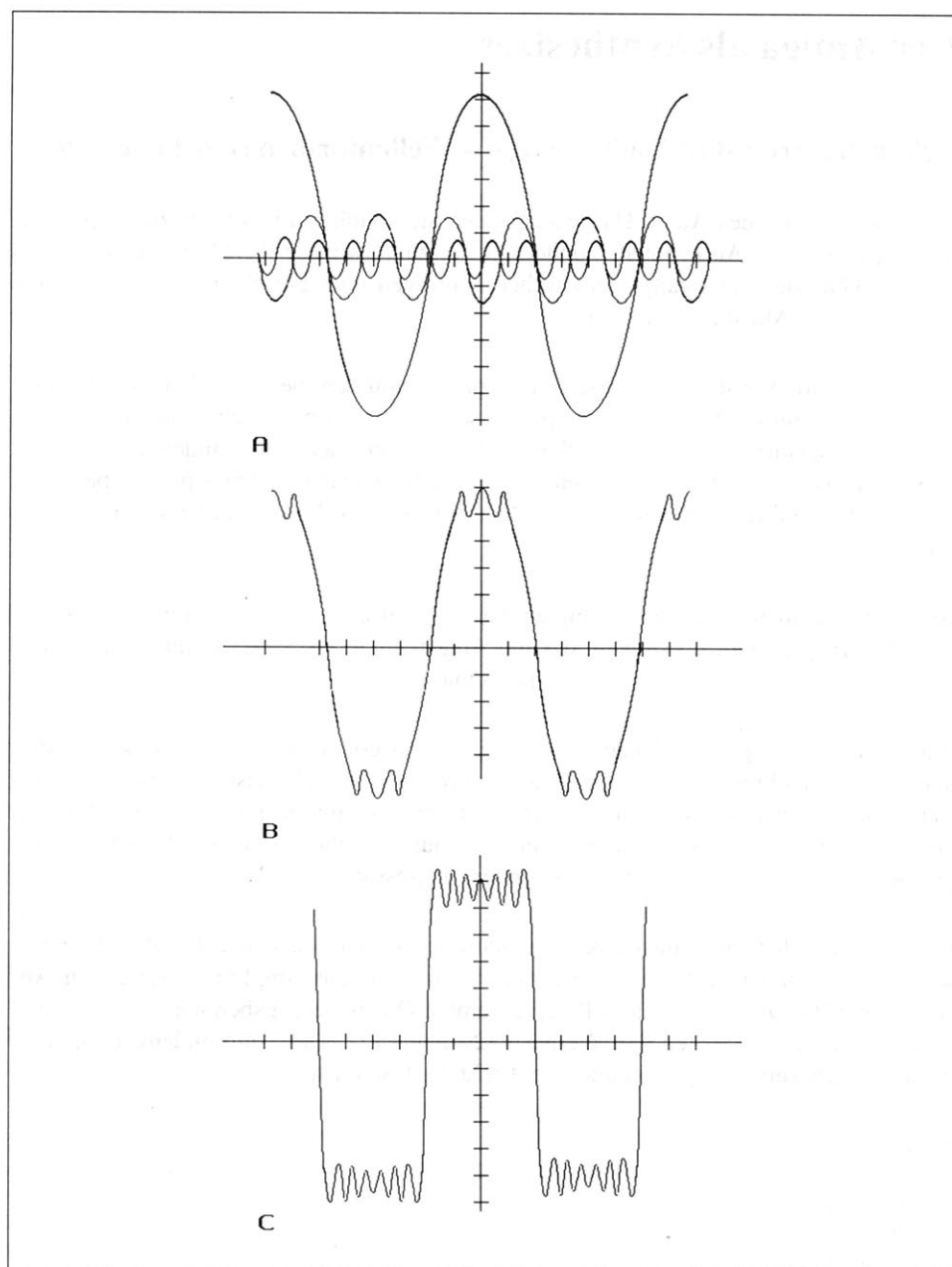



Abb. 8.9: Entstehung einer Rechteckschwingung durch Überlagerung von Sinuswellen

Alles streng nach Noten

Damit lässt sich bereits ein brauchbarer Sinusoszillator programmieren, den man zum Spielen von Melodien und kleinen Musikstücken nutzen kann. Da diese in Form von Noten niedergeschrieben sind, erstellt man am besten eine Umrechnungstabelle für Notenfrequenzen und ihre korrespondierenden Samplingraten.

<i>Note</i> *	<i>Frequenz (Hz)</i> <i>(ideal)</i>	<i>Ausgabeperiode für AUDxLEN = 8</i> <i>(auf- bzw. abgerundet)</i>
------------------	--	--

A	440.0	1006
A#	466.2	949
B	493.9	896
C	523.3	846
C#	554.4	798
D	587.3	753
D#	622.3	711
E	659.3	671
F	698.5	634
F#	740.0	598
G	784.0	564
G#	830.6	533
A	880.0	503
A#	923.3	479
B	987.8	448
C	1046.5	423
C#	1108.7	399

* Mit jedem Halbtonschritt erhöht sich die Tonfrequenz um den Faktor $2^{1/12}$

Die Berechnung der Ausgabeperiode kann man leicht am Beispiel des Kammertons A (440 Hz) nachvollziehen.

Die Samplingrate entspricht dem Produkt aus Tonfrequenz und der Anzahl von Datenwerten, die zur Definition einer Schwingungsperiode benötigt werden. Bei einer Frequenz von 440 Hz und 8 Samples ergibt sich also eine Abtastrate von $440 * 8 = 3520$ Hz. Zur Umrechnung in die obligatorische Anzahl von Buszyklen bildet man den Kehrwert der Samplingrate und dividiert ihn durch die Dauer eines Buszyklus:

$$\text{AUDxPER} = 1/3520\text{Hz}/282\text{ns} = 1006$$

Der Quotient aus Systemtakt und Samplingrate führt schneller zum Ziel:

$$\text{AUDxPER} = 3,54\text{MHz}/3520\text{Hz} = 1005,68$$

Da das Periodenregister nur ganzzahlige Werte akzeptiert, ist man gezwungen, das Ergebnis zu runden. Obwohl dadurch zwangsläufig Abweichungen zur idealisierten Tonfrequenz (wohltemperiertes Klavier) entstehen, sind die Differenzen so gering, daß sie nicht ins Gewicht fallen.

Arbeitserleichterung – eine universelle Initialisierungsroutine

Die Initialisierung eines Tonkanals besteht aus einer Reihe gleichartiger Schritte, die häufig wiederkehren. Folgende Assembler-Routine erleichtert diesen Vorgang. Sie wird wie eine C-Funktion aufgerufen und erwartet eine Parameterübergabe nach dem Schema:

Play (Kanalnummer, Samplepuffer, Samplelänge, Ausgabeperiode, Lautstärke);

#asm

```
public _Play_Play:
lea $DFF0a0,a0          ;Basisadresse der Customchips
                        nach a0

move.l 4(a7),d0          ;Tabellenanfang des
asl.l #4,d0              ;Audiokanals
add.l d0,a0              ;ermitteln und in a0 speichern

move.l 8(a7),(a0)        ;Anfang der Datenliste nach
                        AUDxLCH

move.l 12(a7),d0         ;Bytelänge des Samples vom Stapel
                        holen

asr.l #1,d0              ;durch 2 teilen
move.w d0,4(a0)          ;und in das AUDxLEN-Register
                        schieben

move.l 16(a7),d0         ;Periodenwert vom Stapel holen
move.w d0,6(a0)          ;und in AUDxPER abspeichern

move.l 20(a7),d0         ;Lautstärke vom Stapel holen
move.w d0,8(a0)          ;ins AUDxVOL-Register schieben

move.w #$8000,d1         ;S/C-Bit für DMACON
```

```
    move.l 4(a7),d0          ;Kanalnummer nach d0
    bset d0,d1               ;und im entsprechend setzen
    move.w d1,$DFF096        ;DMA aktivieren
    rts                     ;Rücksprung zur aufrufenden
                           Routine

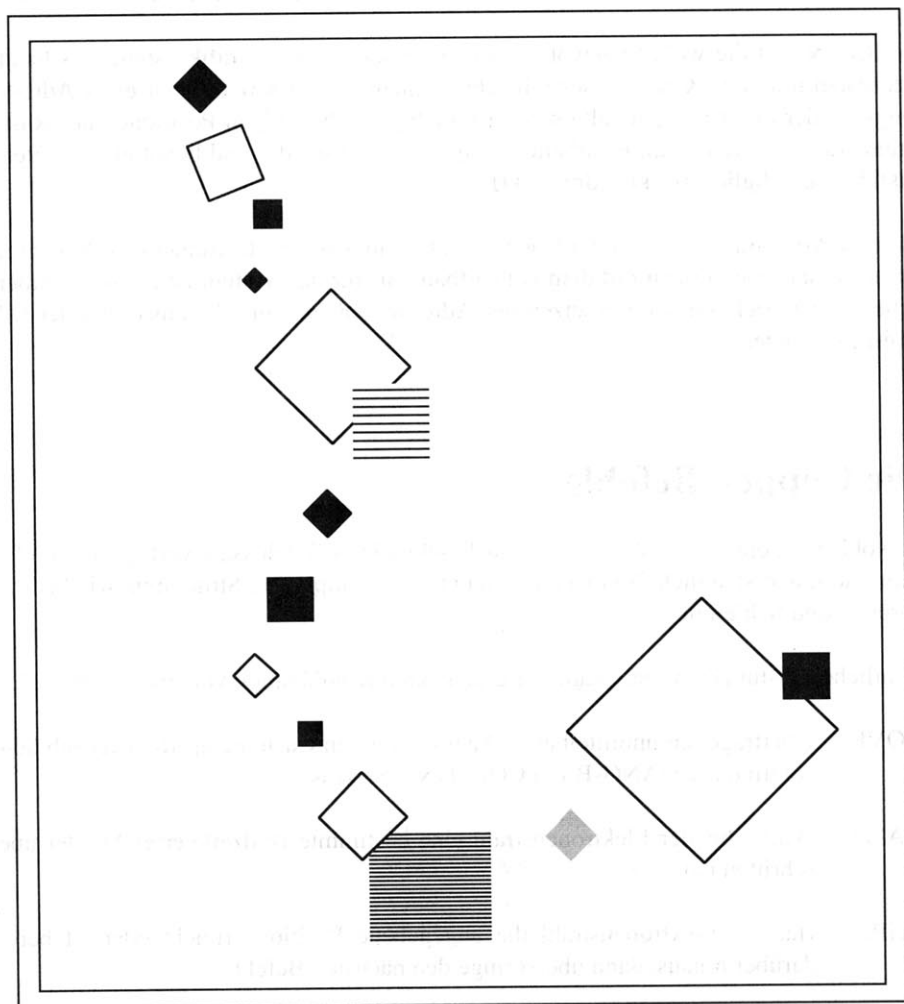
    public _Stop

_Stop:
    move.l 4(a7),d0          ;Kanalnummer vom Stapel holen
    moveq #0,d1              ;d1 löschen
    bset d0,d1               ;Kanalbit setzen
    move.w d1,$DFF096        ;DMA deaktivieren
    rts                     ;Rücksprung

#endasm
```


Kapitel 9

Die Grafik-Hardware



"Alles im Griff" – der Copper

Die vielleicht wichtigste Entwicklungsphilosophie des Amigas ist die ungewöhnliche Transparenz der Grafikdarstellung. Der Programmierer hat nicht nur die Wahl zwischen verschiedenen Auflösungen und Darstellungsmodi, sondern kann auch den Weg des Elektronenstrahls in allen Einzelheiten verfolgen. Die Änderung von Grafikparametern in Abstimmung mit dem Bildaufbau erlaubt Spezialeffekte, die sich auf anderen Systemen – wenn überhaupt – nur mit erheblichem Programmieraufwand realisieren lassen.

Der Copper ist die wohl ungewöhnlichste Komponente des Grafiksystems. Es handelt sich dabei um einen Coprozessor mit sehr kleinem Befehlssatz und nur einer Adressierungsart, der die Registerinhalte der Customchips an beliebigen Positionen des Rasterstrahls modifizieren kann. Er arbeitet weitgehend selbständig und benötigt zur Befehlsausführung lediglich zwei bis drei Takte.

Für den Programmierer stellt der Copper ein leistungsfähiges Instrument zur Synchronisation seiner Operationen mit dem Bildaufbau dar, für das System ist er ein unentbehrliches Hilfsmittel zum Zurücksetzen der Adreßzeiger und zur Aktualisierung der Bildschirmparameter.

Die Copper-Befehle

Obwohl der Copper nur über einen sehr beschränkten Befehlssatz verfügt, ist die Programmierung erstaunlich flexibel und erlaubt auch komplexere Strukturen, wie bedingte Sprünge und Schleifen.

Sämtliche Leistungen basieren auf der geschickten Kombination von nur drei Befehlen:

- MOVE:** Übertrage ein unmittelbares Datenwort in ein Customchip-Register (ab \$040, sofern das CDANG-Bit in COPCON gesetzt ist).
- WAIT:** Warte, bis der Elektronenstrahl eine bestimmte Bildzeile erreicht oder überschritten hat.
- SKIP:** Hat der Elektronenstrahl die angegebene Position erreicht oder ist bereits darüber hinaus, dann überspringe den nächsten Befehl.

Befehlsformat und Ausführungszeit

Copper-Befehle bestehen immer aus zwei 16-Bit-Worten, deren unterstes Bit (Nr. 0) als Befehlskennung dient. Diese Bits bilden einen 2-Bit-OpCode, der die Unterscheidung von vier verschiedenen Befehlen erlaubt, belegt sind aber nur die Bitmuster 00 (MOVE), 10 (WAIT) und 11 (SKIP).

Die Instruktionsworte werden vom DMA-Controller aus dem Chip-RAM-Speicher geholt und in das Befehlsregister COPINS geschrieben. Sie sind in einer Liste zusammengefaßt, der sogenannten Copper-Befehlsliste (oder kurz Copper-List). Um beide Worte einer Instruktions-Zeile zu lesen, benötigt der Controller zwei gerade Buszyklen oder vier Prozessortakte.

Die Copper-Hardware hat eine höhere System-Priorität als Blitter oder CPU, so daß Copper-DMA-Zugriffe immer Vorrang haben.

Durch RISC-ähnliche Architekturmerkmale wie einheitliche Befehlslänge und extrem kurze OpCodes wird die Interpretation und Ausführung der Copper-Befehle soweit verkürzt, daß der Befehlszyklus praktisch auf die Dauer des DMA-Zugriffs zum Holen der Instruktionsworte zusammenschmilzt.

MOVE und SKIP benötigen beide jeweils zwei Buszyklen, der WAIT-Befehl verbraucht einen zusätzlichen Zyklus bei Erfüllung seiner Wartebedingung. Effektiv ist der Befehlszyklus aber doppelt so lang, also vier bzw. sechs Zyklen, da die Copper-DMA an gerade Timeslots gebunden ist und damit nur jeden zweiten Speicherzyklus für den Befehlszugriff nutzen kann.

Das Befehlsformat im Überblick:

Bit	MOVE		WAIT		SKIP	
	1. Wort	2. Wort	1. Wort	2. Wort	1. Wort	2. Wort
15	x	D15	V7	BFD	V7	BFD
14	x	D14	V6	VM6	V6	VM6
13	x	D13	V5	VM5	V5	VM5
12	x	D12	V4	VM4	V4	VM4
11	x	D11	V3	VM3	V3	VM3
10	x	D10	V2	VM2	V2	VM2
9	x	D9	V1	VM1	V1	VM1
8	RA8	D8	V0	VM0	V0	VM0
7	RA7	D7	H8	HM8	H8	HM8

Bit	MOVE		WAIT		SKIP	
	1. Wort	2. Wort	1. Wort	2. Wort	1. Wort	2. Wort
6	RA6	D6	H7	HM7	H7	HM7
5	RA5	D5	H6	HM6	H6	HM6
4	RA4	D4	H5	HM5	H5	HM5
3	RA3	D3	H4	HM4	H4	HM4
2	RA2	D2	H3	HM3	H3	HM3
1	RA1	D1	H2	HM2	H2	HM2
0	0	D0	1	0	1	1

RA8..RA0: Tabellenoffset des adressierten Customchip-Registers

D15..D0: 16-Bit-Datenwort

V7..V0: vertikaler Wartewert

H8..H2: horizontale Warteposition

VM6..VM0: Maskenwert für vertikale Strahlposition

HM8..HM2: Maskenwert für horizontale Strahlposition

Der MOVE-Befehl

Pseudo-Syntax: MOVE #\$xxxx,CustomReg

Format:

Erstes Instruktions-Wort (1. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x	x	x	x	x	x	x	x	[	Registeradresse						]	I

Registeradresse: Tabellenoffset (Nummer) vom Zielregister der Customchip-Hardware.

I: Kennungsbit für den MOVE-Befehl (= 0).

Zweites Instruktions-Wort (2. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[16-Bit-Datenwort (unmittelbar)]															

MOVE ist, wie die Syntax schon nahelegt, der Datentransport-Befehl im Copper-Befehlssatz. Er schreibt ein unmittelbares 16-Bit-Datenwort in ein Register der Customchip-Hardware. Die Registeradresse liegt im ersten Befehlswort, das Datum wird im zweiten Befehlswort übergeben.

Nicht alle Customchip-Register sind für den Copper zugänglich. Der Zugriff ist normalerweise erst ab Adresse \$080 (COP1LCH) erlaubt. Im Copper-Danger-Mode (CDANG-Bit im COPCON-Register) können aber auch die Register des Blitters adressiert werden, ein Zugriff auf den Bereich von \$000 bis \$03F ist allerdings grundsätzlich tabu. (Was sollte der Copper auch mit den Leseadressen am Anfang der Registertabelle anfangen?)

Pseudo-Syntax: WAIT (x,y)

Format:

Erstes Instruktions-Wort (1. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[V7	V6	V5	V4	V3	V2	V1	V0]	[H8	H7	H6	H5	H4	H3	H2]	I

V7 bis V0: vertikale Strahlposition, auf die gewartet werden soll.

H8 bis H2: horizontale Strahlposition.

I: 1. Kennungsbit des WAIT-Befehls (= 1)

Zweites Instruktions-Wort (2. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFD	[VM6	VM5	VM4	VM3	VM2	VM1	VM0]	[HM8	HM7	HM6	HM5	HM4	HM3	HM2]	I

BFD: Blitter-Finished-Disable-Bit. 0 = warten auf Ende der Blitter-Operation / 1 = Registerzugriff ohne Beachtung des Blitter-Status.

VM6 bis VM0: Maske für die vertikale Rasterstrahlposition.

HM8 bis HM2: Maske für die horizontale Strahlposition.

I: 2. Kennungsbit des WAIT-Befehls (= 0)

WAIT macht genau das, was man von einem Befehl mit dieser Bezeichnung erwarten kann, er wartet... und zwar darauf, daß der Elektronenstrahl die im ersten Befehlswort festgelegte horizontale (Hx-Bits) bzw. vertikale (Vx) Rasterposition erreicht oder überschreitet.

Stößt der Copper in seiner Befehlsliste auf eine WAIT-Instruktion, geht er in eine Warteschleife und vergleicht die angegebene Position mit den Rasterstrahlzählern, deren aktueller Stand in den Registern VHPOSr und VPOSr abgegriffen werden kann. Ist der Bildaufbau bereits an dieser Koordinate angelangt bzw. darüber hinaus, verläßt der Coprozessor den WAIT-Modus und bearbeitet die nächste Instruktion in seiner Befehlsliste.

Die Abfragebedingung <Strahlposition größer oder gleich Positionsangabe> verhindert, daß sich der Copper bei der Interpretation von nicht sortierten Wartesequenzen im aktuellen Bildraster "aufhängt" – eine entscheidende Voraussetzung für die flüssige Befehlsverarbeitung bei Sprüngen innerhalb der Copper-Liste oder Wiedereintritt nach einem Rasterstrahl-Interrupt.

```
WAIT (0,100)      ;wartet auf Rasterzeile 100
```

```
WAIT (0,50)      ;da Bedingung bereits erfüllt, sofort weiter
```

Der große Vorteil des WAIT-Befehls liegt in der verdeckten Wartephase. Während der Zählerabfrage verläßt der Copper den Chip-RAM-Bus und belegt keine weiteren Speicherzyklen; erst, wenn die Bedingung erfüllt ist, kehrt er wieder zurück und setzt nach einem zusätzlichen Buszyklus die Befehlsausführung weiter fort.

Die H- und V-Bits der Wartekoordinate korrespondieren mit den entsprechenden Bits in den Rasterstrahlzählern (Wertebereich: 228/227 Buszyklen pro Zeile, 312/313 Rasterzeilen).

```
VHPOSR    $006      r
```

```
Bit :  15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
```

```
Label: V7 V6 V5 V4 V3 V2 V1 V0 H8 H7 H6 H5 H4 H3 H2H0
```

```
VPOSR     $004      r
```

```
Bit :  15    14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
```

```
Label: LOF  x  x  x  x  x  x  x  x  x  x  x  x  x  x  V8
```

Alle Angaben beziehen sich auf Buszyklen, die nach dem Timeslot-Allocation-Schema des DMA-Systems (siehe Kapitel 5) von \$00 bis \$E4 (= 228) durchnummeriert sind. Die maximale Ausdehnung der Bitplane-DMA reicht von \$18 bis \$D8. Die horizontale Austastlücke liegt zwischen \$F bis \$35 (wichtig für die verdeckte Änderung von Darstellungsparametern), die vertikale Austastlücke fällt in den Bereich von Zeile \$00 bis Zeile \$19.

Auffällig ist das Fehlen der Bits H1 und V8 im ersten Wort des WAIT-Befehls, wodurch die Erfassung der horizontalen Strahlposition eine minimale Schrittweite von zwei Buszyklen hat und die vertikale Position auf den Bereich von 0 bis 255 begrenzt ist.

Was ist der Grund für diese Einschränkungen?

Horizontal stößt die Auflösung an die Grenze der Reaktionsgeschwindigkeit des Coppers, die wiederum von der Länge des Befehlszyklus abhängt. Der WAIT-Befehl benötigt zwei gerade Speicherzyklen zum Holen der Instruktionsworte und einen zusätzlichen Zyklus bei Erfüllung der Wartebedingung, so daß Positionsangaben mit einer geringeren Schrittweite als zwei Buszyklen die Copper-Hardware schlicht überfordern.

Da das System pro Buszyklus zwei niedrig- oder vier hochauflösende Punkte ausgibt, beträgt die effektive Auflösung in der Horizontalen vier bzw. acht Pixels. Die Strahlposition läßt sich in Schritten von vier niedrigauflösenden Punkten angeben, die unteren Bits H1 und H0 werden vernachlässigt.

Das Fehlen des V8-Bits hat nichts mit mangelndem Auflösungsvermögen zu tun, sondern liegt einfach daran, daß es sich (als einziges Zähler-Bit) im VPOS-Register (Bit Nr. 0) befindet und von der Vergleichs-Logik nicht ausgewertet wird. Mit den Bits V7 bis V0 kann der Copper lediglich 255 Rasterzeilen erreichen. Ein Zählerstand größer 255 erzeugt einen Überlauf des 8-Bit-Wertebereichs, so daß der Copper nicht zwischen den ersten und letzten Zeilen des Bildrasters unterscheiden kann. Dennoch lassen sich auch die restlichen 56 Rasterzeilen in den Abfrageprozeß einbeziehen.

Die "unverdauliche" Koordinate muß lediglich auf zwei Wait-Befehle verteilt werden, wobei der erste auf die letzte Rasterzeile (\$ff= 255) des regulären Wertebereichs wartet und der zweite den Übertrag abdeckt.

Folgende Sequenz wartet auf Rasterzeile 285:

```
WAIT (0,255)      ;wartet bis zum Überlauf
WAIT (0,30)       ;warte bis zur vertikalen Bildschirmposition
                  255+30=285
```

Das zweite Instruktionswort des WAIT-Befehls enthält einen Maskenwert für die horizontale und vertikale Strahlposition. Dieser Wert legt fest, welche Binärstellen der Wartekoordinate mit den Zählerbits verglichen werden. Der Copper untersucht nur solche Bits, die auch in der Maske gesetzt sind, gelöschte Binärstellen werden ignoriert. Da im zweiten Instruktionswort auch noch ein Kontroll-Bit untergebracht werden mußte, verzichtete man auf das VM7-Bit. Deshalb läßt sich die vertikale Strahlposition lediglich in den Zeilen 0 bis 127 und 256 bis 312 maskieren.

Bei rein sequentieller Bearbeitung der Copper-Befehlsliste spielt der Maskierungsmechanismus nur eine untergeordnete Rolle. Die Maskenbits und das Blitter-Finished-

Disable-Bit liegen auf 1, das Kennungsbit ist gelöscht, so daß das zweite Instruktionswort normalerweise den Wert \$FFFE enthält.

Interessant wird die Sache erst bei der Konstruktion von Copper-Schleifen. Durch geeignete Wahl der Maskenbits kann man die WAIT-Bedingung auf ein Intervall eingrenzen oder so programmieren, daß sie in einem bestimmten Abstand (zyklisch) immer wieder erfüllt wird.

Sehen wir uns dazu zwei Beispiele an:

Im vertikalen Bereich von 64 bis 127 soll mit jeder Rasterzeile ein Interrupt ausgelöst werden. Da die unteren Bits V0 bis V5 die Wartebedingung immer erfüllen müssen, werden sie ausgeblendet. Lediglich V6 soll mit dem Zählerbit verglichen werden. Der korrespondierende Maskenwert ist demnach 1000000 (\$40), die vertikale Warteposition 01000000 (\$40):

```
dc.w $4001,$C000 ;warte auf Koordinate (0,64) mit Maskenwert $40
                    (BFD-Bit, Nr. 15 gesetzt)
```

Ein Wartezyklus, der alle zehn Rasterzeilen erfüllt wird, sieht so aus:

```
dc.w $0A01,$8A00 ;warte auf Koordinate (0,10) mit Maskenwert $0A
```

Der SKIP-Befehl

Pseudo-Syntax: SKIP (x,y)

Format: siehe WAIT

Die Skip-Instruktion ähnelt stark dem WAIT-Befehl. Das Befehlsformat ist bis auf das Kennungsbit (= 1) des zweiten Wortes identisch, und genau wie WAIT vergleicht der SKIP-Befehl die aktuelle Strahlposition mit den Angaben im Befehlswort.

Allerdings wartet er nicht, bis die Bedingung erfüllt ist, sondern trifft sofort eine von zwei möglichen Entscheidungen.

Ist die Position des Rasterstrahls größer oder gleich der spezifizierten Koordinate, wird der nächste Befehl der Copper-Liste übersprungen, ist sie kleiner, arbeitet der Copper normal weiter.

Das mag auf den ersten Blick nicht besonders aufregend erscheinen, erweitert aber den Programmierungsspielraum enorm. In Verbindung mit der Fähigkeit des Coppers, seinen Programmzähler über eine MOVE-Sequenz zu verändern (umbiegen des Adreßzeigers und anschließender Zugriff auf das Strobe-Register COPJMP1), kann man Schleifen konstruieren, die an einer bestimmten Rasterzeile beginnen und nach x Durchläufen an einer anderen Zeile wieder enden.

Aufbau einer Copper-Befehlsliste

Bei der Erstellung einer Copper-Liste sollte man beachten, daß der Copper nicht nur für spezielle Anwendungen konzipiert wurde, sondern im System eine Reihe essentieller Aufgaben erfüllt, wie beispielsweise das zyklische Zurücksetzen der Bitplane-Adreßzeiger. Diese grundlegenden Funktionen sind fester Bestandteil eines Copper-Programms und bilden den Kopf der Befehlsliste. Danach kann man eigene Ideen realisieren, weitere Listen konstruieren und anschließend alle Teile zu einem kompakten Copper-Programm zusammenfügen, in dem sämtliche Aufgaben innerhalb des Bildschirmrasters definiert sind.

Da der Copper damit in den wenigsten Fällen bis zum Ende des Bildaufbaus beschäftigt sein wird, muß es eine Möglichkeit geben, den Befehls-Hol-Mechanismus nach Bearbeitung des letzten Befehls "auf Eis" zu legen. Da hierfür kein spezieller Befehl vorgesehen ist, muß man eben in die "Trick-Kiste" greifen. So eignet sich ein WAIT-Aufruf mit unerreichbarer Strahlposition hervorragend als Endmarkierung einer Copper-Befehlsliste:

```
WAIT ($FE,$FF) bzw. dc.w $FFFF,$FFFE ;warte auf VP = 00, HP = 254
```

Zur Erinnerung: Der letzte Buszyklus einer langen PAL-Rasterzeile hat die Nummer \$E4, größere Werte in der Horizontalen sind nicht möglich.

Die Initialisierung der Copper-Hardware

Die Copper-Register:

```
COP1LCH  $080      w
```

(COPper first LoCation register High = Adreßregister der ersten Copper-Befehlsliste, obere drei Bits)

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

[18–16] = höherwertige drei Bits der 19-Bit-Speicheradresse

COP1LCL \$082 w

(COPper first LoCation register Low = Adreßregister der zweiten Copper-Befehlsliste, untere 16 Bits)

COP1LCH und COP1LCL ergänzen sich zu einem 19-Bit-Adreßregister, das die Chip-RAM-Position der ersten Copper-Liste enthält. Das Registerpaar stellt nicht den Programmzähler dar, sondern dient lediglich zur Pufferung der Speicheradresse. Um die Befehlsausführung zu starten, muß der Adreßzeiger korrekt initialisiert werden und anschließend ein Zugriff auf die Strobe-Adresse COPJMP1 (Offset: \$088) erfolgen.

Während des Strahlrücklaufs (vertikale Austastlücke) wird COP1LC(H/L) automatisch in den internen Programmzähler übertragen.

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

[niederwertiger Teil der 19-Bit-Speicheradresse]

COP2LCH \$084 w

(COPper second LoCation register High = Adreßzeiger für die zweite Copper-Befehlsliste, obere drei Bits)

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

[18–16] = höherwertige drei Bits der 19-Bit-Speicheradresse

COP2LCL \$086 w

(COPper second LoCation register Low = Adreßzeiger für die zweite Copper-Befehlsliste, untere 16 Bits)

COP2LCH und COP2LCL bilden wieder ein Adreßregister. Es enthält die Startadresse der zweiten Copper-Befehlsliste, die für den Aufbau einer Interlace-Darstellung benötigt wird, da jedes Halbbild eine eigene Copper-Liste verwendet.

COPCON \$02E w

(COPper CONTROL register = Copper-Kontrollregister)

Dieses Register enthält zur Zeit nur ein relevantes Bit (Nr. 01). Es erlaubt den Copper-Zugriff auf die Register der Blitter-Hardware (Offset \$040 bis \$07E), die normalerweise lediglich von DMA-Controller und CPU angesprochen werden. Nach einem Neustart ist das Bit gelöscht.

BIT 00 CDANG Copper-DANger-mode.

COPINS \$08C w

(COPper INStruction fetch identify = Befehlsregister des Coppers)

DMA-Pufferregister für die Befehle der Copper-Listen.

COPJMP1 \$088 s

(COPper JuMP register 1 = Strobe-Signal zur Initialisierung des Programmzählers auf die erste Copper-Befehlsliste COP1LC(H/L)).

Ein Zugriff auf die Adresse \$DFF088 initialisiert den Programmzähler der Copper-Hardware auf den Beginn (COP1LC[H/L]) der ersten Befehlsliste im Speicher.

Der Copper kann dieses Register selbst beschreiben, so daß sich auch weite Sprünge in der Copperliste realisieren lassen.

COPJMP2 \$08A s

(COPper JuMP register 2 = Strobe-Signal zur Initialisierung des Programmzählers auf die zweite Copper-Liste COP2LC(H+L)).

Bitbelegung und Funktion (für die zweite Befehlsliste) wie COPJMP1.

Wie die CPU besitzt auch der Copper einen Programmzähler, der immer auf den aktuellen Befehl im Speicher zeigt. Genaugenommen sind dieser Programmzähler und das dazugehörige Adreß-Unit keine eigenständigen Einheiten des Coppers, sondern Teile des zentralen Adreßgenerators von FAT AGNUS.

Zur Initialisierung des Programmzählers muß eines der beiden Adreßregister (normalerweise ist das COP1LC(H/L)) auf den Beginn der Copper-Liste zeigen und an-

schließlich ein Zugriff auf die zugehörige Strobe-Adresse (COPJMP1) erfolgen. Das erscheint zwar etwas umständlich, bietet aber entscheidende Vorteile.

Der Programmierer kann die Adreßzeiger im laufenden Betrieb umbiegen, ohne den Copper gleich aus dem aktuellen Befehlszyklus zu reißen; die Verzweigung erfolgt erst mit der Reinitialisierung zu Beginn des nächsten Bildrasters.

Das lästige Zurücksetzen der Adreßzeiger entfällt, da die Register gepuffert sind und nicht vom Adreßgenerator verändert werden.

Und schließlich ist die Konstruktion von Schleifen bei der begrenzten Sprungweite der SKIP-Instruktion ohne die indirekte Zugriffstechnik undenkbar. Zum Umbiegen der 19-Bit-Sprungadresse benötigt der Copper zwei MOVE-Sequenzen, SKIP kann aber nur einen Befehl überspringen. Beim indirekten Zugriff auf den Programmzähler werden die Adreßpuffer bereits vor Beginn der Schleife initialisiert. Zum Auslösen des Sprungs bedarf es lediglich eines MOVE-Befehl (schreibt in das zugehörige Strobe-Register (COPJMPx)), der von der SKIP-Instruktion problemlos übersprungen werden kann.

Sehen wir uns eine typische Initialisierungssequenz für die Copper-DMA an:

```

COP1LCH = $080
COP1LCL = $082
COPJMP1 = $088
DMACON = $096
lea $DFF000,a5          ;Basisadresse nach a5
move CopperList1,COP1LCH(a5) ;höherwertiger Teil der Adresse
move CopperList2,COP1LCL(a5) ;niederwertiger Adreßteil
move #$0000,COPJMP1      ;Initialisierung des Programmzählers
                           (Zugriff beliebig)
move #$8180,DMACON       ;Setzen von COPEN und DMA-Hauptschal-
                           ter zur Aktivierung der Copper-DMA!
```

Die Copper-Programmierung

Der Copper besitzt Fähigkeiten, die über das sequentielle Bearbeiten von MOVE-Sequenzen weit hinausgehen.

Der folgende Abschnitt gibt deshalb ein paar Anregungen für die "höhere" Copper-Programmierung. Sie sollen einen kleinen Eindruck davon geben, welche phantastischen Möglichkeiten in der Ausnutzung dieser Fähigkeiten liegen.

Programmbeispiel 1: Schleifen in der Copper-Befehlsliste

Sehen wir uns einmal die Schleifenkonstruktion an folgendem Beispiel an. Die Routine löst zwischen Rasterzeile 100 und 150 mit jeder neuen Zeile einen Rasterstrahl-Interrupt (Zugriff auf INTREQ) aus:

```

CopperList:
ständige Initialisierungsaufgaben      ;Kopf der Copper-Liste

WAIT(0,100)                             ;warte auf Rasterzeile 100
MOVE #(CopperList+Loop).high,COP1LCH    ;biegt die Adreßpuffer auf
                                         ;den Beginn
MOVE #(CopperList+Loop).low,COP1LCL      ;des Schleifenkopfes um
Loop:                                     ;Sprungmarke (bei n Befehlen
                                         ;ist der Abstand zum Listen-
                                         ;kopf n * 4 Byte)

MOVE #$8010,INTREQ                       ;Rasterstrahl-Interrupt
WAIT (0,x)                               ;warte bis zur nächsten
                                         ;Rasterzeile, x = Anzahl der
                                         ;Schleifendurchläufe

SKIP (0,150)                             ;überspringe nächsten
                                         ;Befehl, wenn Strahlposition
                                         ;größer als 150; sonst
                                         ;normal weiter

MOVE #$0000,COPJMP1                     ;springe zum Schleifenkopf
MOVE #CopperList.high,COP1LCH            ;Adreßpuffer auf Beginn der
MOVE #CopperList.low,COP1LCL             ;Copper-Liste zurücksetzen
WAIT (254,255)                           ;Endmarkierung. Warte auf
                                         ;Rasterende!

```

Nehmen wir an, die Copper-Liste steht an der Adresse \$050000 und die Schleife beginnt mit dem 12. Befehl (Rasterstrahl-Interrupt), dann beträgt ihr relativer Abstand zum Kopf $4 * 11 = 44$ Byte. Sie befindet sich also an der absoluten Adreßposition \$050044. Mit diesem Wissen läßt sich die Makro-Darstellung in ein entsprechendes Deklarationsfeld für den Assembler umsetzen:

```
CopperList:
dc.w $6401,$ffff           ;WAIT (0,100)
dc.w $0080,$0005           ;MOVE #(CopperList+Loop).high,COP1LCH
dc.w $0082,$002c           ;MOVE #(CopperList+Loop).low,COP1LCL
                           ;Schleifenkopf (Loop)
```


Im Prinzip läßt sich diese Umschalttechnik beliebig erweitern. So kann der Copper ein komplexes Feld verschiedener Copper-Listen zyklisch durchlaufen. Durch bestimmte Eingriffe von außen (CPU) sind sogar unterschiedliche Pfade denkbar, auf denen der Copper gezielt geführt wird.

Programmbeispiel 3: Color-Cycling mit dem Copper

Eine potentielle Anwendung multipler Copper-Listen ist das Color-Cycling, ein Animationseffekt, der auf einer zyklischen Rotation der Farbpalette beruht. Im folgenden Beispiel "hangelt" sich der Copper durch drei verschiedene Befehlslisten, in denen die unteren drei Farbreister jeweils mit Farbwert des Vorgängers beschrieben werden:

CopperList0:

Listenkopf für die üblichen Initialisierungsarbeiten

```

.      .      .
;Beginn des anwendungsspezi-
;fischen Teils
MOVE #$0000,COLOR00      ;Farbreister initialisieren!
MOVE #$0fff,COLOR01
MOVE #$0777,COLOR02
MOVE #CopperList1.high,COP1LCH ;Adresse der nächsten Copper-
;Liste in den
MOVE #CopperList1.low,COP1LCL ;Adreßpuffer COP1LC(H/L) schieben
WAIT (254,255)             ;Warten bis Rasterende. Zu Beginn
;der vertikalen Austastlücke
;springt der Copper automatisch
;in die nächste Liste

```

CopperList1:

Listenkopf für die üblichen Initialisierungsarbeiten

```

.      .      .
;Beginn des anwendungsspezi-
;fischen Teils
MOVE #$0777,COLOR00      ;Farbreister initialisieren
MOVE #$0000,COLOR01      ;(Rotation nach rechts!)
MOVE #$0fff,COLOR02
MOVE #CopperList2.high,COP1LCH ;Adresse der nächsten Copper-
;Liste in den
MOVE #CopperList2.low,COP1LCL ;Adreßpuffer COP1LC(H/L) schieben
WAIT (254,255)           ;Warten bis zum Rasterende,
;danach Sprung zu CopperList 2

```

```

CopperList2:
Listenkopf für die üblichen Initialisierungsarbeiten
. . . . .
                                ;Beginn des anwendungsspezi-
                                ;fischen Teils
MOVE #$0fff,COLOR00             ;Farbregister initialisieren
MOVE #$0777,COLOR01             ;(Rotation nach rechts!)
MOVE #$0000,COLOR02
MOVE #CopperList0.high,COP1LCH  ;Adresse der ersten Copper-Liste
                                ;in den
MOVE #CopperList0.low,COP1LCL   ;Adreßpuffer COP1LC(H/L) schieben
WAIT (254,255)                  ;Warten bis Rasterende,
                                ;anschließend Rücksprung
                                ;in die erste Copper-Liste

```

Modifikation der Copper-Liste

Da häufig auch die Farbregister vom Copper gesetzt und damit alle 50stel Sekunde re-initialisiert werden, läßt sich die Farbpalette nicht mehr durch einen direkten Zugriff auf die entsprechenden Register modifizieren. Natürlich kann man eine neue Copper-Liste erstellen und anstatt der alten installieren, für kleinere Vorhaben (beispielsweise die Neubelegung eines einzigen Farbregisters) stellt dieser Aufwand allerdings kaum eine befriedigende Lösung dar.

Sofern Struktur und Position der Copper-Befehlsliste bekannt sind, erspart die Veränderung der laufenden Copper-Liste eine Menge Arbeit. Dabei gilt es lediglich zu beachten, daß der Copper nicht gerade in dem Augenblick auf das eine Befehlswort zugreift, während das andere von der CPU überschrieben wird. Da der Copper aber außerordentlich zügig arbeitet, kann man bereits am Ende der vertikalen Austastlücke (je nach Länge der Copper-Liste) die ersten Befehle mit "gutem Gewissen" überschreiben. Zur Ermittlung dieses Zeitpunktes bietet sich die WaitTOF-Funktion der Graphics-Library an. Nach dem Aufruf kehrt sie erst dann zurück, wenn sämtliche Interrupt-Server des Vertical-Blanking-Interrupts bearbeitet sind.

Ein Beispiel:

Der zehnte Befehl der laufenden Copper-Liste lädt das Farbregister COLOR00 mit dem Farbwert \$0000 (schwarz). Um den Hintergrund auf weiß (\$0FFF) zu setzen, wird das 21. Wort bzw. 42. Byte (= 2 * 21) der Befehlsliste (zweites Befehlswort des MOVE-Befehls) mit dem Wert \$0FFF überschrieben:

```
lea _CopperList,a5 ;Basisadresse der Copper-Liste nach a5
move.w #$0fff,42(a5) ;Änderung des unmittelbaren Datenwertes auf
                      $0fff
```

Rasterstrahl-Interrupts

Rasterstrahl-Interrupts, auf vielen Systemen nur mit den ausgefallensten Tricks zu realisieren und Ausdruck höchster Programmierkunst, erfordern beim Amiga keine großen Klimmzüge.

Da der Copper auf sämtliche Register oberhalb des Tabellenoffsets \$080 uneingeschränkt zugreifen kann, ist er natürlich auch in der Lage, jeden beliebigen System-Interrupt durch Setzen des entsprechenden Bits im INTREQ-Register (Offset 09c) zu simulieren. Bit 4 ist sogar speziell für Copper-Interrupts reserviert.

Folgende Sequenz löst eine Unterbrechung zu Beginn der Rasterzeile 100 aus:

```
WAIT (0,100)
MOVE #$8010,INTREQ ;Set-Bit (Nr. 15) und Copper-Bit (Nr. 4)
                  setzen!
```

Rasterstrahl-Interrupts sind ein leistungsfähiges Instrument zur Synchronisation des Prozessors mit dem Videobild. Wenn die Komplexität der Operation das Eingreifen der CPU erfordert, stellen sie eine sinnvolle Ergänzung zum Copper-Befehlssatz dar.

Synchronisation zwischen Copper und Blitter

Setzt man das CDANG-Bit im COPCON-Register, hat der Copper Zugriff auf die Register des Blitters. Um den Blitter zu steuern, muß die Copper-Hardware wissen, wann eine Blitter-Operation beendet ist – eine Veränderung der Registerinhalte bei laufendem Blitter wäre fatal. Deshalb wartet der WAIT-Befehl normalerweise bis zum Ende der Blitter-Operation und untersucht erst dann die aktuelle Rasterstrahl-Position. Den Blitter-Status ermittelt WAIT durch Abfrage des BBUSY-Bit im DMACONR-Register (Offset \$002, 1 = Blitter aktiv, 0 = inaktiv).

Da sich dieses Verhalten im normalen Copper-Betrieb allerdings äußerst störend auswirkt, kann man die Sicherheitsperre durch Setzen des BFD-Bits (Blitter Finish Disable, Bit 15) im zweiten Instruktionswort des WAIT-Befehls gezielt abschalten. Wie der

Name schon sagt, verhindert das Bit die Einbeziehung des Blitter-Finish-Signals in die Wartebedingung der Wait-Instruktion. In den meisten Fällen wird man also Bit 15 des zweiten Befehlswortes löschen.

Sprites

"Verspielte Elfe oder Fee" lautet die Übersetzung von "Sprite", wie man sie in einem Wörterbuch finden kann. Der Begriff ist zu einem festen Bestandteil des Computervokabulars geworden (dessen Neigung zur bildhaften Beschreibung hinlänglich bekannt ist) und bezeichnet kleine, animierte Grafikobjekte.

Hardwareunterstützte Sprites gehören bei Home-Computern zum guten Ton, im Profisektor sind sie eher eine Seltenheit und müssen für Spiele aufwendig simuliert werden; beim Amiga bilden sie zusammen mit den Playfields die Grundlage des Grafiksystems.

Die Video-Hardware besitzt acht Sprite-Generatoren, die unabhängig von der aktuellen Bildschirmauflösung arbeiten und über eigene DMA-Kanäle verfügen. Amiga-Sprites sind bis zu 16 Punkte breit, unbegrenzt hoch und besitzen drei verschiedene Farben plus Hintergrund-Transparenz. Wahlweise lassen sich auch zwei Generatoren zur Darstellung eines 16-farbigem Multi-Color-Sprites kombinieren.

Aufbau der Datenliste

Farbselektion

Sämtliche Daten über Form, Farbauswahl und Größe eines Sprites sind in einer Datenliste zusammengefaßt, die sich im Prinzip als kleines Playfield mit einer Tiefe von zwei Bitplanes auffassen läßt. Allerdings liegen die Planes eines Playfields getrennt im Speicher, während die Dateninformationen in der der Spritelisten zeilenweise verzahnt sind.

Bei dreifarbigem Darstellung und einer Breite von 16 Pixeln benötigt die Definition einer Sprite-Zeile zwei Worte. Das höchstwertige Bit eines Wortes stellt den äußerst linken Rand, das unterste Bit den rechten Rand des Sprite-Rasters dar.

Die korrespondierenden Bits des höherwertigen Wortes (an der geraden Adresse) und des niederwertigen 16-Bit-Wertes (an der ungeraden Adresse) ergänzen sich jeweils zu

16 2-Bit-Registervektoren. Die Farbselektion läuft wieder nach dem bekannten Color-Indirection-Schema. Der Farbvektor eines Sprite-Pixels gibt die Nummer des zugehörigen Farbregister an, das den 12-Bit-RGB-Wert für die D/A-Wandler der Video-Hybrid enthält.

Um Playfield- und Sprite-Farben voneinander abzukoppeln, ist für letztere die obere Hälfte des Color-Lookup-Table von Farbregister 16 bis 31 reserviert, so daß beide Systeme erst bei mehr als 16-farbiger (4 Bitplanes) Bildschirmdarstellung kollidieren. Während die Farbwahl des Display-Fensters (unabhängig von der Aktivierung der Sprite-Hardware) keinen Einschränkungen unterliegt, ist die Registerzuweisung der Sprites verbindlich, da das System die Farbvektoren der Sprite-Pixel automatisch auf die Register 16 bis 31 umbiegt.

Leider resultiert daraus eine Limitierung der Farbwahl, schließlich stehen lediglich 16 Farbregister für acht dreifarbige Sprites zur Verfügung. Aus diesem Grunde teilen sich immer ein Sprite mit gerader Nummer und der Nachfolger mit ungerader Ordnungszahl vier Farbregister.

Die Registerverteilung für die acht Hardware-Sprites:

<i>Spritenummer</i>	<i>zugehöriges Farbregister</i>	<i>Adresse</i>
0 und 1	COLOR16 (transparent)	\$DFF1A0
	COLOR17	\$DFF1A2
	COLOR18	\$DFF1A4
	COLOR19	\$DFF1A6
2 und 3	COLOR20 (transparent)	\$DFF1A8
	COLOR21	\$DFF1AA
	COLOR22	\$DFF1AC
	COLOR23	\$DFF1AE
4 und 5	COLOR24 (transparent)	\$DFF1B0
	COLOR25	\$DFF1B2
	COLOR26	\$DFF1B4
	COLOR27	\$DFF1B6
6 und 7	COLOR28 (transparent)	\$DFF1B8
	COLOR29	\$DFF1BA
	COLOR30	\$DFF1BC
	COLOR31	\$DFF1BE

Da die Gitterstruktur des Sprite-Rasters nicht sichtbar sein soll, wird der Farbvektor 00 immer durchsichtig dargestellt, unabhängig vom Inhalt des korrespondierenden Farbre-gisters.

Multi-Color-Sprites

Für manche Anwendungen sind drei Farben und Hintergrund-Transparenz zu wenig, die Ergebnisse unbefriedigend. Abhilfe schafft der Attached-Modus, bei dem beide Ele-mente des Sprite-Paares zu einem 16-farbigen Multi-Color-Sprite mit einer Vektortiefe von vier Bits kombiniert werden. Die Zeilen-Worte des geraden Sprites (z.B. Nummer 0) bilden dabei die Binärstellen 0 und 1 der Registernummer, die korrespondierenden Zeilenwerte des ungeraden Sprites (beispielsweise Nummer 1) ergänzen die höherwer-tigen Bitpositionen 2 und 3. Voraussetzung für eine korrekte Darstellung ist allerdings die exakte Überlagerung der Spritepositionen; ist dies nicht der Fall, ignoriert das Sy-tem den Attached-Modus und bleibt in der normalen dreifarbigen Betriebsart.

Folgendes Beispiel zeigt die ersten Zeilen eines Multi-Color-Sprites und die Interpreta-tion der Datenlisten (die Kontrollworte zu Beginn der Datenliste sind nicht aufgeführt):

Datenliste von Sprite 0

Farbvektoren der Pixel (von links)

Adresse	Zeile	Datenwort	1. Zeile		2. Zeile	
			binär	dezimal	binär	dezimal
n	1	0000000000000000	0110	6	1010	10
n+1	1	1111111111111111	0010	2	1010	10
n+2	2	0000000000000000	0110	6	1010	10
n+3	2	1111111111111111	0010	2	1010	10
<i>Datenliste von Sprite 1</i>			0110	6	1010	10
			0010	2	1010	10
			0110	6	1010	10
Adresse	Zeile	Datenwort	0010	2	1010	10
			0110	6	1110	14
			0010	2	1110	14
			0110	6	1110	14
			0010	2	1110	14
			0110	6	1110	14
			0010	2	1110	14
			0110	6	1110	14
a	1	1010101010101010	0010	2	1110	14
a+1	1	0000000000000000	0110	6	1110	14
a+2	2	0000000011111111	0010	2	1110	14
a+3	2	1111111111111111	0110	6	1110	14
			0010	2	1110	14
			0110	6	1110	14
			0110	2	1110	14

Definition der Bildschirmposition

Die Datenliste eines Sprites beschreibt nicht nur Form und Farbwahl, sondern auch die horizontale (VSTART) bzw. vertikale (VSTOP) Bildschirmposition. Diese Informationen sind in zwei Kontrollworten am Anfang der Liste enthalten.

Vor Beginn der Sprite-Einblendung liest der DMA-Controller die ersten beiden Worte des Listenkopfes und überträgt sie in die Register SPRxCTL bzw. SPRxPOS des Video-Chips. Die Register enthalten neben der XY-Koordinate auch die letzte Zeile der Datenliste, die wieder aus zwei Kontrollworten besteht. Der DMA-Controller ermittelt die Anzahl der erforderlichen Speicherzyklen aus der Differenz zwischen den Werten VSTOP und VSTART. Trifft er in der letzten Zeile auf zwei Kontrollworte mit dem Wert Null, wird der DMA-Kanal für den Rest des Bildschirm-Rasters abgeschaltet.

Bitbelegung von SPRxCTL:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EV7	EV6	EV5	EV4	EV3	EV2	EV1	EV0	ATT	x	x	x	x	SV8	EV8	SH0

EV7 bis EV0: untere acht Bits der vertikalen Stopposition

ATT: Attach-Mode zur Darstellung eines 16 farbigen Sprites

SV8: 9. Bit der vertikalen Startposition

EV8: 9. Bit der vertikalen Stopposition

SH0: niederwertiges Bit des horizontalen Startwertes

Bitbelegung von SPRxPOS:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SV7	SV6	SV5	SV4	SV3	SV2	SV1	SV0	SH8	SH7	SH6	SH5	SH4	SH3	SH2	SH1

SV7 bis SV0: untere acht Bits der vertikalen Startposition

SH8 bis SH1: obere acht Bits der horizontalen Startposition

Für die horizontale und vertikale Bildschirmposition des Sprites steht also jeweils ein 9-Bit-Wert zur Verfügung. Die Angaben beziehen sich allerdings nicht direkt auf den sichtbaren Darstellungsbereich des Display-Fensters, sondern beschreiben den Abstand zur Ursprungsordinate (0,0) im DIWSRT-Register, die den äußerst linken Rand der Strahlabtastung markiert. Als Bezugspunkt am Sprite-Raster dient die linke obere Ecke (also Bit 15 der ersten Zeile).

Die Auflösung ist unabhängig vom aktuellen Darstellungs-Modus, sie beträgt horizontal einen niedrigauflösenden Pixel (siehe DIWSRT) und vertikal eine Non-Interlace-Rasterzeile. Bei der Definition der Bildschirmkoordinate muß der Programmierer selbst darauf achten, daß das Sprite innerhalb des sichtbaren Darstellungsbereiches erscheint.

Für das Standard-PAL-Fenster in der niedrigen Auflösung liegen die garantiert sichtbaren Sprite-Positionen horizontal zwischen 129 und 448 (abzüglich der Spritebreite) und vertikal im Intervall von 41 bis 296 (minus der effektiven Spritehöhe).

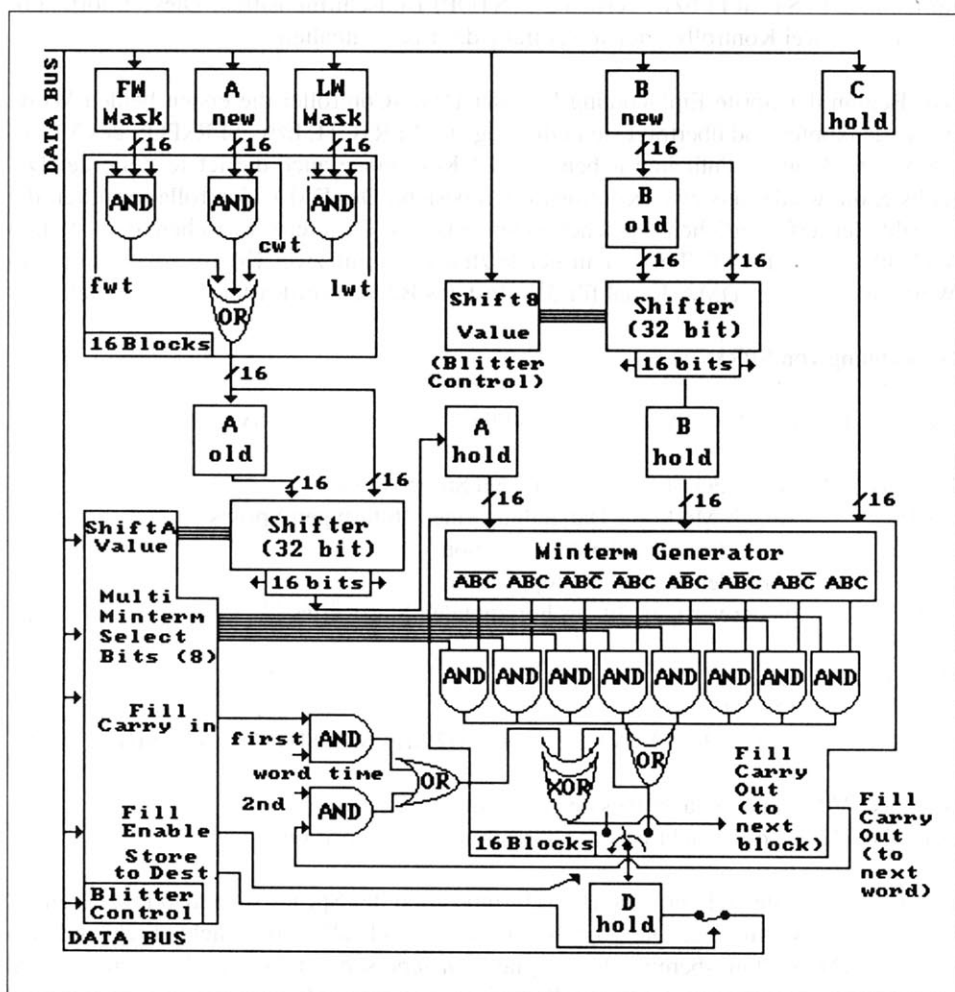


Abb. 9.1: Aufbau und Struktur der Sprite-Datenliste

Nicht immer ist es ungewollt, wenn ein Sprite über das Bildschirmfenster hinausragt und nur noch teilweise sichtbar ist. Viele Animationssequenzen machen sich diesen Clipping-Effekt zunutze.

So kann ein Sprite beispielsweise von einer imaginären Wand reflektiert und in das Bildschirmfenster zurückgeworfen werden; wandert das Objekt aus einem Bildschirmbereich hinaus und tritt am gegenüberliegenden Rand wieder ein, scheint es sich auf einer Art Umlaufbahn zu bewegen.

Zum Abschluß dieses Abschnitts noch ein praktisches Beispiel zum Aufbau einer Sprite-Datenliste. Es soll ein quadratisches Sprite mit transparentem Zentrum in der Mitte eines niedrigauflösenden Bildschirms dargestellt werden. Die Datenliste als Assemblerdefinition:

```
dc.w %0101011110000001,%01010011000011001 ;Position (281,165),
                                         VSTOP = 175
dc.w %00001111111110000,%00000000000000000 ;1. Datenzeile
dc.w %00001111111110000,%00000000000000000 ;2. Datenzeile
dc.w %00001111111110000,%00000000000000000 ;3. Datenzeile
dc.w %0000111001110000,%00000000000000000 ;4. Datenzeile
                                         (Fenster-Beginn!)
dc.w %0000111001110000,%00000000000000000 ;5. Datenzeile
                                         (Fenster-Ende)
dc.w %00001111111110000,%00000000000000000 ;6. Datenzeile
dc.w %00001111111110000,%00000000000000000 ;7. Datenzeile
dc.w %00001111111110000,%00000000000000000 ;8. Datenzeile
dc.w $0000,$0000 ; Ende des DMA-Zyklus!
```

Die Programmierung der Sprite-Hardware

Die Register

SPRxDATA \$144+x*\$008 w

(SPRite x image DATa register A = DMA-Datenregister A für Sprite x)

SPRxDATA enthält zusammen mit SPRxDATB die Bildinformation eines Sprites. Ein Zugriff auf das Register aktiviert die Sprite-Darstellung des DENISE-Chips, der daraufhin die horizontale Position des Rasterstrahls mit der Spaltenangabe im SPRxPOS-

Register (SHx-Bits) vergleicht. Stimmen beide Werte überein, werden die Spritedaten entsprechend ihrer Priorität in die Videoausgabe gemischt.

SPRxDATB $\$146+x*\008 w

(SPRite x image DATA register B = DMA-Datenregister B für Sprite x)

Zweiter 16-Bit-Datenpuffer für die Sprite-Darstellung (siehe SPRxDATA).

SPRxCTL $\$142+x*\008 w

(SPRite x vertical stop position and ConTroL data = Kontrollregister für vertikale Stopposition und Darstellungsparameter von Sprite x)

Bitbelegung:

Nr.	Label	Funktion
15–08	EVx	Untere acht Bits des vertikalen Stoppwertes; wird vom DMA-Controller ständig mit der aktuellen Rasterzeile verglichen, um bei Gleichstand die Sprite-Darstellung abzuschalten.
07	ATT	Attach-Bit (gilt nur für Sprites mit ungeraden Nummern). Ist es gesetzt, so werden die ungeraden und geraden Sprites zu Sprite-Paaren mit 16-farbiger Darstellung kombiniert.
06–03	–	nicht verwendet.
02	SV8	9. Bit des vertikalen Startwerts. Die übrigen acht Bits befinden sich im SPRxPOS-Register.
01	EV8	9. Bit des vertikalen Stoppwerts.
00	SH0	Unterstes Bit des horizontalen Startwerts. Die höherwertigen acht Bits liegen im SPRxPOS-Register.

SPRxPOS $\$140+x*\008 w

(SPRite x vertical and horizontal start POSition data = Bildschirmposition von Sprite x [0...7])

Zweites Kontrollwort für die Bildschirmposition von Sprite x (x = 0–7).

Bitbelegung:

Nr.	Label	Funktion
15–08	SVx	Vertikaler Startwert (untere acht Bits) der Sprite-Darstellung. Das 9. Bit steht in SPRxCTL-Register.

Bitbelegung:

Nr.	Label	Funktion
07–00	SHx	Horizontaler Startwert (obere acht Bits) der Sprite-Darstellung. Das untere 0. Bit liegt in SPRxCTL.

SPRxPTH \$120+x*\$004w

(SPRite x PoinTer High = Adreßzeiger auf Datenliste von Sprite 0 bis 7, obere drei Bits)

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x	x	x	x	x	x	x	x	x	x	x	x	x	x			

[18–16] = höherwertige drei Bits der 19-Bit-Speicheradresse

SPRxPTL \$122+x*\$004w

(SPRite x PoinTer Low = Adreßregister (untere 16 Bits) für die Speicherposition der Datenliste von Sprite 0 bis 7)

Zusammen mit SPRxPTH enthält SPRxPTL die 19-Bit-Adresse der Sprite-Daten im Chip-RAM-Speicher. Da die Adreßzeiger vom DMA-Controller laufend erhöht werden, müssen sie in jeder vertikalen Austastlücke neu initialisiert werden (durch Prozessor oder Copper).

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

[niederwertiger Teil der 19-Bit-Speicheradresse]

Initialisierung

Zur Darstellung des Grafikobjekts wählt man einen der acht Sprite-Kanäle und schreibt die Speicherposition der gewünschten Datenliste in die Adreßzeiger SPRxPT(H/L). Wie gewohnt geschieht dies durch einen 32-Bit-Zugriff auf den höherwertigen Teil des Adreßregisters:

```
move.l spritelist,SPR0PTH ;Initialisierung von Sprite-Generator 0
```

Da für die Sprite-Hardware lediglich ein Bit im DMACON-Register (Nr. 5, SPREN) vorgesehen ist, wirkt die DMA-Aktivierung immer auf alle Kanäle gleichzeitig, so daß

auch die nicht unmittelbar verwendeten Generatoren initialisiert werden müssen. Das klingt aufwendiger als es ist. Während die aktiven Sprite-Kanäle gültige Datenlisten bearbeiten, wird der Rest durch eine Beschäftigungstherapie mitgezogen.

Dazu bildet man eine leere Dummy-Datenliste, die nur die beiden Kontrollworte zum Abschalten des DMA-Controllers enthält (Wert: Null) und lädt die "brachliegenden" Adreßregister mit ihrer Speicherposition:

```
dummy:
dc.w $0000,$0000
move.l spritelist,SPR0TH    ;Sprite 0 normal initialisieren
move.l dummy,SPR1TH        ;alle anderen Kanäle zeigen auf die
                             Dummy-Liste
move.l dummy,SPR2TH
      .      .
      .      .
move.l dummy,SPR7TH
```

Wie die Bitplane-Zeiger sind auch die Adreßregister der Sprite-Kanäle ungepuffert und werden mit jedem Speicherzyklus inkrementiert. Deshalb müssen sie nach Ablauf der Darstellungssequenz zurückgesetzt werden, eine Aufgabe, die gewöhnlich dem Copper zufällt. Auch hier sollte man daran denken, alle Kanäle zu aktualisieren; unabhängig davon, ob sie gültige oder leere Datenlisten bearbeiten.

Eine derartige Sequenz sieht in der Makro-Darstellung etwa so aus:

```
MOVE #spritelist.HIGH,SPR0PTH
MOVE #spritelist.LOW,SPR0PTL
MOVE #dummy.HIGH,SPR1PTH
MOVE #dummy.LOW,SPR1PTL
      .      .
      .      .
MOVE #dummy.HIGH,SPR7PTH
MOVE #dummy.LOW,SPR7PTL
```

Sind alle Vorbereitungen abgeschlossen, die Adreßzeiger korrekt initialisiert und die Copper-Liste für die Zurücksetzung erstellt, wird die Sprite-DMA durch den Befehl `move.w #$83A0,$DFF09a` aktiviert.

Animation

Die Bewegung eines Sprites erreicht man durch Änderung der Positionsangabe im Listenkopf, der Rest bleibt dem Video-Controller überlassen; dabei muß allerdings das Timing zwischen DMA- und CPU-Zugriff stimmen.

Überschreibt der Prozessor die Kontrollworte, während der DMA-Controller gerade auf diese zugreifen will, kann es zu "merkwürdigen" Ergebnissen kommen, da die Positions-Bits recht eigenwillig über die Register verteilt sind und eventuell nicht mehr zueinander passen. Ein unsynchronisierter Zugriff kommt also nicht in Frage, unproblematisch dagegen ist die Modifikation des Listenkopfes innerhalb des Vertical-Blanking-Interrupts.

Es bietet sich an, die Positionen sämtlicher aktiver Sprites in einer Tabelle zusammenzufassen und sie im Verlauf der vertikalen Austastlücke in die Kontrollworte der korrespondierenden Datenlisten zu übertragen. Die Aktualisierung der Positionstabelle erfolgt dann im normalen Programmfluß.

Der Vertical-Blanking-Interrupt ist bereits vom Betriebssystem "beschlagnahmt", so daß die Interrupt-Routine als Server installiert werden muß.

Da der Copper in der Austastlücke gerade mit der Abarbeitung seiner Befehlsliste beginnt, kann es passieren, daß die Adreßzeiger SPRxPT(H/L) noch nicht initialisiert sind, wenn unsere Routine die Datenlisten anspricht. Deshalb greift man bei der Adressierung besser auf gepufferte Werte zurück oder wartet bis zum Ende der Copper-Liste (Copper-Interrupt).

Den elegantesten Kompromiß zwischen Programmieraufwand und Leistung bietet die WaitTOF-Funktion (Wait TOp of Frame, Offset -270) der Graphics-Library, mit der sich beliebige Operationen auf den Vertical-Blanking-Interrupt synchronisieren lassen.

Wenn vor jedem Zugriff auf die Datenliste WaitTOF aufgerufen und unmittelbar danach die aktuellen Kontrollworte geschrieben werden, gibt es keine Probleme:

```
lea _GfxBase, a5
jsr -270(a5)
move #xx, SPRxCTL
move #xx, SPRxPOS
```


Definition der Bildschirmpriorität

Bei der Komplexität des Grafik-Systems bedarf es einer strengen Hierarchie, anhand derer die Video-Logik entscheidet, welche Flächen sichtbar sind und welche von Elementen höherer Priorität verdeckt werden.

Die Darstellungs-Priorität der Sprites ist statisch und nimmt mit steigender Ordnungszahl ab. Wenn es zu Überschneidungen kommt, erscheint Sprite 0 immer vor Sprite 7 dagegen immer hinter allen anderen Sprites. Diese Hierarchie sollte man bei Auswahl der Sprite-Kanäle bedenken.

Direkt beeinflussen läßt sich die Darstellungspriorität gegenüber den Playfields, im Dual-Playfield-Modus sogar getrennt für gerade bzw. ungerade Bitplanes. Allerdings beziehen sich die Positionsangaben nicht auf einzelne Sprites, sondern (wie schon bei der Farbselektion im Attached-Modus) auf Sprite-Paare. Das Playfield kann vor bzw. hinter den Sprites oder irgendwo zwischen den Paaren liegen. Insgesamt gibt es fünf verschiedene Positionen, die jeweils getrennt für gerade (PF1Px-Bits) bzw. ungerade Bitplanes (PF2Px-Bits) von einem 3-Bit-Wert im BPLCON2-Register repräsentiert werden.

Stellt man sich die Grafikelemente als Stapel vor, dann sieht die Zuordnung von oben nach unten folgendermaßen aus:

Wert: 000	001	010	011	100
Playfield	Sprite 0 und 1	Sprite 0 und 1	Sprite 0 und 1	Sprite 0 und 1
Sprite 0 und 1	Playfield	Sprite 2 und 3	Sprite 2 und 3	Sprite 2 und 3
Sprite 2 und 3	Sprite 2 und 3	Playfield	Sprite 4 und 5	Sprite 4 und 5
Sprite 4 und 5	Sprite 4 und 5	Sprite 4 und 5	Playfield	Sprite 6 und 7
Sprite 6 und 7	Sprite 6 und 7	Sprite 6 und 7	Sprite 6 und 7	Playfield

Die Differenzierung zwischen geraden und ungeraden Bitplanes ist natürlich nur im Dual-Playfield-Modus sinnvoll. Befindet sich das System im normalen Darstellungsmodus, werden die Positions-Bits für die ungeraden Planes (PF2Px) und das Prioritäts-Flag PF2PRI ignoriert.

BPLCON2 \$104 w

(Bit PLane CONTROL register 2 = Bitplane-Kontrollregister 2)

Dieses Register steuert die Darstellungs-Priorität der verschiedenen Grafikobjekte (Playfields, Bobs, Sprites).

Bitbelegung:

Nr.	Label	Funktion
15-07	—	nicht verwendet.
06	PF2PRI	Playfield 2 (ungerade Bitplanes) besitzt erhöhte Priorität gegenüber Playfield 1 (gerade Playfields) und erscheint immer vor den geraden Bitplanes (Funktion auf den Dual-Playfield-Modus begrenzt).
05-03	PF2Px	Diese Bits beschreiben die Position der ungeraden Bitplanes zu den vier Sprite-Paaren. Es sind Werte von 0 (Bitplanes vorne) bis 4 (Bitplanes ganz hinten) möglich.
02-00	PF1Px	Analoges für Playfield 1 (gerade Bitplanes).

Die Kollisionserkennung

Für die Kollisionsregistrierung animierter Objekte stellt das Betriebssystem die Funktion DoCollision (Graphics-Library) bereit. Anstatt die leistungsfähige Detections-Logik auf dem DENISE-Chip zu nutzen, arbeitet DoCollision jedoch mit komplexen Kollisionsmasken und aufwendigen Positionsvergleichen und wird dadurch für ein schnelles Animationssystem schlicht unbrauchbar.

Man kommt also in den wenigsten Fällen um eine direkte Abfrage der Kollisions-Hardware herum. Dabei ist der Programmieraufwand kaum größer als bei Verwendung der System-Routinen.

Das Kontrollregister CLXCON enthält die Spritenummer bzw. Pixelkombinationen der zu untersuchenden Objekte, und das CLXDAT-Register registriert die Zusammenstöße. Erkannt werden Sprite/Sprite, Sprite/Playfield und Playfield/Playfield-Kollisionen.

Die Detections-Logik ist (wie der Prioritäts-Filter) mit den Bitplane- und Sprite-Sequenzen verbunden und analysiert die eintreffenden Bitströme. Für jeden Pixel der Bildschirmdarstellung werden die Farbvektoren der Sprite- und Playfield-Hardware mit den aktuellen Parametern des Kontrollregisters CLXCON verglichen und im Falle eines gültigen Zusammenstoßes durch Setzen des entsprechenden Bits im Datenregister CLXDAT vermerkt.

Leider ist das CLXDAT-Register nicht mit dem Interrupt-System gekoppelt, so daß es zyklisch überprüft werden muß. Je nach Anwendung und Komplexität des Animations-systems geschieht dies innerhalb der Bewegungssequenz eines Grafikobjektes oder durch regelmäßige Abfrage (beispielsweise im Vertical-Blanking-Interrupt).

Die Bitbelegung der Kollisions-Register:

CLXCON \$098 w

(CoLLision CONtrol = Kollisions-Kontrollregister)

Kontrollregister der Kollisions-Erkennungs-Logik.

Bitbelegung:

Nr.	Label	Funktion
15	ENSP7	Kollisionskontrolle von Sprite 7 einschalten.
14	ENSP5	Kollisionskontrolle von Sprite 5 einschalten.
13	ENSP3	Kollisionskontrolle von Sprite 3 einschalten.
12	ENSP1	Kollisionskontrolle von Sprite 1 einschalten.
11-06	ENBPx	Kollisionskontrolle von Bitplane x (Bit 11 für Plane 6, Bit 10 für Plane 5 usw.) aktivieren.
05-00	MVBPx	Vergleichswert für Bitplane x setzen. Nur wenn Match-Bit und Bitplane an der Kollisionstelle übereinstimmen, wird die Kollision registriert.
CLXDAT	\$00E	r

(CoLLision DATa register = Datenregister der Kollisions-Logik)

Dieses Register reflektiert den aktuellen Kollisions-Status. Nach dem Zugriff wird es automatisch zurückgesetzt.

Besonderheit:

Zu Playfield 1 gehören alle aktiven ungeraden Bitplanes, zu Playfield 2 alle eingeschalteten, geraden Bitplanes.

Bitbelegung:

Nr.	Funktion
15	nicht verwendet.
14	Sprite 4 (oder 5) kollidiert mit Sprite 6 (oder 7).
13	Sprite 2 (oder 3) kollidiert mit Sprite 6 (oder 7).
12	Sprite 2 (oder 3) kollidiert mit Sprite 4 (oder 5).
11	Sprite 0 (oder 1) kollidiert mit Sprite 6 (oder 7).
10	Sprite 0 (oder 1) kollidiert mit Sprite 4 (oder 5).
09	Sprite 0 (oder 1) kollidiert mit Sprite 2 (oder 3).

Bitbelegung:

Nr.	Funktion
08	Sprite 6 (oder 7) kollidiert mit Playfield 2.
07	Sprite 4 (oder 5) kollidiert mit Playfield 2.
06	Sprite 2 (oder 3) kollidiert mit Playfield 2.
05	Sprite 0 (oder 1) kollidiert mit Playfield 2.
04	Sprite 6 (oder 7) kollidiert mit Playfield 1.
03	Sprite 4 (oder 5) kollidiert mit Playfield 1.
02	Sprite 2 (oder 3) kollidiert mit Playfield 1.
01	Sprite 0 (oder 1) kollidiert mit Playfield 1.
00	Playfield 1 kollidiert mit Playfield 2.

Wie man sieht, ist die Auflösung der Kollisionserfassung bei Sprites auf Paare beschränkt, Zusammenstöße zwischen den einzelnen Elementen eines Paares können nicht registriert werden. Für eine korrekte Abfrage von 16-farbigen Multi-Color-Sprites im Attached-Mode muß auch das korrespondierende ungerade Sprite in den Erkennungsprozeß eingeblendet werden (Setzen des ENSPx-Bits).

Sehr flexibel wird dagegen die Abfrage der Bitplanes gehandhabt. Die Kollisionslogik untersucht nur solche Planes, die durch Setzen der zugehörigen ENBPx-Bits ("Enable BitPlane x") angemeldet sind und vergleicht sie mit dem 6-Bit-Matchwert der MVBPx-Bits ("Match Value for BitPlane x").

Durch diese Technik läßt sich die Kollisionskontrolle gezielt auf einen bestimmten Bereich von Farbvektoren eingrenzen. Geschickte Auswahl der Bitmuster und entsprechende Verteilung auf die Grafikobjekte erlauben die Definition von Hot-Spots und die Bildung von Kollisionsmasken für Playfield-Animationen.

Sehen wir uns dazu zwei Beispiele an:

Es seien nur Kollisionen an den Punkten erlaubt, die ihre Pixelfarbe aus Farbreghister Nummer 8 holen. Dazu müssen sämtliche Bitplanes analysiert (ENBPx = 111111) und mit dem Matchwert 001000 (8) verglichen werden:

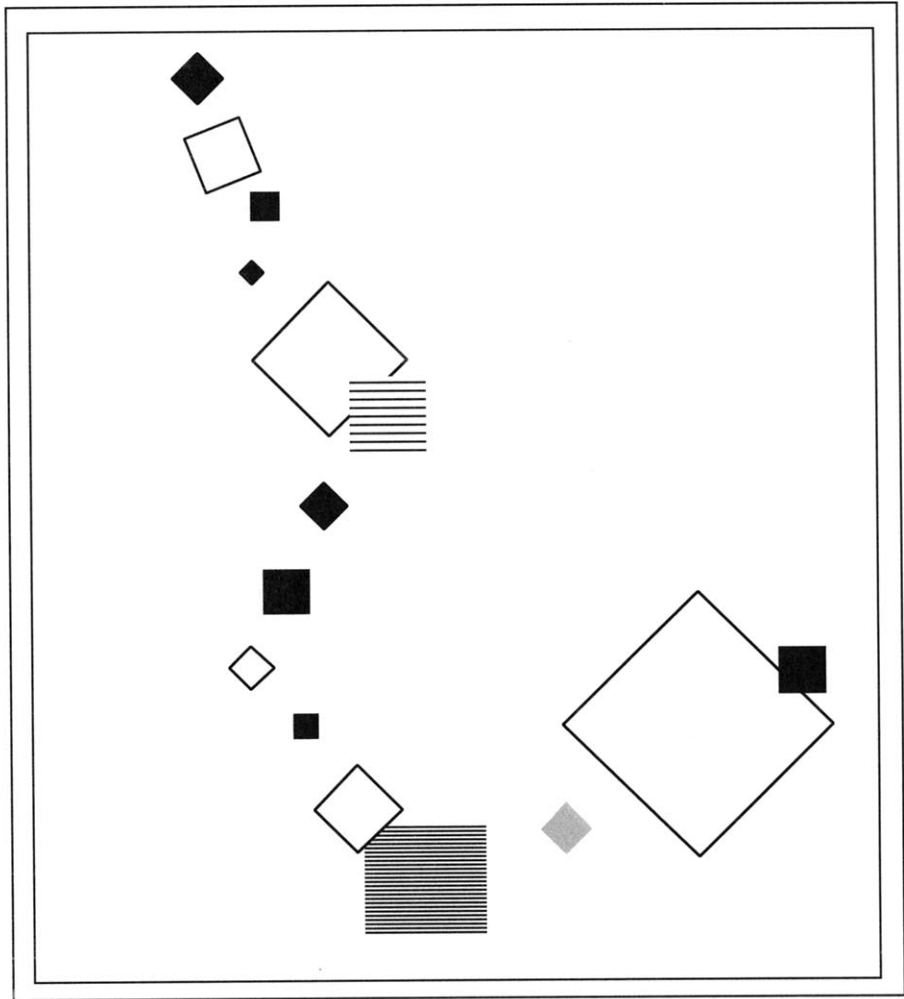
```
move.w %0000111111001000,COLXCON
```

Ein Hot-Spot-Feld der höchsten vier Farbreghister (28 bis 31) enthielte die Bitmuster 011100, 011101, 011110, 011111. Der entsprechende Matchwert ist 0111xx (die unteren beiden Bits werden nicht berücksichtigt), die ENBPx-Bits sollten auf 111100 liegen (nur die höchsten vier Binärstellen sind statisch und müssen untersucht werden):

```
move.w %0000111100011100,COLXCON
```

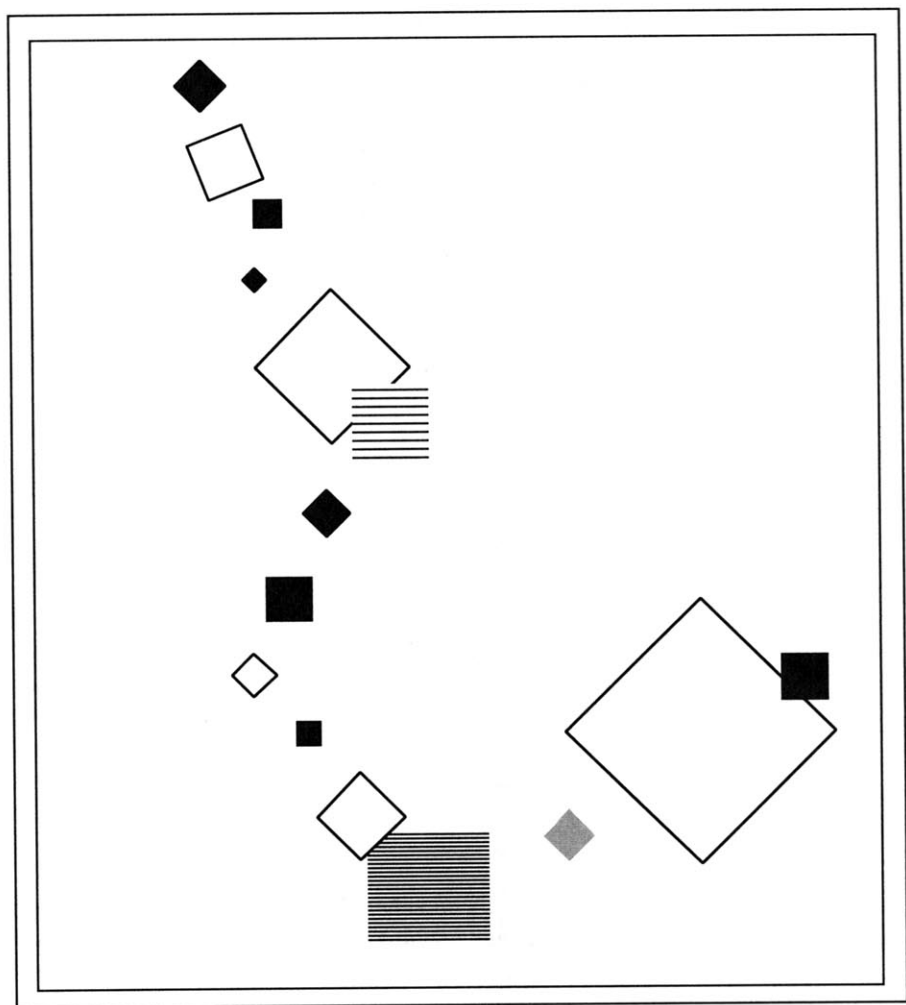

Teil 2

Die Systemsoftware



Kapitel 1

Exec



Am Anfang ist die ExecBase...

Bevor beim Amiga überhaupt etwas gemacht werden kann, muß die ExecBase initialisiert sein. Und da auch sonst recht viel über sie läuft, ist dieser Abschnitt einigen Ausschnitten der ExecBase gewidmet.

```

                                struct ExecBase {
$0000      0      struct Library  LibNode;
$0022     34      UWORD    SoftVer;
$0024     36      WORD     LowMemChkSum;
$0026     38      ULONG    ChkBase;
$002a     42      APTR     ColdCapture;
$002e     46      APTR     CoolCapture;
$0032     50      APTR     WarmCapture;
$0036     54      APTR     SysStkUpper;
$003a     58      APTR     SysStkLower;
$003e     62      ULONG    MaxLocMem;
$0042     66      APTR     DebugEntry;
$0046     70      APTR     DebugData;
$004a     74      APTR     AlertData;
$004e     78      APTR     MaxExtMem;
$0052     82      UWORD    ChkSum;
$0054     84      struct    IntVector IntVects[16];
$0054     84      TBE
$0060     96      DSKBLK
$006C    108      SOFTINT
$0078    120      PORTS
$0084    132      COPER
$0090    144      VERTB
$009C    156      BLIT
$00A8    168      AUD0
$00B4    180      AUD1
$00C0    192      AUD2
$00CC    204      AUD3
$00D8    216      RBF
$00E4    228      DSKSYNC
$00F0    240      EXTER
$00FC    252      INTEN
$0108    264      NMI

```

```

$0114 276      struct Task *ThisTask;
$0118 280      ULONG IdleCount;
$011c 284      ULONG DispCount;
$0120 288      UWORD Quantum;
$0122 290      UWORD Elapsed;
$0124 292      UWORD SysFlags;
$0126 294      BYTE IDNestCnt;
$0127 295      BYTE TDNestCnt;
$0128 296      UWORD AttnFlags;
$012a 298      UWORD AttnResched;
$012c 300      APTR ResModules;
$0130 304      APTR TaskTrapCode;
$0134 308      APTR TaskExceptCode;
$0138 312      APTR TaskExitCode;
$013c 316      ULONG TaskSigAlloc;
$0140 320      UWORD TaskTrapAlloc;
$0142 322      struct List MemList;
$0150 336      struct List ResourceList;
$015e 350      struct List DeviceList;
$016c 364      struct List IntrList;
$017a 378      struct List LibList;
$0188 392      struct List PortList;
$0196 406      struct List TaskReady;
$01a4 420      struct List TaskWait;
$01b2 434      struct SoftIntList SoftInts[5];
$0202 514      LONG LastAlert[4];
$0212 530      UBYTE VBlankFrequency;
$0213 531      UBYTE PowerSupplyFrequency;
$0214 532      struct List SemaphoreList;
$0222 546      APTR KickMemPtr;
$0226 550      APTR KickTagPtr;
$022a 554      APTR KickCheckSum;
$022e 558      UBYTE ExecBaseReserved[10];
$0238 568      UBYTE ExecBaseNewReserved[20];
};

```

Da ohne diese Struktur so ziemlich nichts im Amiga laufen kann, ist ihr auch als einziger Software-Komponente eine feste Adresse zugeordnet: In der Speicherstelle 4 findet man den Zeiger auf sie.

Doch nun zu den eigentlichen Einträgen:

LibNode	Library-Struktur zur internen Verwaltung der Exec-Library.
SoftVer	Eine Nummer, die der internen Release-Nummer entspricht.
LowMemChkSum	Eine Checksum über die Werte von SoftVer bis MaxExtMem (eigentliche CheckSum steht in ChkSum).
ChkBase	Komplement der Basis-Adresse. Exklusiv-Oder-Verknüpfung der Basis-Adresse und dieses Wertes muß -1L ergeben. Ist dies nicht der Fall, wird die Struktur beim Reset komplett neu erstellt.
ColdCapture	Vektor für einen Einsprung in die Reset-Routine. Ist dieser Vektor 0, so wird keine Routine angesprungen. Achtung! An dieser Stelle ist im System noch so ziemlich nichts richtig gesetzt. Daher muß die Reset-Routine auch mit JMP(A5) zurückkehren und nicht mit RTS. Außerdem sollte man nach Möglichkeit den Stack nicht verwenden, da auch er nicht gesetzt ist. Schreibt man eine Reset-Routine, muß man als erstes den Vektor wieder setzen, da er vor dem Einsprung in die Routine gelöscht wird.
CoolCapture	Entspricht ColdCapture, nur ist hier das System weitgehend initialisiert. Daher kann nun auch fast ganz normal gearbeitet werden. Zur Reset-Routine gelangt man mit RTS.
WarmCapture	Dieser Vektor wird z.Zt. nicht (mehr?) angesprungen.
SysStkUpper SysStkLower	Obere und untere Grenze des Supervisor-Stacks. Diese Werte stehen fast immer an der obersten Grenze des Speichers.
MaxLocMem	Größe des Chip-Speichers.
DebugEntry	Adresse des Debuggers.
DebugData	Adresse der Debugger-Daten (normalerweise 0).
AlertData	Vermutlich Daten für den Aufruf der Alert-Routine (undokumentiert).
MaxExtMem	Obere Grenze des zur Verfügung stehenden Speichers.

ChkSum Diese ChkSum über denselben Bereich wie LowMemChkSum und wird beim Reset überprüft. Berechnet wird sie wie folgt:

```

$FC043E:      MOVEQ      #0,D1
               LEA        SoftVer(A6),A0
               MOVE.W     #$0016,D0
               ADD.W      (A0)+,D1
               DBRA       D0,START+10
               NOT.W      D1
               MOVE.W     D1,ChkSum(A6)

```

IntVects Vektoren der Interrupts.

ThisTask Zeiger auf den aktuell bearbeiteten Task.

IdleCount "Leerlauf"-Zähler.

DispCount Zähler für Task-Wechsel.

Quantum Größe der Zeitscheiben.

Elapsed Anzahl der vergangenen Zeiteinheiten.

SysFlags Wichtige System-Flags.

IDNestCnt Disable/Enable-Zähler. Ist dieser Wert gleich -1, so sind Interrupts erlaubt.

TDNestCnt Forbid/Permit-Zähler. Ansonsten wie IDNestCnt.

AttnFlags Je nach Prozessor werden hier verschiedene Bits gesetzt:

Bit 0: 68010

Bit 1: 68020

Bit 4: 68881

ResModules Zeiger auf ein Array aus residenten Modulen. Diese werden beim Reset aufgerufen.

Über die folgenden fünf Vektoren war bisher leider keine Dokumentation zu bekommen, daher mußte die Bedeutung von der Namensgebung abgeleitet werden:

TaskTrapCode	Adresse des Trap-Handlers des aktuell bearbeiteten Programms.
TaskExceptCode	Adresse des Exception-Handlers.
TaskExitCode	Adresse der Exit-Routine.
TaskSigAlloc	Vom Task reservierte Signale.
TaskTrapAlloc	Vom Task belegte Trap-Vektoren.
MemList	Liste, in der die Memory-Blöcke verwaltet werden.
ResourceList	Liste der Ressourcen im System.
DeviceList	Liste der Devices im System.
IntrList	Liste der Interrupts (zur Zeit nicht benutzt).
LibList	Liste der Libraries.
PortList	Liste der Ports.
TaskReady	Liste der Tasks im READY-Status.
TaskWait	Liste der Tasks im WAIT-Status.
SoftInts	Array der Software-Interrupts, die auf Abarbeitung warten, die Prioritäten sind in der Reihenfolge -32,-16,0,16,32 verteilt.
LastAlert	Hier werden die Daten des nach einem Reset darzustellenden Alerts zwischengespeichert.
VBlankFrequency	Frequenz des VBL-Interrupts.
PowerSupply-Frequency	Netzspannungs-Frequenz.
SemaphoreList	Liste der verwendeten Semaphoren.
KickMemPtr	Zeiger auf eine Liste von Memory-Strukturen.

KickTagPtr	Zeiger auf eine Liste von RomTags. Diese wird jedoch nur eingebunden, wenn alle Speicherteile von KickMemPtr belegt werden konnten.
KickChkSum	Prüfsumme für die letzten Werte. Diese erhält man durch Aufruf der Funktion SumKickData.
ExecBaseReserved	Reservierter Speicherbereich für spätere Erweiterungen. Dieser sollte, ebenso wie ExecBaseNewReserved unter keinen Umständen verwendet werden, da sonst die Kompatibilität zu neuen Kickstarts verloren gehen kann.

Listen

Listenverwaltung

Grundlage für viele Funktionen des Betriebssystems sind Listen. Ohne sie hätte man Probleme mit dem Multitasking (wie will man ohne Listen die Tasks verwalten?), Intuition (Warteschlangen von Messages) oder der Hardware.

Doch glücklicherweise haben wir sie. Stellt sich nur die Frage: Was ist eigentlich eine Liste? Wie ist sie aufgebaut?

Listen werden im Amiga-Betriebssystem mit Hilfe eines Headers (struct List) verwaltet, die einzelnen Einträge sind die Nodes. Eine Liste wird immer über die Adresse des Headers verwaltet. Er besitzt Informationen über den ersten und letzten Eintrag der Liste sowie über die Art der Liste:

```
struct List {
    struct Node *lh_Head;
    struct Node *lh_Tail;
    struct Node *lh_TailPred;
    UBYTE lh_Type;
    UBYTE l_pad;
};
```

Der erste Eintrag der Header-Struktur ist der Zeiger auf das erste Element der Liste, als nächstes findet man das letzte Element der Liste, welches jedoch immer 0 ist. Einen

Zeiger auf das letzte gefüllte Element der Liste befindet sich in `lh_TailPred`. Die Art der Liste kann einen der folgenden Werte annehmen (zu finden in `exec/nodes.h`):

<code>NT_UNKNOWN</code>	0		<code>NT_LIBRARY</code>	9
<code>NT_TASK</code>	1		<code>NT_MEMORY</code>	10
<code>NT_INTERRUPT</code>	2		<code>NT_SOFTINT</code>	11
<code>NT_DEVICE</code>	3		<code>NT_FONT</code>	12
<code>NT_MSGPORT</code>	4		<code>NT_PROCESS</code>	13
<code>NT_MESSAGE</code>	5		<code>NT_SEMAPHORE</code>	14
<code>NT_FREEMSG</code>	6		<code>NT_SIGNALSEM</code>	15
<code>NT_REPLYMSG</code>	7		<code>NT_BOOTNODE</code>	16
<code>NT_RESOURCE</code>	8			

`lh_pad` (so definiert in den C-Includes, nach Assembler-Includes `lh_pad`, welches stimmen dürfte...) sorgt nur dafür, daß die Struktur auf einer Wort-Grenze endet. Sie sollte nicht für Informationen verwendet werden.

Um eine Liste benutzen zu können, muß sie erst einmal initialisiert werden. Dies geschieht mit der Funktion `NewList`, die die Adresse der Liste als Argument erhält (in Assembler würde man das Makro `NEWLIST` verwenden). Dabei geschieht folgendes:

- Die Adresse des Headers wird in das Feld `lh_Head` geschrieben und um 4 erhöht (→ Adresse von `lh_Tail`). `-lh_Tail` wird gelöscht.
- Die Adresse des Headers wird in das Feld `lh_TailPred` geschrieben.

Nun ist die Struktur bereit zum Einsatz, einzelne Elemente können angehängen werden. Diese Nodes sehen folgendermaßen aus:

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE   ln_Type;
    BYTE    ln_Pri;
    char    *ln_Name;
};
```

`ln_Succ` ist das nächste Element der Liste, `ln_Pred` zeigt auf das vorhergehende Element. `ln_Type` entspricht `lh_Type` in der Header-Struktur. Mit `ln_Pri` definiert man die Priorität des Eintrages, `ln_Name` zeigt auf den Namen des Eintrages. Mit dieser Struktur hat man nun sein "Space Shuttle" für die Reise in die Welt der Listen vorliegen, doch wohin packt man die Nutzlast? Die Antwort darauf ist ganz einfach: genau wie beim

Raumschiff auch – alles nach hinten (hinter die Node-Struktur). Will man beispielsweise eine Adressen-Verwaltung mit den Exec-Listen verwirklichen, so definiert man die einzelnen Einträge folgendermaßen:

```
struct Adresse {  
    struct Node node;  
    char Vorname[20];  
    char Nachname[30];  
    char Strasse[50];  
    char Ort[60];  
    char Tel[20];  
};
```

Die zur Verwaltung notwendigen Node-Daten sind direkt am Anfang untergebracht, so daß man die Adressen-Struktur wie eine normale Node-Struktur behandeln kann. Und ob man im eigenen Programm noch die Node-Struktur am Anfang stehen hat oder nicht ist relativ egal, da man die einzelnen Einträge sowieso über den Offset anspricht (z.B. `strcpy(adr->Vorname,vname)`; mit `struct Adresse *adr`).

Routinen zur Listenmanipulation

Um diese Listen verwenden zu können, muß man Teile einfügen und löschen können. Zu diesem Zweck stehen einige Funktionen zur Verfügung, die hier kurz erläutert werden.

```
Insert (list, node, pred)
```

Mit Insert fügt man die Node node in der Liste list hinter der Node pred ein. Ist pred 0 oder gleich dem Header, so wird die Node an der ersten Stelle eingefügt.

```
Remove (node)
```

Remove entfernt die Node node aus einer Liste. Da vom System offensichtlich nicht geprüft wird (oder werden kann), ob sich node überhaupt in einer Liste befindet, könnte das Entfernen nicht eingebundener Nodes leichte Probleme verursachen (= Guru).

Für das Einfügen und Löschen an erster und letzter Stelle einer Liste können Insert und Remove zwar auch verwendet werden, doch sollte man hier auf die folgenden Funktionen zurückgreifen:

```
AddHead (list, node) AddTail (list, node)
```


Hiermit fügt man eine Node am Anfang (Head) bzw. am Ende (Tail) der Liste ein. Die Gegenstücke zum Entfernen von Nodes lauten:

```
RemHead (list) RemTail (list)
```

Falls die Liste bereits leer ist, so wird ein Null-Pointer zurückgegeben.

```
Enqueue (list, node)
```

Diese Funktion entspricht der AddHead-Funktion, jedoch mit einem kleinen Unterschied: Neue Nodes werden nicht einfach am Anfang der Liste eingetragen, sondern nach dem In_Pri-Feld sortiert. Somit wird eine Node nach dem letzten Element gleicher Priorität eingefügt. Um die jeweils wichtigste Node abzuholen, benutzt man RemHead.

```
FindName (list, name)
```

Mit dieser Funktion sucht man in einer Liste nach einem bestimmten Eintrag, von dem man den Namen weiß. Dabei wird immer ab dem nächsten Eintrag gesucht; um also vom Listenanfang an zu suchen, übergibt man nicht etwa den Zeiger auf das erste Element, sondern den Zeiger auf den Header. Außerdem muß man beachten, daß die Suche nach dem Namen Groß- und Kleinschreibung unterscheidet ("dir" ist nicht gleich "Dir").

Auf der Diskette befindet sich ein kleines Beispielprogramm, das demonstriert, wie man mit Listen eine Adressendatei verwalten kann. Neben dem normalen Vor- und Rückwärtslesen aller Namen wird auch das Suchen der Nachnamen mittels der Funktion FindName unterstützt.

Die Libraries

Wie man mit Libraries umgeht

Eine der besten Einrichtungen des Amigas sind die Libraries, Bibliotheken von Systemroutinen. Nicht nur wegen der Menge und Qualität der einzelnen Routinen, sondern besonders deswegen, weil man sie sehr leicht verbessern kann. So wird beispielsweise zur Zeit für die Entwickler von Kickstart-1.4-Software eine Jumpstart ausgeliefert, die die neuen 1.4-Libraries als Software-Update enthält. Diese werden dann so installiert, daß sie nach einem Reset automatisch die Rom-Software (Exec, Intuition, etc.) ersetzen.

Somit haben auch Besitzer von Amiga 500 und 2000 die Möglichkeit, die neueste Version der Libraries zu verwenden, ohne die Roms austauschen zu müssen.

Doch wie arbeitet man eigentlich mit diesen Libraries? Wie auch jede Bibliothek hat eine Library einen Index, in dem alle Routinen nacheinander aufgeführt sind. Um diese Routinen unabhängig von ihrer momentanen Lage ansprechen zu können, werden sie mittels eines Offsets relativ zur Basis-Adresse angesprungen. Diese Basis-Adresse ist beim Amiga normalerweise in A6 zu finden. Demzufolge sieht ein Library-Einsprung so aus:

```
MOVE.L  LibraryBase, a6
JSR      Offset(a6)
```

Des weiteren stehen bei allen System-Libraries die Rückgabe-Werte in D0, egal ob es sich um eine Adresse oder einen normalen Wert handelt. Außerdem kann man davon ausgehen, daß die Register D0, D1, A0 und A1 nach dem Aufruf verändert werden, A6 dagegen behält bei allen System-Libraries seinen Inhalt (genau wie die anderen Register).

Bei der Programmierung eigener Libraries sollte man an die Leute denken, die die Library später einmal benutzen, und dies ebenfalls einhalten.

Libraries Marke "Eigenbau"

Eine Anleitung im SingleStep...

Um eine eigene Library schreiben zu können, muß man erst einmal verstehen, wie Libraries aufgebaut sind.

Im Speicher werden die eingebundenen Libraries durch drei Dinge bestimmt: Der wichtigste ist dabei der Base-Pointer, also die Basis-Adresse, von der aus alle Zugriffe auf die Library geschehen. Die anderen beiden Teile sind der positive und negative Speicherbereich. Im negativen Bereich werden dabei die Sprung-Adressen der Library eingetragen, der positive Teil enthält die Daten des Library-Status, die von Library zu Library total verschieden sind.

Total? Nun, nicht ganz. Alle Libraries haben eines gemeinsam: die Library-Struktur am Anfang der Library-Base. Hier werden die Daten zur internen Verwaltung der Library gespeichert, so z.B. die Größe des positiven und negativen Teils, die CheckSum sowie die Zahl der offenen Zugriffe auf die Library.

Und noch etwas haben die Libraries gemeinsam: vier (4) Funktionen, ohne die nämlich nichts geht (außer schief...).

Diese Funktionen sind in der Reihenfolge ihres Auftretens in der Library:

Open	Diese Funktion wird aufgerufen, sobald irgendein Task im System die Library mit OpenLibrary öffnet. Dabei wird der Zähler in der Library-Struktur hochgezählt.
Close	Ähnlich wie Open wird diese Funktion aufgerufen, sobald im System die Funktion CloseLibrary verwendet wird. Hierbei wird der oben genannte Zähler dekrementiert, und, falls das betreffende Flag gesetzt ist, bei Zähler=0 die Library aus dem System entfernt (Aufruf von Expunge).
Expunge	(=vertreiben, entfernen) bereitet die Library auf das Entfernen vor. Alle offenen Libraries werden geschlossen, der Speicher wird freigegeben.
ExtFunc	Dieser Pointer ist für zukünftige Erweiterungen reserviert, muß aber immer vorhanden sein. Die Routine sollte immer den Wert 0 zurückgeben.

Mit diesen Vorkenntnissen fällt es einem plötzlich wie "Schuppen von der Tastatur", warum die meisten Library-Vektoren bei -30 beginnen: Die bisher aufgeführten Funktionen haben die Offsets -6 (Open), -12 (Close), -18 (Expunge) und -24 (ExtFunc). Somit ist der nächste freie Platz Offset -30.

Doch Moment, wieso immer im Sechser-Schritt?

Die Library-Einsprünge

Jede "normale" Library (d.h. alle außer DOS, aber die is' ja nich' ganz normal...) hat die Library-Einsprünge nach folgendem Prinzip aufgebaut:

```
jmp $xxxxxx
jmp $yyyyyy
```

wobei \$xxxxxx und \$yyyyyy die Adresse der eigentlichen Routine sind. So weit, so gut.

Da das DOS dank seiner Abstammung eine kleine Extrawurst haben muß, wird hier ein etwas anderes Vorgehen verwendet:

Bei den Offsets wird D0 mit Werten für eine Sprungtabelle geladen (moveq #offset,d0), danach wird an die Stelle der Library gesprungen (bra DosBase+\$4e bei Kickstart 1.3), die für die Umrechnung und den Ansprung der eigentlichen Funktion sorgt. Das ergibt zusammen wieder 6 Byte pro Eintrag, womit wenigstens nach außen hin der Schein einer ordentlichen Library gewahrt wird.

Egal, damit wollen wir uns nicht weiter belasten. Wenn wir Glück haben, wird die Kickstart 1.4 bereits über ein DOS verfügen, das nicht mehr in BCPL, sondern in C geschrieben ist.

Der Library-Knoten

Bei der Definition der Library wird, wie schon erwähnt, eine Library-Struktur zur Verwaltung verwendet. Diese hat folgendes Aussehen:

```
extern 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_IdString;  
    ULONG lib_Sum;  
    UWORD lib_OpenCnt;  
};
```

lib_node ist eine normale Node-Struktur, die dazu verwendet wird, die Library in die System-Listen einzubinden.

lib_Flags Hier werden Informationen über den Zustand der Library gespeichert. Diese sind:

LIBF_SUMMING

Über die Library wird gerade eine Checksum berechnet.

LIBF_CHANGED

Die Library wurde (gerade) verändert.

LIBF_SUMUSED

Wird dieses Flag geändert, so wird darauf geachtet, ob die Checksum der Library stimmt.

LIBF_DELEXP

Delayed expunge, d.h. die Library soll entfernt werden, ist aber noch nicht von allen Tasks freigegeben.

lib_NegSize und lib_PosSize

geben die Größe des positiven und negativen Teils wieder.

lib_Version, lib_Revision und lib_idString

Wichtig ist hier eigentlich für das System nur die Versions-Nummer, da mit ihr entschieden wird, ob ein Benutzer die von ihm angeforderte Library bekommt oder nicht. Der andere Wert und der idString sind nur zur Information, welche Unterversion man benutzt und von wann sie stammt. Der idString sollte dabei folgende Informationen enthalten: Name der Library Version.Revision (Datum) sowie anschließend die ASCII-Werte 13, 10 und 0.

lib_Sum

enthält die CheckSum der Library.

lib_OpenCnt

ist der wohl wichtigste Wert. Hier wird gezählt, wie oft die Library geöffnet wurde.

Legen wir los...

Doch nun zu unserer eigenen Library.

Angenommen, wir wollen einige CLI-Befehle schreiben, die uns Arbeiten im System erledigen sollen, deren Aufgaben aber so universell sind, daß wir sie auch für andere Dinge verwenden wollen. Daher setzen wir uns hin und bauen unsere eigene Library.

Zuerst bekommt das Kind einen Namen. Nennen wir sie also MyLib. Klingt gut und tut unserem Ego noch besser...

Nun zu den Funktionen. Für den Anfang reichen zwei aus: eine, die auf einen Mausklick wartet, und eine, die den Bildschirm blinken läßt, wobei wir die Anzahl übergeben wollen.

Die Namen für diese Funktionen sind dann auch recht schnell gefunden: WaitMouse für die erste, Blink für die zweite Funktion.

So weit, so gut. Kommen wir zu den Problemen: Wie geht's weiter?

Beginnen wir bei der Definition der Library-Base (File mylibbase.h). Dazu brauchen wir zuerst einmal ein paar Includes, was in den ersten paar Zeilen erledigt wird.

```
#ifndef MYLIB_BASE_H
#define MYLIB_BASE_H
#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif EXEC_TYPES_H
#ifndef EXEC_LISTS_H
#include <exec/lists.h>
#endif EXEC_LISTS_H
#ifndef EXEC_LIBRARIES_H
#include <exec/libraries.h>
#endif EXEC_LIBRARIES_H
```

Danach wird die Base definiert. Wie schon gesagt, hat jede Library eine Library-Struktur, die wir als erstes einfügen:

```
struct MyLibBase {
    struct Library LibNode;
```

Nun noch ein paar Speicherstellen für die Verwaltung und Library-Pointer, dann sind wir schon mit der Library-Base fertig:

```
    UBYTE Flags;
    UBYTE pad;
    ULONG SysLib;
    ULONG IntuiLib;
    ULONG DosLib;
    ULONG SegList;
};
```

Seien wir noch nett und definieren den Namen unserer Library:

```
#define MYLIBNAME      "MyLib.library"
#endif MYLIB_BASE_H
```

Das war's schon. Nun kommt noch das Equivalent in Assembler (befindet sich auf der Diskette). Ist das erledigt, dann können wir uns um das C-Interface kümmern. Auch hier brauchen wir ein einige Vorarbeiten:

```
INCLUDE "exec/types.i"
INCLUDE "exec/libraries.i"
    SECTION _LVO
    DSEG
```

Als nächstes definieren wir die Library-Einsprünge:

```
LIBINIT
LIBDEF _LVOWaitMouse
XDEF _LVOWaitMouse
LIBDEF _LVOBlink
XDEF _LVOBlink
```

Danach werden dem restlichen Code einige Werte publik gemacht:

```
SECTION CSTUB
CSEG
XREF _MyLibBase
XDEF _WaitMouse
XDEF _Blink
```

Nun folgen die Routinen, die vom C-Code aus aufgerufen werden und die Ansteuerung der Library übernehmen:

```
_WaitMouse:
    MOVE.l  a6, -(sp)
    move.l  _MyLibBase, a6
    jsr     _LVOWaitMouse(a6)
    move.l  (sp)+, a6
    rts

_Blink:
    move.l  a6, -(sp)
    move.l  8(sp), d0
```

```
                ; Anzahl Blinken nach D0
move.l  _MyLibBase, a6
jsr     _LVOBlink(a6)
move.l  (sp)+, a6
rts
END
```

Damit ist dieses File (genannt mylibln.asm) fertig. Es muß nur noch assembliert werden und kann danach zu C-Programmen gelinkt werden. Jetzt kommen wir zu dem eigentlichen Library-Code. Auch dazu brauchen wir eine Reihe Includes:

```
SECTION        Section
include        "exec/types.i"
include        "exec/libraries.i"
include        "exec/lists.i"
include        "exec/alerts.i"
include        "exec/initializers.i"
include        "exec/resident.i"
include        "libraries/dos.i"
include        "intuition/intuitionbase.i"
include        "mylibbase.i"
```

Als nächstes geben wir die einzelnen Vektoren bekannt:

```
XDEF        InitLib
XDEF        Open
XDEF        Close
XDEF        Expunge
XDEF        ExtFunc
XDEF        MyLibName
XDEF        WaitMouse
XDEF        Blink
```

Der nun folgende Teil bindet die Offsets der Libraries ein, damit später beim Linken keine Library mehr eingebunden werden muß:

```
include        "libvecs/exec_lib.i"
include        "libvecs/intuition_lib.i"
include        "libvecs/dos_lib.i"
```


Dieser Teil wird bei den Aztec-Disketten allerdings nicht mitgeliefert.

Alle LibVecs-Includes wurden mit dem Programm ConvertToInc auf Diskette aus den FD-Files der Extras-Diskette erzeugt.

Um zu verhindern, daß die Library als normales Programm gestartet wird, wird als erstes in der Library der Rücksprung zum DOS verankert:

```
Start:
        moveq    #-1,d0
        rts
```

Nun folgt die Resident-Struktur, die zur Verwaltung der Library im Speicher benötigt wird:

```
RTSTRUC:
        DC.W      RTC_MATCHWORD
        DC.L      RTSTRUC
        DC.L      EndLib
        DC.B      RTF_AUTOINIT
        DC.B      MYLIBVERSION
        DC.B      NT_LIBRARY
        DC.B      MYLIBPRI
        DC.L      MyLibName
        DC.L      LibId
        DC.L      InitLib
MyLibName: MYLIBNAME
MYLIBVERSION: EQU      34
MYLIBREVISION: EQU     2
MYLIBPRI:     EQU      0
LibId:        DC.B 'MyLib.Library 34.2 (9. Aug 89)<',13,10,0
```

Außerdem haben wir gleich noch einige zusätzliche Daten definiert, so den Namen, die Version und den Identifikations-String.

Durch das Flag RTF_AUTOINIT (s.o.) wird festgelegt, daß es sich bei dem Wert InitLib um einen Zeiger auf diese Tabelle handelt.

Wäre das Bit nicht gesetzt gewesen, so würde es sich bei InitLib um einen Zeiger auf die aufzurufende Funktion handeln.

Weiter geht es nun mit den LibInit-Daten:

```

InitLib:      ds.w          0

              dc.l          MyLibBase_SIZEOF
              dc.l          Functions
              dc.l          Datas
              dc.l          LibInitRoutine; Init-Routine

Functions:
              dc.l          Open
              dc.l          Close
              dc.l          Expunge
              dc.l          ExtFunc ; bis hier Standard-Fkt
              dc.l          WaitMouse ; unsere Fkt
              dc.l          Blink
              dc.l          -1      ; Ende der Funktionen

Datas:
              INITBYTE      LN_TYPE,NT_LIBRARY
              INITLONG      LN_NAME,MyLibName
              INITBYTE      LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
              INITWORD      LIB_VERSION,MYLIBVERSION
              INITWORD      LIB_REVISION,MYLIBREVISION
              INITLONG      LIB_IDSTRING,LibId
              dc.l          0

```

Jetzt fehlt noch die eigentliche Initialisierungs-Routine. Bei der Gelegenheit werden noch zusätzliche Daten definiert:

```

dosName:      DOSNAME
intuiName:    dc.b          'intuition.library',0
              ds.w          0

LibInitRoutine:
              move.l        a5,-(sp)
              move.l        d0,a5
              move.l        a6,sb_SysLib(a5)
              move.l        a0,sb_SegList(a5)
              lea           dosName(pc),a1
              moveq         #0,d0
              CALLLIB       OpenLibrary
              move.l        d0,sb_DosLib(a5)
              bne.s         dosOK
              ALERT         AG_OpenLib!AO_DOSLib

```

```

dosOK:
        lea            intuiName(pc),a1
        moveq          #0,d0
        CALLLIB        OpenLibrary
        move.l         d0,sb_IntuiLib(a5)
        bne.s          intuiOK
        ALERT          AG_OpenLib!AO_Intuition

intuiOK:
        move.l         a5,d0
        move.l         (sp)+,a5
        rts

```

Ist unser Amiga erst mal so weit, dann kann eigentlich nichts mehr passieren. Vom System wird an dieser Stelle dem aufrufenden Task unsere Basis-Adresse übergeben. Allerdings wird noch folgende Routine aufgerufen:

```

Open:
        addq.w         #1,LIB_OPENCNT(a6)
        bclr           #LIBB_DELEXP,sb_Flags(a6)
        move.l         a6,d0
        rts

```

Danach folgt noch der Code zum Schließen und Entfernen der Library:

```

Close:
        moveq          #0,d0
        subq.w         #1,LIB_OPENCNT(a6)
        bne.s          endclose
        btst           #LIBB_DELEXP,sb_Flags(a6)
        beq.s          endclose
        bsr            Expunge

endclose:
        rts

Expunge:
        movem.l        d2/a5/a6,-(sp)
        move.l         a6,a5
        move.l         sb_SysLib(a5),a6
        tst.w          LIB_OPENCNT(a5)
        beq.s          noopen
        bset           #LIBB_DELEXP,sb_Flags(a5)
        moveq          #0,d0
        bra.s          ExpungeEnd

```

```

noopen:
    move.l    sb_SegList(a5),d2
    move.l    a5,a1
    CALLLIB   Remove
    move.l    sb_DosLib(a5),a1
    CALLLIB   CloseLibrary
    move.l    sb_IntuiLib(a5),a1
    CALLLIB   CloseLibrary
    moveq     #0,d0
    move.l    a5,a1
    move.w    LIB_NEGSIZE(a5),d0
    sub.l     d0,a1
    add.w     LIB_POSSIZE(a5),d0
    CALLLIB   FreeMem
    move.l    d2,d0

ExpungeEnd:
    movem.l   (sp)+,d2/a5/a6
    rts

```

Nicht zu vergessen, auch noch die reservierte Funktion:

```

ExtFunc:
    moveq     #0,d0
    rts

```

Das war auch schon die ganze Vorarbeit. Jetzt binden wir unsere Funktionen ein:

```

WaitMouse:
    btst      #6,$bfe001
    bne.s     NoLeft
    moveq     #1,d0
    rts

NoLeft:
    btst      #2,$dff016
    bne.s     WaitMouse
    moveq     #0,d0
    rts

```

Bei der Funktion WaitMouse ist zu beachten, daß das nicht gerade die feine englische Art ist. Eigentlich sollte hier auf die Daten nicht direkt über die Hardware, sondern über das GamePort.Device zugegriffen werden. Da es hier allerdings um die Libraries geht, wollen wir diese kleine Nebensächlichkeit übersehen.

Als nächstes folgt die Routine Blink:

```
Blink:
        movem.l      d4/a0/a5/a6, -(sp)
        move.l       a6, a5
        move.l       d0, d4
dbl:    move.l       sb_IntuiLib(a5), a6
        sub.l        a0, a0
        CALLLIB      DisplayBeep
        move.l       sb_DosLib(a5), a6
        moveq        #2, d1
        CALLLIB      Delay
        dbra         d4, dbl
        moveq        #0, d0
        movem.l      (sp)+, d4/a0/a5/a6
        rts
```

Um die Library fertigzustellen, fehlen jetzt nur noch zwei Zeilen:

```
EndLib:
        END
```

Das war's. Dies kann jetzt assembliert und gelinkt werden, danach landet das fertig File im LIBS:-Directory und ist fertig zum Einsatz.

Fehlt eigentlich nur noch ein Programm, das die Library testet:

```
#include <exec/types.h>
#include <exec/libraries.h>
#include <libraries/dos.h>
#include "mylibbase.h"
struct MyLibBase *MyLibBase, *OpenLibrary();
main()
{
    long g;
    struct Library *lib;
    if (!(MyLibBase=OpenLibrary(MYLIBNAME, 0L))) {
        printf("Couldn't open...\n");
        exit(-1L);
    }
    printf("Ich warte...\n");
    g=WaitMouse();
```

```
printf("WaitMouse: %ld\n",g);  
Blink(3L);  
CloseLibrary(MyLibBase);  
exit(0L);  
}
```

Wird das Programm gestartet, so wird erst auf einen Druck auf die linke oder rechte Maustaste gewartet, danach blinkt der Bildschirm dreimal – und das alles mit unserer eigenen Library!

Die Tasks

Noch vor vier Jahren war das Betreiben mehrerer Programme gleichzeitig, also das Multitasking, für Benutzer von Heimcomputern eine Zukunftsvision.

Zwar gab es schon eine Reihe Programme, die durch Abbarbeiten von kleineren Routinen während eines Interrupts das Feeling von paralleler Verarbeitung aufkommen ließen, doch waren das keine vollwertigen Programme, sondern eben nur einzelne Routinen.

Das wahre Multitasking für den Heimbereich kam erst mit dem Amiga auf. Das Betriebssystem des Amigas hat im Bezug auf die Verarbeitung mehrerer paralleler Aufgaben Leistungsmerkmale, die man bis dahin eigentlich nur Rechnern von Workstations aufwärts zutraute.

Doch was ist eigentlich ein Task? Unser Institut für Normung hat dafür folgende, einprägsame (!?) DIN-Norm definiert:

Task (Prozeß):

Arbeits- und Verwaltungseinheit, deren Ablauf in der Ausführung eines strikt sequentiellen Programms mit unbestimmter aber endlicher Geschwindigkeit in einem gegebenen Adreßraum besteht.

(DIN 66201)

Stellt sich die Frage: Was wollten uns die Damen und Herren damit sagen? Auf deutsch heißt das etwa folgendes:

Ein Task ist der kleinste Teil eines Programms, der einen zusammenhängenden Adreßraum belegt. So gesehen fahren wir also auf dem Amiga keineswegs Tasks, sondern irgendetwas anderes, denn Programme können im Speicher des Amigas kreuz und quer verteilt sein. (Hilfe, mein Amiga entspricht nicht der DIN-Norm...)

In folgendem Bild wollen wir uns ansehen, wie solch eine parallele Verarbeitung ablaufen könnte (hier bei drei Tasks):

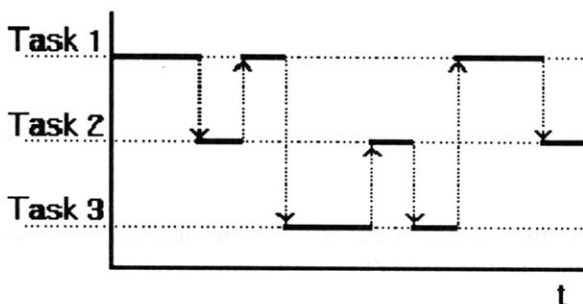


Abb. 1.1: Task-Wechsel

Zur Verwaltung eines Tasks werden eine Reihe Daten gebraucht:

- Beschreibung (Name, Art, Priorität, ...)
- Laufparameter (Registerinhalte, Zustand des Prozessors, ...)
- Verwaltungsdaten (belegte Speicherbereiche, Wartelisten, ...)

Abhängig vom Betriebssystem kann dabei die Liste der Daten größer oder kleiner sein, doch existieren wird ein solcher Verwaltungsblock für Tasks in jedem Multitasking-System. Beim Amiga werden dabei, neben einer Reihe interner Task-Daten, Zeiger auf den Stack, Anfang und Ende des Stacks sowie die Liste der Memory-Blöcke verwaltet.

Zustände eines Tasks

Ein Task kann verschiedene Zustände annehmen. Diese sind:

- | | |
|---------|---|
| Waiting | Der Task wartet auf irgendein Ereignis oder die Zuteilung eines Betriebsmittels (Disk, Drucker, Port...). |
|---------|---|

Ready	Der Task ist bereit, weiterzuarbeiten und wartet nur noch auf die Zuteilung der CPU.
Active	Der Task arbeitet zur Zeit.
Suspended	Der Task wurde angehalten (z.B. wegen eines Fehlers, zum Debuggen, ...)

Folgendes Bild verdeutlicht den "Kreislauf", in dem sich Tasks befinden. Ein Pfeil verdeutlicht jeweils einen möglichen Zustandswechsel.

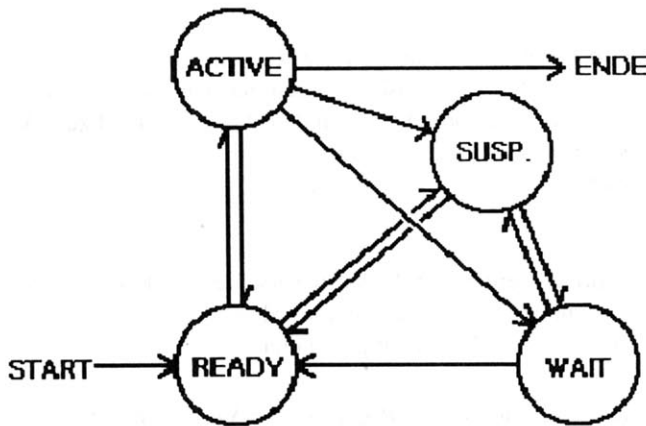


Abb. 1.2: Zustände der Tasks

Steuerung der Tasks

Um die konkurrierende, also quasi-parallele, Verarbeitung verschiedener Tasks zu ermöglichen, muß man dafür sorgen, daß ein übergeordnetes Programm einen Task unterbricht (also in den Zustand **READY** zurückführt), um einen anderen Task **ACTIVE** werden zu lassen.

Dieses Programm ist der Task-Dispatcher. Ein zweites Programm übernimmt die Vergabe der Plätze in der **READY**-Liste: der Scheduler (Aussprache etwa: Skedschuler). Er sorgt dafür, daß Tasks, deren Bedingung zum Weiterarbeiten erfüllt ist, einen Platz weiterrücken. Bei dem Task-Switching, also der Umschaltung **READY/ACTIVE**, gibt es verschiedene Methoden, nach denen man vorgehen kann. Einige gebräuchliche sind:

sequentiell Hierbei werden die READY-Tasks nach dem FIFO-Prinzip (der erste Task, der in die Warteliste eingetragen wurde, wird auch als erster wieder herausgenommen) in den Zustand ACTIVE überführt. Falls noch Prioritäten vergeben werden, werden neue READY-Tasks jeweils hinter dem letzten Task ihrer Priorität eingereiht (statt am Ende der Liste). Der Nachteil dieser Methode ist, daß Tasks nicht unterbrochen werden können. Somit blockieren Tasks mit hoher Rechenzeit andere, die nur wenig rechnen und eigentlich schnell fertig sein könnten.

Zyklische Verfahren:

Round-Robin Hierbei wird jedem Task eine gewisse Rechenzeit (=Zeitscheibe) zugeordnet. Alle READY-Tasks werden nacheinander bis zum Ende ihrer Zeitscheibe (oder bis sie in den WAIT-Zustand zurückkehren) bearbeitet. Bei diesem Verfahren ist keine Prioritätsvergabe möglich.

zyklisch mit Priorität

Je nach Priorität werden die Tasks in verschiedene Listen eingetragen, die dann zyklisch bearbeitet werden. Ist eine Liste leer, wird auf die nächst niedrigere umgeschaltet.

Beim Amiga wird das Task-Switching nach einem Round-Robin-Verfahren mit Prioritätsvergabe vorgenommen: Alle Tasks gleicher Priorität erhalten bestimmte Zeiten (also nicht bis zum nächsten WAIT), in denen sie arbeiten können.

3 – 2 – 1 – Feuer! Wir starten einen Task...

Um einen neuen Task zu starten, müssen erst einige Vorarbeiten verrichtet werden. Zuerst muß eine Task-Struktur vorhanden sein. Diese wird initialisiert und in Exec eingebunden. Im einzelnen ist folgendes zu tun:

- Anlegen der Task-Struktur (incl. Speicher für den Task-Namen). Dies sollte mit AllocMem() getan werden; als Flags für den Speicher empfiehlt sich MEMF_PUBLIC, MEMF_CLEAR sollte auf jeden Fall gesetzt werden.
- Reservieren des Stack-Speichers.

- Als letztes müssen nun verschiedene Werte in der Task-Struktur gesetzt werden. Diese sind:

tc_SPLower	untere Stack-Grenze
tc_SPUpper	obere Stack-Grenze
tc_SPReq	Stack Pointer (am Anfang=tc_SPUpper)
tc_Node.In_Type	Art (NT_TASK)
tc_Node.In_Name	Name des Tasks

- Aufruf der Routine AddTask mit der Adresse der Task-Struktur, der ersten Adresse des Tasks sowie der Endadresse des Tasks (die auf den Stack des Tasks geschoben wird, so daß sie als Return-Adresse funktioniert).

... und halten ihn wieder an

Wurde als Endadresse eines Tasks 0 angegeben, so werden vom System alle "Aufräumarbeiten" übernommen. Hat man jedoch eine eigene Routine angegeben, so kann man noch einiges mehr tun. Man denke da zum Beispiel an eine Routine, die während einer Textverarbeitung parallel zum eigentlichen Programm läuft und die Rechtschreibung überprüft, oder an ein Programm, das nebenbei mathematische Berechnungen anstellt.

In beiden Fällen könnte das Hauptprogramm noch vor Freigabe des Speichers die gesammelten Daten sichern oder verarbeiten. Egal, was man macht, man muß sich natürlich um alle Speicherblöcke kümmern und sie freigeben. Während man allerdings Dinge wie Libraries und Devices ohne Probleme vom Vater-Prozeß holen kann, muß man beachten, daß Signale nur vom jeweiligen Task geholt werden können.

Interprozeßkommunikation oder: CB-Funk für Tasks

Wenn mehrere Leute an einer Sache arbeiten, dann wird es von Zeit zu Zeit notwendig, sich über den Stand der Dinge auf dem Laufenden zu halten, damit nicht ein Task Daten verarbeitet, die der andere noch gar nicht erzeugt hat.

Sehen wir uns dazu einmal ein kleines Fallbeispiel an: ein einfaches Erzeuger/Verbraucher-Problem. Zwei Tasks laufen nebeneinander, wobei der eine Daten erzeugt, der andere verbraucht sie (gibt sie also beispielsweise zum Drucker). Beide verfügen über einen gemeinsamen Speicherbereich von 1 KByte, in dem die Daten enthalten sind. Ohne Kommunikation könnte folgendes passieren:

*Task 1 (Erz.)**Task 2 (Verb.)*

Füllt Speicher

Startet Task 2

Füllt Speicher bis

512 Byte

Füllt Speicher bis

Ende

Gibt Speicher aus

Gibt Speicher (1 KByte) aus

Gibt Speicher (1 KByte) aus

Hier sieht wohl jeder, daß die Ausgabe-Daten in keiner Weise mit Daten übereinstimmen, die man eigentlich vom Erzeuger haben will.

Um dies zu verhindern, bedient man sich der Signale. Diese werden vom System für viele Aufgaben verwendet, doch letztlich sorgen sie immer für eine gute Verständigung zwischen verschiedenen Tasks (auf Interrupts!). Diese Signale werden unabhängig verwaltet, d.h. nur der "Besitzer" eines Signals sowie die Tasks, die seine Bedeutung kennen, werden über ein Signal informiert.

Dies ist etwa vergleichbar mit einem Funkgerät, das 32 Kanäle hat, und anderen Funkgeräten, die wissen müssen, auf welchem Kanal ihr Signal kommt, um es zu empfangen.

Signale bilden bei der Kommunikation zwischen Tasks die unterste Stufe; sie werden z.B. intern auch beim Message-System verwendet.

Wir beantragen einen Kanal ...

Bevor wir ein solches Signal verwenden können, muß es beim System "beantragt" werden, nämlich mit `AllocSignal`. Diese Funktion erwartet als Parameter die Nummer des Signals, das man verwenden möchte. Normalerweise sollte man hier `-1` übergeben, damit man einfach irgendein Signal bekommt. Sollte man dennoch ein bestimmtes Signal haben wollen, so braucht man gar nicht erst zu versuchen, eines der unteren 16 zu bekommen, da diese für das System verwendet werden. Ist noch ein Signal (bzw. das bestimmte Signal) frei, so wird man die Nummer des Signals zurückbekommen. Ansonsten kehrt `AllocSignal` mit `-1` zurück. Um dieses Signal nun zu verwenden, sollte man darauf achten, daß dieses Signal die Nummer des Signals ist. Somit muß man erst in eine Bit-Maske verwandeln:

```
Maske = 1 << Signal;
```

Sobald ein Signal nicht mehr benötigt wird, sollte man es dem System mit FreeSignal zurückgeben.

Warten auf das Signal ...

Einem gut programmierten Task wird es im Amiga nicht so gehen wie der Gruppe, die einst folgende Zeile in ihrem Repertoire hatte: "Ich steh' in der Kälte und wart' auf 'n Taxi, aber's kommt nicht". Unser Task macht in der Zeit einfach einen kleinen Winterschlaf, sprich: er fällt in den WAIT-State zurück und verbraucht somit keine Rechnerzeit. Das Signal, auf das er wartet, wirkt somit wie eine Art Wecker. Damit es auch wirklich interessant wird, kann man sich von verschiedenen Dingen wecken lassen:

```
Signal = Wait (Maskel | Maske2 | Maske3)
```

Wird eines der Signale gesetzt, so kehrt Wait mit der Maske des Signals zurück. Da jedoch mehr als ein Signal gesetzt sein kann, kann man die Abfrage nicht mit SWITCH aufbauen, sondern muß alle Bits einzeln abfragen:

```
if (Signal & Maskel) {  
    printf("Task beendet\n");  
    :  
}  
if (Signal & Maske2) {  
    :  
    :  
}
```

Sohn an Vater: bitte kommen... oder: Wie man Signale erzeugt

Signale empfangen und entscheiden, was sie eigentlich bedeuten, können wir jetzt, aber wie erzeugt man sie? Nun, nichts einfacher als das: Dazu verwendet man die Funktion Signal, der man einen Zeiger auf die Task-Struktur des Empfängers sowie die Bit-Maske der Signale übergibt, also z.B.:

```
Signal (Vater, 0x40000);
```

welches das Signal Nummer 18 setzt.

Ampeln im Amiga oder: Wir stoppen Tasks ...

Bei einigen Operationen im System sollte man dafür sorgen, daß kein anderer Task oder Interrupt auf Daten zugreift. Dies ist oft der Fall bei Auslesen oder Ändern von System-Listen. Zu diesem Zweck gibt es zwei System-Routinen, mit denen man andere Tasks anhalten kann. Die Erste ist `Forbid()`. Hiermit wird das Task-Scheduling angehalten, so daß das Multitasking solange abgeschaltet wird, wie der aufrufende Task `RUNNING` ist. Interrupts sind aber weiterhin erlaubt.

Kehrt der Task in den `WAITING`-Status zurück, wird das Multitasking solange wieder aufgenommen, bis der Task wieder `READY` ist. Dann werden wieder alle Tasks angehalten. Um das Task-Switching erneut einzuschalten, muß der Aufruf `Permit` verwendet werden.

Die `Forbid/Permit`-Bereiche können durchaus geschachtelt sein. In dem Fall wird erst dann wieder auf das normale Multitasking zurückgeschaltet, wenn der letzte `Forbid` durch `Permit` beendet wurde.

Doch Achtung: `Forbid` hält nur andere Tasks auf, Interrupts können weiterhin ungehindert auftreten. Um diese noch zu verhindern, braucht man den Befehl `Disable`. Hiermit wird wirklich alles angehalten. Um die anderen Tasks und Interrupts weiterlaufen zu lassen, benutzt man den Befehl `Enable`.

Ähnlich wie auch `Forbid` werden alle Tasks gestartet, sobald der aktuelle in den `Wait`-Status zurückkehrt. Auf eines muß man jedoch achten: Wird das Multitasking zu lange angehalten, so kann es ernste Probleme geben. Nach dem "Rom Kernel Reference Manual: Exec" sollte man möglichst unter 250 Mikrosekunden bleiben.

Ausnahmen – aber nur nach den Regeln!

Um auf bestimmte Situationen besonders schnell und von jeder beliebigen Position im Programm aus reagieren zu können, hat man die Möglichkeit, Exceptions, also Ausnahmen, zu definieren. Dieser Interrupt wird durch die Funktion `Sig_Except` aufgerufen. Die Funktion verwendet die Daten, die in Feldern `tc_ExceptCode` (Adresse der Exception-Routine) und `tc_ExceptData` (Pointer auf zusätzliche Datenfelder) der Task-Struktur stehen. Als Übergabeparameter erhält das Exception-Programm in `D0` die Maske der Exception, `A1` enthält den Pointer aus `tc_ExceptData`. Auf dem Stack kann man die Register des "normalen" Taskverlaufs finden (inclusive PC und Status-Register).

Durch einen besonderen Service von Exec braucht man sich nicht darum zu kümmern, daß der Exception-Code mehrfach aufgerufen wird. Als Rückgabewert sollte in D0 derselbe Wert wie beim Aufruf zurückgegeben werden.

Wird der Exception-Code mit RTE beendet, so stellt das System die gesamte Umgebung wieder auf den Zustand, wie er beim Aufruf der Routine ursprünglich war.

Vorsicht Falle!

Als nächstes wollen wir uns ansehen, wie man neben der Benutzung der Task-Exceptions noch weitere Exceptions be- und ausnutzen kann. Diese Ausnahme-Routinen werden vom Prozessor direkt verwaltet und heißen TRAPS, also Fallen. Sobald ein Befehl in eine solche Falle tappt, wird ein Programm aufgerufen, um darauf zu reagieren. Doch welche Traps gibt es? Zu diesem Zweck sehen wir uns folgende Tabelle an:

<i>Trap</i>	<i>Bedeutung</i>
(0)	Reset: Initial SSP
(1)	Reset: Initial PC
2	Bus Error
3	Address Error
4	Illegal Instruction
5	Divide by Zero
6	CHK Instruction
7	TRAPV Instruction
8	Pivilege Violation
9	Trace
10	Line-A Emulation
11	Line-F Emulation
12-14	(Unassigned, reserved)
15	Uninitialized Interrupt-Vector
16-23	(Unassigned, reserved)
24	Spurious Interrupt
25-31	Interrupt Autovectors (Level 1-7)
32-47	TRAP Instruction Vectors
48-63	(Unassigned, reserved)
64-255	User Interrupt Vectors

Wie man sieht, sind eine Reihe Traps schon belegt, andere sind zwar nicht belegt, sollten aber in weiser Voraussicht nicht verwendet werden (andere Betriebssystem-Version,

anderer Prozessor). Bleiben also neben den User-Interrupt-Vektoren noch die ausdrücklich für die Traps vorbehaltenen Vektoren.

Zuerst einmal ein Wort zur Programmiersprache. Zwar ist es möglich, Trap Handler in C zu schreiben, doch würde ich persönlich dazu tendieren, solche Routinen ausschließlich in Assembler zu schreiben. Außerdem sollte man sich überlegen, ob man wirklich Traps verwenden muß (die ja im Prinzip Erweiterungen des Sprachumfangs des Prozessors sind).

Doch zurück zu den Traps. Ist man sich wirklich sicher, einen Trap benutzen zu müssen, sollte man zuerst einmal dem System Bescheid sagen, sprich: den Befehl AllocTrap absetzen, um so ein freies Trap-Register zu erhalten. Weiterhin gilt es zu verhindern, daß ein anderes (ebenfalls ordentlich programmiertes) Programm sich eben jenen Trap geben läßt und den Inhalt so umändert, daß unser Programm sozusagen "leicht verwirrt aus der Wäsche gucken" wird. Eine kurze Anmerkung noch zu AllocTrap: Am Ende des Programms sollte man den Trap wieder freigeben, nämlich mit FreeTrap. Der Ablauf für das Anlegen eines Traps sieht nun etwa folgendermaßen aus:

```
move.l    $4,a6
moveq     #-1,d0          ; irgendeinen Trap
jsr       AllocTrap(a6)
tst.l     d0              ; Return-Wert
bmi.s     Error_No_Trap   ; <0, Fehler
addq      #32,d0          ; wahre Nummer
asl.l     #2,d0           ; Adresse des Offsets
move.l     d0,a0
move.l     #ErrorHandler,(a0) ; Handler-Adresse
```

Damit ist der Trap-Handler installiert und kann verwendet werden. (Es sei denn, es war kein Trap mehr frei.) Braucht man aus programmtechnischen Gründen einen bestimmten Trap, kann man vor dem Aufruf von AllocTrap die Nummer des Traps in D0 laden.

Interrupts

Der einzige Weg, wie man auf "kleinen" Rechnern wie Heimcomputern oder PCs kleine Programme nebenbei abarbeiten kann, sind die Interrupts. Viele Amiga-Freaks (auch ich) kriegen heute noch leuchtende Augen, wenn sie an die guten alten Zeiten auf ihrem C64 denken, das Umstellen des Vektors bei \$314/\$315, Beenden der Routine mit JMP \$EA31, ach, war das doch alles einfach...

Bei einem System wie dem Amiga hat man da zwar eine Menge mehr Möglichkeiten, doch ist das Ganze schon ein paar Schritte schwieriger. Doch kommen wir zuerst einmal zu den Interrupts. Folgende Interrupts treten im Amiga-System auf:

<i>Interrupt</i>	<i>Level</i>	<i>Pseudo-Prio.</i>	<i>Art</i>
NMI	7	15	Nichtmaskierbarer Interrupt
INTEN	6	14	Copper
EXTER	6	13	8520B, externe Ebene 6
DSKSYNC	5	12	Disk Byte
RBF	5	11	Serial Input
AUD3	4	10	Audio Kanal 3
AUD2	4	9	Audio Kanal 2
AUD1	4	8	Audio Kanal 1
AUD0	4	7	Audio Kanal 0
BLIT	3	6	Blitter fertig
VERTB	3	5	Vertical Blank
COPER	3	4	Copper
PORTS	2	3	8520A, externe Ebene 2
TBE	1	2	Serial Output
DSKBLK	1	1	Disk Block fertig
SOFTINT	1	0	Software Interrupts

Dabei können Interrupts nur von Interrupts höherer Level (=Priorität) unterbrochen werden. Außerdem sollte beachtet werden, daß ein Interrupt-Code so wenig Zeit wie möglich in Anspruch nehmen sollte, um die niedrigeren Interrupts nicht zu blockieren. Sollten größere Aufgaben bevorstehen, so sollte entweder ein Software-Interrupt verwendet werden oder im Interrupt-Code nur noch ein Signal gesetzt werden, so daß ein anderer Task die Funktionen übernimmt.

Bitte nicht unterbrechen...

Der NMI, der Non-Maskable-Interrupt (nicht maskierbar, also nicht zu verhindern), ist ein etwas ungewöhnlicherer Interrupt. Er kann nicht vom System selber erzeugt werden, außerdem läuft er nicht durch den 4703-Interrupt-Chip. Sollte ein solcher Interrupt von externer Hardware erzeugt werden, so kann sich ein Server auf nichts verlassen – das System wurde an irgendeiner Stelle angehalten, die Inhalte der Register enthalten irgendwas, nur keine sinnvollen Werte. Auch wird die Disable-Sperre nicht beachtet. Ein solcher Interrupt-Server muß also "ganz unten" anfangen.

Bearbeiten von Interrupts

Im Amiga werden Interrupts von Handlern und Servern bearbeitet. Der Unterschied zwischen den beiden besteht darin, daß ein Handler einen Interrupt exklusiv bearbeitet, während mehrere Server von einem einzigen Interrupt aufgerufen werden können.

Beim Hochfahren des Systems werden für die Interrupts PORTS, COPER, VERTB, BLIT, EXTER und den NMI Server-Ketten eingerichtet, alle anderen Interrupts werden von Handlern bearbeitet.

Definiert werden die Server und Handler durch folgende Struktur:

```
struct Interrupt {
    struct Node      is_Node;
    APTR             is_Data;
    VOID             (*is_Code) ();
};
```

Zur Initialisierung müssen zum einen Werte in der Node-Struktur gesetzt werden (`is_Node.ln_Type = NT_INTERRUPT`, `ln_Pri = z.B. 0`), zum anderen müssen die Adressen `is_Data` und `is_Code` eingetragen werden. Danach kann man sie mit der Routine `SetIntVector` einbinden, die die Adresse des letzten Handlers zurückgibt. Um den eigenen Interrupt wieder zu entfernen, muß nur `SetIntVector` mit dieser Adresse aufgerufen werden. Zu beachten ist allerdings, daß dies alles nur für Interrupt-Handler gilt. Für Interrupt-Server muß die Funktion `AddIntServer` (bzw. `RemIntServer` zum Entfernen) verwendet werden.

Wenn ein Interrupt ausgelöst und somit ein Server oder Handler gestartet wurde, kann man von folgender Belegung der Register ausgehen:

Register	Übergabe
D0	—
D1	(Handler: INTENA&INTREQ, Server: -)
A0	(H: Adr. CustomChips, S:-)
A1	<code>is_Data</code> -Zeiger
A5	wird beim Aufruf verwendet
A6	(H: ExecBase, S:-)

Diese Register brauchen nicht wieder hergestellt zu werden. Zu beachten ist noch, daß in einer Server-Kette das Z-Flag des Prozessors am Ende eines Server-Durchlaufs ge-

setzt werden muß, da sonst keine weiteren Server aus der Kette aufgerufen werden. Außerdem muß wegen eines Fehlers in der GraphicsLibrary bei Servern mit Priorität >10 in A0 die Adresse der Custom-Chips (= \$DFF000) zurückgegeben werden.

Software-Interrupts

Um auch größere Programmteile noch als einen Interrupt ablaufen zu lassen, gibt es die Software-Interrupts. Diese werden mit einer Priorität größer als der des normalen Tasks, aber kleiner als der von Hardware-Interrupts abgearbeitet.

Dabei wird dieselbe Interrupt-Struktur verwendet. Aufgerufen wird der SI (=Software-Interrupt) mittels der Funktion Cause. Wurde diese von einem normalen Task aus aufgerufen, so wird dieser sofort unterbrochen und der SI-Code angesprungen. Wurde Cause() von einem Hardware-Interrupt aus aufgerufen, wird sie nach Beendigung des letzten Hardware-Interrupts gestartet.

Neben Cause() kann man auch mit Messages, die von Ports kommen, einen Software-Interrupt auslösen. Dabei wird das Flag PA_SOFTINT gesetzt. Somit könnte man Menü- oder Gadget-Aufrufe in einem Interrupt abarbeiten, während das Programm normal weiterläuft.

Software-Interrupts werden, genau wie Interrupt-Server auch, über eine Priorität verwaltet. Doch sind hier nur fünf verschiedene Prioritäten möglich: 32, 16, 0, -16 und -32. Dabei werden Werte, die außerhalb des Bereichs liegen, ignoriert.

"Störung unerwünscht"

Da es oft mal Routinen gibt, die möglichst nicht durch Interrupts angehalten werden dürfen, hat man natürlich auch die Möglichkeit, die Interrupts abzuschalten.

Dazu gibt es die beiden Befehle Disable und Enable. Diese Funktionen erfüllen folgende Aufgaben:

- Interrupts werden durch Ändern des Hardware-Registers INTENA (\$DFF09A) erlaubt (\$C000) bzw. verboten (\$4000).
- Durch Benutzen des IDNestCnt-Wertes in der ExecBase ist es möglich, Disable/Enable-Paare bis zu einer Tiefe von 126 zu schachteln.

Im Assembler-Code sehen die Makros so aus:

```
DISABLE    MACRO    * [scratchReg]
            IFC      '\1', ''
            MOVE.W   #$04000, _intena
            ADDQ.B   #1, IDNestCnt (A6)
            ENDC
            :
            ENDM
ENABLE     MACRO    * [scratchReg]
            IFC      '\1', ''
            SUBQ.B   #1, IDNestCnt (A6)
            BGE.S    ENABLE\@
            MOVE.W   #$0C000, _intena
ENABLE\@:
            ENDC
            :
            ENDM
```

Messages und Ports

Kommunikation im System

Um eine Abstimmung von Tasks untereinander oder den Austausch von Daten zu ermöglichen, braucht jedes Multitasking-System eine Art Message-System. Beim Amiga bemüht man dazu die Ports und Messages. Dabei sind Ports die Ankunfts- und Abfahrts-Orte, von denen oder an die Messages gehen. Dabei ist das ganze System so flexibel und schnell, daß es praktisch überall verwendet wird, wenn Daten zwischen den Tasks zu übergeben sind.

Widmen wir uns nun zuerst den Ports. Die Port-Struktur sieht folgendermaßen aus:

```
struct MsgPort {
    struct Node mp_Node;
    UBYTE   mp_Flags;
    UBYTE   mp_SigBit;
    struct Task *mp_SigTask;
    struct List mp_MsgList;
};
```

mp_Node	Eine normale Node-Struktur (siehe Listenverwaltung).
mp_Flags	Hier werden verschiedene Aktionen bei der Ankunft der Message eingestellt. Diese sind: PA_SIGNAL Bei Ankunft der Message wird ein Signal ausgelöst. Dies geschieht unabhängig davon, wieviele Messages schon angekommen sind. PA_SOFTINT entspricht PA_SIGNAL, hier wird lediglich ein Interrupt ausgelöst. PA_IGNORE Die Message wird nur in die Message-Liste des Empfänger-Ports angehängt.
mp_SigBit	Dies ist die Nummer des Signals, mittels dem die Ankunft einer Message signalisiert wird.
mp_SigTask	Hier steht ein Zeiger auf die Task-Struktur des Tasks, der benachrichtigt werden soll. Sollen statt Signalen Interrupts ausgelöst werden, so muß hier ein Zeiger auf eine Interrupt-Struktur stehen.
mp_MsgList	Hier findet man den Listen-Kopf für die eingegangenen Messages (siehe Listenverwaltung).

Achtung! Da bei jeder Message das Signal-Bit gesetzt bzw. ein Interrupt ausgelöst wird (sofern dies in mp_Flags eingestellt ist), diese jedoch nur binäre Information enthalten, kann man nicht einfach bei gesetztem Bit oder im Interrupt die erste Message abholen, sondern man muß alle eingegangenen Messages sichten.

Wir bauen einen Hafen...

Um jedoch Messages überhaupt benutzen zu können, muß vorher ein Port angelegt werden. Dazu gibt es die Routine CreatePort, die folgende Aufgaben erfüllt:

- Belegen eines Signals sowie des benötigten Speichers.
- Definieren der Node-Struktur (Name, Priorität, Art).
- Setzen von mp_Flags, mp_SigBit und mp_SigTask.
- Aufruf der Funktion AddPort() zum eigentlichen Anlegen des Ports selber

Will man allerdings einen Port haben, der im System nicht mit FindPort zu finden sein muß, dann reicht zum Einbinden auch ein Aufruf von:

```
NewList (&port->mp_MsgList)
```

Das Gegenstück zu AddPort, DeletePort, löscht den Port mit RemPort und gibt den Speicher sowie das Signal wieder frei.

Ab geht die Post...

Kommen wir nun zu der Message. Der Kopf jeder Message sieht so aus:

```
struct Message {
    struct Node mn_Node;
    struct MsgPort *mn_ReplyPort;
    UWORD mn_Length;
};
```

mn_Node Eine normale Node-Struktur für die Listenverwaltung.

mn_ReplyPort Der Port, an den die Rückantwort gehen soll.

mn_Length Die Länge der Daten, also der eigentlichen Nachricht, in Bytes.

Will man nun eine Message "verschicken", so braucht man nur die eigentlichen Daten an eine Message-Struktur anhängen:

```
struct MyMes {
    struct Message msg;
    struct RastPort *rp;
    int width, height, depth;
    int command;
} *mmsg;
```

Vor dem Absenden muß man noch die Adresse des Reply-Ports und die Art der Node eintragen (`mmsg->msg.mn_ReplyPort` und `mmsg->msg.mn_Node.In_Type`). Eine Initialisierungs-Routine für das Absenden einer Message könnte so aussehen:

```
struct MsgPort *mp;
struct MyMsg *mmsg, *AllocMem();

PrepareMsg()
{
    if ((mp = CreatePort( NULL, 0L)) &&
        mmsg = AllocMem( (long)sizeof(struct MyMsg),
                        MEMF_PUBLIC|MEMF_CLEAR))
    {
        mmsg->msg.mn_ReplyPort = mp;
        mmsg->msg.mn_Node.ln_Type = NT_MESSAGE;
        return(1);
    }
    else {
        cleanup();
        return(0);
    }
}
```

Abschicken und Empfangen

Um die fertige Message abzuschicken, verwendet man die Routine PutMsg(). Diese erhält als Parameter den Port, an den die Message gehen soll, sowie die Adresse der Message-Struktur:

```
mp = FindPort ("MyClient");
if (mp) {
    PutMsg(mp, mmsg);
}
```

Da die Message nicht kopiert wird, muß der Absender dafür sorgen, daß die Message-Struktur nicht verwendet wird, bevor nicht ein ReplyMsg() erfolgt ist.

Um eine Message abzuholen, sind zwei Befehle vorhanden. Zum einen der Befehl WaitPort(), zum anderen GetMsg().

Bei WaitPort() wird der Task in den Status WAIT zurückgestellt, so daß keine Rechnerleistung verbraucht wird. Allerdings muß dazu das mp_SigTask und mp_SigBit der Port-Struktur richtig vorbereitet sein (siehe Beschreibung im Abschnitt über die Tasks). Als Return-Wert liefert WaitPort die Adresse der ersten Message, wobei die Message selbst jedoch nicht aus der Liste herausgenommen wird. Dies muß noch mit GetMsg() gemacht werden.

Im Programm könnte die Abfrage eines Ports folgendermaßen aussehen:

```
while (RUN==TRUE) {  
  
    WaitPort(myport);  
    while (msg = GetMsg(myport)) {  
        :  
        :  
        ReplyMsg(msg);  
    }  
}
```

Wichtig ist dabei zum einen, daß WaitPort nur auf neu angekommene Messages reagiert, so daß man die ganze Liste mit GetMsg() leeren muß. Außerdem müssen in der Regel alle Messages beantwortet werden, was mit ReplyMsg() geschieht. Damit wird der Absender darüber informiert, daß die Message angekommen ist und somit neu verwendet werden kann. Achtung! Messages dürfen erst bestätigt werden, wenn sie nicht mehr gebraucht werden, da der Speicher sonst unter Umständen falsche Werte enthält!

IO

Allgemeine Ein- und Ausgaben

Alle Devices des Amiga-Systems haben eines gemeinsam: Sie sind alle mittels der selben IO-Struktur (IO=Input/Output = Eingabe/Ausgabe) anzusprechen.

Ein gutes Beispiel dafür ist der PD-Diskettenmonitor Sectorama, der jedes beliebige Laufwerk editieren kann, solange ein Gerätetreiber vorhanden ist (neben DF0: bis DF3: alle Festplatten und Ram-Disks mit fester Speicherbelegung).

Somit bedeutet das, daß jeder Gerätetreiber eine Verbindung zwischen einer abstrakten Ansteuerung und der physikalischen Seite herstellt.

Die allgemeine Seite benötigt erst einmal eine Struktur, in der einige Dinge beschrieben bzw. verankert sind. Diese sieht so aus:

```
struct IORequest {  
    struct Message io_Message;  
    struct Device *io_Device;  
    struct Unit *io_Unit;  
    UWORD io_Command;  
    UBYTE io_Flags;  
    BYTE io_Error;  
};
```

- io_Message** Eine ganz normale Message-Struktur, die zum einen dazu dient, einen beendeten IORequest zu melden, zum anderen, um die Requests intern zu verwalten.
- io_Device** Dieses Feld wird automatisch von einer Exec-Routine beim Öffnen des Device vorbereitet.
- io_Unit** Hier stehen system-interne Daten, die abhängig von dem jeweiligen Gerätetreiber sind und nicht von eigenen Programmen angesprochen werden sollten.
- io_Command** In dieses Feld wird der eigentliche Befehl eingetragen. Je nach angesprochenem Device können das Standard-Befehle oder device-spezifische Kommandos sein.
- io_Flags** Hier trägt man bestimmte Bits ein, um andere Modi oder Optionen auszuwählen. Dabei sind die unteren vier Bits für Exec reserviert, die anderen vier Bits sind für das Device verfügbar.
- io_Error** Gab es bei der Durchführung eines IORequests Probleme, so wird hier der Code eines Fehlers eingetragen.

Neben der IORequest-Struktur gibt es noch eine weitere, den 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;  
};
```

Die zusätzlichen Felder haben dabei folgende Bedeutung:

io_Actual	Anzahl der übertragenen Bytes. Achtung! Nur gültig, falls der Request ordnungsgemäß beendet wurde.
io_Length	Anzahl der zu übertragenden Bytes.
io_Data	Zeiger auf den Datenpuffer.
io_Offset	Offset, ab dem der IOStdReq durchgeführt werden soll. Bei Disketten muß dieser beispielsweise ein Vielfaches der Blockgröße sein.

Zugriff auf die Devices

Um auf die Devices zugreifen zu können, stellt Exec folgende vier Routinen zur Verfügung, die als Parameter alle den Zeiger auf eine IORequest-Struktur (auch erweiterte wie IOStdReq) erhalten:

DoIO()	DoIO dürfte die wohl am meisten verwendete Routine sein. Sie übernimmt das Absenden des Requests und wartet, bis der Device-Treiber den Auftrag erledigt hat. Erst dann kann der aufrufende Task weiterarbeiten.
SendIO()	Entspricht DoIO, doch wird hier nicht auf das Ende des Requests gewartet. Somit kann der Task weiterarbeiten, während der Treiber die Arbeit macht.
WaitIO()	Diese Routine wird benutzt, um auf das Ende eines mit SendIO gestarteten Requests zu warten. Wie auch DoIO() kehrt sie erst nach Beendigung des Requests zurück.
CheckIO()	Hiermit prüft man, ob ein mit SendIO() gestarteter Request schon fertig ist. Erhält man als Return-Wert NULL, so ist er noch nicht fertig; ansonsten bekommt man den Zeiger auf die IORequest-Struktur.

Zusätzlich zu diesen Routinen gibt es noch zwei weitere, die direkt Vektoren im Device-Treiber anspringen, nämlich BeginIO und AbortIO. Da diese Funktionen kaum Geschwindigkeitsvorteile bringen und nur bei guter Kenntnis des Device-Treibers angewendet werden sollten, wird hier nicht näher darauf eingegangen.

Standard-IO-Befehle

Vom Amiga-Betriebssystem werden zur Zeit (bis Kickstart 1.3) folgende Befehle unterstützt:

CMD_RESET	Hiermit wird ein Device in seinen Urzustand zurückgesetzt. Dabei werden auch alle noch ausstehenden IORequests entfernt.
CMD_READ	Führt einen Lese-Zugriff durch. Dabei werden io_Length-Daten gelesen. Die Anzahl der tatsächlich gelesenen Daten findet man in io_Actual.
CMD_WRITE	Entspricht CMD_READ, nur werden hier Daten geschrieben.
CMD_UPDATE	Mit diesem Befehl werden eventuell vorhandene Ram-Puffer auf den Datenträger geschrieben. Dies wird zwar normalerweise auch von Zeit zu Zeit von dem Device-Treiber automatisch durchgeführt, doch kann man hiermit diese Aktion explizit verlangen.
CMD_CLEAR	Hiermit löscht man die Puffer, ohne daß sie auf den Datenträger geschrieben werden.
CMD_STOP CMD_START	Mit diesen beiden Befehlen kann man die Abarbeitung von IORequests stoppen. Zwar werden von der Device weiterhin Requests angenommen, doch wird die Abarbeitung erst nach CMD_START wieder aufgenommen.
CMD_FLUSH	Hiermit löscht man alle noch ausstehenden Requests; sie werden mit einer Fehlermeldung zurückgegeben.
CMD_NONSTD	Ab hier können Devices eigene Komandos definieren.

Memory

Im Gegensatz zu Single-Task-Computern, bei denen jeweils nur ein Programm im Speicher ist und bearbeitet wird, muß man bei einem Rechner wie dem Amiga sicherstellen, daß verschiedene Programme sich nicht gegenseitig ausschließen. Daher kann man Programme, die unter dem Multitasking laufen sollen, nicht so schreiben, daß sie auf einen ganz bestimmten Speicherbereich angewiesen sind.

Deshalb sollten alle Programme, die für die Ausführung Speicher benötigen, dem System mitteilen, wieviel Speicher sie brauchen, und sollten sich diesen zuteilen lassen. Dabei wird außerdem angegeben, welche Anforderungen noch an den Speicher gestellt werden:

MEMF_CHIP Der Speicher liegt im Chip-Memory, also den (im Moment) unteren 512 KByte (bei den neueren A2000 sind ja bereits jetzt 1MB Chip-Ram eingebaut). Dies ist beispielsweise nötig, wenn Daten von oder zur Diskette übertragen werden sollen, oder wenn der Blitter zur Grafikdarstellung darauf zugreifen soll.

MEMF_FAST Hiermit wird Fast-Memory, also alles außer Chip-Memory, angefordert. Dies hat den Vorteil, daß der Speicher nicht durch Zugriffe der Coprozessoren verlangsamt wird.

Werden weder **MEMF_CHIP** oder **MEMF_FAST** gesetzt, so wird zuerst versucht, den Speicher im Fast-Ram zu bekommen, danach im Chip-Ram.

MEMF_PUBLIC Setzt man dieses Flag, so wird der Speicher, den man mit **MEMF_CHIP/MEMF_FAST** (siehe oben) definiert, belegt. Zusätzlich wird dieser Block allerdings noch als Public markiert, damit (bei Verwendung einer MMU) mehrere Tasks auf denselben Speicher zugreifen können.

MEMF_CLEAR Speicher, der mit diesem Flag angefordert wird, wird vom System automatisch gelöscht, so daß man leeren Speicher erhält.

AllocMem

Doch nun zum Anfordern des Speichers. Die wohl gebräuchlichsten Routinen sind dabei **AllocMem()** und **FreeMem()**. Im Programm würde der Zugriff beispielsweise so aussehen:

```

char *mem, *AllocMem();
:
main()
{
    if (!(mem=AllocMem(512L, MEMF_PUBLIC|MEMF_CLEAR))) {
        printf("Kein Speicher\n");
        exit();
    }
    :
    :
    FreeMem(mem, 512L);
}

```

Wichtig ist dabei, daß der gesamte Speicher wieder zurückgegeben wird, da das System keine Ahnung hat, wer wieviel Speicher belegt hat, und so der Speicher bis zum nächsten Reset verloren wäre.

Außer AllocMem() gibt es auch noch die Routine AllocAbs(), mit der man einen bestimmten Speicherbereich anfordert, z.B.:

```
mem=AllocAbs(1024L, 0x12345L);
```

Sollte auch nur ein Teil dieses Speichers bereits belegt sein, so wird die Routine 0 zurückliefern.

Im Betriebssystem wird oft ein zweites Paar Routinen verwendet: AllocEntry() und FreeEntry(). Diese beiden Routinen erhalten eine ganze Liste von Speicherbereichen, die nacheinander belegt werden. Dafür wird zuerst die Struktur MemList gefüllt:

```

struct MemList {
    struct Node ml_Node;
    UWORD ml_NumEntries;
    struct MemEntry ml_ME[1];
};

```

ml_Node sorgt dafür, daß man als Benutzer mehrere Listen zusammenhängen kann. Von AllocEntry wird dies allerdings ignoriert.

ml_NumEntries gibt die Anzahl der Blöcke an, die man haben möchte.

In der Struktur MemEntry werden jetzt die Größen und Arten der Speicherblöcke übergeben:

```
struct MemEntry {
    union {
        ULONG   meu_Reqs;
        APTR    meu_Addr;
    } me_Un;
    ULONG   me_Length;
};
```

Dabei enthält me_Un.meu_Reqs die Art des Speichers, me_Length wird mit der Länge des Speichers belegt. In meu_Addr wird von AllocEntry() die Adresse des Speichers zurückgegeben. Im Programm würde das so aussehen:

```
struct MemList *aml, *AllocEntry();
struct ml {
    struct MemList ml;
    struct MemEntry me[MEMS];    /* MEMS=Anzahl der Blöcke-1 */
} ml;
:
ml.ml.ml_NumEntries = MEMS+1;
ml.ml.ml_ME[0].me_Un.meu_Reqs = MEMF_CHIP|MEMFCLEAR;
ml.ml.ml_ME[0].me_Length = 512L;
ml.ml.ml_ME[1].me_Un.meu_Reqs = MEMF_PUBLIC;
:
aml = AllocEntry(&ml);
```

Als Return-Wert erhält man einen Zeiger auf eine Kopie der MemList-Struktur, die allerdings mit Zeigern auf den Speicher gefüllt ist. Sollte der angeforderte Speicher nicht geholt werden können, so erhält man als Return-Wert die Art des Speichers, der nicht allokiert werden konnte; außerdem wird das höchste Bit gesetzt:

```
if ((ULONG)(aml)&(1<<31)) {
    printf("Nicht genug Speicher!\n");
    exit();
}
```

Um den Speicher wieder freizugeben, benutzt man FreeEntry mit der Adresse der neuen MemList als Parameter:

```
FreeEntry(aml);
```

Allocate

Eine letzte Art der Speicherverwaltung bieten die Routinen `Allocate()` und `Deallocate()`. Diese arbeiten auf der Basis von `MemHeader`-Strukturen:

```
struct MemHeader {
    struct Node mh_Node;
    UWORD    mh_Attributes;
    struct MemChunk *mh_First;
    APTR     mh_Lower;
    APTR     mh_Upper;
    ULONG    mh_Free;
};
```

Hierbei dient die `Node`-Struktur zur Verwaltung von mehreren Listen; `mh_Attributes` wird zur Zeit noch ignoriert. `mh_First` zeigt auf die erste `MemChunk`-Struktur:

```
struct MemChunk {
    struct MemChunk *mc_Next;
    ULONG    mc_Bytes;
};
```

Die letzten drei Werte enthalten Start- und Endadresse des eigenen Speichers sowie die freien Bytes.

Vorteil von `Allocate` und `Deallocate` ist, daß ein Programm seinen eigenen Speicher vorbelegt, von dem dann jeweils Teile belegt werden. Somit spart man sich unter anderem bei größeren `Allocate`-Aktionen auch das Freigeben, da man vor dem Verlassen des Programms ein einziges `FreeMem()` machen kann, mit dem der gesamte Speicher wieder frei ist; `Deallocate()`-Aufrufe sind nicht nötig. Im Einsatz könnte eine Routine so aussehen:

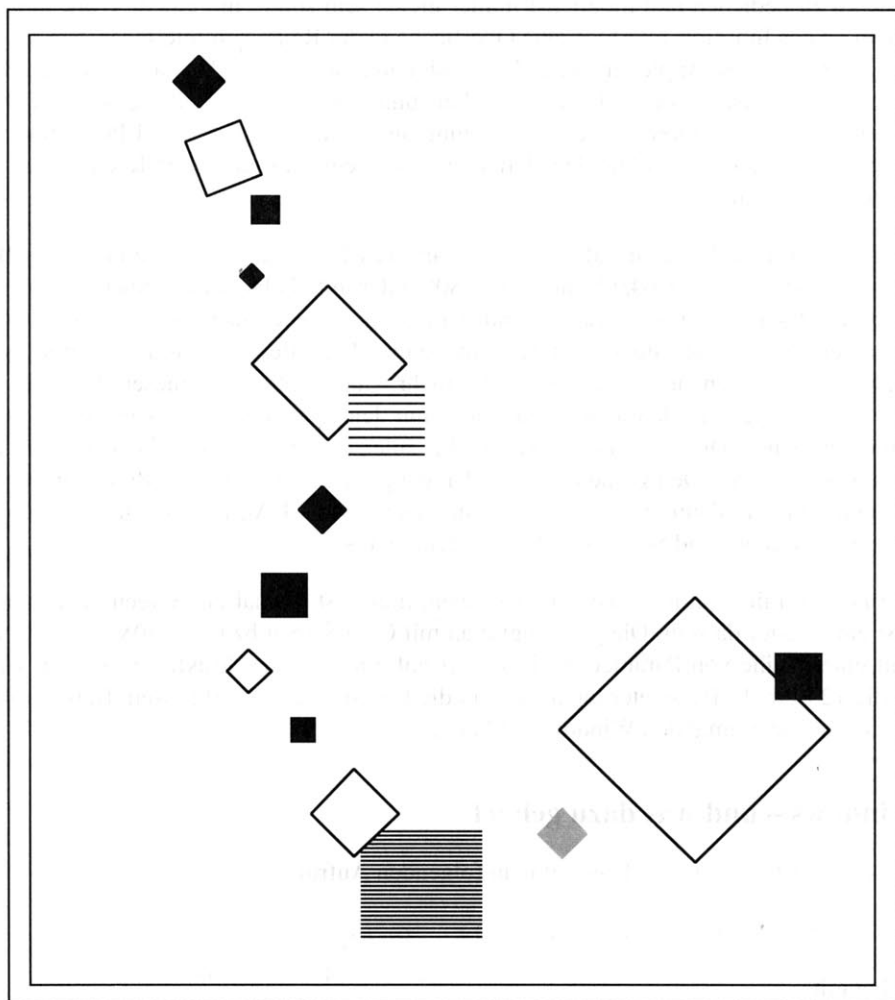
```
struct MemHeader *mh;
struct MemChunk *mc;
long temp;
:
temp = (long)(sizeof (struct MemHeader)+8192L);

mh = (struct MemHeader *)AllocMem(temp, MEMF_CLEAR|MEMF_CHIP);
if (mh) {
    mh->mh_Node.ln_Type = NT_MEMORY;
    mh->mh_Node.ln_Name = "Memorypool";
```

```
mh->mh_First = mh + sizeof(struct MemHeader);
mh->mh_Lower = mh->mh_First;
mh->mh_Upper = mh->mh_First+8192L;
mh->mh_Free = 8192L;
mc = mh->mh_First;
mc->mc_Next = 0L;
mc->mc_Bytes = 8192L;
:
    myMem = Allocate (mh, Size);
    :
    Deallocate (mh, myMem, Size);
:
FreeMem (mh, temp);
}
```

Kapitel 2

Intuition



Intuition

Intuition ist beim Amiga der Teil, der für den Dialog mit dem Benutzer zuständig ist. Dabei werden dem Programmierer eine Menge "Bausteine" in die Hand gegeben, die es ihm ermöglichen, sich nur noch um das eigentliche Programm kümmern zu können. Welche Teile das sind und wie man sie einfach benutzt, wird in diesem Kapitel gezeigt.

Ziel von Intuition war es, dem Benutzer eine Oberfläche zu geben, die einfach und angenehm zu bedienen und möglichst immer gleich sein sollte; mit einem Wort: intuitiv. Dabei ist das Intuition-Interface eine Oberfläche in der Reihe von Interfaces, wie sie vor einigen Jahren von Apple auf dem Macintosh eingeführt wurde. Damals noch eine Sensation, ist eine Benutzeroberfläche mit Maus und grafischer Darstellung auf dem Bildschirm heute, dank ihrer starken Verbreitung auf Amiga, Atari ST und PC, schon fast ein Standard geworden, ohne den sich kaum noch jemand traut, einen Rechner auf den Markt zu bringen.

Und dies ja nicht ohne Grund: Bei keiner anderen Bedienung, sei es nun mittels BASIC-Befehlen wie bei C64/128 und Atari 800, oder mittels DOS bzw. Shell bei PC und Unix, ist das Erlernen der grundlegenden Funktionen so einfach. So kann man schon nach weniger als einer Stunde auf dem Amiga eine Diskette kopieren, Files verdoppeln, verschieben, umbenennen und noch vieles mehr. Und das Beste an diesem Interface ist, daß Programme, die damit arbeiten, nicht nur leicht zu bedienen, sondern auch zu schreiben sind. Dies verdankt man nicht zuletzt den zur Zeit 74 Routinen der "Intuition.Library". Den grundsätzlichen Umgang mit den wichtigsten Routinen werden wir nun erklären, damit auch Sie sich nicht weiter mit CLI-Argumenten herumschlagen müssen. Windows und Screens – ohne sie geht nichts.

Um normal mit Intuition arbeiten zu können, muß erst einmal ein Screen oder zumindest ein Fenster da sein. Diese erzeugt man mit OpenScreen bzw. OpenWindow, wobei statt einer Reihe von Parametern ein Zeiger auf eine spezielle Struktur übergeben wird, da die 12 bzw. 18 Parameter nicht mehr in die Register des 68000 passen. Dabei wollen wir uns zuerst einmal den Windows widmen.

Windows – und was dazu gehört

Um ein Fenster zu öffnen, benötigt man folgenden Aufruf:

```
win = OpenWindow ( &NewWinDef );
```

wobei gilt:

```
struct Window *win, *OpenWindow();  
struct NewWindow NewWinDef;
```

Ist `win == 0`, so konnte das Fenster nicht geöffnet werden.

Wichtig ist hier der Aufbau der Struktur `NewWindow`:

```
struct NewWindow
{
    SHORT   LeftEdge, TopEdge;
    SHORT   Width,    Height;
    UBYTE   DetailPen,BlockPen;
    ULONG   IDCMPFlags;
    ULONG   Flags;
    struct  Gadget *FirstGadget;
    struct  Image  *CheckMark;
    UBYTE   *Title;
    struct  Screen *Screen;
    struct  BitMap *BitMap;
    SHORT   MinWidth,  MinHeight;
    USHORT  MaxWidth,  MaxHeight;
    USHORT  Type;
};
```

In die einzelnen Felder muß man dabei folgendes eintragen:

`LeftEdge, TopEdge` Die Position der linken oberen Ecke.

`Width, Height` Breite und Höhe des Fensters.

`DetailPen, BlockPen` Die Farbe, in der Text und Gadgets bzw. Blöcke im Fenster gezeichnet werden. Bei -1 werden Standardwerte verwendet.

`IDCMPFlags` Hier werden die Flags angegeben, die bei Eintreten bestimmter Situationen eine Message über den Messageport des Fensters schicken sollen. Folgende Flags können dabei ausgewählt werden:

Event
Bedingung

ACTIVEWINDOW
Aktivieren des Fensters.

CLOSEWINDOW
Selektieren des Close-Gadgets.

DELTAMOVE

MOUSEMOVE-Werte als Differenz zur letzten Position.

DISKINSERTED

Einlegen von Disks.

DISKREMOVED

Entfernen von Disks.

GADGETUP

Loslassen eines Gadgets.

GADGETDOWN

Selektieren eines Gadgets.

INACTIVELWINDOW

Deaktivieren des Windows.

INTUITICKS

Gibt mit einer Frequenz von maximal 10 Hertz Messages an den Port, wobei neue Nachrichten nur geschickt werden, wenn die letzte INTUITICKS-Message bereits Reply(ed) wurde.

MENUPICK

Auswahl eines Menüs.

MENUVERIFY

Wird gesetzt, damit Menüs erst gezeichnet werden, wenn alle Ausgaben auf das Fenster beendet sind (ergibt keine Nachricht!).

MOUSEBUTTONS

Drücken einer Maustaste (rechts oder links).

MOUSEMOVE

Bewegen der Maus (Achtung: wenn dieses Flag gesetzt wird, sollte man dafür sorgen, daß die Messages so schnell wie möglich bearbeitet wird!).

NEWPREFS

Message, sobald neue Preferences-Einstellungen vorgenommen wurden.

RAWKEY

Tastendruck als Raw-Code (s. Anhang "Tastatur").

REFRESHWINDOW

Message jeweils, sobald das Fenster eine Auffrischung benötigt (außerdem sollte in den Flags `SIMPLE_REFRESH` oder `SMART_REFRESH` gesetzt sein!).

REQCLEAR

wird geschickt, sobald alle Requester des Fensters beendet sind.

REQSET

Sobald ein Requester geöffnet ist, bekommt das Programm diese Message.

REQVERIFY

wird gesetzt, damit Requester erst gezeichnet werden, wenn alle Ausgaben auf das Fenster beendet sind (ergibt keine Nachricht!).

SIZEVERIFY

wird gesetzt, damit das Verändern der Fenstergröße erst erlaubt wird, wenn alle Ausgaben auf das Fenster beendet sind (ergibt keine Nachricht!).

VANILLAKEY

Tastendruck als ASCII-Werte (s. Anhang "Tastatur").

WBENCHMESSAGE

Diese Nachricht wird von Intuition geschickt, falls der Befehl `CloseWorkBench` oder `OpenWorkBench` aufgerufen wurde und das Programm sich mit `AlohaWorkbench()` angemeldet hat. Diese Nachricht wird allerdings nicht über den Userport des Fensters geschickt, sondern an den Port, den man übergeben hat (s. "Ordentliche WB-Programmierung").

Flags

In diesem Feld werden die anderen Flags gesetzt, die Eigenschaften bzw. das Erscheinungsbild des Fensters bestimmen:

Flag

Funktion

ACTIVATE

Das Fenster wird beim Öffnen aktiviert.

BACKDROP

Das Fenster wird hinter allen anderen Fenstern geöffnet; es kann vor keine anderen Fenster gelegt werden.

BORDERLESS

Es wird kein Rand gezeichnet.

GIMMEZEROZERO

Das Fenster wird aus zwei Bereichen aufgebaut: einem äußeren, auf dem Rahmen, Titel etc. dargestellt werden, und einem inneren, den der Programmierer normal benutzen kann. Im Gegensatz zu normalen Fenstern braucht man hier keinen Offset für Rahmen und Titel zu berechnen. Das Fenster fängt bei (0,0) an; die Größe kann man aus der Struktur Window bekommen, und zwar GZZWidth und GZZHeight. Nachteil dieser Methode sind ein höherer Speicherbedarf und Leistungseinbußen beim Verschieben und anderen Window-Operationen.

SIMPLE_REFRESH

SMART_REFRESH

NOCAREREFRESH

SUPER_BITMAP

Diese Refresh-Arten werden im Grafik-Teil genauer besprochen.

WINDOWREFRESH

Ist gesetzt, falls das Window gerade aufgefrischt wird.

REPORTMOUSE

Einschalten der Erkennung von Mausbewegungen.

RMBTRAP

Abfangen der rechten Maustaste.

SIZEBBOTTOM

Auswahl, wo der Platz für das SIZEBRIGHT-SizeGadget genom-

men werden soll, entweder aus dem unteren oder dem rechten Randbereich. Default ist SIZEBRIGHT.

WINDOWCLOSE

Fenster mit Close-Gadgets.

WINDOWDEPTH

Fenster mit Depth-Gadgets.

WINDOWDRAG

Fenster verschiebbar.

WINDOWSIZING

Fenster in der Größe veränderbar.

Gadgets	Ein Zeiger auf das erste Element einer Liste von Gadgets, die in dem Fenster auftauchen sollen.
CheckMark	Ein Zeiger auf eine Image-Struktur, die den normalen Haken in den Menüs ersetzen soll.
Title	Ein Zeiger auf einen mit Null beendeten String, der in der Titelseile des Fensters ausgegeben werden soll.
Screen	Ein Zeiger auf eine Screen-Struktur. Das Fenster wird dann im Screen geöffnet, wenn Type auf CUSTOMSCREEN gesetzt wurde.
BitMap	Wenn in Flags das Flag SUPER_BITMAP gesetzt wurde, steht hier ein Zeiger auf die BitMap-Struktur des Fensters.
MinWidth MaxWidth MinHeight MaxHeight	Dies sind die Werte, die die minimale bzw. maximale Größe des Fensters angeben, falls es ein Size-Gadget beinhaltet.
Type	Art des Screens. Entweder "WBENCHSCREEN" oder "CUSTOMSCREEN".

Damit ist das Thema Fenster eigentlich erledigt. Um dem Leser die Möglichkeit zu geben, ohne viele Compiler- oder Assembler-Durchläufe die ganzen Möglichkeiten

selbst durchzuprobieren, befindet sich auf der Diskette das Programm FensterTest, mit dem alle Parameter eines Fensters geändert werden können, und so das Aussehen bestimmt werden kann. Verlassen kann man das Programm durch Drücken des Close-Gadgets.

Doch nun zu den Screens. Um beim Amiga etwas darstellen zu können, muß ein Screen vorhanden sein (außer natürlich bei direkter Programmierung der Hardware). Als Standard-Screen ist die Workbench fast immer vorhanden, so daß man sich für einfache Anwendungen nur um das Fenster kümmern muß. Ähnlich wie bei den Fenstern bekommt auch die Routine OpenScreen einen Zeiger auf eine Struktur, die das Aussehen des Screens bestimmt:

```
struct NewScreen {
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
    UBYTE DetailPen, BlockPen;
    USHORT ViewModes;
    USHORT Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap;
};
```

LeftEdge, TopEdge Die linke oberer Ecke des Screens.

Width, Height Breite und Höhe des Screens.

Depth Anzahl der Bitplanes ($\Rightarrow$ Farben: 2^{Depth}).

DetailPen, BlockPen Entspricht Detail- und BlockPen bei Window.

ViewModes Darstellungsart des Screens:

Flag

Art

HIRES

Doppelte X-Auflösung.

LACE

Doppelte Y-Auflösung.

SPRITES

Anzeige von Sprites auf Screen.

DUALPF

Dual-Playfield-Screen.

HAM

Hold-and-Modify-Modus.

VP_HIDE

View-Port wird nicht nach vorne gelegt.

EXTRA_HALFBRITE

Extra-Halfbrite-Modus.

Type

Hier können verschiedene Einstellungen für den Screen selber vorgenommen werden:

Flag

Funktion

WBENCHSCREEN

Öffnen des Screens mit Belegen des nötigen Speichers.

CUSTOMSCREEN

Öffnen des Screens unter Verwendung des mit CustomBitMap definierten Speichers.

SHOWTITLE

Screen-Titel liegt über Fenstern.

CUSTOMBITMAP

Vorhandene BitMap verwenden.

SCREENBEHIND

Screen öffnet sich hinter anderen Screens.

SCREENQUIET

Keine Drag-Bar und Depth-Gadgets in der Titelzeile des Screens anzeigen.

Font	Ein Zeiger auf eine TextAttr-Struktur; der damit definierte Font wird als Default-Font für den Screen und alle in ihm geöffneten Fenster benutzt.
DefaultTitle	Der Titel des Screens.
Gadgets	Ein Zeiger auf eine Gadget-Liste.
CustomBitMap	Ist CUSTOMSCREEN unter Type gesetzt, so wird der hier mittels eines Zeigers auf eine BitMap-Struktur definierte Speicher verwendet.

Gadgets

Ein Hauptproblem bei anderen Rechnern, z.B. C64, Atari 800XL oder auch PCs, war und ist, daß man als Programmierer sämtliche Eingaben "zu Fuß" verarbeiten muß: Bei Stringeingaben muß beachtet werden, daß das Eingabefeld nicht verlassen wird, Eingaben von Zahlen müssen ebenfalls speziell überwacht werden, damit keine anderen Zeichen übernommen werden. Für alle diese und noch viele Aufgaben mehr stellt Intuition ein schnelles und leicht zu benutzendes Hilfsmittel zur Verfügung: die Gadgets. Hiermit kann man Texteingaben vornehmen, Zahlenwerte übernehmen, Menüs aufbauen. Basis für die Programmierung eines Gadgets bildet die Gadget-Struktur:

```
struct Gadget
{
    struct Gadget *NextGadget;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    USHORT Activation;
    USHORT GadgetType;
    APTR GadgetRender;
    APTR SelectRender;
    struct IntuiText *GadgetText;
    LONG MutualExclude;
    APTR SpecialInfo;
    USHORT GadgetID;
    APTR UserData;
};
```

Die einzelnen Elemente haben dabei folgende Bedeutung:

NextGadget Zeiger auf das nächste Gadget.

LeftEdge, TopEdge Linke obere Ecke des Gadgets.

Width, Height Breite und Höhe des Gadgets.

Flags Hiermit werden eine Reihe von Funktionen des Gadgets definiert.
Möglich sind zur Zeit folgende Flags:

Flag

Funktion

GADGHCOMP

Zum Verdeutlichen der Selektion wird das Gadget invertiert.

GADGHBOX

Wie GADGHCOMP, doch wird um das Gadget ein Rahmen gezeichnet.

GADGHIMAGE

Die Selektion wird durch ein zweites Image, also ein anderes Bild, dargestellt.

GADGHNONE

Das Gadget verändert sich beim Anwählen nicht.

GADGIMAGE

Zu dem Gadget gehört ein Image (=Bild).

GREL...

Die Angaben von Position und Größe sollen jeweils relativ zu den Rändern sein: Bei GRELBOTTOM und GRELRIGHT ist die Positionsangabe der linken oberen Ecke in Y- bzw. X-Richtung zum unteren bzw. rechten Rand des Fensters relativ, bei GRELWIDTH und GRELHEIGHT sind die Angaben von Breite und Höhe relativ zur Höhe des Windows.

SELECTED

Ist dieses Flag gesetzt, wird das Gadget selektiert dargestellt bzw. ist gerade selektiert.

GADGDISABLED

Das Gadget ist abgeschaltet und wird auch in der Anzeige anders dargestellt.

Activation

In diesem Flag werden einige Informationen über das Gadget übergeben:

Flag

Funktion

ALTKEYMAP*

Wird gesetzt, wenn eine andere Keymap bei den String-Gadgets verwendet werden soll. In dem Fall muß außerdem ein Zeiger auf die Keymap in die StringInfo-Variable AltKeyMap eingetragen werden.

BOOLEXTEND

Zusätzliche BoolInfo-Struktur.

...BORDER

Ist ein oder zwei der Flags LEFTBORDER oder RIGHTBORDER und TOPBORDER oder BOTTOMBORDER gesetzt, wird das Gadget in den Rahmen des Fensters eingebaut, wobei die Breite des Rahmens automatisch auf das Gadget eingestellt wird.

ENDGADGET

Ist dieses Flag gesetzt, handelt es sich um ein Gadget mit dem, falls es sich in einem Requester befindet, der Requester beendet wird.

FOLLOWMOUSE

Ist dieses Flag gesetzt, werden alle Maus-Bewegungen übermittelt, solange das Gadget selektiert ist.

GADGIMMEDIATE

Dieses Flag dient dazu, sofort eine Message zu schicken, sobald das Gadget selektiert wird.

LONGINT*

Ist dieses Flag gesetzt, wird in einem String-Gadget nur noch die Eingabe einer Zahl zugelassen. Außerdem muß es sich bei der Vorgabe im String-Puffer ebenfalls um eine Zahl handeln.

RELVERIFY

Hiermit wird nur eine GADGETUP-Message geschickt, falls das Gadget losgelassen wurde und die Maus sich noch über dem Gadget befindet.

STRINGCENTER*

Der Eingabestring wird im Gadget zentriert dargestellt.

STRINGRIGHT*

Der Eingabestring wird im Gadget rechtsbündig dargestellt.

TOGGLESELECT

Der Status des Gadgets wird bei jedem Anklicken invertiert. Vor-einstellen kann man ihn mittels des SELECTED-Flags. Um ihn abzufragen, muß man ebenfalls das SELECTED-Flag überprüfen.

*: Alle so gekennzeichneten Flags beziehen sich nur auf String-Gadgets!

GadgetType

Dieses Feld beinhaltet eins der folgenden drei Flags zur Definition der Gadget-Art:

BOOLGADGET

Das Gadget wird als Boolean-Gadget, also für Ja/Nein-Abfragen und ähnliches eingesetzt.

STRGADGET

Das Gadget ist ein String-Gadget.

PROPGADGET

Bei dem Gadget handelt es sich um ein Proportional-Gadget, also einen Schiebe-Regler.

Außerdem müssen eventuell noch folgende Bits gesetzt werden:

GZZGADGET

Falls das Gadget in ein GIMMEZEROZERO-Fenster soll, muß dieses Flag gesetzt werden, um es in dem Fensterrahmen unterzubringen.

REQGADGET

Dieses Flag muß gesetzt werden, wenn das Gadget in einem Requester-Fenster auftauchen soll.

GadgetRender	In diesem Feld befindet sich ein Zeiger auf eine Border- oder Image-Struktur (für Image muß das Flag GADGIMAGE (s.o.) gesetzt sein). Wird hier NULL eingetragen, so wird kein Rahmen gezeichnet.
SelectRender	Entspricht GadgetRender, nur für das Aussehen des selektierten Gadgets.
GadgetText	Ein Zeiger auf eine IntuiText-Struktur. Durch Verketteten mehrerer Strukturen können für jedes Gadget beliebig viele Texte ausgegeben werden.
MutualExclude	Dieses Feld wird zur Zeit noch von Intuition ignoriert. Um Programme auch unter zukünftigen Kickstarts lauffähig zu halten, sollte hier NULL eingetragen werden.
SpecialInfo	Hier steht ein Zeiger auf eine zusätzliche Struktur für Proportional- und Stringgadgets.
GadgetID	Hier kann ein Wert bzw. eine Gadget-Nummer eingetragen werden, um im Programm verzweigen zu können. Dieses Feld wird, ebenso wie das folgende, von Intuition ignoriert.
UserData	Dieses Feld steht dem Programmierer zur Verfügung und kann beliebig benutzt werden.

Hat man damit alle Parameter des Gadgets eingestellt, fehlt noch der Rahmen oder das Bild. Zuerst wollen wir uns daher die Border-Struktur ansehen:

```
struct Border
{
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    BYTE Count;
    SHORT *XY;
    struct Border *NextBorder;
};
```

Dabei haben die einzelnen Teile folgende Bedeutung:

LeftEdge, TopEdge Position der linken oberen Ecke relativ zum Gadget.

FrontPen, BackPen	Rahmen werden immer mit der FrontPen-Farbe gezeichnet; BackPen wird z.Zt. noch ignoriert.
DrawModeDer	Modus, in dem die Linien gezeichnet werden sollen, also JAM1, JAM2(=JAM1), COMPLEMENT (=exklusiv oder) oder INVERSVID.
Count	Anzahl der Linien, aus denen der Rahmen besteht.
XY	Zeiger auf ein Array, in dem die Werte für den Rahmen stehen.
NextBorder	Zeiger auf die nächste Border-Struktur.

Um statt des Rahmens eine eigene Grafik unterzubringen, muß (neben anderen Flags) ein Zeiger auf folgende Image-Struktur übergeben werden:

```
struct Image
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT Depth;
    USHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
};
```

LeftEdge,TopEdge	Linke obere Ecke relativ zum Gadget.
Width, Height	Breite und Höhe der Grafik.
Depth	Anzahl der Bitplanes in der Grafik.
ImageData	Zeiger auf die Grafik-Daten.
PlanePick PlaneOnOff	Diese beiden Werte sorgen dafür, daß bei Grafiken, in denen die Werte für eine oder mehrere Bitplanes nur 0 oder nur 1 sind, diese nicht abgespeichert werden müssen. Dazu trägt man in PlanePick Bits für die Bitplanes ein, für die Grafikdaten vorhanden sind; für diese Bitplanes wird PlaneOnOff ignoriert. Alle anderen Bitplanes werden auf den Wert gesetzt, den die Bitplane enthalten soll. Als Beispiel seien folgende Werte gegeben: PlanePick=\$6,

PlaneOnOff=\$38. Hier würden für die Bitplanes 1 und 2 die Grafikdaten verwendet, Bitplane 0 wird auf 0 gesetzt, und Bitplanes 3 bis 5 werden auf 1 gesetzt.

Auf diese Weise kann man auch leicht gefüllte Rechtecke als Gadgets erzeugen (zum Beispiel in einem Farb-Requester): man stellt Width und Height auf die gewünschte Größe, Depth auf 0, PlanePick auf 0 und PlaneOnOff auf die Farbe.

NextImage

Hier steht ein Zeiger auf die nächste Image-Struktur.

Proportional-Gadgets

Die Eingabe eines analogen Wertes ist schon für alle Programmierer ein Problem gewesen. Wenn so etwas anstand, mußten Routinen zur Abfrage des Eingabegerätes, zur Darstellung des Eingabefeldes und vieles mehr geschrieben werden. Beim Amiga hat man den Vorteil, daß man sogenannte Proportional-Gadgets verwenden kann, deren Programmierung und Abfrage recht leicht ist. Dazu wird in der Gadget-Struktur im Feld GadgetType der Wert PROPGADGET eingetragen, das Feld SpecialInfo erhält einen Zeiger auf folgende Struktur:

```
struct PropInfo
{
    USHORT Flags;
    USHORT HorizPot;
    USHORT VertPot;
    USHORT HorizBody;
    USHORT VertBody;
    USHORT CWidth;
    USHORT CHeight;
    USHORT HPotRes, VPotRes;
    USHORT LeftBorder;
    USHORT TopBorder;
};
```

Flags

Hier werden einige Optionen für das Gadget gesetzt:

Flag

Funktion

AUTOKNOB

Automatische Verwaltung des Reglers;

Anklicken ober- und unterhalb des Reglers verschiebt ihn um jeweils eine Anzeige-Einheit. Auch die Größe wird dabei automatisch berechnet.

FREEHORIZ

Ist dieses Flag gesetzt, kann man den Regler in X-Richtung bewegen.

FREEVERT

Wie FREEHORIZ, nur in Y-Richtung.

KNOBHIT

Wird von Intuition gesetzt, wenn der Regler gerade benutzt wird.

PROPBORDERLESS

Dieses Flag sorgt dafür, daß kein Rahmen gezeichnet wird.

HorizPot, VertPot Aktuelle Position des Reglers, wobei 0 0% entspricht, 65535 sind 100%.

HorizBody, VertBody

Hier wird die prozentuale Größe des Reglers angegeben.

Um die letzten beiden Daten-Paare zu verändern, muß die Funktion NewModifyProp (siehe unten) verwendet werden. Die restlichen Werte werden von Intuition gesetzt und brauchen nicht beachtet zu werden.

Da man theoretisch zum Modifizieren des Prop-Gadgets dieses aus der Liste herausnehmen, ändern und neu einbinden müßte, stellt Intuition eine Routine zur Verfügung, die dies übernimmt:

NewModifyProp (Gadget, Window, Req, Flags, HPot, VPot, HBody, VBody, Num)

Gadget ist dabei ein Zeiger auf die Gadget-Struktur (nicht auf PropInfo!), Window zeigt auf die Window-Struktur, und Req zeigt auf den Requester (falls vorhanden). Die nächsten fünf Werte entsprechen denen der PropInfo-Struktur. Num gibt an, wieviele Gadgets ab dem Prop-Gadget "refreshed" werden sollen (-1 = bis Ende der Liste).

String-Gadgets

Die dritte Erscheinungsform eines Gadgets ist das String-Gadget. Dieses übernimmt Daten von der Tastatur, wobei entweder allgemeine Strings oder rein numerische Strings erlaubt sind. Zur Definition eines String-Gadgets muß in der Gadget-Struktur im Feld `GadgetType` der Wert `STRGADGET` eingetragen werden, `SpecialInfo` erhält einen Zeiger auf folgende Struktur:

```
struct StringInfo
{
    UBYTE *UndoBuffer;
    SHORT BufferPos;
    SHORT MaxChars;
    SHORT DispPos;
    SHORT UndoPos;
    SHORT NumChars;
    SHORT DispCount;
    SHORT CLeft, CTop;
    struct Layer *LayerPtr;
    LONG LongInt;
    struct KeyMap *AltKeyMap;
};
```

Buffer	ist ein Zeiger auf den Puffer, in dem der Eingabestring gespeichert werden soll. Steht beim Initialisieren ein Text im Puffer, wird er im Gadget angezeigt.
UndoBuffer	Dieser Puffer dient dazu, bei Änderungen im Gadget den alten Inhalt zurückzuholen. Wird dieser Zeiger auf NULL gesetzt, steht kein Undo-Puffer zur Verfügung.
BufferPos	Die Position, an der der Cursor stehen soll.
MaxChars	Anzahl der Zeichen, die maximal im Gadget übernommen werden sollen. Achtung: das Stringende-Kennzeichen wird ebenfalls mitgezählt, <code>MaxChars</code> muß also um 1 größer sein als die maximale Anzahl Zeichen!
DispPos	Position des ersten angezeigten Zeichens.

Die folgenden sechs Felder werden von Intuition gesetzt und verwaltet und sind normalerweise für Programmierer weniger interessant:

UndoPos	Position im Undo-Puffer.
NumChars	Anzahl Zeichen im Puffer.
DispCount	Maximale Anzahl Zeichen in der Anzeige.
CLeft, CTop	Linke obere Ecke bezüglich des Anzeigefeldes.
LayerPtr	Zeiger auf die Layer, in der das Gadget angezeigt wird.
LongInt	Falls es sich bei dem String-Gadget um ein numerisches String-Gadget handelt, wird hier nach der Eingabe der Inhalt des Puffers als Zahl gespeichert.
AltKeyMap	Soll für das Gadget eine eigene Tastaturbelegung verwendet werden, muß hier ein Zeiger auf die Tabelle stehen.

Menüs

Neben den Gadgets sind Menüs das zweite wichtige Eingabemedium. Auch hier muß der Programmierer praktisch nur Zahlenwerte abfragen, der Rest wird von Intuition übernommen. Grundlage eines Menüs ist die Menu-Struktur:

```
struct Menu
{
    struct Menu *NextMenu;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    BYTE *MenuName;
    struct MenuItem *FirstItem;
    SHORT JazzX, JazzY, BeatX, BeatY;
};
```

Menu ist ein Zeiger auf das nächste Menü oder NULL, falls keine weiteren Menüs angehängt werden sollen. Maximal dürfen 31 Menüs definiert werden.

LeftEdge, TopEdge Hiervon wird zur Zeit nur LeftEdge beachtet, da bei allen bisherigen Kickstart-Versionen das Menü in der Kopfzeile des Screens

erscheinen muß. Um auch für zukünftige Versionen von Kickstart korrekt laufende Programme zu schreiben, sollte in `TopEdge` 0 eingetragen werden. `LeftEdge` gibt den linken Rand des Menüs im Screen an.

Width, Height `Width` ist die Breite des Menüs. Ähnlich wie bei `TopEdge` wird hier `Height` von Intuition ignoriert und auf die Höhe der Titelzeile des Screens gesetzt.

Flags Zur Beeinflussung des Menüs existiert zur Zeit nur das Flag `MENUENABLED`. Ist sie gesetzt, können Unterpunkte des Menüs ausgewählt werden, ansonsten ist das Menü mit allen Unterpunkten gesperrt. Intern gibt es noch das Flag `MIDRAWN`. Ist sie gesetzt, dann sind die Unterpunkte im Moment angezeigt.

MenuName Zeiger auf den Text, der in der Menü-Zeile erscheinen soll.

FirstItem Zeiger auf das erste Element der Item-Liste des Menüs.

Die restlichen vier Variablen sind nur für den internen Gebrauch gedacht und sollten auf keinen Fall im Programm verwendet werden, da sich ihre Bedeutung bei anderen Kickstart-Versionen ändern kann.

Die Struktur zur Beschreibung der einzelnen Items, also Unterpunkte des Menüs, sieht folgendermaßen aus:

```
struct MenuItem
{
    struct MenuItem *NextItem;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    LONG MutualExclude;
    APTR ItemFill;
    APTR SelectFill;
    BYTE Command;
    struct MenuItem *SubItem;
    USHORT NextSelect;
};
```

NextItem Ein Zeiger auf das nächste Element der Liste, oder NULL für Ende der Liste.

LeftEdge, TopEdge Linke obere Ecke des Menüpunktes. Während LeftEdge, das relativ zum linken Rand des Menüs ist, auch negativ sein darf, muß TopEdge auf jeden Fall positiv sein, da es sich auf die höchste, von Intuition zugelassene Position bezieht.

Width, Height Breite und Höhe des Menüpunkts.

Flags Hier werden verschiedene Funktionen und das Aussehen des Menüpunktes definiert:

Flag

Funktion

CHECKIT

Ist dieses Flag gesetzt, ist das Menü mit einer Art "Umschalter" versehen: Bei Selektion des Menüpunkts wird der Amiga-Haken vor dem Menü (oder ein anderes, vom Programmierer eingebautes Bild) angeschaltet sowie das Flag CHECKED gesetzt.

CHECKED

ist gesetzt, wenn das CHECKIT-Flag gesetzt ist und der Benutzer den Menüpunkt "eingeschaltet" hat, also der Haken zu sehen ist.

COMMSEQ

Ist dieses Flag gesetzt, kann das Menü mittels einer Tastatur-Abkürzung ausgewählt werden. Diese besteht aus der rechten Amiga-Taste sowie dem Zeichen, das in Command angegeben ist.

HIGHBOX

Befindet sich die Maus über dem Menüpunkt, wird ein Rahmen um ihn herum gezeichnet.

HIGHCOMP

Wie HIGHBOX, nur wird der Menüpunkt invertiert.

HIGHIMAGE

Statt eines Rahmens oder Invertierung wird ein anderes Image verwendet. (Zeiger in SelectFill.)

HIGHITEM

Dieses Flag wird gesetzt, wenn sich die Maus gerade über dem Item befindet.

HIGHNONE

Es findet keine Änderung des Items statt.

ISDRAWN

wird von Intuition gesetzt, wenn die Subitems des Menüpunkts gerade angezeigt werden.

ITEMENABLED

Wenn dieses Flag gesetzt ist, kann der Menüpunkt und seine Untermenüs (falls vorhanden) ausgewählt werden.

ITEMTEXT

Falls gesetzt, sind ItemFill und SelectFill Zeiger auf IntuitText-Strukturen, ansonsten auf Image-Strukturen.

MENUTOGGLE

Ist dieses Flag gesetzt, wird das Flag CHECKED sowie der Haken bei jeder Selektion des Menüpunkts umgeschaltet.

Mutualexclude

In diesem Long-Word werden jeweils die Bits für die Menüpunkte gesetzt, deren CHECKED-Flag bei Selektion dieses Menüpunktes zurückgesetzt werden soll.

ItemFill

Hier steht ein Zeiger auf eine IntuitText- oder Image-Struktur, die in dem Menüpunkt angezeigt werden soll. Entsprechendes gilt für:

SelectFill

Dies ist jedoch für den selektierten Zustand des Menüpunktes zuständig. Die Art der Struktur wird durch das Flag ITEMTEXT bestimmt.

Command

Hier steht das Zeichen, auf das Intuition als Shortcut, also Abkürzung mit der rechten Amiga-Taste, reagieren soll. Außerdem muß das Flag COMMSEQ gesetzt sein.

SubItem

Soll das Menü noch Untermenüs haben, steht hier ein Zeiger auf eine Liste aus MenuItem-Strukturen. Falls es sich bei dem aktuellen Menü schon um ein Untermenü handelt, wird dieser Wert ignoriert.

NextSelect

In diesem Feld steht die Nummer des nächsten selektierten Menüs oder sie ist gleich MENUNULL.

Eine bisher wenig bekannte und daher auch selten benutzte Möglichkeit ist das Auswählen mehrerer Menüpunkte in einem Schritt. Zu diesem Zweck kann der Benutzer die rechte Maustaste drücken, ein Menü ansteuern und dort einen Punkt mit der linken Maustaste auswählen (während er die rechte Maustaste gedrückt hält). Dies kann er mehrfach wiederholen. Sobald er die rechte Maustaste losläßt, wird eine IDCMP-Mesage (falls das IDCMP-Flag `MENUPICK` gesetzt ist) an den UserPort des Fensters geschickt, in der das erste ausgewählte Menü verzeichnet ist. In dem Feld `NextSelect` dieses Menüpunktes wird dann die Nummer des nächsten Menüpunktes stehen, und so weiter, bis schließlich der letzte ausgewählte Punkt den Wert `MENUNULL` enthält. Die Abarbeitung dieser Funktionen bedeutet zwar etwas mehr Programmieraufwand, sollte aber auf jeden Fall durchgeführt werden, um die Arbeit mit den Menüs für den Benutzer zu beschleunigen.

60543 Menüpunkte – und wie man damit umgeht

Zugegeben, dies ist eine recht seltsame Zahl. Doch im Rahmen der Abarbeitung der Menüs im Programm muß man eine Zahl abfragen, die eng im Zusammenhang mit dieser Zahl steht: die Menünummer. Diese setzt sich so zusammen:

S S S S S	U U U U U U	M M M M M
5 Bits	6 Bits	5 Bits
Submenünummer	Menüpunkt	Menünummer

In einem C-Programm würde zum Beispiel die Abfrage wie folgt aussehen (die Funktion `MenuNumber` liefert die Nummer des Menüs in obigem Format; `DoMenus()` verarbeitet die drei Variablen `MenuNum`, `ItemNum` und `SubNum`):

```
num = MenuNumber();
while ( num != MENUNULL ) {
    MenuNum = MENUNUM(num);
    ItemNum = ITEMNUM(num);
    SubNum = SUBNUM(num);
    DoMenus();
    menuitem = ItemAddress (my_menus, num);
    num=menuitem->NextSelect;
}
```

War einer der Teile `MENUNUM`, `ITEMNUM` oder `SUBNUM` nicht ausgewählt, so enthält er den Wert `NOMENU`, `NOITEM` bzw. `NOSUB`.

Requester und Alerts

Fehlermeldungen sind nicht nur für den Benutzer ärgerlich, sondern auch ein besonderer Punkt für den Programmierer. Wo soll man sie plazieren, wie sollen sie aussehen, wie bekommt man die Aufmerksamkeit des Benutzers? Doch das Betriebssystem des Amigas gibt dem Programmierer die Möglichkeit, mittels Requestern und Alerts solche unangenehmen Ausgaben (natürlich nicht nur die) halbwegs standardisiert auszugeben. Dies hat vor allem für den Benutzer den Vorteil, daß er sich nicht bei jedem Programm erst einmal überlegen muß, wo denn hier die Fehlermeldungen auftauchen.

Die (für den Programmierer) einfachere Methode ist dabei der Alert. Hierbei braucht er nämlich nur einen Zeiger auf einen Text sowie die Größe der Alert-Box übergeben, dazu noch eine Alert-Nummer, die allerdings zur Zeit kaum Bedeutung hat. Das Ganze sieht also in einem Programm beispielsweise so aus:

```
DisplayAlert(RECOVERY_ALERT, Text, Höhe);
```

"Höhe" ist dabei die Höhe, die der Alert haben soll.

Der Zeiger auf den Alert-String, in diesem Fall "Text", ist aus einzelnen Strings folgender Struktur aufgebaut:

Als erstes kommt die X- und Y-Position des Strings, wobei erstere als 16-Bit-, die Y-Koordinate als 8-Bit-Zahl übergeben wird. Danach folgt der eigentliche String, der mit einem NULL-Byte beendet sein muß. Als letztes kommt das sogenannte Continuation Byte. Ist dieses ungleich 0, so folgt ein weiterer String gleicher Struktur. Ist es 0, ist der Alert-Text damit beendet und wird angezeigt. Der Return-Wert gibt an, wie der Benutzer den Alert quittiert hat. Ist D0 = 0, so wurde die rechte Maustaste betätigt, ansonsten wurde der Alert mit der linken Maustaste beendet.

Wenn man jedoch den Benutzer nicht so sehr erschrecken will, sollte man doch den "normalen" Requester vorziehen. Dabei ist auch hier eine einfache Version von Seiten des Systems zu beziehen: die Routine `AutoRequest()`. Hierbei braucht man im Prinzip nur einige wenige Werte zu übergeben. Das Erzeugen und Verwalten des Requesters sowie die Abfrage der Gadgets wird vom System übernommen:

```
AutoRequest(win, Text, Ja, Nein, JaFlags, NeinFlags, X, Y)
```

Als erstes Argument braucht `AutoRequest` dabei einen Zeiger auf ein Fenster, zu dem der Requester geöffnet werden soll. Ist dieser Wert `NULL`, so wird der Requester auf der Workbench-Screen geöffnet. Das zweite Argument ist ein Zeiger auf eine `IntuiText`-Liste, die in dem Requester ausgegeben wird. Die nächsten beiden Werte sind ebenfalls

wieder Zeiger auf IntuiText-Listen. Einer der beiden, nicht jedoch beide, kann weglassen werden, es wird dann nur ein Requester mit einem Gadget geöffnet. Die nächsten beiden Flags sind für spezielle Beantwortungen des Requesters zuständig. Hier kann man beliebige IDCMP-Codes angeben, so zum Beispiel DISKINSERTED, um den Requester durch Einlegen einer Disk zu beenden. X und Y geben an, wie groß der Requester sein soll. Man sollte darauf achten, daß die Texte nicht größer als der Requester selber sind. Zwar kann dabei nichts passieren, da automatisch ein Clipping an den Requester-Rändern vorgenommen wird, doch sieht es nicht allzu ästhetisch aus, wenn die Hälfte vom Text nicht zu sehen ist.

Die zweite Möglichkeit, einen Requester zu erzeugen, ist, ihn praktisch von Anfang an selbst zusammenzusetzen. Dabei muß man der Funktion Request() einen Zeiger auf eine Struktur Requester sowie einen Zeiger auf eine Window-Struktur übergeben. Diese Window-Struktur wird dann als Requester geöffnet. Zu beachten ist dann allerdings, daß man mindestens ein Gadget mit gesetztem ENDGADGET-Flag in der Gadget-Liste hat, ansonsten wird der Requester nie beendet.

Intuition – IO für Anfänger?

Alle IO-Operationen (IO: englisch: Input/Output, deutsch: Ein-/Ausgabe) werden vom Amiga-System fast ausschließlich gleich behandelt. Doch da wir uns hier im Rahmen von Intuition bewegen, bleiben von den vielen Möglichkeiten der IO-Geräte nur einige wenige übrig, als da wären Maus und Tastatur, die als primäre Eingabe-Geräte verwendet werden, und der Bildschirm, wo in einzelnen Fenstern die Ergebnisse der Eingaben zu finden sind (sofern der Rechner nicht mal wieder abgestürzt ist...).

Eingaben leicht gemacht!?

Von einigen fortgeschrittenen Programmierern hört man manchmal, daß die Eingabebehandlung über Intuition viel zu primitiv sei, daß der "wahre Profi" über die diversen Devices oder gar direkt an die Hardware geht. Daß diese Aussage nicht nur ein wenig überheblich, sondern auch falsch ist, wollen sie meist nicht einsehen. Gut, es mag Stellen geben, an denen die direkte Verwendung der Console.device-Daten nötig ist, manche Spiele kommen vielleicht nicht ohne Abfrage der Hardware aus. Doch um ein Programm zu schreiben, das zum einen multitasking-fähig, zum anderen garantiert noch unter Kickstart 3.0 (?) läuft (sofern Commodore sich an seine eigenen Vereinbarungen hält...), muß man einen anderen Weg gehen, der einen sicher zum Ziel führt: die Eingabe-Abfrage mittels IDCMP-Events.

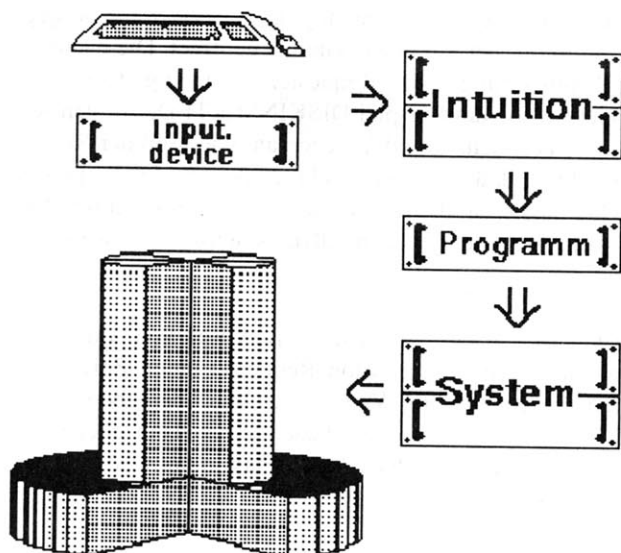


Abb. 2.1: IDCMP-Events im System

Um den Ablauf einer Eingabe zu verstehen, sehen wir uns an dieser Stelle einmal die Abbildung 2.1 an. Man sieht, wie die Eingaben von Maus und Tastatur über das Input.device an Intuition geliefert werden, das wiederum mit dem Programm kommuniziert. Dies geschieht mittels der Intuition-Messages, die mit einigen wenigen Hand- bzw. Programmgriffen bearbeitet werden können. Doch bevor man solche Nachrichten empfangen kann, muß zuerst einmal ein Briefträger da sein, der sie einem zustellt. Dies wird im Fall des Amigas von Exec übernommen, das sich dazu der Message-Ports bedient. Doch wie macht man das denn nun?

Zuerst einmal braucht man ein Fenster, das, sobald irgendwelche IDCMP-Flags gesetzt sind, auch über einen Message-Port verfügt. Also gut, öffnen wir also ein Fenster, mit dem wir die Tastatur abfragen wollen. Die dazugehörige NewWindow-Struktur könnte beispielsweise so aussehen:

```
struct NewWindow windef = {
    0,0,640,100,          /* Größe */
    0,1,                  /* DetailPen, BlockPen */
    CLOSEWINDOW VANILLAKEY, /* IDCMP-Flags */
    WINDOWDRAG WINDOWCLOSE, /* Flags */
```

```

NULL, NULL,
(UBYTE *) ">>> Test-Fenster ! <<<", /* Titel */
NULL, NULL,
0, 0, -1, -1, WBENCHSCREEN
};

```

Wie man sieht, sind zwei IDCMP-Flags gesetzt: CLOSEWINDOW und VANILLA-KEY. Während die erste nur für das Close-Gadget des Fensters zuständig ist, sorgt die zweite dafür, daß wir benachrichtigt werden, sobald eine Taste gedrückt wird.

Warten – aber richtig!

Weil Programme sowieso selten etwas anderes tun, warten wir auch bei der Abfrage der Messages. Hier gibt es zwei Wege – einen guten und einen schlechten. Der schlechte ist das Busy-Wait, also die ständige Abfrage des Message-Ports im Programm. Sie zieht selbst bei niedriger Priorität unnötig viel Rechenzeit und sollte daher unter allen Umständen vermieden werden.

Richtig dagegen macht man es, wenn man mit WaitPort() oder Wait() wartet. Hierbei wird der eigene Task vom Scheduler in den Zustand Waiting gesetzt, das Programm kehrt erst dann in den Ready-Zustand zurück, wenn die Bedingung, im unseren Fall die Ankunft der Message, erfüllt ist:

```

struct IntuiMessage *msg, *GetMsg();
struct Window *win;
:
:
main() {
:
:
    WaitPort(win->UserPort);

```

Das wäre also der Teil. Als nächstes holen wir die Message ab:

```
msg=GetMsg(win->UserPort);
```

Um den IDCMP-Code der Message herauszufinden, muß das Feld Class in der Message abgefragt werden:

```

while (msg) {
    my_IDCMP = msg -> Class;

```

Nun können wir aufgrund der IDCMP-Class verzweigen:

```
switch (my_IDCMP) {
    case CLOSEWINDOW:
        :
        :
        break;
    case VANILLAKEY:
        :
        :
        break;
}
```

Als letztes müssen wir noch den Empfang der Message quittieren; dies geschieht mit ReplyMsg():

```
ReplyMsg(msg);
```

Da Wait/WaitPort nur auf neue Messages reagiert, müssen wir vor Aufruf von weiteren Wait/WaitPort noch die restlichen Messages abrufen. Dies wurde bereits durch while (msg) vorbereitet, wir brauchen hier nur noch zu schreiben:

```
msg=GetMsg(win->UserPort);
}
```

Das war's schon. Doch im Zusammenhang mit WaitPort() kann es auch manchmal zu einem kleinen Problem kommen: Was macht man, wenn man die Ports von zwei oder mehr Windows abfragen will? In so einem Fall muß man auf den Befehl Wait() ausweichen, da er mehrere Bits abfragen kann. Für die beiden Windows win1 und win2 würde dies dann beispielsweise so aussehen:

```
Wait(1<<(win1->UserPort->mp_SigBit)
     1<<(win2->UserPort->mp_SigBit));
```

Sobald an einem der beiden Ports eine Message ankommt, wird das mp_SigBit gesetzt, und das Programm kann die Message(s) von den Ports abholen.

Zum Abschluß der Messages wollen wir noch einen kurzen Blick auf die Struktur der Intuition-Messages werfen, da hier noch einige andere Informationen als nur die Art der IDCMP-Message abzulesen sind:

```
struct IntuiMessage
```

```

{
    struct Message ExecMessage;
    ULONG Class;
    USHORT Code;
    USHORT Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
};

```

In ExecMessage werden die Daten, die Exec für die Verwaltung der Message braucht (Empfänger, ReplyPort), verwaltet. In Class wird, wie schon angesprochen, die Art der IntuitionMessage übergeben, Code enthält Zusatzinformationen (selektiertes Menü, gedrückte Taste). In Qualifier werden eventuell gedrückte Qualifier-Tasten (Shift, Alt etc.) übergeben. In IAddress findet man z.B. solche Dinge wie einen Zeiger auf Gadget, sofern dies selektiert wurde. Damit kann man dann leicht die GadgetID herausbekommen:

```
GadID = (struct Gadget *) (msg->IAddress)->GadgetID
```

In MouseX und MouseY findet man die Position der Maus zu dem Zeitpunkt, als die Message abgeschickt wurde; ebenso steht die Uhrzeit in Seconds und Micros. IDCMPWindow ist ein Zeiger auf das Fenster, zu dem die Intui-Message gehört. Der letzte Zeiger ist für das System vorbehalten.

Grafik unter Intuition

Um die hervorragende Grafik des Amigas auszunutzen, stellt das Betriebssystem eine Reihe von Routinen zur Verfügung. Doch sind sie nicht nur in der Graphics-Library zu finden, sondern auch bei Intuition.

Im Prinzip hat Intuition drei Grundtypen von Grafik-Unterstützung: Images (Grafiken), Borders (Rahmen) und IntuiTexte. Zur Darstellung kümmert sich, im Gegensatz zu den Routinen der Graphics-Library, Intuition fast vollständig. Der Programmierer braucht nur relative Werte anzugeben, dazu noch die Information, worin das jeweilige Element dargestellt werden muß.

Dazu kommt noch, daß man mittels Verkettung einzelner Elemente mit Pointern viele Rahmen, Images oder Texte mit einem Betriebssystemaufruf durchführen kann.

Border

Einen Rahmen (der nicht nur ein Rechteck sein muß) definiert man mit folgender Struktur:

```
struct Border
{
    SHORT          LeftEdge, TopEdge;
    UBYTE          FrontPen, BackPen;
    UBYTE          DrawMode;
    BYTE           Count;
    SHORT          *XY;
    struct Border  *NextBorder;
};
```

Mit LeftEdge und TopEdge wird dabei die linke obere Ecke definiert, zu der die Koordinaten des Rahmens relativ gezeichnet werden. FrontPen und BackPen werden vor dem Zeichnen der Grafik gesetzt (wobei BackPen zur Zeit nicht benutzt wird), ebenso der DrawMode. (Da nur der FrontPen beachtet wird, sind nur JAM1 und XOR sinnvoll.)

In Count steht die Anzahl der zu zeichnenden Eckpunkte. XY ist der Zeiger auf das Array, in dem die einzelnen Koordinaten-Paare zu finden sind.

Der letzte Wert sorgt dafür, daß andere Rahmen, falls vorhanden, automatisch gezeichnet werden. Zu diesem Zweck schreibt man einfach die Adresse der nächsten Border-Struktur hinein.

Ist die Border-Struktur fertig definiert, kann man sie entweder im Zusammenhang mit einem Gadget (über GadgetRender und SelectRender) oder mit der Routine DrawBorder() ausgeben.

Intuition – Texte

Bei der Verwendung von IntuiText-Strukturen kann man zum einen, wie auch bei den Borders, Texte verketteten, so daß viele Texte mit einem Aufruf ausgegeben werden. Zum anderen kann man jedem Text seine eigene Farbe, einen Stil und Font zuordnen:

```
struct IntuiText
{
    UBYTE          FrontPen, BackPen;
    UBYTE          DrawMode;
    SHORT          LeftEdge;
    SHORT          TopEdge;
    struct TextAttr *ITextFont;
    UBYTE          *IText;
    struct IntuiText *NextText;
};
```

FrontPen, BackPen und DrawMode dürften inzwischen bekannt sein. LeftEdge und TopEdge geben die Position des Textes an. Mit dem ITextFont kann man auf einen anderen Font umschalten. Will man nur den Standard-Font benutzen, trägt man hier NULL ein. In IText schließlich steht der Text selber, der mit einer Null (0) enden muß. Mit NextText verkettet man mehrere IntuiTexte. Auch diese Struktur wird vielfältig verwendet, so zum Beispiel in Menüs und Gadets. Um eigene Ausgaben machen zu können, steht die Funktion PrintIText() zur Verfügung.

Images

Images werden in der Intuition-Oberfläche recht häufig benutzt. Dazu gehören die Bilder in den Size, Close- und Depth-Gadgets, die Pfeile in den Scroll-Bars der Workbench-Windows und vieles mehr. Diese Images werden auch wieder mit einer Struktur verwaltet, die folgendermaßen aussieht:

```
struct Image
{
    SHORT          LeftEdge, TopEdge;
    SHORT          Width, Height;
    SHORT          Depth;
    USHORT         *ImageData;
    UBYTE          PlanePick, PlaneOnOff;
    struct Image   *NextImage;
};
```

LeftEdge, TopEdge, Width und Height geben dabei Position und Größe an, Depth definiert die Anzahl der Bitplanes. In dem Array ImageData werden die eigentlichen Grafik-Daten abgelegt. (Achtung! Dieses Array muß im Chip-Ram liegen!) PlanePick und PlaneOnOff bieten eine einfache Möglichkeit, Speicher zu sparen. Für alle Bitplanes, in denen Grafik dargestellt werden soll, wird in PlanePick das jeweilige Bit auf 1 gesetzt. Sobald eine Bitplane komplett 0 oder 1 ist, läßt man das dazugehörige Bit auf 0 und definiert an der selben Stelle in PlaneOnOff, auf welchen Wert die Bitplane gesetzt werden soll. Als Beispiel dafür soll die folgende Grafik dienen:

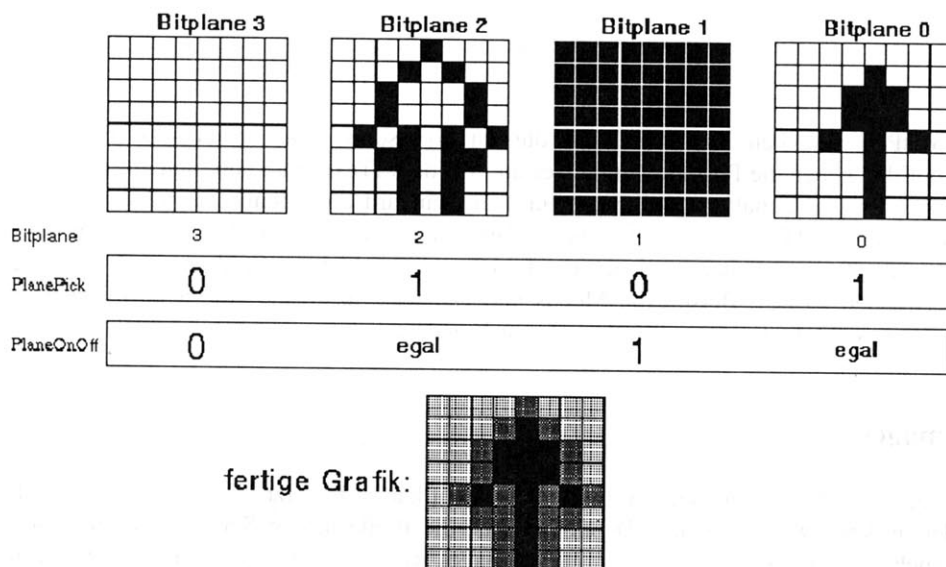


Abb. 2.2: Verwendung von PlanePick und PlaneOnOff

Wie man sieht, haben die Bitplanes 0 und 2 Grafik, Bitplanes 1 und 3 sind ganz 1 bzw. 0. Damit erhält man für $\text{PlanePick} = 2^0 + 2^2$, PlaneOnOff ist gleich 2^1 .

Was es sonst noch so gibt ...

Eine Funktion, die man entweder leicht vergißt oder aber gern Exec zuordnet, ist die Speicher-Anforderung, wobei vom System Buch geführt wird, basierend auf der Funktion AllocRemember. Zu dem Zweck muß erst einmal eine Struktur Remember vorbereitet sein:

```
struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};
```

Um die Funktion zu verwenden, muß das Programm etwa so aussehen:

```
struct Remember *MeinSpeicher;
char *Buf1, *Buf2;

:
MeinSpeicher = NULL;
:
Buf1 = AllocRemember (&MeinSpeicher, 512L,
                      MEMF_CHIP MEMF_CLEAR);
Buf2 = AllocRemember (&MeinSpeicher, 10240L,
                      MEMF_PUBLIC);
..(Prüfen ob erfolgreich)..
:
FreeRemember (&MeinSpeicher, TRUE);
```

Wie man sieht, werden an die Funktion AllocRemember dieselben Parameter übergeben, wie sie auch AllocMem bekommt. Zusätzlich dazu wird jedoch noch die Adresse des Pointers auf die Remember-Struktur übergeben. Der Zeiger muß dabei anfangs auf Null gesetzt sein.

Zwar verbraucht man mit AllocRemember pro belegtem Speicherblock 12 Byte mehr, doch hat man die Möglichkeit, den gesamten belegten Speicher mit einem Aufruf von FreeRemember wieder freizugeben. Und sollte man den Speicher der Remember-Strukturen benötigen, so kann man ihn auch gesondert freigeben:

Wird FreeRemember nämlich mit dem zweiten Parameter FALSE aufgerufen, so wird nur der Speicher der Verwaltungsstrukturen zurückgegeben, nicht jedoch der andere Speicher.

"Ordentliche" WB-Programmierung

So ziemlich alle Programmierer, die Utilities für den "normalen", sprich: Workbench-Gebrauch programmieren, denken nicht an einen Sonderfall: Versucht irgendein Programm, Speicher im System durch Schließen der Workbench zu sparen, kann die Funktion `CloseWorkBench` dies nicht tun, sobald auch nur ein einziges Fenster einer Anwendung offen ist.

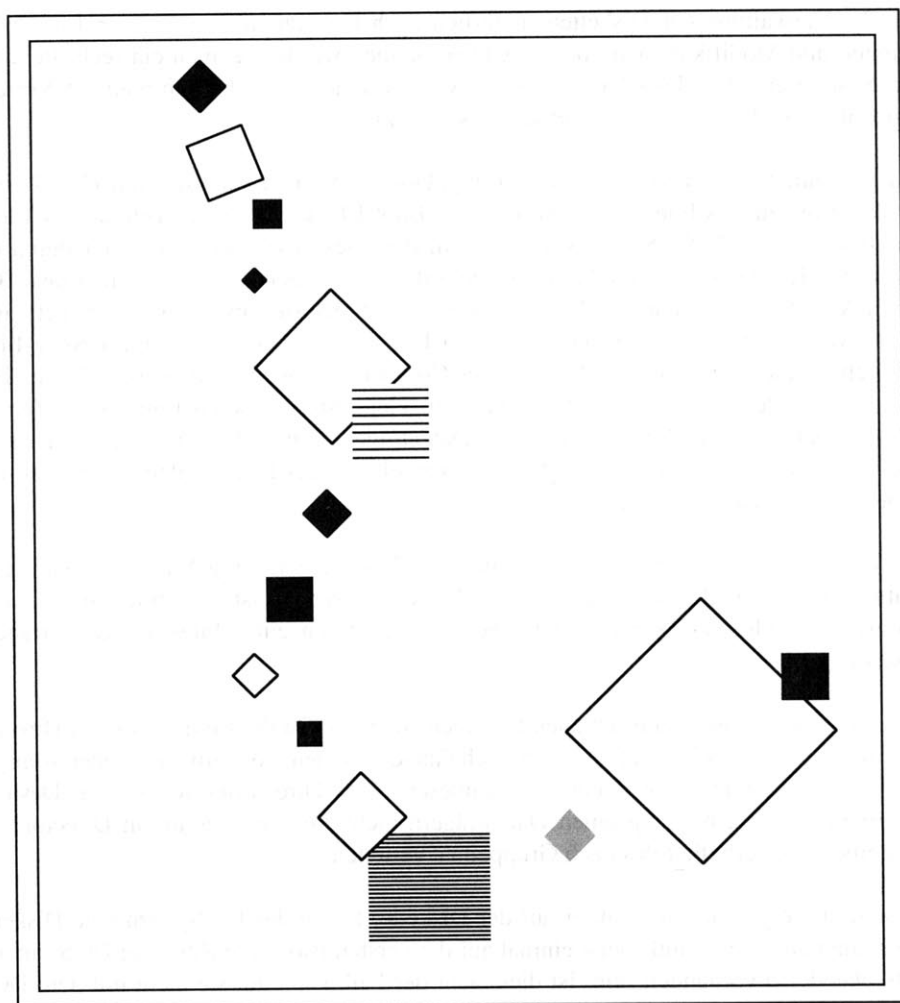
Um dies zu verhindern, gibt es die Routine `AlohaWorkbench`. Damit gibt man der Workbench die Information, daß man, falls die Workbench geschlossen werden soll, unter einem bestimmten Port zu erreichen ist. Sobald dann `CloseWorkBench` oder `OpenWorkBench` aufgerufen wird, schickt Intuition an alle eingetragenen Ports eine `IntuitionMessage` mit `Class=WBENCHMESSAGE`. In Code wird `WBENCHCLOSE` bzw. `WBENCHOPEN` übergeben.

Hat man sich allerdings angemeldet, so muß man die Messages auch wirklich bearbeiten, d.h. die Fenster schließen, bevor man die Message "replied", da Intuition davon ausgeht, daß alle Arbeiten nach einem `ReplyMsg` beendet sind.

`AlohaWorkBench` erhält entweder die Adresse eines MessagePorts oder 0. Die erste Variante dient zum Anmelden, mit der zweiten meldet man sich bei Intuition wieder ab.

Kapitel 3

Das Disk Operating System



Disketten-Format

Das FileSystem des Amigas verwaltet Disketten mittels einer Reihe von Blöcken sowie einiger Zusatzdaten auf den Datenblöcken selber. Dabei taucht auch in gewissem Maße Redundanz, als doppelte Speicherung der Verwaltungsinformationen, auf. Daher sind auf "normalen" DOS-Disketten statt 880 KByte lediglich maximal 835 KByte. Warum das so ist, und wie diese Blöcke aufgebaut sind, erfahren Sie in diesem Abschnitt.

Da die Verwaltung der Disketten (natürlich auch Festplatten) mit leichten Einschränkungen und Modifikationen von Unix übernommen wurde, hat man ein recht flexibles FileSystem erhalten. Dies hat eine Reihe von Vorteilen, jedoch kann man nie Vorteile ganz ohne Nachteile erhalten, und so ist es auch hier.

Zuerst einmal zu den Vorteilen. Das AmigaDos-File-System kann theoretisch auf einer Diskette bis zu 878 Files ($((1760 \text{ Blöcke} - 2 \text{ Boot-Blöcke} - 1 \text{ Block Bitmap} - 1 \text{ Block Rootblock}) / 2 = 878)$). Somit ist die Anzahl der Files pro Diskette also nur durch die Disketten-Kapazität beschränkt (im Gegensatz z.B. zu den alten Versionen des TOS beim Atari ST, das ja nur eine kleine Anzahl von Directories und Files pro Diskette und Platte verwalten konnte). Im Gegensatz zu Unix wurde jedoch auf eine I-Node-Liste verzichtet, die sich bei dem Single-User-Computer Amiga nicht gelohnt hätte. Des weiteren werden Directories nicht als Files abgelegt. Statt dessen hat man alle Informationen über das File (Position auf dem Datenträger, Länge, Datum, Name) in einem (bzw. mehreren) Blöcken abgelegt, deren Verzeichnis als Liste in dem jeweiligen Parent-Directory zu finden ist.

Da die Einträge in dieser Tabelle mittels Hashing (die Eintrag-Nummer erhält man mittels einer Formel aus dem File-Name) berechnet werden, ist der Amiga in der Lage, so ziemlich alle gängigen Computer beim Lokalisieren eines Files auf Datenträgern abzuhängen.

Doch dies alles sorgt natürlich auch für einen Nachteil. Da die Einträge in den Directories nicht mehr schön chronologisch nacheinander stehen, sondern mehr oder weniger gestreut auf der Diskette, dauert das Einlesen eines Directories recht lang. Deshalb empfiehlt es sich, bei langsamen Datenträgern nicht zuviele Files in ein Directory zu schreiben, sondern die Files nach Gruppen zu verteilen.

Nun zu der eigentlichen Struktur auf der Diskette. Damit das FileSystem eine Diskette überhaupt annimmt, muß zuerst einmal auf dem ersten Block die Kennung DOS mit abschließender 0 vorhanden sein. Ist dies nicht der Fall, kann auf sie nicht mit dem DOS (aber trotz allem natürlich auf direktem Weg über das Trackdisk-Device) zugegriffen. Als nächstes wird zu dem sogenannten Root-Block, also der "Wurzel", verzweigt.

Der Root-Block

Der Root-Block entspricht in seinem Aufbau im großen und ganzen dem eines normalen Directories, mit der Ausnahme, daß er kein Parent-Directory hat und sein sekundärer Block-Typ anders ist. Außerdem steht er an einer festen Stelle: bei Disketten ist das Block 880, allgemein gesprochen: in der Mitte des Datenträgers. Damit wird erreicht, daß die durchschnittliche Zugriffszeit kürzer ist als bei einer anderen Position.

0	T.SHORT	primärer Block-Typ
1	0	Header
2	0	Highest Sequence
3	HT Size	Größe der Hash-Tabelle
4	0	
5	Checksum	Prüfsumme
6	Hash Table	
	⋮	
L-51		
L-50	BM Flag	Bitmap korrekt ?
L-49	Bitmap pages	Bitmap Blöcke
	⋮	
L-24		
L-23	Days	Datum des
L-22	Mins	letzten
L-21	Ticks	Schreibzugriffs
L-20	Disk	Name der Disk
	Name	(BCPL-String)
L-7	CreateDays	Datum, an dem
L-6	CreateMins	Disk angelegt
L-5	CreateTicks	(formatiert) wurde
L-4	0	Hash Chain
L-3	0	Parent
L-2	0	Extension
L-1	ST.ROOT	sekundärer Block-Typ

Abb. 3.1: Der Root-Block

Anmerkung zur Grafik: Werte, die mit Zahlen besetzt sind, haben zwar eine bestimmte Bedeutung, werden aber vom "normalen" FileSystem ausschließlich mit der jeweiligen Zahl belegt. Auf diese wird auch nicht weiter eingegangen.

Die HashTableSize gibt an, wieviele Long-Words, und damit Einträge, die HashTable aufnehmen kann. Die Prüfsumme sorgt dafür, daß bei Addition ohne Übertrag aller Words im Block Null ergibt.

In der HashTable stehen die Blöcke der Einträge im Root-Directory. Das BitMap-Flag gibt an, ob die BitMap, also das Verzeichnis der freien Blöcke, gültig ist (TRUE für gültig, FALSE für validate). Darauf folgen Long-Words, in denen die Bitmap-Blöcke verzeichnet sind.

Auf jedem dieser Blöcke können 4064 Blöcke verwaltet werden ($512 \text{ Byte} * 8 - 1$ Long-Word Checksum), somit ist die maximale Datenträger-Kapazität (pro Partition) auf 105664 Blöcke beschränkt ($4064 \text{ Blöcke/Bitmap-Block} * 26 \text{ Bitmap-Blöcke}$), also etwa 52MB.

Als nächstes folgt das Datum, an dem zuletzt auf die Disk geschrieben wurde.

Der Name der Diskette wird als BCPL-String abgespeichert, d.h. zuerst kommt die Länge des Strings als Byte, danach der String selber. Das CreationDate, also der Zeitpunkt, zu dem die Diskette formatiert wurde, wird unter anderem auch vom DOS verwendet, um zwei Disketten gleichen Namens zu unterscheiden.

Das letzte Long-Word unterscheidet den Root-Block vom normalen Directory-Block, in dem statt ST.ROOT ST.USERDIR steht.

Der Directory-Block

Der Platz, der durch das Fehlen der Bitmap-Informationen im Root-Block bei Directory-Blöcken freigeworden ist, wird für die Protection-Bits (HSPARWED) sowie für den Comment, also die Bemerkungen zum Directory, verwendet.

Außerdem sind hier die beiden Werte für den nächsten Eintrag in der Hash-Chain sowie das Parent-Directory belegt. An dieser Stelle stellt sich die Frage, was die Hash-Chain überhaupt ist.

Der Amiga verwaltet die File-Namen über ein Hashing, also die Berechnung einer Position im Directory mittels des File-Namens. Dabei entstehen jedoch doppelte Belegungen (so haben z.B. die Namen zoo, fc und runb denselben Hash-Wert).

Statt jedoch für solche Fälle neue Directory-Blöcke anzulegen, wo diese Files verzeichnet werden, wird an der Position HashChain der Block eingetragen, in dem die File- oder Directory-Informationen zu finden sind.

0	T.SHORT	primärer Block-Typ
1	Header Key	Zeiger auf sich selbst
2	0	Highest Sequence
3	0	
4	0	
5	Checksum	Prüfsumme
6	Hash Table	
	:	
L-51		
L-50	(unbenutzt)	
L-48	Protect	Protection-Bits
L-47	0	(unbenutzt)
L-46	Comment	Dir-Comment
	:	(BCPL-String)
L-24		
L-23	Days	Datum des
L-22	Mins	letzten
L-21	Ticks	Schreibzugriffs
L-20	Directory Name	Name des Directories (BCPL-String)
L-4	Hashchain	Nächster Eintrag (s. Text)
L-3	Parent	vorhergehendes Directory
L-2	0	Extension
L-1	ST.USERDIR	sekundärer Block-Typ

Abb. 3.2: Der Directory-Block

Der File-Header-Block

Auch hier ist die erste Kennung des Blocks T.SHORT, zur Unterscheidung vom Directory ist die sekundäre Kennung ST.FILE. In der HighestSeq wird verzeichnet, wie viele Blöcke der maximal von einem Block verwalteten 72 verwendet werden.

0	T.SHORT	primärer Block-Typ
1	Header Key	Zeiger auf sich selbst
2	Highest Seq	Anzahl der Block-Verweise im Block
3	Data Size	Anzahl der verwendeten Block-Zeiger (s. Kommentar)
4	First Data	Erster Datenblock
5	Checksum	Prüfsumme
6	:	Zeiger auf die Datenblöcke
	Data Blk 1	
	Data Blk 0	
L-51	(unbenutzt)	
L-48	Protect	Protection-Bits
L-47	Byte Size	Größe des Files in Bytes
L-46	Comment	File-Comment
	:	(BCPL-String)
L-24		
L-23	Days	Datum des
L-22	Mins	letzten
L-21	Ticks	Schreibzugriffs
L-20	File	Name des Files
	Name	(BCPL-String)
L-4	Hashchain	Nächster Eintrag (s. Text)
L-3	Parent	Zeiger auf Directory
L-2	Extension	Zeiger auf nächsten Block
L-1	ST.FILE	sekundärer Block-Typ

Abb. 3.3: Der File-Header-Block

Data Size soll eigentlich die Anzahl der verwendeten Datenblock-Einträge enthalten, doch wird von den aktuellen DOS-Versionen hier nur 0 eingetragen.

Die restlichen Werte sind schon von den anderen Blöcken her bekannt, lediglich der Extension-Block ist bisher noch nicht in Aktion getreten.

Jeder File-Header-Block kann nur maximal 72 Blöcke verwalten. Sobald mehr als diese Blöcke in einem File vorhanden sind, wird ein File-List-Block angelegt, in dem die restlichen Blöcke verzeichnet sind. Seine Block-Adresse wird an der Extension-Stelle eingetragen.

Der File-List-Block

Im Prinzip ist dieser Block identisch mit dem normalen File-Header-Block, mit der Ausnahme, daß Informationen über File-Namen, Protection-Bits und Comment natürlich nicht nötig sind. Demzufolge können in jedem File-List-Block auch wieder nur 72 Blöcke notiert werden. Zudem wird in dem Parent-Feld der Zeiger auf den File-Header-Block eingetragen.

0	T.LIST	primärer Block-Typ
1	Header Key	Zeiger auf sich selbst
2	Block Count	Anzahl der verwendeten Block-Zeiger
3	Data Size	s. Kommentar File-Header-Block
4	First Data	Erster Datenblock
5	Checksum	Prüfsumme
6	⋮	Zeiger auf die Datenblöcke
	Data Blk N+1	
	Data Blk N+0	
L-51	(unbenutzt)	
L-50		
L-4	0	hash Chain
L-3	Parent	File Header Block
L-2	Extension	Zeiger auf nächsten Block
L-1	ST.FILE	sekundärer Block-Typ

Abb. 3.4: Der File-List-Block

Der Data-Block

Nach all diesen Verwaltungsblöcken sind wir endlich dort gelandet, wo die Daten sind, die wir brauchen. Leider (manchmal aber zum Glück) werden auch hier noch eine Reihe zusätzlicher Daten gespeichert. Angefangen mit dem Typ des Blocks, gefolgt vom Zeiger auf den File-Header-Block, dem Block-Zähler, der Länge der Daten im Block und dem Zeiger auf den nächsten Datenblock. Lediglich der Checksum mag man noch einen gewissen Wert zusprechen. Doch so schlecht diese Redundanz (immerhin etwa 4,5% pro Block) für die Kapazität des Datenträgers ist, so nützlich ist sie beim Wiederherstellen einer defekten Disk-Struktur. Bei diesem FileSystem können sämtliche Verwaltungsblöcke gelöscht sein, und dennoch kann die gesamte Struktur (bis auf die Namen, Datum, Protection ...) der Disk wieder auf den alten Stand gebracht werden.

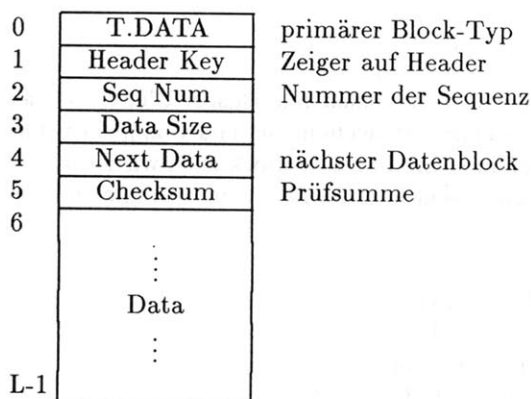


Abb. 3.5: Der Data-Block

Das FastFileSystem, das vor allem für Festplatten recht nützlich ist, spart sich diese 24 Byte am Anfang jedes Datenblocks, wobei man pro Megabyte altes FileSystem etwa 50 KByte mehr an Daten unterbringen kann. Doch sind hier einmal die Header-Informationen eines Files defekt, hat man so gut wie keine Möglichkeit mehr, es noch zu restaurieren.

Das FastFileSystem

Ein FileSystem hat die Aufgabe, dem Programmierer die Verwaltung von Speichermedien abzunehmen. Es muß Funktionen zum Erzeugen, Ändern und Lesen von Files bereitstellen, Inhaltsverzeichnisse anlegen und verwalten. Dabei stützt es sich meist auf physikalische Devices, die das Interface zur Hardware darstellen. Es ist aber auch möglich, daß ein FileSystem seine Ressourcen selbst verwaltet (Ram-Handler).

Es gibt noch eine abgespeckte Version eines FileSystems, das ist der Handler. (Der Ram-Handler heißt zwar Handler, ist aber ein FileSystem!) Ein Handler unterstützt weniger Funktionen als ein FileSystem, ihm fehlen z.B. alle Funktionen, die mit Directories arbeiten.

Auf dem Amiga ist ein FileSystem als Process realisiert, dem der User seine Wünsche mitteilt. Dieser macht dann die erforderlichen Zugriffe und meldet eventuell aufgetretene Fehler. Die Befehle werden dem FileSystem in Form sogenannter Packets

(englisch packet = Paket) übergeben. Das ist eine normale Exec-Message, die im Feld `mn_Node.In_Name` einen Zeiger auf eine `DosPacket`-Struktur enthält. Diese Struktur sieht wie folgt aus:

```
struct DosPacket {  
    struct Message *dp_Link;  
    struct MsgPort *dp_Port;  
    LONG dp_Type;  
    LONG dp_Res1;  
    LONG dp_Res2;  
    LONG dp_Arg1;  
    LONG dp_Arg2;  
    LONG dp_Arg3;  
    LONG dp_Arg4;  
    LONG dp_Arg5;  
    LONG dp_Arg6;  
    LONG dp_Arg7;  
};
```

- | | |
|----------------------------|---|
| <code>dp_Link</code> | Ein Zeiger zurück zur entsprechenden Message. |
| <code>dp_Port</code> | zeigt auf den Port, zu dem das Packet nach Beendigung der Aufgabe zurückgeschickt wird. Erst wenn die Message wieder in diesem Port ankommt, ist das Kommando erledigt. Dieser Zeiger muß bei jedem Abschicken des Packets neu eingetragen werden, da er vom FileSystem überschrieben wird. |
| <code>dp_Type</code> | Das eigentliche Kommando. (Dazu später.) |
| <code>dp_Res1</code> | Die Rückmeldung: Hier kann <code>DOSTRUE</code> oder <code>DOSFALSE</code> (-1 oder 0) stehen oder ein <code>BPTR</code> auf ein Handle oder Lock. |
| <code>dp_Res2</code> | gibt eine nähere Fehlerbeschreibung, wenn in <code>dp_Res1</code> <code>DOSFALSE</code> zurückkam. |
| <code>dp_Arg1</code> bis 7 | Argumente, die das Kommando zur Ausführung braucht. Wieviel Argumente übergeben werden müssen und welche Bedeutung sie haben, hängt von dem auszuführenden Befehl ab. Allerdings handelt es sich immer bei Zeigern um <code>BPTR</code> und bei Zeichenketten um <code>BSTRings</code> . |

Wie man sieht, ist es mit etwas Aufwand verbunden, die Dienste eines FileSystems in Anspruch zu nehmen. Um z.B. eine Diskette umzubenennen, sind folgende Schritte nötig:

- Initialisieren eines Packets.
- Initialisieren einer Message.
- Herausfinden, an welchen FileHandler das Packet zu schicken ist.
- Die Message zum FileSystem schicken.
- Auf Rückkehr des Packets warten.

Um dies zu vereinfachen, gibt es die `dos.library`. In ihr sind die meisten Packets in einfach aufzurufenden Funktionen erreichbar. Natürlich gibt es bei Sache einen Haken: Der Aufruf durch die `dos.library` ist synchron, d.h. man kann immer nur eine Funktion rufen.

Die FileSystem-Kommandos

Die Kommandos sind wie folgt definiert:

`ACTION_NIL` führt keine Aktion aus.

`ACTION_GET_BLOCK`
Internes Kommando.

`ACTION_SET_MAP`
Internes Kommando.

`ACTION_DIE` Das FileSystem sollte sich, wenn möglich, entfernen (Im Moment nicht implementiert!).

`ACTION_EVENT`
Internes Kommando.

`ACTION_CURRENT_VOLUME`
gibt in `Res1` einen Zeiger auf die `VolumeNode` der eingelegten Disk oder 0 (keine Disk) zurück.

`ACTION_LOCATE_OBJECT`
stellt einen Lock auf ein File oder Directory zur Verfügung:
`Arg1=Lock of ParentDir Arg2=Name Arg3=Mode`
(`ACCESS_READ` oder `ACCESS_WRITE`).

DOSCall: `Lock(Name,Mode)`.

ACTION_RENAME_DISK

gibt einer Disk einen anderen Namen: Arg1=Neuer Diskname.

ACTION_WRITE schreibt in ein File: Arg1=FileHandle Arg2=Buffer Arg3=Bytes.

DOSCall: Write(FileHandle,Buffer,Bytes).

ACTION_READ liest aus einem File: Arg1=FileHandle Arg2=Buffer Arg3=Bytes.

DOSCall: Read(FileHandle,Buffer,Bytes).

ACTION_FREE_LOCK

Freigeben eines Locks: Arg1=Lock.

DOSCall: UnLock(Lock).

ACTION_DELETE_OBJECT

Löschen eines Files: Arg1=Lock of ParentDir Arg2=Name.

DOSCall: DeleteFile(Name).

ACTION_RENAME_OBJECT

Umbenennen eines Files oder Directories: Arg1=Lock of OldParentDir Arg2=OldName Arg3=Lock of NewParentDir Arg4=NewName.

DOSCall: Rename(OldName,NewName).

ACTION_MORE_CACHE

ändert die Anzahl der Puffer eines FileSystems: Arg1=Anzahl der zusätzlichen Puffer.

ACTION_COPY_DIR

dupliziert ein Lock Arg1=Lock.

DOSCall: DupLock(Lock).

ACTION_WAIT_CHAR

wartet begrenzte Zeit auf Zeichen (Ist nur beim Console-Handler sinnvoll!): Arg1=FileHandle Arg2=Timeout in Ticks.

DOSCall: WaitForChar(FileHandle,Timeout).

ACTION_SET_PROTECT

setzt die Protection-Bits eines Files oder Dirs: Arg1=Unused
Arg2=Lock of ParentDir Arg3=Comment Arg4=ProtBits.

DOSCall: SetProtection(Name,ProtBits).

ACTION_CREATE_DIR

erzeugt Directory: Arg1=Lock of ParentDir Arg2=Name.

DOSCall: CreateDir(Name).

ACTION_EXAMINE_OBJECT

holt Infos über ein File oder Directory: Arg1=Lock
Arg2=FileInfoBlock.

DOSCall: Examine(Lock,FileInfoBlock).

ACTION_EXAMINE_NEXT

holt Infos über den nächsten Eintrag eines Directories: Arg1=Lock
auf Dir Arg2=FileInfoBlock.

DOSCall: ExNext(Lock,FileInfoBlock).

ACTION_DISK_INFO

holt Infos über eingelegte Disk: Arg1=InfoData.

ACTION_INFO

holt Infos über eingelegte Disk: Arg1=Lock Arg2=InfoData
DOSCall: Info(Lock,InfoData).

ACTION_FLUSH

Das FileSystem schreibt seine internen Puffer auf das Medium.

ACTION_SET_COMMENT

setzt Kommentar zu File oder Directory: Arg1=Unused
Arg2=Lock of ParentDir Arg3=Name Arg4=Comment.

DOSCall: SetComment(Name,Comment).

ACTION_PARENT

holt Lock auf ParentDir: Arg1=Lock.

DOSCall: ParentDir(Lock).

ACTION_TIMER Internes Packet.

ACTION_INHIBIT hält das FileSystem von Zugriffen ab: Arg1=Mode
(DOSTRUE=Kein Zugriff, DOSFALSE=Zugriff).

ACTION_DISK_TYPE
Internes Packet.

ACTION_DISK_CHANGE
Internes Packet.

ACTION_SET_DATE
ändert das Datum eines Files/Dirs: Arg1=Lock Arg2=DateStamp.

ACTION_SCREEN_MODE
schaltet Console-Handler zwischen RAW- und CON-Mode um:
Arg1=Mode (DOSTRUE=RawMode, DOSFALSE=ConMode).

ACTION_READ_RETURN
Internes Packet.

ACTION_WRITE_RETURN
Internes Packet.

ACTION_SEEK ändert die Leseposition in einem File: Arg1=FileHandle
Arg2=Mode Arg3=Offset.

DOSCall: Seek(FileHandle,Mode,Offset).

ACTION_FINDUPDATE
öffnet altes File exklusiv, d.h. es klappt nicht, wenn schon jemand
das File geöffnet hat: Arg1=FileHandle Arg2=Lock of ParentDir
Arg3=Name.

DOSCall: Open(Name,MODE_READWRITE).

ACTION_FINDINPUT
öffnet altes File: Arg1=FileHandle Arg2=Lock of ParentDir
Arg3=Name.

DOSCall: Open(Name,MODE_OLDFILE).

ACTION_FINDOUTPUT

öffnet neues File (und löscht evt. ein altes): Arg1=FileHandle
Arg2=Lock of ParentDir Arg3=Name.

DOSCall: Open(Name,MODE_NEWFILE).

ACTION_END

schließt File: Arg1=FileHandle.

DOSCall: Close(FileHandle).

ACTION_TRUNCATE

kürzt File ab Leseposition: Arg1=FileHandle.

ACTION_WRITE_PROTECT

schützt Volume vor Schreibzugriffen. Ist der Schutz schon aktiv, kann nur mit dem korrekten Code (LONG) etwas geändert werden: Arg1=Mode (DOSTRUE=geschützt, DOSFALSE= ungeschützt) Arg2=Code.

Bis Kickstart 1.2 gab es nur zwei richtige FileSystems: das FileSystem für Disketten und den Ram-Handler. Das Disk-FileSystem ist sehr flexibel geschrieben, so daß es sich die Größe des Mediums aus dem EnvironmentVector holt. (Der Mount-Befehl legt einen solchen an.) So läßt es sich nicht nur für normale Disketten verwenden, sondern auch für Festplatten. Allerdings treten dabei einige Probleme auf (keine Partition mit mehr als 52 MByte, sehr langsam), die Commodore veranlassten, ein neues FileSystem speziell für Festplatten zu schreiben: das FastFileSystem.

Dieses bietet im wesentlichen drei Vorteile gegenüber dem normalen FileSystem:

1. Es ist in Assembler geschrieben und dadurch in der Ausführung um Längen schneller als der Code des BCPL-Compilers.
2. Die Datenblöcke enthalten keine Verwaltungsinformation mehr. Dadurch entfällt ein nachträgliches Umkopieren der Daten im Speicher.
3. Liegen mehrere Datenblöcke wirklich hintereinander, werden sie auch auf einmal geladen. (Das alte FileSystem ließ sich vom Device jeden Block einzeln geben.)

Durch diese Verbesserungen liegt die Geschwindigkeit des FFS um Größenordnungen höher als die des alten FileSystems. Auch die maximale Größe einer Partition ist nicht mehr auf 52MB berenzt.

Die Datenorganisation auf Disk

Das FFS verwaltet seine Blöcke genauso wie das alte FileSystem, allerdings mit zwei Ausnahmen:

Die sechs Long-Words am Anfang eines Datenblocks fehlen. Sind also der File-Header-Block oder ein File-List-Block zerstört, sieht es sehr schlecht aus. Dafür können die Daten direkt "am Stück" von der Platte in den vom User gewünschten Bereich (`Read(FileHandle,BUFFER,Len)`) geladen werden.

Das ist natürlich wesentlich schneller, als die Daten nachträglich umzukopieren. (Allerdings kann der Spareffekt auch zunichte gemacht werden, wenn man als Buffer eine ungerade Adresse wählt: 16-Bit-DMA-Devices dürften damit ihre Schwierigkeiten haben, so daß im Endeffekt doch wieder der Prozessor kopiert.)

Die BitMap des alten FileSystems konnte maximal 26 Blöcke umfassen. Diese Einschränkung wird durch BitMapList-Blocks aufgehoben. Reichen 25 Einträge im Root-Block nicht aus, enthält der letzte Eintrag den Key auf den ersten BitMapList-Block. Dieser besteht aus 127 Keys, in denen BitMap-Blocks stehen, und dem letzten Eintrag, in dem der Key des nächsten BitMapList-Blocks steht.

Am Ende der Liste steht Key 0. Das heißt aber auch, daß im Root-Block nur noch 25 BitMap-Blocks zur Verfügung stehen, da sonst der letzte als BitMapList-Block interpretiert würde. Anders als die anderen Blöcke haben FFS-BitMapList-Blocks keine Checksum.

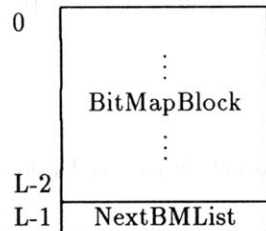


Abb. 3.6: Der FFS-BitMapList-Block

Wie man ein File findet...

Hashing beim AmigaDOS/FileSystem

Der Amiga ist zur Zeit der Rechner, der Files auf einem Datenträger schneller als alle anderen gängigen Rechner findet, sofern der Name bekannt ist. Erreicht wird dies dadurch, daß die Position des Eintrages aufgrund des Namens berechnet wird. Dies nennt man hashing (deutsch: zerschneiden).

Als erstes muß allerdings angemerkt werden, daß diese Berechnung nur für das normale FileSystem (DOS\0) sowie das FastFileSystem von Kickstart 1.3 (DOS\1) gilt.

Beim Hashing wird auf eine Tabelle zugegriffen, die jeweils im Root- oder Directory-Block steht. Dabei wird im Root-Block im vierten Long-Word (Offset 12) angegeben, wie groß die Tabelle ist (Angabe in Long-Words, normal: 72). Zum Aufbau des Root- und der Directory-Blocks möchte ich auf den Abschnitt über die Blockstruktur verweisen.

Doch nun zu der eigentlichen Umrechnung. Den ersten Bestandteil des Hash-Wertes bildet die Länge des Strings. Danach werden die einzelnen Buchstaben nach folgendem Prinzip hinzuaddiert:

- Alle Kleinbuchstaben (a-z, nicht aber ä,ö und ü !) werden in Großbuchstaben umgerechnet.
- Der bisherige Hashing-Wert wird mit 13 multipliziert, danach wird der ASCII-Wert des Zeichens hinzuaddiert.
- Zuletzt folgt eine Und-Verknüpfung mit \$7FF.

Den fertigen Hash-Wert erhält man durch Modulo-Division mit der Größe der Hash-Table.

Eine Unterroutine zur Berechnung könnte also folgendermaßen aussehen:

```
int CalcHash(file, tableSize)
char *file;
int tableSize;
{
```

```
int curHash;
char c;
curHash = strlen(file);
while (c=*file++) {
    if ((c>='a') && (c<='z'))
        c&=0xE0;
    curHash= (curHash*13 + c) & 0x7FF;
}
return (curHash % tableSize);
}
```

Wie schon erwähnt, darf man nicht davon ausgehen, daß auf allen Devices die Größe der Hash-Table 72 Long-Words beträgt, daher muß man diesen Wert aus dem Root-Block auslesen.

Des weiteren kann diese Routine nicht mit Unterdirectories umgehen, d.h., man darf ihr nur den Namen eines Directories oder Files, jeweils ohne : oder / übergeben.

Hat man den Eintrag der Hash-Tabelle gefunden, so ist der dort eingetragene Block nicht unbedingt der gesuchte Block. An dieser Stelle muß erst überprüft werden, ob der Name mit dem gesuchten Text übereinstimmt. Ist dies nicht der Fall, so muß man sich über das Feld Hash Chain zu dem gesuchten Eintrag "vorarbeiten", was allerdings viel schneller ist als die Verfahren von anderen DOS-Systemen.

Struktur der Binärfiles

Programme auf dem Amiga sind im Normalfall (d.h., wenn sie vom DOS lad- und ausführbar sind) relocatibel, also beliebig im Speicher platzierbar. Dies ist bei einem Multitasking-System, das ja mehrere Programme gleichzeitig ausführen kann, auch unbedingt notwendig, denn sonst würde es schnell zu Überlagerungen einzelner Programme kommen.

Um dies zu erreichen, sind eine Reihe zusätzlicher Informationen nötig, anhand derer der SegLoader Files als ausführbar erkennt, das Programm im Speicher verteilt und die Adressen korrigiert.

Programme sind aus einzelnen Hunks (deutsch: Stück, Brocken) aufgebaut, die (bis auf wenige Ausnahmen) folgende Grobstruktur haben:

Hunk-Kennung
Länge
Daten

Abb. 3.7: Die allgemeine Hunk-Struktur

Dabei wird die Länge grundsätzlich in Long-Words angegeben, die einzelnen Hunks werden mit Nullen auf die Long-Word-Grenze expandiert.

Programm-Files müssen folgende Struktur haben, wobei bei den einzelnen Hunks ein oder mehrere Teile weggelassen werden können (lediglich `hunk_end` muß vorhanden sein):

`hunk_header`

Hunks:

`hunk_name`

`hunk_code/data/bss`

`hunk_reloc-Daten` (nur bei `hunk_code` oder `hunk_data`)

`hunk_ext`

`hunk_symbol`

`hunk_debug`

`hunk_end`

Im übrigen ist noch anzumerken, daß die Standard-Linker-Libraries und Object-Files ebenfalls aus solchen Hunks aufgebaut werden.

An dieser Stelle sollten Sie vielleicht, bevor Sie weiterlesen, einmal das Programm Hunk auf der Diskette ausprobieren. Geben Sie dazu Hunk sowie einen File-Namen ein. Nachdem Sie sich klar gemacht haben, aus wieviel Teilen manche Programme bestehen, werde ich Ihnen dann erklären, wie die einzelnen Teile aussehen.

Folgende Hunks werden zur Zeit (Kickstart 1.3) vom Amiga Betriebssystem verwendet:

`hunk_unit` 999 / \$3E7

Dieser Hunk kennzeichnet den Anfang einer Programm-Einheit. Er enthält lediglich deren Namen sowie seine Länge.

<i>hunk_unit</i>
N
N
Long Words
Name

Abb. 3.8: *hunk_unit*

`hunk_name` 1000 / \$3E8

Hiermit definiert man den Namen eines Hunks. Befinden sich in einer Library oder in einem Object-File mehrere Hunks gleichen Namens, so werden sie beim Linken in einem Hunk zusammengekommen; 8- und 16-Bit-Relokations-Hunks können dabei nur zwischen Hunks gleichen Namens aufgelöst werden.

<i>hunk_name</i>
N
N
Long Words
Name

Abb. 3.9: *hunk_name*

`hunk_code` 1001 / \$3E9

In diesem Hunk sind Programm-Daten zu finden. Sein Aufbau ist in Abbildung 3.10 zu sehen.

<i>hunk_code</i>
N
N Long Words Programmdaten

Abb. 3.10: *hunk_code*

hunk_data 1002 / \$3EA

In diesem Block sind vordefinierte Daten zu finden. Ebenso wie auch im Falle eines Code-Hunks können alle Daten in diesem Hunk mittels der Reloc-Hunks, je nach Bedarf, noch geändert werden.

<i>hunk_data</i>
N
N Long Words initialisierte Daten

Abb. 3.11: *hunk_data*

hunk_bss 1003 / \$3EB

Um leere Datenblöcke automatisch zu reservieren, wird der BSS-Hunk verwendet. Er hat als einziges Argument die Größe des Blocks in Long-Words.

<i>hunk_bss</i>
N

Abb. 3.12: *hunk_bss*

hunk_reloc32 / 16 / 8 1004 – 1006 / \$3EC – \$3EE

Diese Hunks sind das Herzstück des Loaders, sorgen sie doch dafür, daß alle Adressen und Zeiger innerhalb der Programme und Daten auf die jeweilige Speicherverteilung eingestellt werden. Dabei werden an den von der Offset-Tabelle definierten Stellen jeweils die Adressen der angesprochenen Hunks hinzuaddiert. Da dies nur beim Laden

des Files geschieht, kann ein fertig verschobenes File nicht mehr im Speicher verschoben werden, da diese Daten nicht mehr im Speicher gehalten werden. Bei den Reloc16- und Reloc8-Hunks ist zu beachten, daß nur Hunks gleichen Namens damit versehen werden dürfen.

<i>hunk_reloc32/16/8</i>
N1
Hunk Nummer
N1 Offsets
N2
Hunk Nummer
:
Nn
Hunk Nummer
Nn Offsets
0

Abb. 3.13: *hunk_reloc32/16/8*

hunk_ext 1007 / \$3EF

Dieser Hunk enthält Information für Symbole, wobei sowohl der Name als auch die Referenz enthalten ist. Er wird praktisch ausschließlich für zu linkende Files verwendet. Seine Grundstruktur ist in Grafik H8 zu sehen.

<i>hunk_ext</i>
Symbol Data Unit
Symbol Data Unit
:
0

Abb. 3.14: *hunk_ext*

Zur Zeit (Kickstart 1.3) gibt es dabei acht verschiedene Typen, wobei einer allerdings nur bei Symbol-Hunks auftritt.

Name:	Wert:	Bedeutung:
ext_symb	0	Nur bei <i>hunk_symbol</i>
ext_def	1	relokatable Definition
ext_abs	2	absolute Definition
ext_res	3	Definition für residente Library
ext_ref32	129	32-Bit Referenz auf ein Symbol
ext_common	130	32-Bit Referenz auf <i>Common</i> (s.Text)
ext_ref16	131	32-Bit Referenz auf ein Symbol
ext_ref8	132	32-Bit Referenz auf ein Symbol

Abb. 3.15: Arten von Symbol-Hunks

Alle anderen Werte sollten vom Linker beanstandet werden. Abhängig von der Art der Referenz ist auch das Aussehen der Symbol-Data-Units unterschiedlich:

Typ	N
N Long Words	
Name des Symbols	
Wert des Symbols	

ext_def/abs/res

Typ	N
N Long Words	
Name des Symbols	
R	
R Long Words	
Symbol-Referenzen	

ext_ref32/16/8

130	N
N Long Words	
Name des Symbols	
Größe des Common-Blocks	
R	
R Long Words	
Symbol-Referenzen	

ext_common

Abb. 3.16: Aufbau der einzelnen Extension-Hunks

ext_def	Das Datenwort, also der Wert des Symbols, gilt als Offset vom Anfang des Hunks.
ext_abs	Der Wert des Symbols wird direkt in den Code übernommen.
ext_res	Der Linker behandelt diesen Typ wie ext_def, nur daß vorausgesetzt wird, daß es sich bei dem Namen um den Namen der Library und bei dem Offset um einen Wert in ihr handelt. Dieser wird daher direkt an das fertige File übergeben.
ext_ref	Diese drei Arten werden von einem Zähler sowie einer Reihe Offsets, die relativ zum Hunk-Anfang sind, gefolgt.
ext_common	Dies entspricht in etwa ext_ref, zusätzlich wird jedoch noch die Größe des Common-Blocks angegeben.
ext_symb	Dieser Code wird nur von dem Symbol-Hunk verwendet und sollte bei hunk_ext nicht auftreten.

hunk_symbol 1008 / \$3F0

Dieser Hunk entspricht hunk_ext, nur wird hunk_symbol hauptsächlich für das Einbinden von Debug-Informationen verwendet. Auch seine Struktur ist gleich:

<i>hunk_symbol</i>
Symbol Data Unit
Symbol Data Unit
⋮
0

Abb. 3.17: *hunk_symbol*

Die Symbol-Data-Unit ist dieselbe wie bei ext_def, abs und res, lediglich wird der Code 0 für ext_symb verwendet.

0	N
N Long Words	
Name des Symbols	
Wert des Symbols	

Abb. 3.18: *Symbol-Data-Unit für hunk_symbol*

`hunk_debug` 1009 / \$3F1

Dieser Hunk wird vom DOS unterstützt, um Programmen die Möglichkeit zu geben, Informationen im Programm unterzubringen, die zum Beispiel zum Debuggen benötigt werden.

Dabei wird kein bestimmtes Format vorgeschrieben, lediglich die Kennung (\$000003F1) sowie die Länge der folgenden Daten müssen vorhanden sein.

<i>hunk_debug</i>
N
N Long Words Debug Daten

Abb. 3.19: *hunk_debug*

`hunk_end` 1010 / \$3F2

Jeder Hunk muß mit diesem Kennzeichen beendet werden.

<i>hunk_end</i>

Abb. 3.20: *hunk_end*

`hunk_header` 1011 / \$3F3

Mit diesem Hunk muß jedes Programm-File beim Amiga beginnen.

Er enthält eine Reihe wichtiger Information: welche residenten Libraries geöffnet werden müssen; die maximale Anzahl der Hunks, die geladen werden; welche Hunks als nächste kommen und wie viele geladen werden sollen, sowie die (maximalen) Längen der einzelnen Hunks.

<i>hunk_header</i>
N1
N1 Long Words Name
N2
N2 Long Words Name
:
0
Anzahl Hunks insgesamt
erster Hunk E
letzter Hunk L
Größen der Hunks E - L

Abb. 3.21: *hunk_header*

In einem normalen Programm wird der Header-Hunk nur ganz am Anfang des Files auftreten, E wird auf 0 stehen und L die Nummer des letzten Hunk im File enthalten. Arbeitet man dagegen mit Overlays (wie Deluxe Paint), so sieht die Sache ganz anders aus. Hier ist zum Beispiel ein Ausschnitt aus dem Hunk-Listing von DPaint:

```

Hunk #0:
hunk_header:
    Number of hunks: 11
    First hunk: 0
    Last hunk: 2
    Sizes of hunks:
    Hunk  0:  162608 / $00027B30  Bytes
    Hunk  1:  37864 / $000093E8  Bytes (MEMF_CHIP)
    Hunk  2:      4 / $00000004  Bytes
hunk_code:
    HunkLen: 162608 / $27B30
hunk_reloc32:
    27 references to hunk 0
    13 references to hunk 1
hunk_end:
    (End of Hunk)

```

Man sieht, daß es zwar elf Hunks (von 0 bis 10) gibt, daß aber nur die ersten drei geladen werden sollen. Jeder einzelne der Overlay-Hunks hat dann wieder einen Header-Hunk. Hunk Nummer 3 sieht beispielsweise so aus:

```
Hunk #3:
hunk_header:
    Number of hunks: 3
    First hunk: 3
    Last hunk: 3
    Sizes of hunks:
    Hunk   3: 8312 / $00002078  Bytes
hunk_code:
    HunkLen:  8312 / $2078
hunk_end:
    (End of Hunk)
```

In diesem Fall bedeutet der Wert im Feld der Anzahl der Hunks etwas anderes, nämlich wohin dieser Hunk gehört: an Stelle 3. Außerdem sieht man bei dem ersten Ausschnitt, daß ein Hunk mit der Anmerkung `CHIP_MEM` versehen ist. Diese Information findet man in dem Long-Word, das die Größe des Hunks bestimmt. Zu diesem Zweck stehen die Bits 30 und 31 zur Verfügung:

```
Bit:   31       30
       0        0   :   egal
       0        1   :   MEMF_CHIP
       1        0   :   MEMF_FAST
       1        1   :   noch nicht unterstützt
```

`hunk_overlay` 1013 / \$3F5

Bei der Verwendung von Overlays muß der Loader eine Reihe von Daten haben, damit er den Rest des Programms korrekt einladen kann. Dazu ist dieser Hunk nötig.

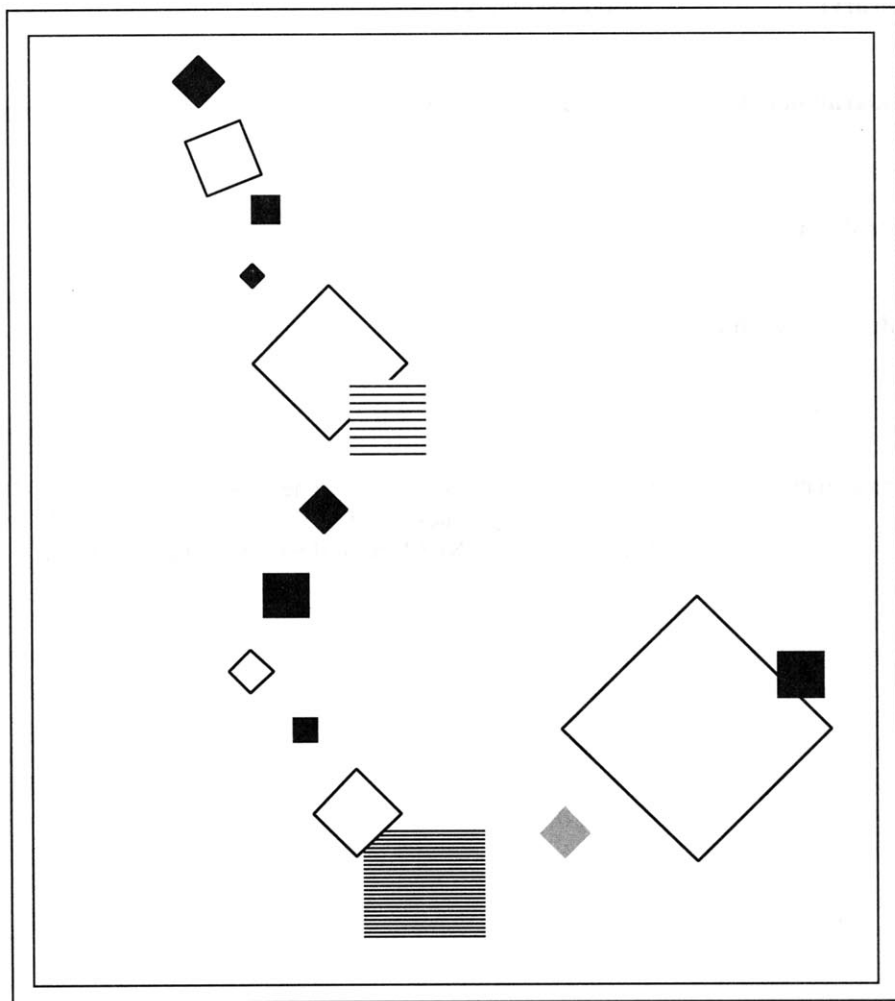
<i>hunk_overlay</i>
Größe der Tabelle
$M + 2$
$M + 1$
Long Words
Overlay Daten

Abb. 3.22: *hunk_overlay*

Da die Aufstellung aller Tabellen und Werte normalerweise vom Compiler/Linker übernommen wird, wird hier nicht weiter darauf eingegangen.

Kapitel 4

Die Library - Routinen



AbortIO

Unterbricht den spezifizierten E/A-Prozeß.

Library: -480(exec.library)

Deklaration in C:

```
long AbortIO(iorequest)
    struct IORequest *iorequest;
```

Aufruf in C:

```
Error = AbortIO(iorequest);
```

Aufruf in Assembler:

```
move.l iorequest,a1
move.l ExecBase,a5
jsr -480(a5)
```

Parameter:

iorequest:	Zeiger auf eine zuvor initialisierte IORequest-Struktur
Error:	NULL bei ordnungsgemäßem Abbruch

ActivateGadget

Aktiviert ein String-Gadget im angegebenen Window bzw. Requester.

Library: -462(intuition.library)

Deklaration in C:

```
BOOL ActivateGadget(gadget, window, requester)
    struct Gadget *gadget;
    struct Window *window;
    struct Requester *requester;
```

Aufruf in C:

```
Success = ActivateGadget
(gadget, window, requester);
```

Aufruf in Assembler:

```
move.l gadget,a0
move.l window,a1
move.l requester,a2
move.l _IntuitionBase,a5
jsr -462(a5)
```

Parameter:	gadget:	Zeiger auf die Struktur des String-Gadgets
	window:	Zeiger auf Window-Struktur
	requester:	gegebenenfalls Zeiger auf Requester-Struktur, andernfalls NULL
	Success:	sind alle Bedingungen erfüllt, dann TRUE, sonst FALSE

Anmerkung:

Zur erfolgreichen Aktivierung eines String-Gadget müssen einige Bedingungen erfüllt sein:

- das Fenster muß aktiv sein
- es darf kein anderes Gadget gerade benutzt werden
- falls das Gadget in einem Requester erscheint, muß dieser ebenfalls aktiv sein
- die rechte Maustaste darf nicht gedrückt sein (keine Menüauswahl!)

ActivateWindow

Aktiviert das angegebene Window.

Library: -450(intuition.library)

Deklaration in C:

```
void ActivateWindow(window)
    struct Window *window;
```

Aufruf in C: `ActivateWindow(Window);`

Aufruf in Assembler:

```
move.l window,a0
move.l _IntuitionBase,a5
jsr -450(a5)
```

Parameter: window: Zeiger auf zuvor erstellte Window-Struktur

AddAnimOb

Fügt ein animiertes Grafikobjekt (AnimOb) zur entsprechenden AnimOb-Liste hinzu.

Library: -156(graphics.library)

Deklaration in C:

```
void AddAnimOb(animob, animkey, rastport)
    struct AnimOb *animob;
    struct AnimOb *animkey;
    struct RastPort *rastport;
```

Aufruf in C: void AddAnimOb(animob, animkey, rastport);

Aufruf in Assembler:

```
move.l animob,a0
move.l animkey,a1
move.l rastport,a2
move.l _GraphicsBase,a5
jsr -156(a5)
```

Parameter:

animob:	Zeiger auf die erstellte AnimOb-Struktur
animkey:	interner Zeiger zur Verwaltung der Objektliste, zeigt auf das zuletzt eingefügte Objekt (bei leerer Liste NULL)
rastport:	Zeiger auf eine RastPort-Struktur

Anmerkung:

- die Timer werden beim Funktionsaufruf zurückgesetzt.
- RastPort->GelsInfo muß initialisiert worden sein.

AddBob

Fügt eine Bob (Blitter Object) in die GEL-Liste (Graphic Element List) des spezifizierten RastPorts ein.

Library: -96(graphics.library)

Deklaration in C:

```
void AddBob(bob, rastport)
    struct Bob *bob;
    struct RastPort *rastport;
```

Aufruf in C: AddBob(bob, rastport);

Aufruf in Assembler:

```
move.l bob, a0
move.l rastport, a1
move.l _GraphicsBase, a5
jsr -96(a5)
```

Parameter:

bob:	Zeiger auf eine zuvor initialisierte Bob-Struktur
rastport:	Zeiger auf den Ziel-Rastport

AddConfigDev

Linkt die Struktur ConfigDev in die Device-Configuration-Liste des Systems.

Library: -30(expansion.library)

Deklaration in C:

```
void AddConfigDev(configdev)
    struct ConfigDev *configdev;
```

Aufruf in C: AddConfigDev(configdev);

Aufruf in Assembler:

```
move.l configdev,a0
move.l _ExpansionBase,a5
jsr -30(a5)
```

Parameter: configdev: Adresse einer ConfigDev-Struktur

AddDevice

Fügt der Device-Liste des Systems ein neues Device hinzu.

Library: -432(exec.library)

Deklaration in C:

```
void AddDevice(device)
    struct Device *device;
```

Aufruf in C: `AddDevice(device);`

Aufruf in Assembler:

```
move.l device,a1
move.l ExecBase,a5
jsr -432(a5)
```

Parameter: device: Adresse der Device-Struktur

AddDosNode

Ergänzt das DOS um einen neuen Floppy-Treiber. Bei bereits aktivem DOS erfolgt die Einbindung sofort, ansonsten unmittelbar nach Initialisierung der DOS-Library.

Library: -150(expansion.library)

Deklaration in C:

```

BOOL AddDosNode(bootpri, flags, dosnode)
    BYTE bootpri;
    long Flags;
    struct DeviceNode *dosnode;

```

Aufruf in C: `ok = AddDosNode(bootpri, flags, dosnode);`

Aufruf in Assembler:

```

move.b #bootpri,d0
move.l #flags,d1
move.l dosnode,a0
move.l ExpansionBase,a5
jsr -150(a5)

```

Parameter:

bootpri:	spezifiziert den Massenspeicher von dem gebootet werden soll.
----------	---

Priorität	System
-----------	--------

+5	internes Laufwerk
0	Harddisk
-5	Laufwerk in einem Netzwerk
-128	sollte nicht benutzt werden

flags:	dosnode:	Zeiger auf die DeviceNode-Struktur
---------------	----------	------------------------------------

AddFont

Ergänzt die Liste der Systemzeichensätze um den angegebenen Zeichensatz.

Library: -480(graphics.library)

Deklaration in C:

```
void AddFont(textfont)
    struct TextFont *textfont;
```

Aufruf in C:

```
void AddFont(textfont);
```

Aufruf in Assembler:

```
move.l textfont,a1
move.l _GraphicsBase,a5
jsr -480(a5)
```

Parameter: textfont: Zeiger auf TextFont-Struktur

Anmerkung: - Zeichensatz und Font-Struktur müssen im PUBLIC-RAM liegen

AddFreeList

Ergänzt die FreeList des Speicherverwaltungssystems um den durch die Parameter mem und size spezifizierten Speicherblock.

Library: -72(icon.library)

Deklaration in C:

```

BOOL AddFreeList (freelist, mem, size)
    struct FreeList *freelist;
    UBYTE *mem;
    unsigned long size;
  
```

Aufruf in C: status = AddFreeList (freelist, mem, size);

Aufruf in Assembler:

```

move.l freelist, a0
move.l mem, a1
move.l #size, a2
move.l _IconBase, a5
jsr -72(a5)
  
```

Parameter:	freelist:	Zeiger auf die FreeList-Struktur
	mem:	Zeiger auf den betreffenden Speicherblock
	size:	Byte-Länge des zu ergänzenden Speicherbereichs
	status:	TRUE, wenn erfolgreich, ansonsten FALSE

AddGadget

Fügt ein Gadget an die durch position spezifizierte Stelle der Window-Gadget-Liste ein.

Library: -42(intuition.library)

Deklaration in C:

```
short AddGadget(window, gadget, position)
    struct Window *window;
    struct Gadget *gadget;
    short position;
```

Aufruf in C: realpos = AddGadget(window, gadget, position);

Aufruf in Assembler:

```
move.l window,a0
move.l gadget,a1
move.w #position,d0
move.l IntuitionBase,a5
jsr -42(a5)
```

Parameter:

window:	Zeiger auf Window-Struktur
gadget:	Zeiger auf eine Gadget-Struktur
position:	Listenposition des neuen Gadgets (bei -1 wird es ans Ende der Liste gehängt)
realpos (d0):	Listenposition an der das Gadget effektiv eingefügt wurde (nicht immer mit dem Wert von position identisch!)

AddGList

Fügt eine Anzahl von Gadgets, die bereits durch Initialisierung der NextGadget-Zeiger miteinander verbunden sind, in die Gadget-Liste des angegebenen Windows ein.

Library: -438(intuition.library)

Deklaration in C:

```
short AddGList
(window, gadget, position, numgad, requester)
    struct Window *window;
    struct Gadget *gadget;
    short position, numgad;
    struct Requester *requester;
```

Aufruf in C:

```
realpos = AddGList
(window, gadget, position, numgad, requester);
```

Aufruf in Assembler:

```
move.l window,a0
move.l gadget,a1
move.w #position,d0
move.w #numgad,d1
move.l requester,a2
move.l IntuitionBase,a5
jsr -438(a5)
```

Parameter:

window:	Zeiger auf Window-Struktur
gadget:	Zeiger auf Gadget-Struktur des ersten Gadgets in der Liste
position:	Listenposition an der die neuen Gadgets eingelinkt werden sollen (-1 für Endposition innerhalb der Window-Gadget-Liste)
numgad:	Anzahl der einzufügenden Gadgets
realpos (d0):	Listenposition an der sie effektiv eingefügt wurden (nicht immer mit dem Wert von position identisch!)

AddHead

Hängt eine neue Node-Struktur an den Anfang der durch list spezifizierten Liste.

Library: -240(exec.library)

Deklaration in C:

```
void AddHead(list, node)
    struct List *list;
    struct Node *node;
```

Aufruf in C: AddHead(list, node);

Aufruf in Assembler:

```
move.l list,a0
move.l node,a1
move.l ExecBase,a5
jsr -240(a5)
```

Parameter:

list:	Adresse der List-Struktur, in die der neue Knoten eingefügt werden soll
node:	Adresse der Node-Struktur

AddIntServer

Hängt einen Interrupt-Server in die angegebene Server-Kette ein. Die Priorität der Node-Struktur entscheidet über seine Kettenposition und damit auch über die Reihenfolge der Bearbeitung.

Library: -168(exec.library)

Deklaration in C:

```
void AddIntServer(intnumber, interrupt)
    long intnumber;
    struct Node *interrupt;
```

Aufruf in C: AddIntServer(intnumber, interrupt);

Aufruf in Assembler:

```
move.l #intnumber,d0
move.l interrupt,a1
move.l ExecBase,a5
jsr -168(a5)
```

Parameter:

intnumber:	Nummer des zu bearbeitenden System-Interrupts [0..14]
interrupt:	Zeiger auf die Interrupt-Node-Struktur des erstellten Servers

Anmerkung:

Beim Aufruf der Server-Routine sieht die Registerbelegung wie folgt aus:

- D0 Zeiger auf nächsten Interrupt
- D1 beliebig
- A0 beliebig
- A1 Zeiger auf Datenblock des Servers
- A5 Adresse der Interrupt-Routine
- A6 Basisadresse der Library

Alle anderen Register müssen vom Interruptprogramm gesichert und nach der Bearbeitung restauriert werden.

AddLibrary

Fügt eine mit der MakeLibrary-Funktion erstellte Library in die Liste der System-Libraries ein. Die neue Funktionsbibliothek ist damit für sämtliche Tasks transparent.

Library: -396(exec.library)

Deklaration in C:

```
void AddLibrary(library)
    struct Library *library;
```

Aufruf in C:

```
AddLibrary(library);
```

Aufruf in Assembler:

```
move.l library,a1
move.l ExecBase,a5
jsr -396(a5)
```

Parameter: library: Zeiger auf eine Library-Struktur

AddMemList

Fügt den deklarierten Speicherbereich der Speicherverwaltungsliste (MemList) hinzu.

Library: -618(exec.library)

Deklaration in C:

```
long AddMemList
(size, attributes, pri, base, name)
    ULONG size;
    UWORD attributes;
    BYTE pri;
    ULONG base;
    char *name;
```

Aufruf in C:

```
Error = AddMemList
(size, attributes, pri, base, name);
```

Aufruf in Assembler:

```
move.l #size,d0
move.l #attributes,d1
move.b #pri,d2
move.l base,a0
move.l name,a1
move.l ExecBase,a5
jsr -618(a5)
```

Parameter:	size:	Speichergröße in Byte
	attributes:	Zugriffsattribute (z.B. MEMF_CHIP)
	pri:	Priorität (-10 = Chip, 0 = Fast)
	base:	Basisadresse des Systemspeichers
	name:	Bezeichnung des neuen Speicherbereichs (Eintragung im Kopf)
	Error (d0):	Fehlernummer

Anmerkung:

"name" wird nicht kopiert, sondern lediglich als Zeiger in den Memory-Header übernommen. Falls die Gefahr besteht, daß der String in der Belegungsphase des Speichers überschrieben wird, sollte der Wert NULL übergeben werden.

AddPort

Hängt den spezifizierten MessagePort in die PortList des Systems ein.

Library: -354(exec.library)

Deklaration in C:

```
void AddPort(port)
    struct MsgPort *port;
```

Aufruf in C: `AddPort(port);`

Aufruf in Assembler:

```
move.l port,a1
move.l ExecBase,a5
jsr -354(a5)
```

Parameter: `port:` Zeiger auf eine zuvor initialisierte MsgPort-Struktur

AddResource

Integriert eine Resource in die Liste der System-Ressourcen, so daß sie für sämtliche Tasks transparent wird.

Library: -486(exec.library)

Deklaration in C:

```
void AddResource(resource)
    struct Node *resource;
```

Aufruf in C: AddResource(resource);

Aufruf in Assembler:

```
move.l resource,a1
move.l ExecBase,a5
jsr -486(a5)
```

Parameter: resource: Adresse der Resource-Node-Struktur

AddSemaphore

Hängt eine neue Semaphore in die Systemliste ein.

Library: -600(exec.library)

Deklaration in C:

```
void AddSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
```

Aufruf in C: `AddSemaphore(sigsem);`

Aufruf in Assembler:

```
move.l sigsem,a1
move.l ExecBase,a5
jsr -600(a5)
```

Parameter: sigsem: Zeiger auf die zu ergänzende Signal-Semaphore-Struktur

AddTail

Hängt eine Node-Struktur an das Ende der durch list adressierten Liste.

Library: -246(exec.library)

Deklaration in C:

```
void AddTail(list, node)
    struct List* list;
    struct Node* node;
```

Aufruf in C: AddTail(list, node);

Aufruf in Assembler:

```
move.l list,a0
move.l node,a1
move.l ExecBase,a5
jsr -246(a5)
```

Parameter:

list:	Zeiger auf die Ziel-Liste
node:	Adresse der einzufügenden Node-Struktur

AddTask

Übergibt dem System einen neuen Task, der normalerweise sofort in die TaskReady-Liste übernommen wird.

Library: -282(exec.library)

Deklaration in C:

```
void AddTask(task, initialpc, finalpc)
    struct Task *task;
    ULONG initialPC;
    ULONG finalPC;
```

Aufruf in C: AddTask(task, initialpc, finalpc);

Aufruf in Assembler:

```
move.l task,a0
move.l initialpc,a1
move.l finalpc,a3
jsr -282(a5)
```

Parameter:	task:	Zeiger auf die zuvor initialisierte Task-Struktur
	initialpc:	Startadresse der Task-Routine
	finalpc:	Rücksprungadresse, die vor der Bearbeitung des Tasks auf dem Stapel abgelegt wird. Hier kann entweder eine eigene Routine angesprungen werden oder man verwendet die standardmäßige FinalPC-Routine des Betriebssystems (Parameter auf NULL setzen).

Anmerkung:

Der Stackgröße hängt von der jeweiligen Anwendung ab. 80 ist der kleinste zulässige Wert. Ein Task, der lediglich Exec-Funktionen aufruft, kommt mit ca. 200 Byte aus. Bei anderen Systemaufrufen sollten mindestens 4 KByte reserviert werden.

AddTime

Addiert die Zeiten zweier timeval-Strukturen und speichert das Ergebnis in der ersten Struktur.

Library: -42(timer.library)

Deklaration in C:

```
void AddTime(destination, source)
    struct timeval *destination;
    struct timeval *source;
```

Aufruf in C: AddTime(destination, source);

Aufruf in Assembler:

```
move.l destination,a0
move.l source,a1
move.l TimerBase,a5
jsr -42(a5)
```

Parameter:

destination:	Ziel-Struktur
source:	Quell-Struktur

Anmerkung: Register A6 enthält einen Zeiger auf das Timer-Device

AddVSprite

Fügt der GEL-Liste des angegebenen RastPorts ein VSprite (Virtual Sprite) hinzu.

Library: -102(graphics.library)

Deklaration in C:

```
void AddVSprite(vsprite, rastport)
    struct VSprite *vsprite;
    struct RastPort *rastport;
```

Aufruf in C: AddVSprite(vsprite, rastport);

Aufruf in Assembler:

```
move.l vsprite,a0
move.l rastport,a1
move.l GraphicsBase,a5
jsr -102(a5)
```

Parameter:

vsprite:	Adresse der VSprite-Struktur
rastport:	Zeiger auf Ziel-RastPort

Alert

Ausgabe einer Fehlermeldung.

Library: -108(exec.library)

Deklaration in C:

```
void Alert(alertnum, parameters)
    ULONG alertnum;
    ULONG *parameters;
```

Aufruf in C: `Alert(alertNum, parameters);`

Aufruf in Assembler:

```
move.l #alertnum,d7
move.l parameters,a5
move.l ExecBase,a6
jsr -108(a6)
```

Parameter:

alertnum:	Fehlernummer
parameters:	Zeiger auf ein Parameter-Feld (zur Identifikation des abstürzenden/warnenden Tasks)

AllocAbs

Versucht den durch die Parameter `size` und `location` (absolute Positionsangabe) definierten Speicherbereich zu reservieren.

Library: -204(exec.library)

Deklaration in C:

```
ULONG *AllocAbs(size, location)
        ULONG size, location;
```

Aufruf in C: `memory = AllocAbs(size, location)`

Aufruf in Assembler:

```
move.l #size,d0
move.l #location,a1
move.l ExecBase,a5
jsr -204(a5)
```

Parameter:

<code>size:</code>	Speichergröße in Byte
<code>location:</code>	Position des Speicherblocks
<code>memory (d0):</code>	bei erfolgreicher Reservierung Zeiger auf belegten Speicherblock, sonst NULL

Allocate

Sucht nach einem Speicherblock der gewünschten Größe (FreeList), allokiert diesen und gibt (bei Erfolg) einen Zeiger darauf zurück.

Library: -186(exec.library)

Deklaration in C:

```
ULONG *Allocate(freelist, size)
    struct MemHeader *freelist;
    ULONG size;
```

Aufruf in C: memory = Allocate(freelist, size);

Aufruf in Assembler:

```
move.l freelist,a0
move.l #size,d0
move.l ExecBase,a5
jsr -186(a5)
```

Parameter:

freelist:	Zeiger auf MemHeader-Struktur, die den Speicherbereich verwaltet, in dem der spezifizierte Block reserviert werden soll
size:	Speichergröße in Byte
memory (d0):	bei erfolgreicher Belegung Zeiger auf den reservierten Speicherblock, sonst NULL

AllocBoardMem

Reserviert Speicherplatz im Expansion-Memory.

Library: -42(expansion.library)

Deklaration in C:

```
long AllocBoardMem(slotspec)
    long slotspec;
```

Aufruf in C:

```
start = AllocBoardMem(slotspec);
```

Aufruf in Assembler:

```
move.l #slotspec,d0
move.l ExpansionBase,a5
jsr -42(a5)
```

Parameter:

slotspec:	Spezifikationen (Typ-Identifikation) der Erweiterungskarte
start (d0):	Nummer des ersten Slots, der reserviert werden konnte, oder -1 bei Mißerfolg

AllocCList

Reserviert eine leere CList im CList-Pool.

Library: -36(clist.library)

Deklaration in C: `long AllocCList (clpool)
 UBYTE *clpool;`

Aufruf in C: `CList = AllocCList (clpool)`

Aufruf in Assembler: `move.l clpool,a1
move.l CListBase,a5
jsr -36(a5)`

Parameter: `clpool:` Zeiger auf CList-Pool
`CList (d0):` Identifikationsnummer der neuen CListe oder
-1 bei Speichermangel

AllocConfigDev

Initialisiert eine ConfigDev-Struktur und übergibt ihre Adresse.

Library: -48(expansion.library)

Deklaration in C: `struct ConfigDev *AllocConfigDev()`

Aufruf in C: `configDev = AllocConfigDev();`

Aufruf in Assembler `move.l ExpansionBase,a5
jsr -48(a5)`

Parameter: configDev (d0): Zeiger auf ConfigDev-Struktur oder NULL

AllocEntry

Multiple Speicherbelegung über eine MemList-Struktur.

Library: -222(exec.library)

Deklaration in C:

```
struct MemList *AllocEntry(entry)
    struct MemList entry;
```

Aufruf in C: memList = AllocEntry(entry);

Aufruf in Assembler:

```
lea entry,a0
move.l ExecBase,a5
jsr -222(a5)
```

Parameter:

entry:	Auswahlvariable einer MemList-Struktur, in der sämtliche Speichereinträge definiert sind
memList:	Zeiger auf eine MemList mit den Adressen der reservierten Speicherblöcke oder ein Wert kleiner NULL bei Mißerfolg

Anmerkung: Konnte der gewünschte Speicher nicht belegt werden, so enthält der Rückgabeparameter die Anforderungen, die nicht erfüllt werden konnten und Bit 31 ist gesetzt.

AllocExpansionMem

Die Funktion allokiert Speicher für die spezifizierte Anzahl von Slots (mit der Größe E_SLOTSIZE) im Expansion-Memory. Die Wahl der Speichergrenzen hängt von dem durch slotalign definierten Modulwert ab.

Library: -54(expansion.library)

Deklaration in C:

```
ULONG AllocExpansionMem
(numslots, slotalign, slotoffset)
    long numslots, slotalign, slotoffset;
```

Aufruf in C:

```
start = AllocExpansionMem
(numslots, slotalign, slotoffset);
```

Aufruf in Assembler:

```
move.l #numslots,d0
move.l #slotalign,d1
move.l #slotoffset,d2
move.l ExpansionBase,a5
jsr -54(a5)
```

Parameter:	numslots:	Anzahl der zu reservierenden Slots
	slotalign:	Modulwert der Speichergrenzen (es gilt: [start-slotoffset] MOD slotalign = 0)
	slotoffset:	Offset des ersten Slots
	start (d0):	Nummer des ersten Slots (Start-Slot)

AllocMem

Die Standardfunktion zur dynamischen Speicherbelegung. Reserviert einen Speicherbereich mit den spezifizierten Attributen und gibt einen Zeiger darauf zurück.

Library: -198(exec.library)

Deklaration in C:

```
ULONG *AllocMem(size, requirements)
        ULONG size;
        ULONG requirements;
```

Aufruf in C: `memory = AllocMem(size, requirements);`

Aufruf in Assembler:

```
move.l #size,d0
move.l #requirements,d1
move.l ExecBase,a5
jsr -198(a5)
```

Parameter:

size:	Byte-Größe des zu reservierenden Speicherbereichs
requirements:	Speicherattribute; zur Zeit sind folgende Eigenschaften definiert:
	MEMF_PUBLIC (1L<<0) Öffentlicher Speicher für beliebige Tasks
	MEMF_CHIP (1L<<1) CHIP-RAM, Zugriff durch DMA-Controller und CPU

MEMF_FAST (1L<<2)

FAST-RAM, Zugriff nur durch den Prozessor möglich (keine Verlangsamung durch Customchip-Aktivitäten)

MEMF_CLEAR (1L<<16)

automatisches Löschen des belegten Speicherbereichs

MEMF_LARGEST (1L<<17)

Allokierung des größten, zur Verfügung stehenden Speicherblocks

memory (d0):

Zeiger auf belegten Speicherblock oder NULL bei Mißerfolg

AllocPotBits

Initialisiert das POTGO-Register (Offset \$034) und verteilt die Zugriffsberechtigung für die Analogeingänge der Game-Ports.

Library: -6(potgo.library)

Deklaration in C:

```
long AllocPotBits(bits)
    long bits;
```

Aufruf in C: `allocated = AllocPotBits(bits);`

Aufruf in Assembler:

```
move.l #bits,d0
move.l PotgoBase,a5
jsr -6(a5)
```

Parameter:

bits:	Registermaske (die Binärstellen entsprechen den Bitfunktionen im POTGO-Register)
allocated (d0):	effektiv reservierte Bits (nicht unbedingt mit dem Wert von bits identisch)

AllocRaster

Reserviert Speicherplatz für ein Raster (Bitplane) der Ausdehnung `width * height`.

Library: -492(graphics.library)

Deklaration in C:

```
PLANEPTR AllocRaster(width, height)
short width, height;
```

Aufruf in C:

```
planeptr = AllocRaster(width, height);
```

Aufruf in Assembler:

```
move.w #width,d0
move.w #height,d1
move.l GraphicsBase,a5
jsr -492(a5)
```

Parameter:

<code>width:</code>	Breite des Rasters in Pixel
<code>height:</code>	Zeilenhöhe
<code>planeptr (d0):</code>	Zeiger auf den reservierten Speicherbereich oder NULL

AllocRemember

Erweiterung der AllocMem()-Funktion für die Intuition-Benutzeroberfläche. Bei jedem Aufruf von AllocRemember() wird der belegte Speicher in einer Remember-Struktur (remember = erinnern) vermerkt. Beim Verlassen des Programms reicht dann ein Aufruf von FreeMem(), um sämtliche Speicherblöcke zu löschen.

Library: -396(intuition.library)

Deklaration in C:

```
ULONG *AllocRemember(rememberkey, size, flags)
struct Remember *rememberkey;
ULONG size, flags;
```

Aufruf in C:

```
memory = AllocRemember
(rememberkey, size, flags);
```

Aufruf in Assembler:

```
move.l rememberkey,a0
move.l #size,d0
move.l #flags,d1
move.l IntuitionBase,a5
jsr -396(a5)
```

Parameter:

rememberkey:	Zeiger auf eine Remember-Struktur zur Speicherverwaltung
--------------	--

Anmerkung:

Die Struktur (Include-File "inuition/intuition.h") hat folgendes Aussehen:

```
struct Remember
```

```
{
```

```
    struct remeber *NextRemember;
```

```
    ULONG RemeberSize;
```

```
    UByte *Memory;
```

```
    } rememberkey;
```

size: SpeichergroÙe in Byte

flags: Standardattribute des Speichers (siehe AllocMem)

memory (d0): Adresse des allokierten Speicherblocks oder NULL

AllocSignal

Belegt das angegebene Task-Signal.

Library: -330(exec.library)

Deklaration in C: `long AllocSignal(signalnum)
long signalnum;`

Aufruf in C: `signal = AllocSignal(signalnum);`

Aufruf in Assembler: `move.l #signalnum,d0
move.l ExecBase,a5
jsr -330(a5)`

Parameter:

signalnum:	Nummer des zu belegenden Signals oder -1 bei beliebiger Allokierung
signal (d0):	Effektiv reserviertes Signal oder -1, wenn kein Signal mehr frei war

Anmerkung: AllocSignal() und FreeSignal() dürfen nicht von einem Exception-Handler aufgerufen werden.

AllocTrap

Belegt einen System-Trap.

Library: -342(exec.library)

Deklaration in C: long AllocTrap(trapnum)
 long trapnum;

Aufruf in C: trap = AllocTrap(trapnum);

Aufruf in Assembler: move.l #trapnum,d0
 move.l ExecBase,a6
 jsr -342(a5)

Parameter:

trapnum:	Nummer des spezifizierten Traps [0...15] oder -1 für die Belegung des nächsten freien Traps
trap (d0):	Nummer des erfolgreich belegten System-Traps oder -1 bei Fehlschlag

Anmerkung: AllocTrap() und FreeTrap() dürfen ebenfalls nicht von einem Exception-Handler aktiviert werden.

AllocWBOject

Allokiert Speicher für ein Workbench-Objekt (WBOject).

Library: -66(icon.library)

Deklaration in C: `struct WBOject *AllocWBOject()`

Aufruf in C: `object = AllocWBOject();`

Aufruf in Assembler: `move.l ExecBase,a5
jsr -66(a5)`

Parameter: `object (d0):` Zeiger auf das reservierte Objekt oder NULL

AlohaWorkbench

Library: -402(intuition.library)

Deklaration in C: void AlohaWorkbench(wbport)

Aufruf in C: AlohaWorkbench(wbport);

Aufruf in Assembler: move.l wbport,a0
move.l IntuitionBase,a5
jsr -402(a5)

Parameter: wbport:

AndRectRegion

AndRectRegion() führt eine logische UND-Verknüpfung zwischen allen Clipping-Rechtecken der Region und dem durch rectangle spezifizierten Bereich durch und löscht anschließend alle Rechtecke bis auf das angegebene.

Library: -504(graphics.library)

Deklaration in C:

```
void AndRectRegion(region, rectangle)
    struct Region *region;
    struct rectangle *rectangle;
```

Aufruf in C:

```
void AndRectRegion(region, rectangle)
```

Aufruf in Assembler:

```
move.l region,a0
move.l rectangle,a1
move.l GraphicsBase,a5
jsr -504(a5)
```

Parameter:

region:	Zeiger auf Region-Struktur
rectangle:	Adresse der Rectangle-Struktur, die nicht gelöscht wird

AndRegionRegion

Ausblendung der nicht in beiden Regions vorhandenen Bereiche durch zweifache logische UND-Verknüpfung (destination AND [source AND destination])

Library: -624(graphics.library)

Deklaration in C:

```
BOOL AndRegionRegion(source, destination)
    struct Region *source, *destination;
```

Aufruf in C:

```
status = AndRegionRegion(source, destination);
```

Aufruf in Assembler:

```
move.l source, a0
move.l destination, a1
move.l GraphicsBase, a5
jsr -624(a5)
```

Parameter:

source:	Quell-Region (bleibt unverändert)
destination:	Ziel-Region
status (d0):	TRUE bei Gleichheit, sonst FALSE und Modifikation der Ziel-Region

Animate

Animationsroutine zur Bewegung der zuvor definierten AnimObjects.

Library: -2(graphics.library)

Deklaration in C:

```
void Animate(animkey, rastport)
    struct AnimOb *animkey;
    struct RastPort *rastport;
```

Aufruf in C: Animate(animkey, rastport);

Aufruf in Assembler:

```
move.l animkey,a0
move.l rastport,a1
move.l GraphicsBase,a5
jsr -162(a5)
```

Parameter:

animkey:	Zeiger auf eine AnimOb-Struktur
rastport:	Zeiger auf RastPort-Struktur

AreaDraw

Definiert den Eckpunkt eines Polygons in der AreaInfo-Struktur des angegebenen Rast-Ports.

Library: -258(graphics.library)

Deklaration in C:

```
long AreaDraw(rastport, x, y)
    struct RastPort *rastport;
    short x, y;
```

Aufruf in C:

```
error = AreaDraw(rastport, x, y);
```

Aufruf in Assembler:

```
move.l rastport,a1
move.w #x,d0
move.w #y,d1
move.l GraphicsBase,a5
jsr -258(a5)
```

Parameter:

rastport:	Zeiger auf zugehörige RastPort-Struktur (RastPort->AreaInfo)
x,y:	Koordinate des Polygonpunktes
error(d0):	NULL: erfolgreiche Definition des Punktes
	-1: kein Platz mehr in der AreaInfo-Struktur

AreaEllipse

Definiert eine Ellipse in der AreaInfo-Struktur eines RastPorts.

Library: -186(graphics.library)

Deklaration in C:

```
long AreaEllipse
(rastport, centerx, centery, xr, yr)
    struct RastPort *rastport;
    short centerx, centery, xr, yr;
```

Aufruf in C:

```
error = AreaEllipse
(rastport, centerx, centery, xr, yr);
```

Aufruf in Assembler:

```
move.l rastport,a1
move.w centerx,d0
move.w centery,d1
move.w xr,d2
move.w yr,d3
move.l GraphicsBase,a5
jsr -186(a5)
```

Parameter:	rastport:	Zeiger auf zugehörige RastPort-Struktur (RastPort->AreaInfo)
	center(x/y):	Mittelpunktskordinate der Ellipse
	xr,yr:	X- und Y-Radius
	error (d0):	siehe AreaDraw()

AreaEnd

Zeichnet die durch AreaInfo definierte Figur (Polygon, Ellipse) und füllt die Fläche mit dem aktuellen Füllmuster.

Library: -264(graphics.library)

Deklaration in C:

```
long AreaEnd(rastport)
    struct RastPort *rastport;
```

Aufruf in C:

```
error = AreaEnd(rastport);
```

Aufruf in Assembler:

```
move.l rastport,a1
move.l GraphicsBase,a5
jsr -264(a5)
```

Parameter:

rastport:	Zeiger auf zugehörige RastPort-Struktur (RastPort->AreaInfo)
error (d0):	siehe AreaDraw(), AreaEllipse()

AreaMove

Definiert den Anfangspunkt eines Polygons. Noch nicht beendete Polygone werden beim Aufruf dieser Funktion automatisch geschlossen.

Library: -252(graphics.library)

Deklaration in C:

```
long AreaMove(rastport, x, y)
    struct RastPort *rastport;
    short x, y;
```

Aufruf in C: error = AreaMove(rastport, x, y);

Aufruf in Assembler:

```
move.l rastport,a1
move.w #x,d0
move.w #y,d1
move.l GraphicsBase,a5
jsr -252(a5)
```

Parameter:

rastport:	Zeiger auf zugehörige RastPort-Struktur (RastPort->AreaInfo)
x,y:	Koordinate des Polygonpunktes
error/d0:	NULL: erfolgreiche Definition des Punktes -1: kein Platz mehr in der AreaInfo-Struktur

AskFont

Ermittelt die Textattribute des aktuellen Zeichensatzes.

Library: -474(graphics.library)

Deklaration in C:

```
void AskFont(rp, textattr)
    struct RastPort *rp;
    struct TextAttr *textattr;
```

Aufruf in C: AskFont(rp, textattr);

Aufruf in Assembler:

```
move.l rp,a1
move.l textattr,a0
move.l GraphicsBase,a5
jsr -474(a5)
```

Parameter:

rastport:	Zeiger auf RastPort-Struktur des untersuchten RastPorts
textattr:	Adresse der TextAttr-Struktur, die mit den aktuellen Parametern initialisiert wird

AskSoftStyle

Ermittelt die wählbaren Darstellungsarten des aktuellen Zeichensatzes. Sie können mit der SetSoftStyle()-Funktion aktiviert werden.

Library: -84(graphics.library)

Deklaration in C:

```
ULONG AskSoftStyle(rp)
    struct RastPort *rp;
```

Aufruf in C:

```
style = AskSoftStyle(rp);
```

Aufruf in Assembler:

```
move.l rp,a1
move.l GraphicsBase,a5
jsr -84(a5)
```

Parameter:

rp:	Adresse der RastPort-Struktur
style (d0):	Font-Style-Flags der unterstützten Schriftarten, beispielsweise:

```
FSF_NORMAL [=0]
FSF_UNDERLINED [=1]
FSF_BOLD [=2]
FSF_ITALIC [=4]
```

AttemptLockLayerRom

Privilegierter Zugriff auf das spezifizierte Layer (aufrufender Task erhält das alleinige Zugriffsrecht).

Library: -654(graphics.library)

Deklaration in C:

```
BOOL AttemptLockLayerRom(layer)
    struct Layer *layer;
```

Aufruf in C:

```
ok = AttemptLockLayerRom(layer);
```

Aufruf in Assembler:

```
move.l layer,a5
move.l GraphicsBase,a6
jsr -654(a6)
```

Parameter:

layer:	Zeiger auf eine Layer-Struktur
ok (d0):	FALSE, falls das betreffende Layer bereits privilegiert ist, ansonsten TRUE

AttemptSemaphore

Siehe unter ObtainSemaphore()

Library: -576(exec.library)

Deklaration in C: `BOOL AttemptSemaphore(sigsem)
struct SignalSemaphore *sigsem;`

Aufruf in C: `success = AttemptSemaphore(sigsem);`

Aufruf in Assembler: `move.l sigsem,s0
move.l ExecBase,a5
jsr -576(a5)`

Parameter:

sigsem:	Adresse der zuvor initialisierten Signal-Semaphore-Struktur
success (d0):	FALSE, wenn bereits von einem anderen Task gelockt, sonst TRUE

AutoRequest

Erzeugt einen Auto-Requester mit den angegebenen Parametern und gibt die Entscheidung des Anwenders als BOOL-Variable zurück (baut auf der Funktion BuildSysRequest() auf).

Library: -348(intuition.library)

Deklaration in C:

```

BOOL AutoRequest
(window,body,ptext,ntext,pflag,nflag,w,h)
    struct Window *window;
    struct IntuiText *btext,*ptext,*ntext;
    ULONG pflag,nflag;
    short w,h;

```

Aufruf in C:

```

action = AutoRequest
(window,body,ptext,ntext,pflag,nflag,w,h);

```

Aufruf in Assembler:

```

move.l window,a0
move.l btext,a1
move.l ptext,a2
move.l ntext,a3
move.l #pflag,d0
move.w #nflag,d1
move.w #w,d2
move.w #h,d3
move.l IntuitionBase,a5
jsr -348(a5)

```

Parameter:

window:	Zeiger auf Window-Struktur des Fensters, in dem der Requester erscheinen soll
btext:	Zeiger auf IntuiText-Struktur für den Informationstext

ptext:	Zeiger auf IntuiText-Struktur für den positiven Entscheidungstext (rechtes Auswahlgadget)
ntext:	Adresse der IntuiText-Struktur für den negativen Text (linkes Gadget)
pflag:	IDCMP-Flags für Auswahl des rechten Gadgets
nflag:	IDCMP-Flags für Auswahl des linken Gadgets
w,h:	Breite und Höhe des Requesters
action (d0):	TRUE für rechtes Gadget, FALSE bei Auswahl des linken Gadgets

AvailFonts

Legt eine Informationstruktur (AvailFontsHeader) mit denen im Speicher bzw. auf Diskette verfügbaren Zeichensätzen an.

Library: -36(diskfont.library)

Deklaration in C:

```
long AvailFonts(buffer, bufbytes, flags)
    BYTE *buffer;
    short bufbytes, flags;
```

Aufruf in C: error = AvailFonts(buffer, bufbytes, flags);

Aufruf in Assembler:

```
move.l buffer,a0
move.w #bufbytes,d0
move.w #flags,d1
move.l DiskfontBase,a5
jsr -36(a5)
```

Parameter:	buffer:	Speicherbereich zur Aufnahme der Informationsstruktur
	bufbytes:	Größe dieses Puffers in Byte
	flags:	AFF_MEMORY für speicherresidente Zeichensätze und AFF_DISK für Disketten-Fonts
	error (d0):	NULL, wenn die Puffergröße ausreichte, sonst Anzahl der zusätzlich benötigten Byte

AvailMem

Durchsucht die System-Speicherliste (MemList) nach freien Bereichen mit den spezifizierten Attributen, ermittelt deren Größe und gibt sie als Ergebnis zurück.

Library: -216(exec.library)

```
Deklaration in C:    ULONG AvailMem(requirements)
                      ULONG requirements;
```

Aufruf in C: `size = AvailMem(requirements);`

```
Aufruf in Assembler:  move.l #requirements,d1
                        move.l ExecBase,a5
                        jsr -216(a5)
```

Parameter:	requirements:	die üblichen Speicherattribute (siehe Alloc-Mem)
	size:	Größe des ermittelten Bereichs in Byte

BeginRefresh

Bereitet das spezifizierte Fenster auf einen Refreshzyklus vor.

Library: -354(intuition.library)

Deklaration in C:

```
void BeginRefresh(window)
    struct Window *window;
```

Aufruf in C: BeginRefresh(window);

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a5
jsr -354(a5)
```

Parameter: window: Zeiger auf eine Window-Struktur

BeginUpdate

Die Funktion wandelt die Damage-List (enthält die durch Überlagerung zerstörten Teile) des spezifizierten Layers in eine ClipRect-List um.

Library: -78(layers.library)

Deklaration in C: `BOOL BeginUpdate(layer)
struct Layer *layer;`

Aufruf in C: `ok = BeginUpdate(layer);`

Aufruf in Assembler: `move.l layer,a0
move.l LayerBase,a5
jsr -78(a5)`

Parameter: `layer:` Zeiger auf eine Layer-Struktur
`ok (d0):` TRUE bei erfolgreicher Bearbeitung, ansonsten FALSE

Anmerkung: Ist die Operation gescheitert, sollte die Funktion EndUpdate(layer,FALSE) aufgerufen werden, um die alte ClipRect-Liste wieder zu rekonstruieren.

BehindLayer

Legt das spezifizierte Layer hinter die anderen Layer. Enthält der Layer-Stapel auch BACKDROP-Layer und handelt es sich bei dem angegebenen um kein BACKDROP-Layer, so erscheint es hinter den normalen aber noch vor den BACKDROP-Layern.

Library: -54(layers.library)

Deklaration in C:

```

BOOL BehindLayer(li, layer)
    struct LayerInfo *li;
    struct Layer *layer;

```

Aufruf in C: ok = BehindLayer(li, layer);

Aufruf in Assembler:

```

move.l li,a0
move.l layer,a1
move.l LayerBase,a5
jsr -54(a5)

```

Parameter:

li:	Adresse einer LayerInfo-Struktur
layer:	Zeiger auf die Layer-Struktur
ok (d0):	TRUE bei erfolgreicher Bearbeitung, sonst FALSE

BltBitMap

Standard-Kopier-Routine für den Blitter, kopiert ein Raster aus der Quell-BitMap in die Ziel-BitMap unter Berücksichtigung der angegebenen Parameter.

Library: -30(graphics.library)

Deklaration in C:

```
ULONG BltBitMap
(srcbm, srcx, srcy, destbm, destx, desty, sizex,
sizey, minterm, mask, buffer)
    struct BitMap *srcbm, *destbm;
    short srcx, srcy;
    short destx, desty, sizex, sizey;
    UBYTE minterm, mask;
    PLANEPTR buffer;
```

Aufruf in C:

```
planeCnt = BltBitMap
(srcbm, srcx, srcy, destbm, destx, desty, sizex,
sizey, minterm, mask, tempa);
```

Aufruf in Assembler:

```
move.l srcbm, a0
move.w #srcx, d0
move.w #srcy, d1
move.l destbm, a1
move.w #destx, d2
move.w #desty, d3
move.w #sizex, d4
move.w #sizey, d5
move.b #minterm, d6
move.b #mask, d7
move.l buffer, a2
move.l GraphicsBase, a5
jsr -30(a5)
```

Parameter:	srcbm:	Zeiger auf BitMap-Struktur des Quellbereichs
	srcx,srcy:	X,Y-Koordinate der linken, oberen Ecke des Quellrasters
	dest:	Zeiger auf BitMap-Struktur des Zielbereichs
	destx,desty:	X,Y-Koordinate der linken, oberen Ecke des Zielrasters
	sizeX,sizeY:	Breite bzw. Höhe des Rasters
	minterm:	verwendete Minterms für die logische Verknüpfung
	mask:	der Maskenwert gibt an, welche Bitplanes in die Operation einbezogen werden (die unteren sechs Binärstellen korrespondieren mit den Bitplanes 0 bis 5)
	buffer:	Zwischenspeicher bei Überlagerung von Ziel- und Quellraster (sollte groß genug für eine Zeile des Planes sein)
	planeCnt (d0):	von der Blitter-Operation betroffene Planes, entspricht dem Maskenwert

Anmerkung: BltBitMap() verbraucht etwa 300 Byte Stackspeicher.

BltBitMapRastPort

Kopiert einen rechteckig-organisierten Speicherbereich aus der Quell-Bitmap in einen Ziel-RastPort (Funktion und Parameter ansonsten wie bei BltBitMap).

Library: -606(graphics.library)

Deklaration in C:

```
void BltBitMapRastPort
(srcbm, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm)
    struct BitMap *srcbm;
    short srcx, srcy;
    struct RastPort *destrp;
    short destx, desty, sizex, sizey;
    UBYTE minterm;
```

Aufruf in C:

```
BltBitMapRastPort
(srcbm, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm);
```

Aufruf in Assembler:

```
move.l srcbm, a0
move.w #srcx, d0
move.w #srcy, d1
move.l destrp, a1
move.w #destx, d2
move.w #desty, d3
move.w #sizex, d4
move.w #sizey, d5
move.b #minterm, d6
move.l GfxBase, a5
jsr -606(a5)
```

Parameter:	srcbm:	wie BltBitMap()
	srcx, srcy:	dito

destrp:	dito
destx,desty:	dito
size,ysize:	dito
minterm:	dito

BltClear

Löscht einen beliebigen Speicherbereich.

Library: -300(graphics.library)

Deklaration in C:

```
void BltClear(memory, size, flags)
    APTR memory;
    ULONG size;
    ULONG flags;
```

Aufruf in C: BltClear(memory, size, flags);

Aufruf in Assembler:

```
move.l memory,a1
move.l #size,d0
move.l flags,d1
move.l GfxBase,a5
jsr -300(a5)
```

Parameter:

memory:	Zeiger auf den zu löschenden Speicherbereich
size:	Speichergröße, siehe flags

flags:

bestimmt die Interpretation von size

Bit 1 gesetzt -> höherwertige 16 Bits von size spezifizieren die Höhe, niederwertiger Teil gibt die Breite (in Byte) eines rechteckig-organisierten Speicherbereichs an

Bit 1 gelöscht -> size gibt die Gesamtlänge (ebenfalls in Byte) eines zusammenhängenden Speicherbereichs an

Bit 0 legt fest ob der Task während der Blitter-Operation gestoppt werden soll.

BltMaskBitMapRastPort

Die Funktion kopiert einen rechteckigen Speicherbereich aus der Quell-BitMap in den Ziel-RastPort unter Verwendung eines Masken-Planes.

Library: -636(graphics.library)

Deklaration in C:

```
void BltMaskBitMapRastPort
(srcbm, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm, mask)
    struct BitMap *srcbm;
    short srcx, srcy;
    struct RastPort *destrp;
    short destx, desty, sizex, sizey;
    UBYTE minterm;
    APTR mask
```

Aufruf in C:

```
BltMaskBitMapRastPort
(srcbm, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm, mask);
```

Aufruf in Assembler:

```
move.l srcbm, a0
move.w #srcx, d0
move.w #srcy, d1
move.l destrp, a1
move.w #destx, d2
move.w #desty, d3
move.w #sizex, d4
move.w #sizey, d5
move.b #minterm, d6
move.l mask, a2
move.l GfxBase, a5
jsr -636(a5)
```

Parameter: srcbm: wie BltBitMap()

srcx,srcy:	dito
destrp:	dito
destx,desty:	dito
sizex,sizey:	dito
minterm:	0xe0 Kopie mit unveränderten Werten
	0x20 Invertierung des Quellbereichs
mask:	Zeiger auf Schablonen-Plane

BltPattern

Füllt ein Rechteck (mit Blitter-Unterstützung) gemäß der aktuellen RastPort-Parameter (Zeichenfarbe, Füllmuster, Zeichenmodus).

Library: -312(graphics.library)

Deklaration in C:

```
void BltPattern
(rastport,mask,x,y,maxx,maxy,fillbytes)
    struct RastPort *rastport;
    APTR mask;
    short x,y,maxx,maxy;
    short fillbytes;
```

Aufruf in C:

```
BltPattern
(rastport,mask,x,y,maxx,maxy,fillbytes);
```

Aufruf in Assembler:

```
move.l rastport,a1
move.l mask,a0
move.w #x,d0
move.w #y,d1
move.w #maxx,d2
move.w #maxy,d3
move.w fillbytes,d4
move.l GfxBase,a5
jsr -312(a5)
```

Parameter:	rastport:	Zeiger auf RastPort-Struktur des betreffenden RastPorts
	mask:	Zeiger auf Zeichemaske (Schablone bestehend aus einem Bitplane)
	x,y:	linke, obere Ecke des zu füllenden Rechtecks
	maxx,maxy:	rechte, untere Ecke dieses Rechtecks
	fillbytes:	Breite der Schablone in Byte

BltTemplate

Liest rechteckig-organisierte Speicherblöcke aus einem gepackten Datenfeld und schreibt sie in ein Ziel-Raster.

Library: -36(graphics.library)

Deklaration in C:

```
void BltTemplate
(source, srcx, srcmod, destrp, destx, desty, sizex,
sizey)
    APTR source;
    short srcx, srcmod;
    struct RastPort *destrp;
    short dstx, desty, sizex, sizey;
```

Aufruf in C:

```
BltTemplate
(source, srcx, srcmod, destrp, destx, desty, sizex,
sizey);
```

Aufruf in Assembler:

```
move.l source, a0
move.w #srcx, d0
move.w #srcmod, d1
move.l destrp, a1
move.w #destx, d2
move.w #desty, d3
move.w #sizex, d4
move.w #sizey, d5
move.l GfxBase, a5
jsr -36(a5)
```

Parameter:

source:	Adresse des Quellbereichs (gepacktes Datenfeld)
srcx:	horizontaler Abstand (in Bits) vom linken Feldanfang

srcmod:	Modulo-Wert für die Zeileninkrementierung
destrp:	Zeiger auf RastPort-Struktur des Ziel-Rast-Ports
destx:	X-Koordinate des Ziel-Rasters
desty:	Y-Koordinate des Ziel-Rasters
sizeX:	Breite des Ziel-Rasters
sizeY:	Höhe dieses Speicherbereichs

BuildSysRequest

Erzeugt einen System-Requester mit den angegebenen Parametern.

Library: -360(intuition.library)

Deklaration in C:

```
struct Window *BuildSysRequest
(window,btext,ptext,ntext,flags,w,h)
    struct Window *window;
    struct IntuiText *btext,*ptext,*ntext;
    ULONG flags;
    short w,h;
```

Aufruf in C:

```
rwindow = BuildSysRequest
(window,btext,ptext,ntext,flags,w,h);
```

Aufruf in Assembler:

```
move.l window,a0
move.l btext,a1
move.l ptext,a2
move.l ntext,a3
move.l #flags,d0
move.w #w,d1
move.w #h,d2
move.l IntuitionBase,a5
jsr -360(a5)
```

Parameter:

window:	Zeiger auf Window-Struktur des Fensters, in dem der Requester erscheinen soll
btext:	Zeiger auf IntuiText-Struktur für den Informationstext
ptext:	Zeiger auf IntuiText-Struktur für den positiven Entscheidungstext (rechtes Auswahlgadget)

ntext:	Adresse der IntuiText-Struktur für den negativen Text (linkes Gadget)
flags:	IDCMP-Flags des Requester-Windows
w:	Breite des Requesters
h:	Höhe des Requesters
rwindow:	= window

Anmerkung:

Falls die Funktion den Requester aus Speicherplatzmangel nicht darstellen kann, wird ein Alert mit denselben Textargumenten erzeugt.

BumpRevision

Erzeugt aus einem Dateinamen die Sequenz "copy of <Dateinamen>"

Library: -108(icon.library)

Deklaration in C:

```
char* BumpRevision(newname, oldname)
    UBYTE *newname;
    UBYTE *oldname;
```

Aufruf in C: newbuffer = BumpRevision(newname, oldname);

Aufruf in Assembler:

```
move.l newname,a0
move.l oldname,a1
move.l IconBase,a5
jsr -108(a5)
```

Parameter:

newname:	Zeichenpuffer für den neuen Dateinamen mit der Bezeichnung "copy of..." (sollte mindestens 31 Zeichen puffern können!)
oldname:	Zeiger auf String des alten Dateinamens
newbuffer (d0):	zeigt auf neuen Puffer (newname)

Cause

Softwaremäßige Auslösung eines Interrupts.

Library: -180(exec.library)

Deklaration in C:

```
void Cause(interrupt)
    struct Node *interrupt;
```

Aufruf in C: `Cause(interrupt);`

Aufruf in Assembler:

```
move.l interrupt,a1
move.l ExecBase,a5
jsr -180(a5)
```

Parameter: interrupt: Adresse einer Interrupt-Node-Struktur

CBump

Biegt den Befehlszeiger des Coppers auf den nächsten Befehl der User-Copperliste um.

Library: -366(graphics.library)

Deklaration in C:

```
void CBump(ucoplist)
    struct UCopList *ucoplist;
```

Aufruf in C:

```
CBump(ucoplist);
```

Aufruf in Assembler:

```
move.l ucoplist,a1
move.l GfxBase,a5
jsr -366(a5)
```

Parameter: ucoplist: Zeiger auf eine User-Copperliste

Anmerkung: Die Funktion wird nur selten benötigt, da die Befehlsmakros CMOVE(), CWAIT() und CEND() eine bequemere Initialisierung der Copper-Liste erlauben.

CDInputHandler

Übergibt die vom Input-Task empfangenen Eingaben an das Console-Device.

Library: -42(console.library)

Deklaration in C:

```
void CDInputHandler(events, consoledevice)
    struct Events *events;
    struct Device *consoledevice;
```

Aufruf in C: CDInputHandler(events, consoledevice);

Aufruf in Assembler:

```
move.l events,a0
move.l consoledevice,a1
move.l ConsoleBase,a5
jsr -42(a5)
```

Parameter:

event:	Adresse der Ereignis-Liste
device:	Zeiger auf das Console-Device

Anmerkung: Die Funktion wird in den neueren Betriebssystemversionen nicht mehr benötigt, ist aber aus Kompatibilitätsgründen erhalten geblieben. Normalerweise sollte man die Funktion WriteEvent() verwenden.

ChangeSprite

Verändert das Aussehen eines SimpleSprites (Hardware-Sprite) durch Umbiegen des Datenvektors.

Library: -420(graphics.library)

Deklaration in C:

```
void ChangeSprite
(viewport, simplesprite, daten)
    struct ViewPort *viewport;
    struct SimpleSprite *simplesprite;
    struct Daten *daten;
    /* vom Anwender definierte Struktur */
```

Aufruf in C:

```
void ChangeSprite
(viewport, simplesprite, daten)
```

Aufruf in Assembler:

```
move.l viewport,a0
move.l simplesprite,a1
move.l daten,a2
move.l GfxBase,a5
jsr -420(a5)
```

Parameter:

viewport:	Zeiger auf ViewPort-Struktur
simplesprite:	Adresse der zuvor definierten SimpleSprite-Struktur

daten:

Zeiger auf selbst-definierte Datenstruktur.
Diese Struktur sieht folgendermaßen aus:

```
struct Daten
{
    UWORD postctl[2];
    UWORD data[Spritehöhe][2];
    UWORD reserved[2];
} daten;
```

Sie muß wegen der dynamischen Größe des Datenfeldes (data[[]]) per Hand eingebunden werden.

CheckIO

Überprüft, ob ein begonnener E/A-Prozeß bereits beendet worden ist.

Library: -468(exec.library)

Declaration in C:

```
struct IORequest *CheckIO(iorequest)
    struct IORequest *iorequest;
```

Aufruf in C: `result = CheckIO(iorequest);`

Aufruf in Assembler:

```
move.l iorequest,a1
move.l ExecBase,a5
jsr -468(a5)
```

Parameter:

iorequest:	Zeiger auf IORequest-Struktur
result (d0):	falls E/A-Prozeß bereits beendet, zeigt result auf die zugehörige IORequest-Struktur, ansonsten liefert es den Wert NULL

ClearDMRequest

Löscht einen zuvor initialisierten Double-Menu-Request.

Library: -48(intuition.library)

Deklaration in C:

```
BOOL ClearDMRequest(window)
    struct Window *window;
```

Aufruf in C: `Success = ClearDMRequest(window);`

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a5
jsr -48(a5)
```

Parameter:

window:	Zeiger auf Window-Struktur des Fensters mit aktiver Double-Menu-Request-Abfrage
Success (d0):	bei Erfolg TRUE, sonst FALSE

ClearEOL

Löscht eine Textzeile ab der aktuellen Position des Grafik-Cursors (Operation korrespondiert mit der Ausdehnung des RastPort-Zeichensatzes).

Library: -42(graphics.library)

Deklaration in C: void ClearEOL(rastport)
 struct RastPort *rastport;

Aufruf in C: ClearEOL(rastport);

Aufruf in Assembler: move.l rastport,a1
 move.l GfxBase,a5
 jsr -42(a5)

Parameter: rastport: Zeiger auf RastPort-Struktur

ClearMenuStrip

Löscht die komplette Menüleiste des angegebenen Fensters.

Library: -54(intuition.library)

Deklaration in C:

```
void ClearMenuStrip(window);  
struct Window *window;
```

Aufruf in C:

```
ClearMenuStrip(window);
```

Aufruf in Assembler:

```
move.l window, a0  
move.l IntuitionBase, a5  
jsr -54(a5)
```

Parameter: window: Window-Struktur des betroffenen Fensters

ClearPointer

Löscht die individuelle Mauszeiger-Definition und ersetzt sie durch den Default-Zeiger der Intuition-Benutzeroberfläche.

Library: -60(intuition.library)

Deklaration in C:

```
void ClearPointer(window)
    struct Window *window;
```

Aufruf in C: ClearPointer(window);

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a5
jsr -60(a5)
```

Parameter: window: Zeiger auf eine Window-Struktur

ClearRectRegion

Löscht ein Clipping-Rechteck in der angegebenen Region.

Library: -552(graphics.library)

Deklaration in C:

```
BOOL ClearRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
```

Aufruf in C: status = ClearRectRegion(region, rectangle);

Aufruf in Assembler:

```
move.l region,a0
move.l rectangle,a1
move.l GfxBase,a5
jsr -522(a5)
```

Parameter:

region:	Zeiger auf Region-Struktur
rectangle:	Adresse einer Rectangle-Struktur
status (d0):	bei erfolgreichem Löschvorgang TRUE, sonst FALSE

Anmerkung: In den ersten Kickstart-Versionen gab es die ClearRectRegion()-Funktion noch nicht. Später wurde ihre Einsprungsadresse an der für diese Erweiterung reservierten Position (-522) in die Graphics-Library eingefügt.

ClearRegion

Die Funktion löscht die komplette Region des zugehörigen Layers.

Library: -528(graphics.library)

Deklaration in C: void ClearRegion(region)
 struct Region *region;

Aufruf in C: ClearRegion(region);

Aufruf in Assembler: move.l region,a0
 move.l GfxBase,a5
 jsr -528(a5)

Parameter: region: Adresse der Region-Struktur

ClearScreen

Löscht den Bildschirm ab der aktuellen Position des Grafik-Cursors.

Library: -48(graphics.library)

Deklaration in C: void ClearScreen(rastport);

Aufruf in C: ClearScreen(rastport);

Aufruf in Assembler: move.l rastport,a1
 move.l GfxBase,a5
 jsr -48(a5)

Parameter: rastport: Zeiger auf RastPort-Struktur

ClipBlit

Kopiert rechteckige Speicherblöcke mit Blitter-Unterstützung. Im Gegensatz zu Blt-Bitmap() stammen diese Speicherbereiche aber nicht aus einer BitMap, sondern aus einem RastPort.

Library: -552(graphics.library)

Deklaration in C:

```
void ClipBlit
(srcrp, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm)
    struct RastPort *srcrp, *destrp;
    short srcx, srcy, destx, desty, sizex, sizey;
    UBYTE minterm;
```

Aufruf in C:

```
ClipBlit
(srcrp, srcx, srcy, destrp, destx, desty, sizex,
sizey, minterm);
```

Aufruf in Assembler:

```
move.l srcrp, a0
move.w #srcx, d0
move.w #srcy, d1
move.l destrp, a1
move.w #destx, d2
move.w #desty, d3
move.w #sizex, d4
move.w #sizey, d5
move.b #minterm, d6
move.l GfxBase, a5
jsr -552(a5)
```

Parameter:	srcrp:	zeigt auf RastPort-Struktur des Quell-Rastports
	srcx:	X-Koordinate des zu kopierenden Rasterausschnitts
	srcy:	Y-Koordinate des zu kopierenden Rasterausschnitts
	destrp:	zeigt auf Struktur des Ziel-RastPorts
	destx:	X-Koordinate des Zielbereichs
	desty:	Y-Koordinate des Zielbereichs
	size:	horizontale Ausdehnung des Raster (Anzahl von Spalten)
	sizey:	vertikale Ausdehnung des Rasters (Zeilen)
	minterm:	logische Verknüpfung (MinTerm-Bits)

Close

Schließt das durch die Funktion `Open()` geöffnete File.

Library: `-36(dos.library)`

Deklaration in C:

```
void Close(file)
    struct FileHandle *file;
```

Aufruf in C: `Close(file);`

Aufruf in Assembler:

```
move.l file,d1
move.l DOSBase,a5
jsr -36(a5)
```

Parameter:

<code>file:</code>	Zeiger auf eine <code>FileHandle</code> -Struktur (dient zur Bearbeitung des Files)
--------------------	---

CloseDevice

Schließt das mittels der angegebenen IORequest-Struktur bearbeitete Device.

Library: -450(exec.library)

Deklaration in C:

```
void CloseDevice(iorequest);  
    struct IORequest *iorequest;
```

Aufruf in C: `CloseDevice(iorequest);`

Aufruf in Assembler;

```
move.l iorequest,a1  
move.l ExecBase,a5  
jsr -450(a5)
```

Parameter: iorequest: Adresse der IORequest-Struktur

CloseFont

Schließt den durch die Funktionen `OpenFont()` bzw. `OpenDiskFont()` geöffneten Zeichensatz.

Library: `-78(graphics.library)`

Deklaration in C:

```
void CloseFont(textfont)
    struct TextFont *textfont;
```

Aufruf in C: `CloseFont(textfont);`

Aufruf in Assembler:

```
move.l textfont,a1
move.l GfxBase,a5
jsr -78(a5)
```

Parameter:

<code>textfont:</code>	Zeiger auf die <code>TextFont</code> -Struktur des zu schließenden Zeichensatzes
------------------------	--

CloseLibrary

Schließt die angegebene Funktionsbibliothek.

Library: -414(exec.library)

Deklaration in C: void CloseLibrary(library)
 struct Library *library;

Aufruf in C: CloseLibrary(library);

Aufruf in Assembler: move.l library, a1
 move.l ExecBase, a5
 jsr -414(a5)

Parameter: library: Zeiger auf die zu schließende Library

CloseScreen

Schließt den durch den Zeiger screen adressierten Screen.

Library: -66(intuition.library)

Deklaration in C:

```
void CloseScreen(screen)
    struct Screen *screen;
```

Aufruf in C: CloseScreen(screen);

Aufruf in Assembler:

```
move.l screen,a0
move.l IntuitionBase,a5
jsr -66(a5)
```

Anmerkung: Eventuell noch sichtbare Fenster werden nicht automatisch geschlossen, sondern sollten vor Aufruf der CloseScreen-Funktion verlassen werden.

CloseWindow

Schließt das durch window spezifizierte Fenster.

Library: -72(intuition.library)

Deklaration in C:

```
void CloseWindow(window)
    struct Window *window;
```

Aufruf in C: `CloseWindow(window);`

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a5
jsr -72(a5)
```

Parameter: window: Zeiger auf das zu schließende Window

Anmerkung: Beim Schließen des Fensters werden alle IDCMP-Messages freigegeben. Bei der Verwendung eines Message-Ports sollte darauf geachtet werden, daß vor Aufruf der Funktion CloseWindow sämtliche Messages ordnungsgemäß quittiert sind (ReplyMsg-Funktion).

CloseWorkBench

Die Funktion verläßt die WorkBench-Oberfläche durch Schließen des Fensters.

Library: -78(intuition.library)

Deklaration in C: `BOOL CloseWorkBench()`

Aufruf in C: `Succes = CloseWorkBench();`

Aufruf in Assembler: `move.l IntuitionBase,a5
jsr -78(a5)`

Parameter: Succes: bei erfolgreichem Schließen der WorkBench
TRUE, sonst FALSE

Anmerkung: Sollte noch irgendein Programm-Fenster sichtbar sein, so wird die WorkBench nicht geschlossen.

CMove

Plaziert einen Copper-MOVE-Befehl in die angegebene User-Copperliste.

Library: -372(graphics.library)

Deklaration in C:

```
void CMove(ucoplist, register, data)
    struct UCoplist *ucoplist;
    APTR register;
    short data;
```

Aufruf in C: CMove(ucoplist, register, data);

Aufruf in Assembler:

```
move.l ucoplist,a1
move.w register,d0
move.w data,d1
move.l GfxBase,a5
jsr -372(a5)
```

Parameter:	ucoplist:	Zeiger auf eine User-Copperliste
	register:	Tabellen-Offset des Zielregisters
	data:	unmittelbarer Datenwert, der in das Custom-chip-Register geschrieben werden soll

CmpTime

Vergleicht die Zeiten zweier timeval-Strukturen.

Library: -54(timer.library)

Deklaration in C:

```
short CmpTime(destination, source)
struct timeval *destination, *source;
```

Aufruf in C:

```
result = CmpTime(destination, source)
```

Aufruf in Assembler:

```
move.l destination,a0
move.l source,a1
move.l TimerBase,a5
jsr -54(a5)
```

Parameter:

destination:	Ziel-timeval-Struktur
source:	Quell-timeval-Struktur
result (d0):	Ergebnis des Vergleichs

= 0	beide Zeiten sind gleich
= +1	Zielzeit größer als Quellzeit
= -1	Quellzeit größer als Zielzeit

Anmerkung: Register A6 enthält einen Zeiger auf das Timer-Device.

ConcatCList

Verknüpft zwei CListen.

Library: -156(clist.library)

Deklaration in C:

```
long ConcatCList(sourceclist, destclist)
long sourceclist, destclist;
```

Aufruf in C:

```
Error = ConcatCList(sourceclist, destclist);
```

Aufruf in Assembler:

```
move.l sourceclist,a0
move.l destclist,a1
move.l CListBase,a5
jsr -156(a5)
```

Parameter:

sourceclist:	Quell-CListe
destclist:	Ziel-CListe
Error (d0):	NULL bei erfolgreicher Verknüpfung

ConfigBoard

Konfiguriert eine Erweiterungskarte.

Library: -60(expansion.library)

Deklaration in C:

```
long ConfigBoard(board, configdev)
    long board;
    struct ConfigDev *configdev;
```

Aufruf in C:

```
error = ConfigBoard(board, configdev);
```

Aufruf in Assembler:

```
move.l #board,a0      ;Konfigurationsadresse
move.l configdev,a1
move.l ExpansionBase,a5
jsr -60(a5)
```

Parameter:

board:	Adresse an der die Karte eingebunden werden soll
configdev:	Zeiger auf eine ConfigDev-Struktur
error (d0):	NULL, wenn erfolgreich

ConfigChain

Bindet alle Erweiterung ab Adresse E_EXPANSIONBASE in die Konfigurationsliste des Systems ein.

Library: -66(expansion.library)

Deklaration in C:

```
long ConfigChain(baseaddr)
    long baseaddr;
```

Aufruf in C: `error = ConfigChain(baseaddr)`

Aufruf in Assembler:

```
move.l #baseaddr,a0      ;Basisadresse der
                           Erweiterungskarten
move.l ExpansionBase,a5
jsr -66(a5)
```

Parameter: `board:` Basisadresse
(normalerweise E_EXPANSIONBASE)
`error (d0):` NULL, wenn erfolgreich, sonst Wert kleiner
als 0

CopyCList

Erzeugt eine Kopie der angegebenen CListe (im selben CPool).

Library: -144(clist.library)

Deklaration in C:

```
long CopyCList(clist)
long clist;
```

Aufruf in C: newclist = CopyCList(clist);

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a5
jsr -144(a5)
```

Parameter:

clist:	Identifikationsnummer der zu kopierenden CListe
newclist (d0):	Identifikationsnummer der Kopie oder ein Wert kleiner NULL bei Abbruch wegen Speichermangel

CopyMem

Kopiert einen Speicherbereich von der Quelladresse an die Zieladresse.

Library: -624(exec.library)

Deklaration in C:

```
void CopyMem(source, destination, size)
    ULONG *source, *destination;
    ULONG size;
```

Aufruf in C: CopyMem(source, destination, size);

Aufruf in Assembler:

```
move.l source,a0
move.l destination,a1
move.l #size,d0
move.l ExecBase,a5
jsr -624(a5)
```

Parameter:

source:	Adresse des Quellbereichs
destination:	Zieladresse
size:	Größe des Speicherblocks in Byte

CopyMemQuick

Optimierte Version der Kopierroutine CopyMem().

Library: -630(exec.library)

Deklaration in C:

```
void CopyMemQuick(source, destination, size)
    ULONG *source, *destination;
    ULONG size;
```

Aufruf in C: CopyMemQuick(source, destination, size);

Aufruf in Assembler:

```
move.l source,a0
move.l destination,a1
move.l #size,d0
move.l ExecBase,a5
jsr -630(a5)
```

Parameter:

source:	Adresse des Quellbereichs
destination:	Zieladresse
size:	Größe des zu kopierenden Speicherblocks in Byte

Anmerkung: Die Adreßangaben müssen Longword-aligned sein, das heißt, der Wert von size ist ohne Rest durch 4 teilbar.

CopySBitMap

Invertierte Form von SyncSBitMap(), kopiert Teile aus der Super-Bitmap in die Layer-Bitmap. Beim Aufruf der Funktion sollte das betreffende Layer mit einem Zugriffs-Lock versehen sein.

Library: -450(graphics.library)

Deklaration in C:

```
void CopySBitMap(layer)
    struct Layer *layer;
```

Aufruf in C:

```
CopySBitMap(layer);
```

Aufruf in Assembler:

```
move.l layer,a0
move.l GfxBase,a5
jsr -450(a5)
```

Parameter: layer: Zeiger auf Layer-Sruktur eines LAYERSUPER-Layers

CreateBehindLayer

Erzeugt ein Layer im Hintergrund des Layer-Stapels (hinter den anderen Layers).

Library: -42(layers.library)

Deklaration in C:

```
struct Layer *CreateBehindLayer
(li,bm,x0,y0,x1,y1,flags,bm2)
    struct Layer_Info *li;
    struct BitMap *bm;
    LONG x0,y0,x1,y1;
    LONG flags;
    struct Bitmap *bm2;
```

Aufruf in C:

```
layer = CreateBehindLayer
(li,bm,x0,y0,x1,y1,flags,bm2);
```

Aufruf in Assembler:

```
move.l li,a0
move.l bm,a1
move.l #x0,d0
move.l #y0,d1
move.l #x1,d2
move.l #y1,d3
move.l #flags,d4
move.l bm2,a2
move.l LayersBase,a5
jsr -42(a5)
```

Parameter:

li:	Zeiger auf eine Layer_Info-Struktur
bm:	Adresse der Layer-Bitmap
x0,y0:	linke, obere Ecke des Layers
x1,y1:	rechte, untere Ecke

flags: übliche Layer-Flags (LAYERSIMPLE, LAYERSMART, LAYERSUPER, BACK-DROP)

bm2: Zeiger auf Super-Bitmap

layer (d0): Zeiger auf Layer-Struktur des erzeugten Layers

CreateDir

Legt auf dem aktuellen Laufwerk das Unterverzeichnis name an.

Library: -120(dos.library)

Deklaration in C: struct FileLock *CreateDir(name)
 UBYTE *name;

Aufruf in C: newlock = CreateDir(name);

Aufruf in Assembler: move.l name,d1
 move.l DosBase,a5
 jsr -120(a5)
 name:
 dc.b "Name",0 ;NULL-terminierter
 Bezeichnungsstring

Parameter: name: Bezeichnung des Unterdirectories
 newlock (d0): BCPL-Zeiger auf neuen Lock

CreateProc

Die Funktion initialisiert eine Process-Struktur und startet das Programm in der Segment-Liste.

Library: -138(dos.library)

Deklaration in C:

```
struct Process *CreateProc
(name, pri, seglist, stack)
    UBYTE *name;
    LONG pri;
    BPTR* seglist;
    LONG stack;
```

Aufruf in C: process = CreateProc(name, pri, seglist, stack)

Aufruf in Assembler:

```
move.l name,d1
move.l #pri,d2
move.l seglist,d3
move.l #stack,d4
move.l DosBase,a5
jsr -138(a5)
```

Parameter:

name:	Bezeichnung des Prozesses
pri:	Priorität [-128...-127]
seglist:	Zeiger auf Segment-Liste (BCPL!)
stack:	Programm-Stack in Byte
process:	BCPL-Zeiger auf Message-Port der Process-Struktur oder NULL bei Mißerfolg

CreateUpfrontLayer

Erzeugt ein Layer im Vordergrund des Layer-Stapels (vor den anderen Layers).

Library: -36(layers.library)

Deklaration in C:

```
struct Layer *CreateUpfrontLayer
(li,bm,x0,y0,x1,y1,flags,bm2)
{
    struct LayerInfo *li;
    struct Bitmap *bm;
    LONG x0,y0,x1,y1;
    LONG flags;
    struct Bitmap *bm2;
```

Aufruf in C:

```
layer = CreateUpfrontLayer
(li,bm,x0,y0,x1,y1,flags,bm2);
```

Aufruf in Assembler:

```
move.l li,a0
move.l bm,a1
move.l #x0,d0
move.l #y0,d1
move.l #x1,d2
move.l #y1,d3
move.l #flags,d4
move.l bm2,a2
move.l LayersBase,a5
jsr -36(a5)
```

Parameter:

li:	Zeiger auf eine Layer_Info-Struktur
bm:	Adresse der Layer-Bitmap
x0,y0:	linke, obere Ecke des Layers
x1,y1:	rechte, untere Ecke

flags:	übliche Layer-Flags (LAYERSIMPLE, LAYERSMART, LAYERSUPER, BACK-DROP)
bm2:	Zeiger auf Super-Bitmap
layer (d0):	Zeiger auf Layer-Struktur des erzeugten Layers

CurrentDir

Erhebt das durch lock definierte Directory zum aktuellen Verzeichnis (siehe CLI-Befehl CD).

Library: -126(dos.library)

Deklaration in C:

```
struct FileLock *CurrentDir(lock)
    struct FileLock *lock;
```

Aufruf in C: oldlock = CurrentDir(lock);

Aufruf in Assembler:

```
move.l lock,d1
move.l DosBase,a5
jsr -126(a5)
```

Parameter:

lock:	BCPL-Zeiger auf FileLock-Struktur des neuen Directories
oldlock (d0):	BCPL-Zeiger auf das alte Verzeichnis

CurrentTime

Kopiert die aktuelle Systemzeit in die angegebenen Puffer.

Library: -84(intuition.library)

Deklaration in C: `void CurrentTime(seconds, micros)
ULONG *seconds, *micros;`

Aufruf in C: `CurrentTime(seconds, micros);`

Aufruf in Assembler: `move.l seconds,a0
move.l micros,a1
move.l IntuitionBase,a5
jsr -84(a5)`

Parameter: `seconds:` Puffer für Sekunden
`micros:` Puffer für Mikrosekunden

Anmerkung: Die Funktion liefert keine genauen Werte, da diese maximal 60 mal pro Sekunde neu gesetzt werden. Bei Normalbetrieb geschieht die Aktualisierung noch weitaus seltener.

CWait

Plaziert einen Copper-WAIT-Befehl in die angegebene User-Copper-Liste.

Library: -378(graphics.library)

Deklaration in C:

```
void CWait(ucoplist, x, y)
    struct UCopList *ucoplist;
    short x, y;
```

Aufruf in C: CWait(ucoplist, x, y);

Aufruf in Assembler:

```
move.l ucoplist,a1
move.w #x,d0
move.w #y,d1
```

Parameter:

ucoplist:	Zeiger auf eine User-Copperliste
x,y:	Wartekoordinate für den Copper

DateStamp

Speichert die aktuelle Zeit in dem angegebenen Puffer ab.

Library: -192(dos.library)

Deklaration in C: `void DateStamp(date)
LONG *date;`

Aufruf in C: `DateStamp(date);`

Aufruf in Assembler: `move.l date,d1
move.l DosBase,a5
jsr -192(a5)`

Parameter: `date:` Zeiger auf einen Puffer bestehend aus 3 Langwörtern bzw. 12 Byte, der nach Aufruf der Funktion folgende Werte enthält:

1. Langwort: Anzahl der seit dem 1.1.1978 vergangenen Tage
2. Langwort: Anzahl der Minuten seit Mitternacht
3. Langwort: Anzahl der Netzpulse (1 Tick = 1/50s) seit der letzten vollen Minute

Deallocate

Deallocate() gibt den mit der Allocate() reservierten Speicherblock frei und fügt ihn wieder der FreeList des Speicherverwaltungssystems hinzu.

Library: -192(exec.library)

Deklaration in C:

```
void Deallocate(freelist, memory, size)
    struct MemHeader *freelist;
    ULONG *memory;
    ULONG size;
```

Aufruf in C: Deallocate(freelist, memory, size);

Aufruf in Assembler:

```
move.l freelist,a0
move.l memory,a1
move.l #size,d0
move.l ExecBase,a5
jsr -192(a5)
```

Parameter:

freelist:	Zeiger auf die zugehörige MemHeader-Struktur
memory:	Adresse des zurückgegebenen Speicherblocks
size:	seine Größe in Byte

Debug

Debug() ruft den Debugger des Systems auf.

Library: -114(exec.library)

Deklaration in C: void Debug()

Aufruf in C: Debug();

Aufruf in Assembler: move.l ExecBase,a5
jsr -114(a5)

Delay

Friert den Programmfluß für den durch ticks beschriebenen Zeitraum ein.

Library: -198(dos.library)

Deklaration in C: void Delay(ticks)
 LONG ticks;

Aufruf in C: Delay(ticks);

Aufruf in Assembler: move.l #ticks,d1
 move.l DosBase,a5
 jsr -198(a5)

Parameter: ticks: Anzahl der Ticks, die abgewartet werden sollen (1/50s)

DeleteFile

Löscht das File mit der Bezeichnung name im aktuellen Directory.

Library: -72(dos.library)

Deklaration in C: `BOOL DeleteFile(name)
 UBYTE *name;`

Aufruf in C: `success = DeleteFile(name);`

Aufruf in Assembler: `move.l name,d1
 move.l DosBase,a5
 jsr -72(a5)`

Parameter:

name:	Zeiger auf einen nullterminierten Namensstring
success (d0):	TRUE, falls die Löschoperation erfolgreich war, sonst FALSE

DeleteLayer

Löscht den für von einer Layer-Struktur belegten Speicherbereich.

Library: -90(layers.library)

Deklaration in C:

```

BOOL DeleteLayer(li, layer)
    struct Layer_Info *li;
    struct Layer *layer;

```

Aufruf in C: result = DeleteLayer(li, layer);

Aufruf in Assembler:

```

move.l li,a0
move.l layer,a1
move.l LayersBase,a5
jsr -90(a5)

```

Parameter:

li:	Zeiger auf die Layer_Info-Struktur
layer:	Adresse der zu löschenden Layer-Struktur
resultd0:	bei erfolgreicher Operation TRUE, sonst FALSE

DeviceProc

Ermittelt den zuständigen Treiber des durch name bezeichneten Gerätes.

Library: -174(dos.library)

Deklaration in C:

```
struct Process *DeviceProc(name)
    UBYTE *name;
```

Aufruf in C:

```
process = DeviceProc(name);
```

Aufruf in Assembler:

```
move.l name,d1
move.l DosBase,a5
jsr -174(a5)
```

Parameter:

name:	Bezeichnung des Gerätes, dessen Treiber gesucht wird
process (d0):	BCPL-Zeiger auf den Message-Port der Process-Struktur des gesuchten Gerätetreibers oder NULL bei erfolgloser Suche

Disable

Sperrt die Systeminterrupt durch Negation des Interrupt-Hauptschalters im INTENA-Register. Enable() hebt die Sperrung wieder auf.

Library: -120(exec.library)

Deklaration in C: void Disable()

Aufruf in C: Disable();

Aufruf in Assembler: move.l ExecBase,a5
jsr -120(a5)

Anmerkung: Jeder Disable()-Aufruf muß irgendwann durch Enable() neutralisiert werden. Wenn der Task, der Disable() aufgerufen hat, in den Wait-Status übergeht, kehrt das System solange zum normalen Ablauf zurück, bis der Task wieder aktiv wird.

DisownBlitter

Hebt die privilegierte Blitter-Benutzung eines Tasks auf und übergibt die Zugriffsberechtigung an das Gesamtsystem (allgemeine Zugriffsberechtigung).

Library: -456(graphics.library)

Deklaration in C: void DisownBlitter()

Aufruf in C: DisownBlitter();

Aufruf in Assembler: move.l fxBase, a5
jsr -456(a5)

Dispatch

Library: -60(exec.library)

Deklaration in C: void Dispatch()

Aufruf in C: Dispatch();

Aufruf in Assembler: move.l ExecBase, a5
jsr -60(a5)

DisplayAlert

Gibt eine Alert-Warnung aus.

Library: -90(intuition.library)

Deklaration in C:

```

BOOL DisplayAlert(alertnumber, string, height)
    ULONG alertnumber;
    UBYTE *string;
    short height;
  
```

Aufruf in C:

```

response = DisplayAlert
(alertnumber, string, height);
  
```

Aufruf in Assembler:

```

move.l #alertnumber,d0
move.l string,a0
move.w height,a1
move.l IntuitionBase,a5
jsr -90(a5)
  
```

Parameter:

alertnumber:	Alert-Vektor
string:	Zeiger auf Alert-Text
height:	Zeilenhöhe des Alert-Kastens (Screen wird bei der Darstellung um diesen Wert nach unten verschoben)
response (d0):	RECOVER-ALERTSs je nach Entscheidung TRUE oder FALSE, DEADEND_ALERTs immer FALSE

DisplayBeep

Löst ein kurzes Aufblinken des Screens aus. (realisiert durch direkte Modifikation von Farbregister 0, wird nach spätestens 1/50 s vom Copper wieder überschrieben).

Library: -96(intuition.library)

Deklaration in C:

```
void DisplayBeep(screen)
    struct Screen *screen;
```

Aufruf in C: `DisplayBeep(screen);`

Aufruf in Assembler:

```
move.l screen,a0
move.l IntuitionBase,a5
jsr -96(a5)
```

Parameter: `screen:` Zeiger auf Screen-Struktur des betroffenen Screens

DisposeLayerInfo

Löscht die angegebene Layer_Info-Struktur und gibt den freigewordenen Speicher an das System zurück.

Library: -150(layers.library)

Deklaration in C:

```
void DisposeLayerInfo(li)
    struct Layer_Info *li;
```

Aufruf in C: DisposeLayerInfo(li);

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a5
jsr -150(a5)
```

Parameter: li: Zeiger auf Layer_Info-Struktur

Anmerkung: Vor Benutzung dieser Funktion muß für jedes Layer Delete-Layer() aufgerufen werden.

DisposeRegion

Löscht die angegebene Region und gibt den Speicher an das System zurück.

Library: -534(graphics.library)

Deklaration in C:

```
void DisposeRegion(region)
    struct Region *region;
```

Aufruf in C: `DisposeRegion(region);`

Aufruf in Assembler:

```
move.l region,a0
move.l GfxBase,a5
jsr -534(a5)
```

Parameter: `region:` zeigt auf Region-Struktur der zu löschenden Region

DoCollision

Software-Kollisionsabfrage im spezifizierten RastPort.

Library: -108(graphics.library)

Deklaration in C: void DoCollision(rp)
 struct RastPort *rp;

Aufruf in C: DoCollision(rp);

Aufruf in Assembler: move.l rp,a1
 move.l GfxBase,a5
 jsr -108(a5)

Parameter: rp: zeigt auf die RastPort-Struktur des unter-
 suchten RastPorts

DoIO

Startet die Ausführung des in der IORequest-Struktur definierten Kommandos.

Library: -456(exec.library)

Deklaration in C:

```
long DoIO(iorequest)
    struct IORequest *iorequest;
```

Aufruf in C: `DoIO(iorequest);`

Aufruf in Assembler:

```
move.l iorequest,a1
move.l ExecBase,a5
jsr -456(a5)
```

Parameter:

iorequest:	zeigt auf IORequest-Struktur des E/A-Prozesses
error (d0):	NULL bei erfolgreicher Operation, sonst Wert kleiner 0

DoubleClick

Bildet die Differenz zweier Zeitangaben und vergleicht das Ergebnis mit dem aktuellen DoubleClick-Intervall (definiert im Preferences).

Library: -102(intuition.library)

Deklaration in C:

```
BOOL DoubleClick  
(sseconds, smicros, cseconds, cmicros)  
    long sseconds, smicros, cseconds, cmicros;
```

Aufruf in C:

```
isdouble = DoubleClick  
(sseconds, smicros, cseconds, cmicros);
```

Aufruf in Assembler:

```
move.l #sseconds,d0  
move.l #smicros,d1  
move.l #cseconds,d2  
move.l #smicros,d3  
move.l IntuitionBase,a5  
jsr -102(a5)
```

Parameter:

sseconds, smicros:	Startzeit
csecond, scmikros:	aktuelle Zeit
isdouble (d0):	TRUE, falls Zeitdifferenz einem Double-Click entspricht

Draw

Line-Draw-Funktion. Verbindet die aktuelle Grafik-Cursor-Position mit der durch den X,Y-Wert definierten Koordinate.

Library: -246(graphics.library)

Deklaration in C:

```
void Draw(rp, x, y);  
    struct RastPort *rp;  
    short x, y;
```

Aufruf in C: Draw(rp, x, y);

Aufruf in Assembler:

```
move.l rp,a1  
move.w #x,d0  
move.w #y,d1  
move.l GfxBase,a5  
jsr -246(a5)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Draw-Rast-Ports
x,y:	Endkoordinate der Linie

DrawBorder

Zeichnet ein Border-Objekt (bzw. mehrere, falls die Border-Strukturen miteinander verkettet sind) in den angegebenen RastPort.

Library: -108(intuition.library)

Deklaration in C:

```
void DrawBorder
(rp, border, leftoffset, toptoffset)
    struct RastPort *rp;
    struct Border *border;
    short leftoffset, rightoffset;
```

Aufruf in C: DrawBorder(rp, border, leftoffset, toptoffset);

Aufruf in Assembler:

```
move.l rp,a0
move.l border,a1
move.w leftoffset,d0
move.w toptoffset,d1
move.l IntuitionBase,a5
jsr -108(a5)
```

Parameter:

rp:	Zeiger auf Draw-RastPort
border:	Adresse einer Border-Struktur (eventuell verkettet)
leftoffset:	X-Offset an der das Objekt in den RastPort gezeichnet wird
topoffset:	Y-Offset vom RastPort-Rand

DrawEllipse

Zeichnet eine Ellipse in den angegebenen RastPort.

Library: -180(graphics.library)

Deklaration in C:

```
void DrawEllipse(rp, centerx, centery, rx, ry)
    struct RastPort *rp;
    short centerx, centery, rx, ry;
```

Aufruf in C: DrawEllipse(rp, centerx, centery, rx, ry);

Aufruf in Assembler:

```
move.l rp,a1
move.w #centerx,d0
move.w #centery,d1
move.w #rx,d2
move.w #ry,d3
move.l GfxBase,a5
jsr -180(a5)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur
centerx,centery:	Mittelpunkts-Koordinate der Ellipse
rx:	X-Radius
ry:	Y-Radius

DrawGList

Stellt die in der GEL-Liste definierten Grafikobjekte dar. (Vor Aufruf der Funktion empfiehlt es sich, die Liste mit SortGList() zu ordnen.)

Library: -114(graphics.library)

Deklaration in C:

```
void DrawGList(rp, vp)
    struct RastPort *rp;
    struct ViewPort *vp;
```

Aufruf in C: DrawGList(rp, vp);

Aufruf in Assembler:

```
move.l rp,a1
move.l vp,a0
move.l GfxBase,a5
jsr -114(a5)
```

Parameter:

rp:	Zeiger auf RastPort, in dem die GEL-Liste definiert ist
vp:	Zeiger auf die Struktur des zugehörigen ViewPorts

Anmerkung: MUSTDRAW ist noch nicht implementiert.

DrawImage

Zeichnet das durch die Image-Struktur definierte Objekt (bzw. Objekte, falls die Struktur verkettet ist) in den spezifizierten RastPort.

Library: -114(intuition.library)

Deklaration in C:

```
void DrawImage
(rp, image, leftoffset, topOffset)
struct RastPort *rp;
struct Image *image;
short leftoffset, topoffset;
```

Aufruf in C: DrawBorder(rp, border, leftoffset, topoffset);

Aufruf in Assembler:

```
move.l rp,a0
move.l border,a1
move.w leftoffset,d0
move.w topoffset,d1
move.l IntuitionBase,a5
jsr -114(a5)
```

Parameter:	rp:	Zeiger auf Draw-RastPort
	border:	Adresse einer Image-Struktur (eventuell verkettet)
	leftoffset:	X-Offset, an der das Objekt in den RastPort gezeichnet wird
	topoffset:	Y-Offset vom RastPort-Rand

DupLock

DupLock() erzeugt eine Kopie des angegebenen Locks und liefert einen Zeiger auf dessen FileLock-Struktur. Es werden allerdings nur Locks vom Typ ACCES_READ bearbeitet.

Library: -96(dos.library)

Deklaration in C:

```
struct FileLock *DupLock(lock)
    struct FileLock *lock;
```

Aufruf in C:

```
newlock = DupLock(lock);
```

Aufruf in Assembler:

```
move.l lock,d1
move.l DosBase,a5
jsr -96(a5)
```

Parameter:

lock:	Zeiger auf FileLock-Struktur des zu duplizierenden Locks
newlock:	Zeiger auf FileLock-Struktur der Kopie

Enable

Hebt die Sperrung des Interrupt-Systems wieder auf (Gegenstück zu Disable).

Library: -126(exec.library)

Deklaration in C: void Enable()

Aufruf in C: Enable();

Aufruf in Assembler: move.l ExecBase,a5
jsr -126(a5)

Anmerkung: Für jeden Disable-Aufruf muß auch einmal die Enable()-Funktion ausgeführt werden.

EndRefresh

Hebt den Refresh-Status nach Aufruf der Funktion BeginRefresh() auf.

Library: -366(intuition.library)

Deklaration in C:

```
void EndRefresh(window, complete)
    struct Window *window;
    BOOL complete;
```

Aufruf in C: EndRefresh(window, complete);

Aufruf in Assembler:

```
move.l window, a0
move.b #complete, d0
move.l IntuitionBase, a5
jsr -366(a5)
```

Parameter:

window:	Zeiger auf eine Window-Struktur
complete:	TRUE, falls Refreshzyklus beendet, sonst FALSE

EndRequest

Beendet die Abfrage des spezifizierten Requesters und entfernt diesen aus dem Darstellungsfenster.

Library: -120(intuition.library)

Deklaration in C:

```
void EndRequest(requester, window)
    struct Requester *requester;
    struct Window *window;
```

Aufruf in C: EndRequest(requester, window);

Aufruf in Assembler:

```
move.l requester,a0
move.l window,a1
move.l IntuitionBase,a5
jsr -120(a5)
```

Parameter:

requester:	Zeiger auf Requester-Struktur, dessen Abfragezyklus beendet wird
window:	Zeiger auf Window-Struktur des zugehörigen Fensters

EndUpdate

Restauriert die ClipRect-Liste nach einem Layer-Refreshzyklus.

Library: -84(layers.library)

Deklaration in C:

```
void EndUpdate(layer, flag)
    struct Layer *layer;
    BOOL flag;
```

Aufruf in C: EndUpdate(layer, flag);

Aufruf in Assembler:

```
move.l layer,a0
move.b #flag,d0
move.l LayersBase,a5
jsr -84(a5)
```

Parameter:

layer:	zeigt auf Layer-Struktur des betroffenen Layers
flag:	ist der Update-Zyklus bereits komplett, wird durch Übergabe von TRUE die Damage-Liste gelöscht (sie bleibt bei FALSE erhalten)

Enqueue

Erweitert die Ziel-Liste um die angegebene Node. Die Listenposition ist abhängig von der in der Node-Struktur definierten Priorität.

Library: -270(exec.library)

Deklaration in C:

```
void Enqueue(list, node)
    struct List *list;
    struct Node *node;
```

Aufruf in C: Enqueue(list, node);

Aufruf in Assembler:

```
move.l list,a0
move.l node,a1
move.l ExecBase,a5
jsr -270(a5)
```

Parameter:

list:	Zeiger auf eine List-Struktur
node:	Adresse der Node-Struktur

Examine

Initialisiert die übergebene FileInfoBlock-Struktur mit den Parametern der durch lock spezifizierten Datei (Größe, Typ, Erstellungsdatum usw.)

Library: -102(dos.library)

Deklaration in C:

```
BOOL Examine(lock, fileinfoblock)
    struct FileLock *lock;
    struct FileInfoBlock *fileinfoblock;
```

Aufruf in C:

```
success = Examine(lock, fileinfoblock);
```

Aufruf in Assembler:

```
move.l lock,d1
move.l fileinfoblock,d2
move.l DosBase,a5
jsr -102(a5)
```

Parameter:

lock:	Adresse der Datei-Lock-Struktur (BCPL-Zeiger)
fileinfoblock:	Zeiger auf eine FileInfoBlock-Struktur
success (d0):	TRUE bei erfolgreicher Initialisierung / sonst FALSE

Exception

Library: -66(exec.library)

Deklaration in C: void Exception()

Aufruf in C: Exception();

Aufruf in Assembler: move.l ExecBase,a5
jsr -66(a5)

Execute

Führt ein CLI-Kommando aus (wird als String übergeben) und lenkt die Ausgabe in den spezifizierten Kanal.

Library: -222(dos.library)

Deklaration in C:

```

BOOL Execute(command, in, out)
    UBYTE *command;
    struct FileHandle *in, *out;

```

Aufruf in C:

```

success = Execute(command, in, out);

```

Aufruf in Assembler:

```

move.l command,d1
move.l in,d2
move.l out,d3
move.l DosBase,a5
jsr -222(a5)

```

Parameter:

command:	CLI-Befehl als nullterminierter String
in:	Zeiger auf eine Datei aus der weitere CLI-Kommandos gelesen werden; ist normalerweise NULL
out:	Zeiger auf Ausgabe-Kanal (bei NULL verwendung des CLI-Fensters)
success (d0):	TRUE bei erfolgreicher Operation

Exit

Verläßt das Programm und kehrt in die Startumgebung zurück.

Library: -144(dos.library)

Deklaration in C: `void Exit(returncode)
long returncode;`

Aufruf in C: `Exit(returncode);`

Aufruf in Assembler: `move.l #returncode,d1
move.l DosBase,a5
jsr -144(a5)`

Parameter: `returncode:` Fehlermeldung beim Programmaustritt
(NULL für ordnungsgemäßem Verlauf)

ExitIntr

Library: -36(exec.library)

Deklaration in C: void ExitIntr()

Aufruf in C: ExitIntr();

Aufruf in Assembler: move.l ExecBase,a5
jsr -36(a5)

ExNext

Untersucht das durch lock definierte Verzeichnis und füllt die FileInfoBlock-Struktur (sie wurde bereits durch Examine() initialisiert) mit Informationen über die gefundene Datei. Um alle Dateien und Unterverzeichnisse zu ermitteln, muß die Funktion so oft aufgerufen werden, bis der Rückgabewert FALSE ist.

Library: -108(dos.library)

Deklaration in C:

```
BOOL ExNext(lock, fileinfoblock)
    struct FileLock *lock;
    struct FileInfoBlock *fileinfoblock;
```

Aufruf in C: success = ExNext(lock, fileinfoblock);

Aufruf in Assembler:

```
move.l lock,d1
move.l fileinfoblock,d2
move.l DosBase,a5
jsr -108(a5)
```

Parameter:

lock:	Zeiger auf Lock-Struktur des untersuchten Verzeichnisses
fileinfoblock:	Adresse der FileInfoBlock-Struktur, die mit den ermittelten Informationen gefüllt wird (Adresse muß Longword-aligned sein)
success (d0):	TRUE, falls noch weitere Dateien vorhanden sind, sonst FALSE

FattenLayerInfo

Library: -156(layers.library)

Deklaration in C:

```
void FattenLayerInfo(li)
    struct Layer_Info *li;
```

Aufruf in C:

```
FattenLayerInfo(li);
```

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a5
jsr -156(a5)
```

Parameter: li: Zeiger auf eine Layer_Info-Struktur

Anmerkung: Wie ThinLayerInfo() stammt auch diese Funktion aus der Amiga-Steinzeit (Kickstart Version 1.0) und sollte daher nicht mehr verwendet werden.

FindConfigDev

Durchsucht die ConfigDev-Liste nach einer Struktur mit den angegebenen Parametern.

Library: -72(expansion.library)

Deklaration in C:

```
struct ConfigDev *FindConfigDev
(oldconfigdev, manufacturer, product)
    struct ConfigDev *oldconfigdev;
    long manufacturer, product;
```

Aufruf in C:

```
configdev = FindConfigDev
(oldconfigdev, manufacturer, product);
```

Aufruf in Assembler:

```
move.l oldconfigdev, a0
move.l manufacturer, d0
move.l product, d1
move.l ExpansionBase, a5
jsr -72(a5)
```

Parameter:

oldconfigdev:	Zeiger auf ConfigDev-Struktur ab der in der Liste gesucht werden soll / NULL für den Listenanfang
manufacturer:	Identifikationsnummer des Herstellers / falls nicht relevant: -1
product:	Produkt-ID / -1, wenn nicht relevant
configdev (d0):	Adresse der gefundenen ConfigDev-Struktur / NULL bei nicht erfolgreicher Suche

FindName

Durchsucht eine Liste nach der angegebenen Node-Struktur und ermittelt ihre Adresse.

Library: -276(exec.library)

Deklaration in C:

```
struct Node *FindName(list, name)
    struct List * list;
    char *name;
```

Aufruf in C:

```
node = FindName(list, name);
```

Aufruf in Assembler:

```
move.l list,a0
move.l name,a1
move.l ExecBase,a5
jsr -276(a5)
```

Parameter:	list:	Zeiger auf List-Struktur der untersuchten Liste
	name:	Bezeichnung der gesuchten Node-Struktur
	node (d0):	Adresse der gefundenen Node-Struktur oder NULL

FindPort

Durchsucht die PortList des Systems nach einem bestimmten Message-Port.

Library: -390(exec.library)

Deklaration in C:

```
struct MsgPort *FindPort(name)
char *name;
```

Aufruf in C:

```
port = FindPort(name);
```

Aufruf in Assembler:

```
move.l name,a1
move.l ExecBase,a5
jsr -390(a5)
```

Parameter:

name:	Bezeichnung des gesuchten Ports (Zeiger auf nullterminierten String)
port (d0):	Adresse der gefundenen Port-Struktur oder NULL, falls unauffindbar

FindResident

Durchsucht die Resident-Strukturen nach einem Modul mit der Bezeichnung `name`.

Library: -96(exec.library)

Deklaration in C:

```
struct Resident *FindResident(name)
    char *name;
```

Aufruf in C:

```
resident = FindResident(name);
```

Aufruf in Assembler:

```
move.l name,a1
move.l ExecBase,a5
jsr -96(a5)
```

Parameter:

<code>name:</code>	Bezeichnung des gesuchten Resident-Moduls (Zeiger auf nullterminierten String)
<code>resident:</code>	Adresse der gefundenen Resident-Struktur oder NULL bei Mißerfolg

FindSemaphore

Sucht nach der spezifizierten Semaphore und liefert einen Zeiger auf ihre Struktur.

Library: -594(exec.library)

Deklaration in C:

```
struct SignalSemaphore *FindSemaphore(sigsem)
    struct SignalSemaphore
    *sigsem;
```

Aufruf in C: semaphore = FindSemaphore(sigsem);

Aufruf in Assembler:

```
move.l sigsem,a1
move.l ExecBase,a5
jsr -594(a5)
```

Parameter:

sigsem:	Zeiger auf eine SignalSemaphore-Struktur
semaphore:	Adresse der gefundenen Semaphore oder NULL

FindTask

Durchsucht die Task-Listen nach einem Task mit der Bezeichnung name und gibt einen Zeiger auf seine Task-Struktur zurück.

Library: -294(exec.library)

Deklaration in C:

```
struct Task *FindTask(name)
char *name;
```

Aufruf in C:

```
task = FindTask(name);
```

Aufruf in Assembler:

```
move.l name,a1
move.l ExecBase,a5
jsr -294(a5)
```

Parameter:

name:	Bezeichnung des gesuchten Tasks
task:	Zeiger auf Task-Struktur oder NULL, wenn nicht gefunden

FindToolType

Sucht im ToolType-Feld nach dem Eintrag mit der Bezeichnung name.

Library: -96(icon.library)

Deklaration in C:

```
UBYTE FindToolType(tooltypearray, name)
UBYTE **tooltypearray;
UBYTE *name;
```

Aufruf in C: value = FindToolType(tooltypearray, name);

Aufruf in Assembler:

```
move.l tooltypearray,a0
move.l name,a1
move.l IconBase,a5
jsr -96(a5)
```

Parameter:

tooltypearray:	Adresse des ToolType-Feldes
name:	Bezeichnung des gesuchten Eintrags
value (d0):	Zeiger auf Stringwert

Flood

Flood-Fill-Operation.

Library: -330(graphics.library)

Deklaration in C:

```
void Flood(rp, mode, x, y)
    struct RastPort *rp;
    ULONG mode;
    short x,y;
```

Aufruf in C: Flood(rp, mode, x, y);

Aufruf in Assembler:

```
move.l rp,a1
move.l #mode,d2
move.w #x,d0
move.w #y,d1
move.l GfxBase,a5
jsr -330(a5)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur
mode:	es sind zwei Modi für die Füll-Operation vorgesehen:
	mode = 0
	aktuelles Füllmuster, aktuelle Farbwahl des AOLPens
	mode = 1
	Farbe des angegebenen Punktes wird als Füllfarbe verwendet
x,y:	Koordinaten eines beliebigen Punktes innerhalb der zu füllenden Fläche

FlushCList

Löscht die spezifizierte Character-Liste.

Library: -48(clist.library)

Deklaration in C:

```
void FlushCList(clist)
    LONG clist;
```

Aufruf in C: `FlushCList(clist);`

Aufruf in Assembler:

```
move.l #clist,a0
move.l CListBase,a5
jsr -48(a5)
```

Parameter: `clist:` Identifikations-Header der zu löschenden CListe

Forbid

Verhindert das Task-Switching des Dispatchers/Schedulers und friert dadurch den Multitaskingbetrieb ein. Dieser Zustand bleibt so lange erhalten, bis der Dispatcher durch Aufruf von Permit() reaktiviert wird.

Library: -132(exec.library)

Deklaration in C: void Forbid()

Aufruf in C: Forbid();

Aufruf in Assembler: move.l ExecBase,a5
jsr -132(a5)

FreeBoardMem

Gibt den mit AllocBoardMem() belegten Speicher wieder frei.

Library: -78(expansion.library)

Deklaration in C:

```
void FreeBoardMem(startslot, slotspec)
    long startslot, slotspec;
```

Aufruf in C: `FreeBoardMem(startslot, slotspec);`

Aufruf in Assembler:

```
move.l #startslot,d0
move.l #slotspec,d1
move.l ExpansionBase,a5
jsr -78(a5)
```

Parameter:

startslot:	Slotnummer
slotspec:	Speicherfeld mit den Slot-Spezifikationen

FreeCList

Löscht CList-Header.

Library: -42(clist.library)

Deklaration in C: void FreeCList (clist)
 long clist;

Aufruf in C: FreeCList (clist);

Aufruf in Assembler: move.l #clist,a0
 move.l CListBase,a5
 jsr -42(a5)

Parameter: clist: Zeiger auf die zu löschende CList-Identifikation

FreeColorMap

Gibt den durch GetColorMap() belegten Speicher für eine Farbtabelle wieder frei.

Library: -576(graphics.library)

Deklaration in C:

```
void FreeColorMap(cm)
    struct ColorMap *cm;
```

Aufruf in C: `FreeColorMap(cm);`

Aufruf in Assembler:

```
move.l cm,a0
move.l GfxBase,a5
jsr -576(a5)
```

Parameter: cm: zeigt auf eine ColorMap-Struktur

FreeConfigDev

Gibt den von der angegebenen ConfigDev-Struktur belegten Speicher frei.

Library: -84(expansion.library)

Deklaration in C:

```
void FreeConfigDev(configdev)
    struct ConfigDev *configdev;
```

Aufruf in C: FreeConfigDev(configdev);

Aufruf in Assembler:

```
move.l configdev,a0
move.l ExpansionBase,a5
jsr -84(a5)
```

Parameter: configdev: Zeiger auf eine ConfigDev-Struktur

FreeCopList

Gibt den Speicher einer durch MakeVPort() erzeugten Teil-Copper-Liste wieder frei.

Library: -546(graphics.library)

Deklaration in C:

```
void FreeCopList(coplist)
    struct CopList *coplist
```

Aufruf in C: `FreeCopList(coplist);`

Aufruf in Assembler:

```
move.l coplist,a0
move.l GfxBase,a5
jsr -546(a5)
```

Parameter: `coplist:` Zeiger auf freizugebende CopList-Struktur

FreeCprList

Dellokiert die spezifizierte Hardware-Copper-Liste (direkt programmierte Copper-Befehlsliste).

library: -564(graphics.library)

Deklaration in C: void FreeCprList(cprlist)
 struct CprList *cprlist;

Aufruf in C: FreeCprList(cprlist);

Aufruf in Assembler: move.l cprlist,a0
 move.l GfxBase,a5
 jsr -564(a5)

Parameter: cprlist: Zeiger auf CprList-Struktur einer Hardware-Copper-Liste

FreeDiskObject

Gibt den gesamten Speicher frei, der von einem Workbench-Disk-Objektes belegt wurde (Funktion baut auf FreeFreeList() auf).

Library: -90(icon.library)

Deklaration in C:

```
void FreeDiskObject(diskobj)
    struct DiskObject *diskobj;
```

Aufruf in C: `FreeDiskObject(diskobj);`

Aufruf in Assembler:

```
move.l diskobj, a0
move.l IconBase, a5
jsr -90(a5)
```

Parameter: `diskobj:` zeigt auf DiskObject-Struktur des zu löschenden Objektes

FreeEntry

Löscht sämtliche Einträge der durch AllocEntry angelegten memList-Struktur.

Library: -228(exec.library)

Deklaration in C:

```
void FreeEntry(entry)
    struct memList *entry;
```

Aufruf in C: FreeEntry(entry);

Aufruf in Assembler:

```
move.l entry, a0
move.l ExecBase, a5
jsr -228(a5)
```

Parameter: entry: Zeiger auf zuvor allokierte memList-Struktur

FreeExpansionMem

Gibt den von AllocExpansionMem() belegten Speicher wieder frei.

Library: -90(expansion.library)

Deklaration in C:

```
void FreeExpansionMem(startslot, numslots)
    long startslot, numslot;
```

Aufruf in C: `FreeExpansionMem(startslot, numslots);`

Aufruf in Assembler:

```
move.l #startslot,d0
move.l #numslot,d1
move.l ExpansionBase,a5
jsr -90(a5)
```

Parameter:	startslot:	Nummer des ersten Slots
	numslot:	Anzahl der belegten Slots

FreeFreeList

Gibt den Speicher von einer Free-Liste verwalteten Speicher und die Liste selbst wieder frei.

Library: -54(icon.library)

Deklaration in C:

```
void FreeFreeList (freelist)
    struct FreeList *freelist;
```

Aufruf in C: FreeFreeList (freelist);

Aufruf in Assembler:

```
move.l freelist,a0
move.l IconBase,a5
jsr -54(a5)
```

Parameter: freelist: Adresse der FreeList-Struktur

FreeGBuffers

Gibt den gesamten Speicher eines GEL-Puffers frei, der vorher durch die Funktion GetGBuffer belegt wurde.

Library: -600(graphics.library)

Deklaration in C:

```
void FreeGBuffers(animob, rp, buffer)
    struct AnimOb *animob;
    struct RastPort *rp;
    BOOL buffer;
```

Aufruf in C: FreeGBuffers(animob, rp, buffer);

Aufruf in Assembler:

```
move.l animob,a0
move.l rp,a1
move.b #buffer,d0
move.l GfxBase,a5
jsr -600(a5)
```

Parameter:

animob:	Zeiger auf eine AnimOb-Struktur, deren Elemente dellokiert werden sollen (jeweils für jede Komponente und jede Sequenz): SaveBuffer BorderLine CollMask/ImageShadow
rp:	Adresse der zugehörigen RastPort-Struktur
buffer:	Double-Buffer-Flag. TRUE => DBufPacket und BufBuffer werden automatisch freigegeben

FreeMem

Gibt den von AllocMem() belegten Speicherbereich frei.

Library: -210(exec.library)

Deklaration in C:

```
void FreeMem(memory, size)
    ULONG *memory;
    ULONG size;
```

Aufruf in C: FreeMem(memory, size);

Aufruf in Assembler:

```
move.l memory,a1
move.l #size,d0
move.l ExecBase,a5
jsr -210(a5)
```

Parameter:

memory:	Zeiger auf einen Speicherbereich
size:	Größe dieses Speichers in Byte

FreePotBits

Gegenstück zu AllocPotBits().

Library: -12(potgo.library)

Deklaration in C: `void FreePotBits(bits)
long bits;`

Aufruf in C: `FreePotBits(bits);`

Aufruf in Assembler: `move.l #bits,d0
move.l PotgoBase,a6
jsr -12(a6)`

Parameter: bits: Registerbits, die gelöscht und freigegeben werden sollen

FreeRaster

Gegenstück zu AllocRaster().

Library: -498(graphics.library)

Deklaration in C:

```
void FreeRaster(planeptr, width, height)
    PLANEPTR planeptr;
    short width, height;
```

Aufruf in C: FreeRaster(planeptr, width, height);

Aufruf in Assembler:

```
move.l planeptr,a0
move.w #width,d0
move.w #height,d1
move.l GfxBase,a5
jsr -498(a5)
```

Parameter:

planeptr:	Zeiger auf den freizugebenden Speicherbereich (Raster beliebiger Definition)
width:	Pixelbreite des Rasters
height:	Zeilenhöhe

FreeRemember

Gibt den von der Remember-Struktur verwalteten Speicher wieder frei.

Library: -408(intuition.library)

Deklaration in C:

```
void FreeRemember(rememberkey, reallyforget)
    struct Remeber *rememberkey;
    BOOL reallyforget;
```

Aufruf in C: FreeRemember(rememberkey, reallyforget);

Aufruf in Assembler:

```
move.l rememberkey,a0
move.b #reallyforget,d0
move.l IntuitionBase,a5
jsr -408(a5)
```

Parameter:

remeberkey:	zeigt auf eine Remember-Struktur (für Definition siehe AllocRemember)
reallyforget:	FALSE => es werden nur die verketteten Knoten der Remember-Struktur gelöscht TRUE => der gesamte, von Remember verwaltete Speicher wird an das System zurückgegeben

FreeSignal

Macht die Signalbelegung von AllocSignal rückgängig.

Library: -336(exec.library)

Deklaration in C:

```
void FreeSignal(signalnum)
    long signalnum;
```

Aufruf in C:

```
FreeSignal(signalnum);
```

Aufruf in Assembler:

```
move.l #signalnum,d0
move.l ExecBase,a5
jsr -336(a5)
```

Parameter: signalnum: Nummer des betroffenen Signals

Anmerkung: AllocSignal() und FreeSignal() dürfen nicht in der Handling-Phase einer Exception ausgeführt werden.

FreeSprite

Gibt das zuvor belegte Hardware-Sprite an das System zurück.

Library: -414(graphics.library)

Deklaration in C: `void FreeSprite (num)
short num;`

Aufruf in C: `FreeSprite (num) ;`

Aufruf in Assembler: `move.w #num,d0
move.l GfxBase,a5
jsr -414(a5)`

Parameter: num: Nummer des freizugebenden Hardware-Sprites

FreeSysRequest

Dellokiert die von der BuildSysRequest()-Funktion belegten Speicherbereiche.

Library: -372(intuition.library)

Deklaration in C:

```
void FreeSysRequest(window)
    struct Window *window;
```

Aufruf in C:

```
FreeSysRequest(window);
```

Aufruf in Assembler:

```
move.l window, a0
move.l IntuitionBase, a5
jsr -372(a5)
```

Parameter: window: Zeiger auf Window-Struktur des Fensters, dem der Requester zugeordnet ist

FreeTrap

Freigabe eines System-Traps.

Library: -348(exec.library)

Deklaration in C: `void FreeTrap(trapnum)
short trapnum;`

Aufruf in C: `FreeTrap(trapnum);`

Aufruf in Assembler: `move.w #trapnum,d0
move.l ExecBase,a5
jsr -348(a5)`

Parameter: trapnum: Nummer des Traps, der freigegeben werden soll

Anmerkung: Dieser Aufruf muß vom selben Task kommen, der den Trap auch mit AllocTrap() belegt hat. Sowohl AllocTrap() als auch FreeTrap() dürfen nicht während des Exception-Handlings ausgeführt werden.

FreeVPortCopLists

Dellokiert sämtliche von MakeVPort() erzeugten Teil-Copper-Listen.

Library: -540(graphics.library)

Deklaration in C: void FreeVPortCopLists(vp)
 struct ViewPort *vp;

Aufruf in C: FreeVPortCopLists(vp);

Aufruf in Assembler: move.l vp,a0
 move.l GfxBase,a5
 jsr -540(a5)

Parameter: vp: Zeiger auf eine ViewPort-Struktur

FreeWBOject

Dellokiert Struktur und FreeList-Einträge des angegebenen Workbench-Objekts und gibt den belegten Speicher an das System zurück.

Library: -60(icon.library)

Deklaration in C:

```
void FreeWBOject(wbobject)
    struct WBOject *wbobject;
```

Aufruf in C: `FreeWBOject(wbobject);`

Aufruf in Assembler:

```
move.l wbobject,a0
move.l IconBase,a5
jsr -60(a5)
```

Parameter: wbobject: Zeiger auf WBOject-Struktur

GetCC

Ermittelt den Zustand des CCR-Register ohne den privilegierten Befehl MOVE sr,<ea> zu verwenden. Die Funktion ist damit kompatibel zu allen Prozessoren der 680xx-Linie.

Library: -528(exec.library)

Deklaration in C: long GetCC()

Aufruf in C: conditions = GetCC();

Aufruf in Assembler: move.l ExecBase,a5
jsr -528(a5)

Parameter: conditions (d0): Inhalt des Condition-Code-Registers

GetCLBuf

Schreibt die durch clist spezifizierte Character-Liste in einen Puffer und löscht die übertragenen Daten in der CListe. Soll die ursprüngliche CListe erhalten bleiben, empfiehlt es sich, die Funktion SubCL() zu verwenden.

Library: -114(clist.library)

Deklaration in C:

```
long GetCLBuf(clist, buffer, length)
long clist;
UBYTE *buffer;
long length;
```

Aufruf in C: `actualelength = GetCLBuf(clist, buffer, length);`

Aufruf in Assembler:

```
move.l #clist,a0
move.l buffer,a1
move.l #length
move.l CListBase,a5
jsr -114(a5)
```

Parameter:	clist:	Header der zu konvertierenden Character-Liste
	buffer:	Zeiger auf Konvertierungspuffer
	length:	Puffergröße in Byte
	actualelength (d0):	Anzahl der effektiv kopierten DatenByte

GetCLChar

Schneidet ein Byte am Anfang der CListe ab und liefert es als Rückgabewert.

Library: -66(clist.library)

Deklaration in C: byte GetCLChar(clist)
 long clist;

Aufruf in C: byte = GetCLChar(clist);

Aufruf in Assembler: move.l #clist,a0
 move.l CListBase,a5
 jsr -66(a5)

Parameter: clist: Header der betroffenen CListe
 byte (d0): abgeschnittenes Byte oder -1 falls Liste leer

GetCLWord

Schneidet ein Wort (2 Byte) vom Anfang der CListe ab und übergibt es an den aufrufenden Task.

Library: -90(clist.library)

Deklaration in C:

```
short GetCLWord(clist)
        ULONG clist;
```

Aufruf in C:

```
word = GetCLWord(clist);
```

Aufruf in Assembler:

```
move.l #clist,a0
move.l CListBase,a5
jsr -90(a5)
```

Parameter:

clist:	Identifikationsnummer der CListe
word (d0):	abgeschnittenes Wort oder -1 bei leerer Liste (bzw. nur einem geholten Byte)

GetColorMap

Initialisiert eine Farbtabelle mit der angegebenen Anzahl von Einträgen, beschafft den benötigten Speicher und übergibt einen Zeiger auf die ColorMap-Struktur.

Library: -570(graphics.library)

Deklaration in C:

```
struct ColorMap *GetColorMap(entries)
    long entries;
```

Aufruf in C:

```
cm = GetColorMap(entries);
```

Aufruf in Assembler:

```
move.w #entries,d0
move.l GfxBase,a5
jsr -570(a5)
```

Parameter:

entries:	Anzahl der Einträge in der ColorMap
cm (d0):	Zeiger auf initialisierte ColorMap-Struktur oder NULL, falls keine Speicher vorhanden

GetCurrentBinding

Holt den Konfigurationsbereich einer Erweiterungskarte.

Library: -138(expansion.library)

Deklaration in C:

```
long GetCurrentBinding(cb, size)
    struct CurrentBinding *cb;
    short size;
```

Aufruf in C: `actuallegth = GetCurrentBinding(cb, size);`

Aufruf in Assembler:

```
move.l cb,a0
move.w #size,d0
move.l ExpansionBase,a5
jsr -138(a5)
```

Parameter:

<code>cb:</code>	Zeiger auf eine CurrentBinding-Struktur
<code>size:</code>	Größe in Byte
<code>actualelength (d0):</code>	ermittelte Größe der CurrentBinding-Struktur

GetDefPrefs

Überträgt die Standard-Spezifikationen (Default) der Preference-Struktur in den angegebenen Puffer.

Library: -126(intuition.library)

Deklaration in C:

```
struct Preferences *GetDefPrefs
(preferences, size)

    struct Preferences
    *preferences;
    short size;
```

Aufruf in C: prefs = GetDefPrefs(preferences, size);

Aufruf in Assembler:

```
move.l preferences,a0
move.w #size,d0
move.l IntuitionBase,a5
jsr -126(a5)
```

Parameter:

preference:	Zeiger auf Datenpuffer (üblicherweise in der Größe der Preference-Struktur)
size:	Puffergröße (sizeof(struct Preferences))
prefs (d0):	Adresse des effektiv belegten Puffers (normalerweise mit dem übergebenen Zeiger identisch)

GetDiskObject

Erweiterte Version von GetIcon(). Holt Speicher für eine DiskObject-Struktur, liest die Parameter der angegebenen Datei ein und erweitert den Dateinamen durch das Anhängsel ".info".

Library: -78(icon.library)

Deklaration in C:

```
struct DiskObject *GetDiskObject(name)
    UBYTE* name;
```

Aufruf in C:

```
object = GetDiskObject(name);
```

Aufruf in Assembler:

```
move.l name,a0
move.l IconBase,a5
jsr -78(a5)
```

Parameter:

name:	Zeiger auf nullterminierten String
object (d0):	Adresse der belegten DiskObject-Struktur oder NULL

GetGBuffers

Beschafft Speicher für sämtliche Komponenten der AnimOb-Struktur.

Library: -168(graphics.library)

Deklaration in C:

```

BOOL GetGBuffers(animobj, rp, buffer)
    struct AnimOb *animobj;
    struct RastPort *rp;
    BOOL *buffer;

```

Aufruf in C: status = GetGBuffers(animobj, rp, buffer);

Aufruf in Assembler:

```

move.l animobj,a0
move.l rp,a1
move.b buffer,d0
move.l GfxBase,a5
jsr -168(a5)

```

Parameter:

animobj:	Zeiger auf AnimOb-Struktur, die sämtliche Informationen über die benötigten Speicherbereiche enthält (SaveBuffer, BorderLine CollMask usw.)
rp:	Zeiger auf RastPort-Struktur
buffer:	Double-Buffer-Flag. TRUE: es wird zusätzlicher Speicher für DBufPacket und BufBuffer benötigt
status (d0):	TRUE bei erfolgreicher Allokierung

GetIcon

Liest die DiskObject-Struktur der angegebenen Datei. Der benötigte Speicher wird automatisch beschafft und in der FreeList-Struktur eingetragen.

Library: -42(icon.library)

Deklaration in C:

```
long GetIcon(name, icon, freelist)
    UBYTE *name;
    struct DiskObject *icon;
    struct FreeList *freelist;
```

Aufruf in C: status = GetIcon(name, icon, freelist);

Aufruf in Assembler:

```
move.l name,a0
move.l icon,a1
move.l freelist,a2
move.l IconBase,a5
jsr -42(a5)
```

Parameter:

name:	Zeiger auf Bezeichnungsstring des gesuchten Objektes (wird durch ".info" erweitert)
icon:	Adresse der zugehörigen DiskObject-Struktur
freelist:	Zeiger auf FreeList
status (d0):	!= 0, wenn erfolgreich

GetMsg

Koppelt die erste Nachricht aus der Liste des angegebenen Port aus und übergibt die Adresse ihrer Message-Struktur. Die Funktion wartet nicht, sondern kehrt auch bei leerem Port sofort zurück.

Library: -372(exec.library)

Deklaration in C:

```
struct Message *GetMsg(port)
    struct MsgPort *port;
```

Aufruf in C:

```
message = GetMsg(port);
```

Aufruf in Assembler:

```
move.l port, a0
move.l ExecBase, a5
jsr -372(a5)
```

Parameter:

port:	Zeiger auf den zu untersuchen Message-Port
message (d0):	Adresse der Message-Struktur der empfangenen Nachricht oder NULL, falls Port leer

GetPacket

Die Funktion versucht ein DOS-Paket abzuholen.

Library: -162(dos.library)

Deklaration in C: `BOOL GetPacket(wait)`
`BOOL wait;`

Aufruf in C: `ok = GetPacket(wait);`

Aufruf in Assembler: `move.b #wait,d1`
`move.l DosBase,a5`
`jsr -162(a5)`

Parameter:

<code>wait:</code>	<code>TRUE:</code> GetPacket() wartet so lange, bis ein Packet ankommt
<code>FALSE:</code>	ist noch kein Packet eingetroffen, sofort zurück
<code>ok (d0):</code>	übergibt FALSE bei erfolgloser Rückkehr

GetPrefs

Holt die aktuellen Parameter der Preference-Struktur und legt sie in einem Puffer ab.

Library: -132(intuition.library)

Deklaration in C:

```
struct Preferences *GetPrefs(preferences, size)
    struct Preferences
    *preferences;
    short size;
```

Aufruf in C:

```
prefs = GetPrefs(preferences, size);
```

Aufruf in Assembler:

```
move.l preferences, a0
move.w #size, a1
move.l IntuitionBase, a5
jsr -132(a5)
```

Parameter:

preferences:	Zeiger auf zuvor erstellten Puffer für die Aufnahme der ermittelten Daten
size:	Größe dieses Puffers
prefs (d0):	Zeiger auf effektiv belegten Pufferbereich (entspricht im Normalfall der übergebenen Pufferadresse)

GetRGB4

Ermittelt den Farbwert eines Eintrags in der ColorMap und gibt ihn als Wort (short) zurück.

Library: -582(graphics.library)

Deklaration in C:

```
short GetRGB4(cm, entry)
    struct ColorMap *cm;
    long entry;
```

Aufruf in C: color = GetRGB4(cm, entry);

Aufruf in Assembler:

```
move.l colormap,a0
move.l #entry,d0
move.l GfxBase,a5
jsr -582(a5)
```

Parameter:

colormap:	Zeiger auf eine ColorMap-Struktur
entry:	Nummer des Farbeintrags, dessen RGB-Wert gesucht wird
color (d0):	ermittelter RGB-Farbwert

GetScreenData

Füllt den angegebenen Puffer mit den Einträgen der untersuchten Screen-Struktur.

Library: -426(intuition.library)

Deklaration in C:

```

BOOL GetScreenData(buffer, size, type, screen)
    CPTR *buffer;
    USHORT size, type;
    struct Screen *screen;
  
```

Aufruf in C:

```

success = GetScreenData
    (buffer, size, type, screen)
  
```

Aufruf in Assembler:

```

move.l buffer,a0
move.w #size,d0
move.w #type,d1
move.l screen,a1
move.l IntuitionBase,a5
jsr -426(a5)
  
```

Parameter:	buffer:	Zeiger auf Datenpuffer
	size:	Puffergröße in Byte
	type:	Screen-Typ (WBENCHSCREEN/CUSTOMSCREEN)
	screen:	Zeiger auf zu untersuchende Screen-Struktur
	success (d0):	TRUE bei erfolgreicher Operation

Anmerkung: Falls die Strukturdaten eines Screens abgerufen werden, der noch nicht geöffnet ist, wird er von der Funktion automatisch dargestellt.

GetSprite

Die Funktion versucht, das angegebene Hardware-Sprite zu allokkieren.

Library: -408(graphics.library)

Deklaration in C:

```
short GetSprite(simplesprite, num)
    struct SimpleSprite *simplesprite;
    short num;
```

Aufruf in C: spritenum = GetSprite(simplesprite, num);

Aufruf in Assembler:

```
move.l simplesprite,a0
move.w #num,d0
move.l GfxBase,a5
jsr -408(a5)
```

Parameter:

simplesprite:	Adresse der zugehörigen SimpleSprite-Struktur
num:	Nummer des Hardware-Sprites
spritenum (d0):	Ordnungsnummer des effektiv belegten Sprites oder -1, falls kein Sprite mehr frei ist

Anmerkung: Wenn keine Sprites mehr verfügbar sind, wird in simplesprite->num ebenfalls der Wert -1 eingetragen.

GetWBOject

Liest eine WBOject-Struktur von der Diskette und hängt das Postfix ".info" an den Dateinamen.

Library: -30(icon.library)

Deklaration in C:

```
struct WBOject *GetWBOject (name)
    UBYTE *name;
```

Aufruf in C:

```
object = GetWBOject (name);
```

Aufruf in Assembler:

```
move.l name, a0
move.l IconBase, a5
jsr -30(a5)
```

Parameter:

name:	Bezeichnung des Workbench-Objektes
object (d0):	Zeiger auf die gelesene WBOject-Struktur oder NULL

IEEEDPAbs

Liefert den Betrag einer Fließkommazahl.

Library: -54(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPAbs (num)
double num;
```

Aufruf in C: result = IEEEDPAbs (num);

Aufruf in Assembler:

```
movem.l #num,d0/d1
move.l MathIeeeDoubBasBase,a5
jsr -54(a5)
```

Parameter:

num:	doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	Betrag dieser Zahl (Absolutwert)

Anmerkung: Zustand des Condition-Code-Registers nach Funktionsaufruf:

Flag	Zustand	Bedingung
N	0	result = 0
Z	1	
V	0	
C	-	
X	-	

IEEEDPAdd

Addiert zwei Fließkommazahlen.

Library: -66(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPAdd(num1, num2)
    double num1, num2;
```

Aufruf in C:

```
result = IEEEDPAdd(num1, num2);
```

Aufruf in Assembler:

```
movem.l num1,d0/d1
movem.l num2,d2/d3
move.l MathIeeeDoubBasBase,a5
jsr -66(a5)
```

Parameter:

num1,num2:	doppelt-genaue Fließkommazahlen im IEEE-Format
result (d0/d1):	Summe der Zahlen

Anmerkung: Zustand des Condition-Code-Registers nach Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result < 0
Z	1	result = 0
V	1	Überlauf
C	—	
X	—	

IEEEDPAsin

Berechnet arcus sinus.

Library: -114(mathieeedoubtrans.library)

Deklaration in C: double IEEEDPAsin(num)
 double num;

Aufruf in C: result = IEEEDPAbs(num);

Aufruf in Assembler: movem.l num,d0/d1
 move.l MathIeeeDoubTransBase,a5
 jsr -114(a5)

Parameter: num: doppelt-genaue Fließkommazahl im IEEE-
 Format
 result (d0/d1): arcus sinus

IEEEDPATan

Berechnet arcus tangens.

Library: -30(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPATan(num)
double num;
```

Aufruf in C:

```
result = IEEEDPATan(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -30(a5)
```

Parameter:

```
num:
```

 doppelt-genaue Fließkommazahl im IEEE-Format

```
result (d0/d1):
```

 arcus tangens von num

IEEEDPCeil

Liefert obere Schranke zu num (kleinste Zahl, die größer oder gleich num ist).

Library: -96(mathieeedoubbas.library)

Deklaration in C: `double IEEEDPCeil(num)`
 `double num;`

Aufruf in C: `result = IEEEDPCeil(num);`

Aufruf in Assembler: `movem.l num,d0/d1`
 `move.l MathIeeeDoubBasBase,a5`
 `jsr -96(a5)`

Parameter: `num:` doppelt-genaue Fließkommazahl im IEEE-Format
 `result (d0/d1):` Näherung von rechts

IEEEDPCmp

Vergleicht zwei Fließkommazahlen.

Library: -42(mathieeedoubbas.library)

Deklaration in C: `double IEEEDPCmp(num1, num2)`
`double num1, num2;`

Aufruf in C: `result = IEEEDPCmp(num);`

Aufruf in Assembler: `movem.l num1,d0/d1`
`movem.l num2,d2/d3`
`move.l MathIeeeDoubBasBase,a5`
`jsr -42(a5)`

Parameter: `num1,num2:` zwei doppelt-genaue Fließkommazahlen im IEEE-Format
`result (d0):` je nach Ergebnis des Vergleichs:

`num1 > num2 => result = +1`

`num1 = num2 => result = 0`

`num1 < num2 => result = -1`

IEEEDPCos

Berechnet cosinus.

Library: -42(mathieeedoubtrans.library)

Deklaration in C: double IEEEDPCos(num)
double num;

Aufruf in C: result = IEEEDPCos(num);

Aufruf in Assembler: movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -42(a5)

Parameter: num: doppelt-genaue Fließkommazahl im IEEE-
Format
result (d0/d1): cosinus von num

IEEEDPCosh

Liefert cosinus hyperbolicus von num.

Library: -66(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPCosh(num)
double num;
```

Aufruf in C:

```
result = IEEEDPCosh(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -66(a5)
```

Parameter:

num:	doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	cosinus hyperbolicus von num

IEEEDPDiv

Dividiert zwei Fließkommazahlen.

Library: -84(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPDiv(num1, num2)
    double num1, num2;
```

Aufruf in C:

```
result = IEEEDPDiv(num1, num2);
```

Aufruf in Assembler:

```
movem.l num1,d0/d1
movem.l num2,d2/d3
move.l MathIeeeDoubBasBase,a5
jsr -84(a5)
```

Parameter:

num1,num2: zwei doppelt-genaue Fließkommazahlen im IEEE-Format

result (d0/d1): Ergebnis der Division

Anmerkung: Zustand des CCR-Registers nach dem Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result < 0
Z	1	result = 0
V	1	Überlauf
C	-	
X	-	

IEEEDPExp

Berechnet die x-te Potenz von e.

Library: -78(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPExp(num)
double num;
```

Aufruf in C: `result = IEEEDPExp(num);`

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -78(a5)
```

Parameter:
num: doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1): Exponentialwert

IEEEDPFieee

Wandelt eine einfach-genaue IEEE-Fließkommazahl in eine doppel-genaue um.

Library: -108(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPFieee (num)
    float num;
```

Aufruf in C:

```
result = IEEEDPFieee (num);
```

Aufruf in Assembler:

```
movem.l num,d0
move.l MathIeeeDoubTransBase,a5
jsr -108(a5)
```

Parameter:

num:	einfach-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	doppelt-genaues Format

IEEEDPFix

Wandelt eine Fließkommazahl ins Integer-Format um.

Library: -30(mathieeedoubbas.library)

Deklaration in C:

```
int IEEEDPFix(num)
double num;
```

Aufruf in C: `result = IEEEDPFix(num);`

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubBasBase,a5
jsr -30(a5)
```

Parameter:
num: doppelt-genaue Fließkommazahl im IEEE-Format
result (d0): Integer-Format

Anmerkung: Zustand der BedingungsCodes nach dem Funktionsaufruf:

Flag	Zustand	Bedingung
N	0	result < 0
Z	1	result = 0
V	0	Überlauf
C	—	
X	—	

IEEEDPFloor

Bildet untere Schranke zu num (größte Zahl, die kleiner oder gleich num ist).

Library: -90(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPFloor(num)
double num;
```

Aufruf in C:

```
result = IEEEDPFloor(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubBasBase,a5
jsr -90(a5)
```

Parameter:

num:	doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	Näherungswert von links

IEEEDPFIt

Konvertierung vom Integer- ins Fließkommaformat.

Library: -36(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPFIt(num)
int num;
```

Aufruf in C:

```
result = IEEEDPFIt(num);
```

Aufruf in Assembler:

```
movem.l num,d0
move.l MathIeeeDoubBasBase,a5
jsr -36(a5)
```

Parameter:
num: Zahl im Integer-Format
result (d0/d1): doppelt-genaues IEEE-Fließkommaformat

Anmerkung: Zustand der BedingungsCodes nach dem Funktionsaufruf:

Flag	Zustand	Bedingung
N	0	result < 0
Z	1	result = 0
V	0	
C	-	
X	-	

IEEEDPLog

Berechnet den natürlichen Logarithmus.

Library: -4(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPLog(num)
double num;
```

Aufruf in C:

```
result = IEEEDPLog(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -84(a5)
```

Parameter:

num:	doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	natürlicher Logarithmus von num

IEEEEDPLog10

Berechnet Logarithmus zur Basis 10.

Library: -126(mathieeedoubtrans.library)

Deklaration in C: `double IEEEEDPLog(num)`
 `double num;`

Aufruf in C: `result = IEEEEDPLog(num);`

Aufruf in Assembler: `movem.l num,d0/d1`
 `move.l MathIeeeDoubTransBase,a5`
 `jsr -126(a5)`

Parameter: `num:` doppelt-genaue Fließkommazahl im IEEE-Format
 `result (d0/d1):` Zehnerlogarithmus

IEEEEDPMul

Multipliziert zwei Fließkommazahlen.

Library: -72(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEEDPMul(num1, num2)
    double num1, num2;
```

Aufruf in C:

```
result = IEEEEDPMul(num1, num2);
```

Aufruf in Assembler:

```
movem.l num1,d0/d1
movem.l num2,d2/d3
move.l MathIeeeDoubBasBase,a5
jsr -72(a5)
```

Parameter:

num1, num2: zwei doppelt-genaue Fließkommazahlen im IEEE-Format

result (d0/d1): Produkt dieser Zahl (Absolutwert)

Anmerkung: Zustand der BedingungsCodes nach dem Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result < 0
Z	1	result = 0
V	1	Überlauf
C	-	
X	-	

IEEEDPNeg

Negation von num (Vorzeichenwechsel).

Library: -60(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPNeg(num)
double num;
```

Aufruf in C:

```
result = IEEEDPNeg(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubBase,a5
jsr -60(a5)
```

Parameter:

num:	doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	Zahl mit umgekehrtem Vorzeichen

Anmerkung: Zustand der BedingungsCodes nach Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result < 0
Z	1	result = 0
V	1	Überlauf
C	—	
X	—	

IEEEDPPow

$\text{num1}^{\text{num2}}$.

Library: -90(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPPow(num2, num1)
double num2, num1;
```

Aufruf in C:

```
result = IEEEDPPow(num2, num1);
```

Aufruf in Assembler:

```
movem.l num1,d2/d3
movem.l num2,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -90(a5)
```

Parameter:

num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	$\text{num1}^{\text{num2}}$

IEEEEDPSin

Berechnet sinus von num.

Library: -36(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEEDPSin(num)
double num;
```

Aufruf in C:

```
result = IEEEEDPSin(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -36(a5)
```

Parameter:

num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	sinus

IEEEDPSincos

Berechnet sinus und cosinus einer Fließkommazahl.

Library: -54(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPSinuscos(buffer,num)
double *buffer, num;
```

Aufruf in C:

```
result = IEEEDPSinuscos(num);
```

Aufruf in Assembler:

```
move.l buffer,a0
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -54(a5)
```

Parameter:

buffer:	Zeiger auf das Ergebnis der Cosinus-Berechnung
num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	sinus dieser Zahl

IEEEDPSinh

Berechnet sinus hyperbolicus.

Library: -60(mathieeedoubtrans.library)

Deklaration in C: `double IEEEDPSinh(num)
double num;`

Aufruf in C: `result = IEEEDPSinh(num);`

Aufruf in Assembler: `movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -60(a5)`

Parameter: `num:` eine doppelt-genaue Fließkommazahl im IEEE-Format
`result (d0/d1):` sinus hyperbolicus von num

IEEEDPSqrt

Liefert Quadratwurzel von num.

Library: -96(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPSqrt (num)
double num;
```

Aufruf in C:

```
result = IEEEDPSqrt (num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -96(a5)
```

Parameter:

num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	Wurzel aus num

IEEEDPSub

num1 – num2.

Library: -72(mathieeedoubbas.library)

Deklaration in C:

```
double IEEEDPSub(num1, num2)
double num1, num2;
```

Aufruf in C:

```
result = IEEEDPSub(num1, num2);
```

Aufruf in Assembler:

```
movem.l num1,d0/d1
movem.l num2,d2/d3
move.l MathIeeeDoubBasBase,a5
jsr -72(a5)
```

Parameter: num1, num2: zwei doppelt-genaue Fließkommazahlen im IEEE-Format
result (d0/d1): ihre Differenz

Anmerkung: Zustand der BedingungsCodes nach Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result < 0
Z	1	result = 0
V	1	result > 0
C	–	
X	–	

IEEEDPTan

Berechnet tangens von num.

Library: -48(mathieeedoubtrans.library)

Deklaration in C:

```
double IEEEDPTan(num)
double num;
```

Aufruf in C:

```
result = IEEEDPTan(num);
```

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -48(a5)
```

Parameter:

num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0/d1):	tangens

IEEDPTanh

Berechnet tangens Hyperbolicus.

Library: -72(mathieeedoubtrans.library)

```
Deklaration in C:    double IEEEEDPTanh (num)
                      double num;
```

Aufruf in C: `result = IEEEEDPTanh(num);`

```
Aufruf in Assembler:  movem.l num,d0/d1
                       move.l MathIeeeDoubTransBase,a5
                       jsr -72(a5)
```

Parameter:	num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
	result (d0/d1):	tangens hyperbolicus

IEEEDPTieee

Wandelt ein doppelt-genaues in ein einfach-genaues IEEE-Fließkommaformat um.

Library: -102(mathieeedoubtrans.library)

Deklaration in C:

```
float IEEEDPTieee(num)
double num;
```

Aufruf in C: result = IEEEDPTieee(num);

Aufruf in Assembler:

```
movem.l num,d0/d1
move.l MathIeeeDoubTransBase,a5
jsr -102(a5)
```

Parameter:

num:	eine doppelt-genaue Fließkommazahl im IEEE-Format
result (d0):	einfach-genaues IEEE-Format

IEEEDPTst

Prüft auf Wert 0.

Library: -48(mathieeedoubbas.library)

Deklaration in C: `long IEEEDPTst (num)
double num;`

Aufruf in C: `result = IEEEDPTst (num);`

Aufruf in Assembler: `movem.l num,d0/d1
move.l MathIeeeDoubBasBase,a5
jsr -48(a5)`

Parameter: `num:` eine doppelt-genaue IEEE-Fließkommazahl
`result (d0):` je nach Ergebnis des Vergleichs:

`num > 0 => result = +1`
`num = 0 => result = 0`
`num < 0 => result = -1`

Anmerkung: Zustand der BedingungsCodes nach dem Funktionsaufruf:

Flag	Zustand	Bedingung
N	1	result = -1
Z	1	result = 0
V	0	
C	-	
X	-	

IncrCLMark

Setzt Markierung der Character-Liste auf das nächste Byte.

Library: -126(clist.library)

Deklaration in C:

```
long IncrCLMark(clist)
long clist;
```

Aufruf in C: error = IncrCLMark(clist);

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a5
jsr -126(a5)
```

Parameter:

clist:	Header der CListe, deren Markierung inkrementiert werden soll
error (d0):	ungleich NULL, falls CListe zuende

Info

Holt Disk-Info und schreibt die Parameter in die InfoData-Struktur, deren Adresse durch den Zeiger parameter übergeben wird.

Library: -114(dos.library)

Deklaration in C:

```
BOOL Info(lock, parameter)
    struct FileLock *lock;
    struct InfoData *parameter;
```

Aufruf in C: success = Info(lock, parameter);

Aufruf in Assembler:

```
move.l lock,d1
move.l parameter,d2
move.l DosBase,a5
jsr -114(a4)
```

Parameter:

lock:	BCPL-Zeiger auf Lock-Struktur (Datei/Verzeichnis der Diskette)
parameter:	Adresse einer InfoData-Struktur
success (d0):	FALSE bei Mißerfolg

InitArea

Initialisiert eine AreaInfo-Struktur für die Verwendung von Area-Funktionen (AreaDraw, AreaMove usw.).

Library: -282(graphics.library)

Deklaration in C:

```
void InitArea
(areainfo, vectorbuffer, vectors)
    struct AreaInfo *areainfo;
    APTR vectortable;
    short vectors;
```

Aufruf in C: InitArea(areainfo, vectortable, vectors);

Aufruf in Assembler:

```
move.l areainfo,a0
move.l vectorbuffer,a1
move.w #vectors,d0
move.l GfxBase,a5
jsr -282(a5)
```

Parameter:

- areainfo: Zeiger auf zu initialisierende AreaInfo-Struktur
- vectorbuffer: Puffer für die Punktvektoren
Mindestgröße: (vectors+1) * 5 Byte
- vectors: maximale Anzahl von Punkten (Vektoren) in der Area

InitBitMap

Initialisiert eine BitMap-Struktur mit den angegebenen Spezifikationen.

Library: -390(graphics.library)

Deklaration in C:

```
void InitBitMap
    (bm, depth, width, height)
    struct BitMap *bm;
    BYTE depth;
    short width, height;
```

Aufruf in C: InitBitMap(bm, depth, width, height);

Aufruf in Assembler:

```
move.l bm,a0
move.b depth,d0
move.w width,d1
move.w height,d2
move.l GfxBase,a5
jsr -390(a5)
```

Parameter:	bm:	Zeiger auf die zu initialisierende BitMap-Struktur
	depth:	Tiefe der BitMap (Anzahl der verwendeten Planes)
	width:	Breite in Pixel
	height:	Höhe in Zeilen

InitCLPool

Initialisiert einen Speicherbereich für die Verwendung als Character-List-Pool, baut die erforderliche Header-Struktur auf und verkettet alle Nodes in einer FreeList.

Library: -30(clist.library)

Deklaration in C:

```
long InitCLPool(poolbuffer, size)
    UBYTE *poolbuffer;
    ULONG size;
```

Aufruf in C: error = InitCLPool(poolbuffer, size);

Aufruf in Assembler:

```
move.l poolbuffer, a0
move.l #size, d0
move.l CListBase, a5
jsr -30(a5)
```

Parameter:

poolbuffer:	Zeiger auf Pool-Speicher
size:	Größe dieses Speicherbereichs in Byte
error (d0):	NULL, falls fehlerfrei

InitCode

Initialisiert sämtliche residenten Code-Module ab der spezifizierten Versionsnummer.

Library: -72(exec.library)

Deklaration in C: void InitCode(startclass, version)
 ULONG startclass, version;

Aufruf in C: InitCode(startclass, version);

Aufruf in Assembler: move.l #startclass,d0
 move.l #version,d1
 move.l ExecBase,a5
 jsr -72(a5)

Parameter:	startclass:	Code-Klasse
	version:	Versionsnummer

InitGels

Bereitet eine GelsInfo-Struktur für die Verwaltung von VSprites und/oder Bobs vor. Die Dummy-Strukturen dienen lediglich zur Initialisierung der GEL-Liste.

Library: -120(graphics.library)

Deklaration in C:

```
void InitGels
(dummyhead, dummytail, gelsinfo)
    struct VSprite *dummyhead,
                                *dummytail;
    struct GelsInfo *gelsinfo;
```

Aufruf in C: InitGels(dummyhead, dummytail, gelsinfo);

Aufruf in Assembler:

```
move.l dummyhead,a0
move.l dummytail,a1
move.l gelsinfo,a2
move.l GfxBase,a5
jsr -120(a5)
```

Parameter:

dummyhead:	Zeiger auf leere VSprite-Struktur (dummy) für Listenkopf
dummytail:	Zeiger auf leere VSprite-Struktur (dummy) für Listenende
gelsinfo:	Adresse der zu initialisierenden GelsInfo-Struktur

InitGMasks

Initialisiert sämtliche Bob-Puffer (CollMask, ImageShadow usw.) eines Animationsobjektes (Speicher muß vorher mit GetGBuffer() beschafft werden).

Library: -174(graphics.library)

Deklaration in C:

```
void InitGMasks (animob)
    struct AnimOb *animob;
```

Aufruf in C: `InitGMasks (animob);`

Aufruf in Assembler:

```
move.l animob, a0
move.l GfxBase, a5
jsr -174(a5)
```

Parameter: animob: Adresse einer AnimOb-Struktur

InitLayers

Überholte Form der Initialisierung einer Layer_Info-Struktur, da ein Aufruf von `FattenLayerInfo()` folgen muß – bequemer geht es mit `NewLayerInfo()`.

Library: `-30(layers.library)`

Deklaration in C:

```
void InitLayers(li)
    struct Layer_Info *li;
```

Aufruf in C: `InitLayers(li);`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a5
jsr -30(a5)
```

Parameter: `li:` Zeiger auf eine Layer_Info-Struktur

InitMasks

Initialisiert VSprite- bzw. Bob-Masken.

Library: -126(graphics.library)

Deklaration in C:

```
void InitMasks(vsprite)
    struct VSprite *vsprite;
```

Aufruf in C: `InitMasks(vsprite);`

Aufruf in Assembler:

```
move.l vsprite,a0
move.l GfxBase,a5
jsr -126(a5)
```

Parameter: vsprite: Zeiger auf eine VSprite-Struktur, die die zu initialisierenden Maskenpuffer enthält (z.B. BorderLine)

InitRastPort

Initialisiert die angegebene RastPort-Struktur mit den Standard-Spezifikationen.

Library: -198(graphics.library)

Deklaration in C:

```
void InitRastPort(rp)
    struct RastPort *rp;
```

Aufruf in C: InitRastPort(rp);

Aufruf in Assembler:

```
move.l rp,a1
move.l GfxBase,a5
jsr -198(a5)
```

Parameter: rp: Adresse einer RastPort-Struktur

Anmerkung: Folgende Default-Werte werden definiert:

```
Mask, FgPen, AOLPen, LinePtrn = -1
DrawMode                      = JAM2
```

InitRequester

Einfache Initialisierungsroutine für Requester-Strukturen, die lediglich sämtliche Einträge auf NULL setzt.

Library: -138(intuition.library)

Deklaration in C:

```
void InitRequester(requester)
    struct Requester *requester;
```

Aufruf in C: InitRequester(requester);

Aufruf in Assembler:

```
move.l requester, a0
move.l IntuitionBase, a5
jsr -138(a5)
```

Parameter: requester: Zeiger auf eine Requester-Struktur

InitResident

Initialisierungsroutine für Resident-Module.

Library: -102(exec.library)

Deklaration in C:

```
void InitResident(resident, seglist)
    ULONG *resident;
    struct List *seglist;
```

Aufruf in C: InitResident(resident, seglist);

Aufruf in Assembler:

```
move.l resident,a0
move.l seglist,d1
move.l ExecBase,a5
jsr -102(a5)
```

Parameter:

resident:	Adresse des zu initialisierenden Resident-Moduls
seglist:	Zeiger auf Segment-Liste

InitSemaphore

Initialisiert eine Semaphore.

Library: -558(exec.library)

Deklaration in C:

```
void InitSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
```

Aufruf in C: `InitSemaphore(sigsem);`

Aufruf in Assembler:

```
move.l sigsem,a0
move.l ExecBase,a5
jsr -558(a5)
```

Parameter: sigsem: Adresse der zu initialisierenden Signal-Semaphore-Struktur

InitStruct

Initialisiert eine Struktur mit den Parametern einer Initialisierungstabelle.

Library: -78(exec.library)

Deklaration in C:

```
void InitStruct
(inittable, memory, size)
    ULONG *inittable, *memory;
    ULONG size;
```

Aufruf in C: InitStruct(inittable, memory, size);

Aufruf in Assembler:

```
move.l inittable,a1
move.l memory,a2
move.l #size,d0
move.l ExecBase,a6
jsr -78(a6)
```

Parameter:	inittable:	Parametertabelle für die Initialisierung
	memory:	Ziel-Speicherbereich für die erstellte Struktur
	size:	Größe der Struktur in Byte

InitTmpRas

Initialisiert den angegebenen Speicherbereich für RastPort-Routinen mit Pufferbedarf (Areafill, Floodfill usw.).

Library: -468(graphics.library)

Deklaration in C:

```
void InitTmpRas(tmpras, buff, size)
    struct TmpRas *tmpras;
    UBYTE *buff;
    USHORT size;
```

Aufruf in C: InitTmpRas(tmpras, buff, size);

Aufruf in Assembler:

```
move.l tmpras,a0
move.l buff,a1
move.w #size,a0
move.l GfxBase,a6
jsr -468(a6)
```

Parameter:

tmpras:	Zeiger auf eine TmpRas-Struktur, die den Pufferspeicher verwaltet
buff:	Adresse des Puffers
size:	seine Größe in Byte

InitView

Löscht sämtliche Einträge der View-Struktur und setzt die Standardwerte für die Zentrierung des Bildfensters (DxOffset, DyOffset).

Library: -360(graphics.library)

Deklaration in C:

```
void InitView(view)
    struct View *view;
```

Aufruf in C: InitView(view);

Aufruf in Assembler:

```
move.l view,a1
move.l GfxBase,a6
jsr -360(a6)
```

Parameter:

view:	Adresse einer View-Struktur
-------	-----------------------------

InitVPort

Initialisiert die angegebene ViewPort-Struktur.

Library: -204(graphics.library)

Deklaration in C:

```
void InitVPort(vp)
    struct ViewPort;
```

Aufruf in C: `InitVPort(vp);`

Aufruf in Assembler:

```
move.l vp, a0
move.l GfxBase, a6
jsr -204(a6)
```

Parameter: `vp:` Zeiger auf zu initialisierende ViewPort-Struktur

Input

Ermittelt FileHandle-Struktur des Standard-Eingabekanals.

Library: -54(dos.library)

Deklaration in C: `struct FileHandle *Input ()`

Aufruf in C: `file = Input ();`

Parameter: file(d0): Zeiger auf FileHandle-Struktur des aktuellen Eingabekanals

Insert

Fügt eine Node-Struktur in die angegebene Liste ein.

Library: -234(exec.library)

Deklaration in C:

```
void Insert(list, node, pred)
    struct List *list;
    struct Node *node, *pred;
```

Aufruf in C: Insert(list, node, pred);

Aufruf in Assembler:

```
move.l list,a0
move.l node,a1
move.l pred,a2
move.l ExecBase,a6
jsr -234(a6)
```

Parameter:

list:	Adresse einer Listen-Struktur
node:	Zeiger auf einzufügende Node
pred:	Adresse der Vorgänger-Node-Struktur, hinter die die angegebene Node eingefügt werden soll

InstallClipRegion

Library: -174(layers.library)

Deklaration in C:

```
struct Region *InstallClipRegion(layer, region)
    struct Layer *layer;
    struct region *region;
```

Aufruf in C: oldclip = InstallClipRegion(layer, region);

Aufruf in Assembler:

```
move.l layer, a0
move.l region, a1
move.l LayersBase, a6
jsr -174(a6)
```

Parameter:

layer:	Adresse einer Layer-Struktur
region:	Zeiger auf Region-Struktur
oldclip(d0):	Adresse der alten ClipRegion

Anmerkung: Bevor das Layer mit DeleteLayer() oder CloseWindow() entfernt wird, muß die ClipRegion mit InstallClipRegion(layer, NULL) gelöscht werden. Sollte beim Einrichten einer ClipRegion nicht genug Speicher zur Verfügung stehen, so wird in layer->Flags das Bit LAYERS_CLIPRECTS_LOST gesetzt.

IntuiTextLength

Ermittelt die Pixelbreite eines Intuition-Textstrings.

Library: -330(intuition.library)

Deklaration in C:

```
long IntuiTextLength(itext)
    struct ItuiText *itext;
```

Aufruf in C:

```
len = IntuiTextLength(itext);
```

Aufruf in Assembler:

```
move.l itext,a0
move.l IntuitionBase,a6
jsr -330(a6)
```

Parameter:

itext:	Adresse der IntuiText-Struktur des zu untersuchenden Textstrings
len(d0):	Breite des Intuition-Textstrings in Pixels

Intuition

Library: -36(intuition.library)

Deklaration in C: void Intuition(ievent)

Aufruf in C: Intuition(ievent);

Aufruf in Assembler: move.l ievent,a0
move.l IntuitionBase,a6
jsr -36(a6)

Parameter: ievent:

IoErr

Liefert genaue Fehlerinformation über eine abgebroche DOS-E/A-Operation.

Library: -132(dos.library)

Deklaration in C: long IoErr()

Aufruf in C: error = IoErr();

Parameter: error(d0): DOS-Fehlervektor

IsInteractive

Überprüft, ob das mittels seiner FileHandle-Struktur angegebene File mit einem virtuellen Terminal verbunden ist und gibt eine BOOL-Antwort.

Library: -216(dos.library)

Deklaration in C:

```
BOOL IsInteractive(file)
    struct FileHandle *file;
```

Aufruf in C:

```
status = IsInteractive(file);
```

Aufruf in Assembler:

```
move.l file,d1
move.l DosBase,a6
jsr -216(a6)
```

Parameter:

file:	Adresse einer FileHandle-Struktur
status(d0):	TRUE, falls Verbindung zum virtuellen Terminal besteht

ItemAddress

Liefert einen Zeiger auf die MenuItem-Struktur des durch menunumber spezifizierten Menüeintrags.

Library: -144(intuition.library)

Deklaration in C:

```
struct MenuItem *ItemAddress
(menustrip, menunumber)
    struct Menu *menustrip;
    USHORT menunumber;
```

Aufruf in C: item = ItemAddress(menustrip, menunumber);

Aufruf in Assembler:

```
move.l menustrip, a0
move.w #menunumber, d0
move.l IntuitionBase, a6
jsr -144(a6)
```

Parameter:

menustrip:	Adresse der ersten Menu-Struktur in der Menüleiste
menunumber:	Menü- und Itemnummer
item(d0):	Zeiger auf MenuItem-Struktur des gesuchten Items oder NULL, falls nicht gefunden

LoadRGB4

Lädt die Farbtabelle der ViewPort-Struktur mit den RGB-Farbwerten aus dem angegebenen Parameterfeld.

Library: -192(graphics.library)

Deklaration in C:

```
void LoadRGB4(vp, colors, count)
    struct ViewPort *vp;
    USHORT *colors, count;
```

Aufruf in C: LoadRGB4(vp, colors, count);

Aufruf in Assembler:

```
move.l vp,a0
move.l colors,a1
move.w #count,d0
move.l GfxBase,a6
jsr -192(a6)
```

Parameter:

vp:	Zeiger auf ViewPort-Struktur, deren Color-Map initialisiert werden soll
colors:	Adresse einer Tabelle mit den RGB-Parametern
count:	Anzahl der Farbeinträge in der Farbtabelle

LoadSeg

Lädt die Segmente eines Moduls (Scatter-Load) und legt sie als Segment-Kette im Speicher ab (siehe auch UnLoadSeg()).

Library: -150(dos.library)

Deklaration in C:

```
BPTR LoadSeg(filename)
        UBYTE *name;
```

Aufruf in C:

```
segment = LoadSeg(filename);
```

Aufruf in Assembler:

```
move.l filename,d1
move.l DosBase,a6
jsr -150(a6)
```

Parameter:

filename:	Adresse eines nullterminierten Bezeichnungsstrings
segment:	BCPL-Zeiger auf das erste Segment oder NULL bei Mißerfolg

LoadView

Verwendet die bereits erstellten Copperlisten zur Bildschirmdarstellung.

Library: -222(graphics.library)

Deklaration in C: void LoadView(view)
 struct View *view;

Aufruf in C: LoadView(view);

Aufruf in Assembler: move.l view,a1
 move.l GfxBase,a6
 jsr -222(a6)

Parameter: view: Adresse einer View-Struktur

Lock

Library: -84(dos.library)

Deklaration in C:

```
struct FileLock *Lock(name, type)
    UBYTE *name;
    ULONG type;
```

Aufruf in C: lock = Lock(name, type)

Aufruf in Assembler:

```
move.l name,d1
move.l #type,d2
move.l DosBase,a6
jsr -84(a6)
```

Parameter:

name:	Adresse eines nullterminierten Bezeichnungsstrings
type:	Zugriffsmodus (ACCESS_READ, ACCESS_WRITE)
lock:	BCPL-Zeiger auf FileLock-Struktur oder NULL, falls nicht erfolgreich.

LockIBase

Library: -414(intuition.library)

Deklaration in C: `ULONG LockIBase(locknum)
ULONG locknum;`

Aufruf in C: `lock = LockIBase(locknum);`

Aufruf in Assembler: `move.l locknum,d0
move.l IntuitionBase,a6
jsr -414(a6)`

Parameter: `locknum:` Nummer des zu untersuchenden Locks
`lock(d0):` Locknummer für UnlockIBase()

Anmerkung: Wenn man LockIBase benutzt, sollte man nicht zur gleichen Zeit andere Locks aufrechterhalten (z.B. auf Layer oder LayerInfo).

LockLayer

Library: -96(layers.library)

Deklaration in C:

```
void LockLayer(li, layer)
    struct Layer_Info *li;
    struct Layer *layer;
```

Aufruf in C: LockLayer(li, layer);

Aufruf in Assembler:

```
move.l li,a0
move.l layer,a1
move.l LayersBase,a6
jsr -96(a6)
```

Parameter:

li:	Adresse einer Layer_Info-Struktur
layer:	Zeiger auf Layer-Struktur

LockLayerInfo

Library: -120(layers.library)

Deklaration in C:

```
void LockLayerInfo(li)
    struct Layer_Info *li;
```

Aufruf in C: `LockLayerInfo(li);`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a6
jsr -120(a6)
```

Parameter: li: Zeiger auf Layer_Info-Struktur

Anmerkung: Die Funktionen der Layer-Library benötigen keine Aufrufe von LockLayerInfo() und UnlockLayerInfo(); lediglich bei eigenen Routinen muß der kritische Abschnitt auf diese Weise gesichert werden.

LockLayerRom

Library: -432(graphics.library)

Deklaration in C: void LockLayerRom(layer)
 struct Layer *layer;

Aufruf in C: LockLayerRom(layer);

Aufruf in Assembler: move.l layer,a5
 move.l GfxBase,a6
 jsr -432(a6)

Parameter: layer: Adresse einer Layer-Struktur

LockLayers

Library: -108(layers.library)

Deklaration in C:

```
void LockLayers(li)
    struct Layer_Info *li;
```

Aufruf in C: `LockLayers(li);`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a6
jsr -108(a6)
```

Parameter: `li:` Zeiger auf Layer_Info-Struktur

MakeDosNode

Library: -144(expansion.library)

Deklaration in C: struct DeviceNode *MakeDosNode (parmpacket)
 struct Packet *packet;

Aufruf in C: node = MakeDosNode (parmpacket);

Aufruf in Assembler: move.l parmpacket,a0
 move.l ExpansionBase,a6
 jsr -144(a6)

Parameter: parmpacket: Zeiger auf Parameterfeld
 node(d0): Adresse der erzeugten Node-Struktur oder
 NULL

MakeFunctions

MakeFunctions() erstellt aus einem Feld von Funktionsvektoren (bestehend aus 32-Bit-Adreßzeigern) eine Sprungtabelle, wie sie von Systembibliotheken, Devices und Resources benötigt wird (siehe auch funcinit von MakeLibrary).

Library: -90(exec.library)

Deklaration in C:

```
long MakeFunctions
(target, functionarray, funcdispbases)
    ULONG *target, *functionarray;
    ULONG funcdispbases;
```

Aufruf in C:

```
tablesiz = MakeFunctions
(target, functionarray, funcdispbases);
```

Aufruf in Assembler:

```
move.l target,a0
move.l functionarray,a1
move.l funcdispbases,a2
move.l ExecBase,a6
jsr -90(a6)
```

Parameter:	target:	Zeiger auf den Speicherbereich (vom Anwender allokiert), in dem die Sprungtabelle erstellt werden soll
	functionarray:	Adresse der Vektortabelle (Adreßzeigern oder Offsets für Funktionseinsprünge) / Aufbau wie funcinit von MakeLibrary()
	funcdispbases:	besteht die Vektortabelle aus Wort-Offsets, so wird hier ihre Basisadresse übergeben (NULL, falls functionarray absolute Zeiger enthält)
	tablesiz:	Größe der erstellten Sprungtabelle

MakeLibrary

Konstruiert aus der angegebenen Tabelle von Funktionseinsparungen eine Library. Die Erstellung der Library-Struktur kann automatisch über eine InitStruct-Parameterliste (siehe auch InitStruct-Funktion) abgewickelt oder von einer eigenen Initialisierungsroutine übernommen werden.

Library: -84(exec.library)

Deklaration in C:

```
struct Library *MakeLibrary
(funcinit, structinit, libinit, datasize, seglist)
    ULONG *funcinit, *structinit, *libinit,
        *seglist;
    ULONG datasize;
```

Aufruf in C:

```
library = MakeLibrary
(funcinit, structinit, libinit, datasize, seglist);
```

Aufruf in Assembler:

```
move.l funcinit, a0
move.l structinit, a1
move.l libtinit, a2
move.l #datasize, d0
move.l codesize, d1
move.l ExecBase, a6
jsr -84(a6)
```

Parameter:	functinit:	Adresse einer Funktionstabelle. Hier stehen entweder Zeiger auf die einzubindenden Funktionen (jeweils 32 Bits lang) oder relative Offsets (16 Bits) zur Librarybasis, an denen die Funktionseinsprünge zu finden sind (letzteres ist durch ein \$FFFF-Wort am Beginn der Tabelle gekennzeichnet). Als Endmarkierung der Funktionstabelle dient in beiden Fällen der Wert -1 (Langwort für Zeigertabelle, Wort für Offsets).
	structinit:	Adresse einer Initialisierungstabelle für die Library-Strukturen oder NULL, falls nicht benötigt. (Achtung: Die Einträge lib_NegSize und lib_PosSize werden bereits von MakeLibrary() gesetzt und sollten nicht von dieser Tabelle überschrieben werden.)
	libtinit:	Zeiger auf eigene Initialisierungsroutine zur Erstellung der Library-Struktur. Die Routine erhält in d0 einen Zeiger auf die Library-Struktur und in a0 die Adresse der Segment-Liste.
	datasize:	Größe des Library-Datenbereichs (einschließlich Library-Knoten)
	seglst:	Adresse einer Segment-Liste (wird vom DOS benötigt)
	library (d0):	Zeiger auf Library-Struktur

MakeScreen

MakeVPort()-Aufruf von Intuition.

Library: -378(intuition.library)

Deklaration in C: void MakeScreen(screen)
 struct Screen *screen;

Aufruf in C: MakeScreen(screen);

Aufruf in Assembler: move.l screen,a0
 move.l IntuitionBase,a6
 jsr -378(a6)

Parameter: screen: Adresse einer Screen-Struktur

MakeVPort

Berechnet die Teil-Copper-Listen eines ViewPorts, die später durch Aufruf von MrgCop() zu einer Befehlsliste (bzw. zwei im Interlace-Betrieb) zusammengesetzt werden.

Library: -216(graphics.library)

Deklaration in C:

```
void MakeVPort (view, vp)
    struct View *view;
    struct ViewPort *vp;
```

Aufruf in C: MakeVPort (view, vp);

Aufruf in Assembler:

```
move.l view, a0
move.l vp, a1
move.l GfxBase, a6
jsr -216(a6)
```

Parameter:

view:	Zeiger auf vollständig initialisierte View-Struktur
vp:	Adresse einer ViewPort-Struktur

MarkCList

Setzt Markierung (Byte-Offset) für Index-Operationen.

Library: -120(clist.library)

Deklaration in C: long MarkCList(clist, offset)
 long clist, offset;

Aufruf in C: error = MarkCList(clist, offset);

Aufruf in Assembler: move.l clist,a0
 move.l #offset,d0
 move.l CListBase,a6
 jsr -102(a6)

Parameter:

clist:	CList-Header
offset:	Abstand der Markierung vom Anfang der Character-Liste
error:	ist offset größer als Listenlänge, dann ungleich NULL

MatchToolValue

Überprüft den Wert einer ToolType-Variablen.

Library: -102(icon.library)

Deklaration in C: `long MatchToolValue(typestring, value)`
`UBYTE *typestring, *value;`

Aufruf in C: `result = MatchToolValue(typestring, value);`

Aufruf in Assembler: `move.l typestring,a0`
`move.l value,a1`
`move.l IconBase,a6`
`jsr -102(a6)`

Parameter:

<code>typestring:</code>	Zeiger auf ToolType
<code>value:</code>	Adresse des Matchwert-Strings
<code>result(d0):</code>	1, falls Variablenwert mit Matchwert übereinstimmt, sonst NULL

ModifyIDCMP

Modifiziert die IDCMP-Flags des Window-Message-Ports.

Library: -150(intuition.library)

Deklaration in C:

```
void ModifyIDCMP(window, flags)
    struct Window *window;
    ULONG flags;
```

Aufruf in C: ModifyIDCMP(window, flags);

Aufruf in Assembler:

```
move.l window, a0
move.l #flags, d0
move.l IntuitionBase, a6
jsr -150(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur
flags:	IDCMP-Flags

ModifyProp

Modifiziert die Einstellungsparameter eines Proportional-Gadgets.

Library: -156(intuition.library)

Deklaration in C:

```
void ModifyProp
(gadget, window, requester, flags, hpot, vpot, hbody,
vbody)
    struct Gadget *gadget;
    struct Window *window;
    struct Requester *requester;
    USHORT flags, hpos, vpos, hbody, vbody;
```

Aufruf in C:

```
ModifyProp
(gadget, window, requester, flags, hpot, vpot, hbody,
vbody);
```

Aufruf in Assembler:

```
move.l gadget, a0
move.l window, a1
move.l requester, a2
move.w #flags, d0
move.w #hpot, d1
move.w #vpot, d2
move.w #hbody, d3
move.w #vbody, d4
move.l IntuitionBase, a6
jsr -156(a6)
```

Parameter:	gadget:	Zeiger auf eine gadget-Struktur
	window:	Adresse der zugehörigen Window-Struktur
	requester:	Zeiger auf einen eventuell vorhandenen Requester (sonst NULL)

flags:	neuer Wert der Flag-Variable		
hpot:	"	"	" HorizPot-Variable
vpot:	"	"	" VertPot-Variable
hbody:	"	"	" HorizBody-Variable
vbody:	"	"	" VertBody-Variable

Anmerkung:

Alle Gadgets, die nach dem modifizierten Proportional-Gadget in der Gadget-Liste stehen, werden von dieser Funktion aufgefrischt. Daher sollte man, insbesondere bei intensiver Nutzung von Prop-Gadgets, dafür sorgen, daß dieser Gadget-Typ am Ende der Liste plaziert ist. Bei der Funktion NewModifyProp() gibt es dieses Problem nicht, hier läßt sich die Anzahl der aufzufrischenden Gadgets explizit angeben.

Move

Verschiebt den Grafik-Cursor.

Library: -240(graphics.library)

Deklaration in C:

```
void Move(rp, x, y)
    struct RastPort *rp;
    short x, y;
```

Aufruf in C: Move(rp, x, y);

Aufruf in Assembler:

```
move.l rp,a1
move.w #x,d0
move.w #y,d1
move.l GfxBase,a6
jsr -240(a6)
```

Parameter:

rp:	Zeiger auf Zeichen-RastPort
x,y:	neue Position des Grafik-Cursors

MoveLayer

Verschiebt ein Layer relativ zur aktuellen Position seiner linken, oberen Ecke.

Library: -60(layers.library)

Deklaration in C:

```
void MoveLayer(li, layer, dx, dy)
    struct Layer_Info *li;
    struct Layer *layer;
    long dx, dy;
```

Aufruf in C: MoveLayer(li, layer, dx, dy);

Aufruf in Assembler:

```
move.l li,a0
move.l layer,a1
move.l #dx,d0
move.l #dy,d1
move.l LayersBase,a6
jsr -60(a6)
```

Parameter:

li:	Zeiger auf Layer_Info-Struktur
layer:	Adresse der zugehörigen Layer-Struktur
dx,dy:	x,y-Offsets (positive Werte bewirken eine Verschiebung nach rechts bzw. unten)

MoveLayerInFrontOf

Plaziert das angegebene Frontlayer vor dem Backlayer, berechnet sämtliche Damage-Listen neu und setzt entsprechend die LAYERREFRESH-Bits.

Library: -168(layers.library)

Deklaration in C: `BOOL MoveLayerInFrontOf(frontlayer, backlayer)
struct Layer *frontlayer, *backlayer;`

Aufruf in C: `status = MoveLayerInFrontOf
(frontlayer, backlayer);`

Aufruf in Assembler: `move.l frontlayer,a0
move.l backlayer,a1
move.l LayersBase,a6
jsr -168(a6)`

Parameter:

frontlayer:	Zeiger auf Layer-Struktur des Layers, das im Vordergrund erscheinen soll
backlayer:	Adresse der Layer-Struktur, dessen zugehöriges Layer in den Hintergrund verlagert wird
status (d0):	FALSE bei Abbruch (beispielsweise aus Speicherplatzmangel)

MoveScreen

Verschiebt einen Screen relativ zur aktuellen Position seiner linken, oberen Ecke.

Library: -162(intuition.library)

Deklaration in C:

```
void MoveScreen(screen, dx, dy)
    struct Screen *screen;
    short dx, dy;
```

Aufruf in C: MoveScreen(screen, dx, dy);

Aufruf in Assembler:

```
move.l screen,a0
move.w #dx,d0
move.w #dy,d1
move.l IntuitionBase,a6
jsr -162(a6)
```

Parameter:

screen:	Adresse einer Screen-Struktur
dx,dy:	relative Offsets zur aktuellen Bildschirmposition der linken, oberen Screenecke (positive Werte bewirken eine Verschiebung nach rechts bzw. unten)

Anmerkung: Um Schwierigkeiten mit zukünftigen Versionen des Amiga-Betriebssystems zu vermeiden, sollte dx auf 0 gesetzt werden, da eine Verschiebung der x-Koordinate zur Zeit ohnehin nicht unterstützt wird.

MoveSprite

Absolut-Verschiebung des angegebenen Hardware-Sprites.

Library: -426(graphics library)

Deklaration in C:

```
void MoveSprite(vp, simplesprite, x, y)
    struct ViewPort *vp;
    struct SimpleSprite *simplesprite;
    short x, y;
```

Aufruf in C: MoveSprite(vp, simplesprite, x, y);

Aufruf in Assembler:

```
move.l vp,a0
move.l simplesprite,a1
move.w #x,d0
move.w #y,d1
move.l GfxBase,a6
jsr -426(a6)
```

Parameter:

vp:	Zeiger auf eine ViewPort-Struktur (wird hier NULL übergeben, bezieht sich die Verschiebung auf die Bildposition der View-Struktur)
simplesprite:	Adresse der SimpleSprite-Struktur
x,y:	Verschiebkoordinaten (Auflösung: 1 Low-Res-Pixel)

MoveWindow

Verschiebt ein Window relativ zur aktuellen Position seiner linken, oberen Ecke.

Library: -168(intuition.library)

Deklaration in C:

```
void MoveWindow(window, dx, dy)
    struct Window *window;
    short dx, dy;
```

Aufruf in C: MoveWindow(window, dx, dy);

Aufruf in Assembler:

```
move.l window, a0
move.w #dx, d0
move.w #dy, d1
move.l IntuitionBase, a6
jsr -168(a6)
```

Parameter:

screen:	Adresse einer Window-Struktur
dx, dy:	relative Offsets zur aktuellen Bildschirmposition der linken, oberen Fensterecke (positive Werte bewirken eine Verschiebung nach rechts bzw. unten)

Anmerkung: MoveWindow() nimmt keine Parameterprüfung vor!

MrgCop

Fügt sämtliche Teil-Copperlisten (z.B. Display, Color, Sprite) zu einer bzw. zwei (Interlace-Modus) Befehlslisten für die Copper-Hardware zusammen.

Library: -210(graphics.library)

Deklaration in C:

```
void MrgCop(view)
    struct View *view;
```

Aufruf in C: `MrgCop(view);`

Aufruf in Assembler:

```
move.l view,a1
move.l GfxBase,a6
jsr -210(a6)
```

Parameter: `view:` Zeiger auf eine View-Struktur

NewFontContents

Ermittelt sämtliche Parameter der Zeichensätze, die sich in dem durch fontlock spezifizierten Verzeichnis befinden und faßt sie in einer FontContentsHeader-Struktur zusammen.

Library: -42(diskfont.library)

Deklaration in C:

```
struct FontContentsHeader *NewFontContents
(fontlock, fontname)
    struct FileLock *fontlock;
    UBYTE *fontname;
```

Aufruf in C: fch = NewFontContents(fontlock, fontname);

Aufruf in Assembler:

```
move.l fontlock,a0
move.l fontname,a1
move.l DiskFontBase,a6
jsr -42(a6)
```

Parameter:

fontlock:	Zeiger auf FileLock-Struktur des Zeichensatzverzeichnis
fontname:	Bezeichnung der FontContents-Datei
fch (d0):	Adresse der erstellten FontContentsHeader-Struktur oder NULL bei Abbruch

NewLayerInfo

Erzeugt eine vollständig initialisierte Layer_Info-Struktur.

Library: -144(layers.library)

Deklaration in C: `struct Layer_Info *NewLayerInfo()`

Aufruf in C: `li = NewLayerInfo();`

Aufruf in Assembler: `move.l LayerBase,a6
jsr -144(a6)`

Parameter: li (d0): Adresse der erzeugten Layer_Info-Struktur
oder NULL bei Abbruch

NewModifyProp

Neue Version der ModifyProp()-Routine zur Modifikation sämtlicher Einstellungsparameter eines Proportional-Gadgets.

Library: -468(intuition.library)

Deklaration in C:

```
void NewModifyProp
(gadget, window, requester, flags, hpot, vpot, hbody,
vbody, num)
    struct Gadget *gadget;
    struct Window *window;
    struct Requester *requester;
    USHORT flags, hpos, vpos, hbody, vbody;
    int num;
```

Aufruf in C:

```
NewModifyProp
(gadget, window, requester, flags, hpot, vpot, hbody,
vbody, num);
```

Aufruf in Assembler:

```
move.l gadget, a0
move.l window, a1
move.l requester, a2
move.w #flags, d0
move.w #hpot, d1
move.w #vpot, d2
move.w #hbody, d3
move.w #vbody, d4
move.l #num, d5
move.l IntuitionBase, a6
jsr -468(a6)
```

Parameter:

gadget:	Zeiger auf eine Gadget-Struktur
window:	Adresse der zugehörigen Window-Struktur

requester: Zeiger auf einen eventuell vorhandenen Re-
quester (sonst NULL)

flags: neuer Wert der Flag-Variable

hpot: " " " HorizPot-Variable

vpot: " " " VertPot-Variable

hbody: " " " HorizBody-Variable

vbody: " " " VertBody-Variable

num: bestimmt die Anzahl der Gadgets, die in den
Refreshzyklus miteinbezogen werden. (Bei
Angabe von -1 verhält sich die Funktion wie
ModifyProp.)

NewRegion

Erzeugt eine neue Region und initialisiert ihre Region-Struktur.

Library: -516(graphics.library)

Deklaration in C: `struct Region *NewRegion()`

Aufruf in C: `region = NewRegion();`

Parameter: `region (d0):` Zeiger auf initialisierte Region-Struktur oder NULL

ObtainConfigBinding

Sichert Zugriffsberechtigung auf die ConfigDev-Struktur bei nachträglicher Einbindung eines Treibers. (Wird beim Systemstart der CLI-Befehl BindDrivers verwendet, erübrigt sich ein Aufruf dieser Funktion.)

Library: -120(expansion.library)

Deklaration in C: `void ObtainConfigBinding();`

Aufruf in C: `ObtainConfigBinding();`

ObtainSemaphore

Privilegierte Zugriffsberechtigung auf einen Message-Port. Falls der Port bereits von einem anderen Task belegt wurde (lock), wartet ObtainSemaphore() auf die Freigabe und sichert unmittelbar danach dem aufrufenden Task das alleinige Zugriffsrecht.

Library: -564(exec.library)

Deklaration in C:

```
void ObtainSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
```

Aufruf in C: `ObtainSemaphore(sigsem);`

Aufruf in Assembler:

```
move.l sigsem,a0
move.l ExecBase,a6
jsr -564(a6)
```

Parameter: sigsem: Zeiger auf eine SignalSemaphore-Struktur

ObtainSemaphoreList

Library: -582(exec.library)

Deklaration in C: `void ObtainSemaphoreList(sigsem)
 struct SignalSemaphore *sigsem;`

Aufruf in C: `ObtainSemaphoreList(sigsem);`

Aufruf in Assembler: `move.l sigsem,a0
 move.l ExecBase,a6
 jsr -582(a6)`

Parameter: sigsem: Zeiger auf eine SignalSemaphore-Struktur

OffGadget

Inaktiviert das angegebene Gadget, so daß es nicht mehr angewählt werden kann (gerasterte, schattenhafte Darstellung).

Library: -174(intuition.library)

Deklaration in C:

```
void OffGadget(gadget, window, requester)
    struct Gadget *gadget;
    struct Window *window;
    struct Requester *requester;
```

Aufruf in C: OffGadget(gadget, window, requester);

Aufruf in Assembler:

```
move.l gadget,a0
move.l window,a1
move.l requester,a2
move.l IntuitionBase,a6
jsr -174(a6)
```

Parameter:

gadget:	Adresse einer Gadget-Struktur
window:	Zeiger auf zugehörige Window-Struktur
requester:	Zeiger auf Requester-Struktur eines eventuell vorhandenen Requesters oder NULL

OffMenu

Inaktiviert das angegebene Menü (bzw. Menüpunkt), so daß es nicht mehr angewählt werden kann (gerasterte, schattenhafte Darstellung).

Library: -180(intuition.library)

Deklaration in C:

```
void OffMenu(window, menunum)
    struct Window *window;
    USHORT menunum;
```

Aufruf in C: OffMenu(window, menunum);

Aufruf in Assembler:

```
move.l window, a0
move.w #menunum, d0
move.l IntuitionBase, a6
jsr -180(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur
menunum:	Nummer des zu deaktivierenden Menüs

OldOpenLibrary

Version 1.0 der OpenLibrary()-Funktion (ohne Prüfung der Versionsnummer).

Library: -408(exec.library)

Deklaration in C:

```
struct Library *OldOpenLibrary(libname)
char *libname;
```

Aufruf in C:

```
library = OldOpenLibrary(libname);
```

Aufruf in Assembler:

```
move.l libname,a1
move.l ExecBase,a6
jsr -408(a6)
```

Parameter:

libname:	Bezeichnung der gesuchten Library
library (d0):	Zeiger auf Library-Struktur oder NULL, falls nicht vorhanden

Anmerkung: Die Funktion OldOpenLibrary() ist nur erhalten geblieben, um auch Programme, die für Kickstart 1.0 geschrieben wurden, verwenden zu können.

OnGadget

Aktiviert das spezifizierte Gadget und hebt die Zugriffssperre wieder auf (siehe auch OffGadget()).

Library: -186(intuition.library)

Deklaration in C:

```
void OnGadget(gadget, window, requester)
    struct Gadget *gadget;
    struct Window *window;
    struct Requester *requester;
```

Aufruf in C: OnGadget(gadget, window, requester);

Aufruf in Assembler:

```
move.l gadget,a0
move.l window,a1
move.l requester,a2
move.l IntuitionBase,a6
jsr -186(a6)
```

Parameter:

gadget:	Adresse einer Gadget-Struktur
window:	Zeiger auf zugehörige Window-Struktur
requester:	Zeiger auf Requester-Struktur eines eventuell vorhandenen Requesters oder NULL

OnMenu

Hebt die Zugriffssperre von OffMenu() wieder auf (siehe auch On/OffGadget).

Library: -192(intuition.library)

Deklaration in C:

```
void OffMenu(window, menunum)
    struct Window *window;
    USHORT menunum;
```

Aufruf in C: OffMenu(window, menunum);

Aufruf in Assembler:

```
move.l window, a0
move.w #menunum, d0
move.l IntuitionBase, a6
jsr -192(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur
menunum:	Nummer des betroffenen Menüs

Open

Öffnet eine Datei für Lese- bzw. Schreiboperationen und liefert einen Zeiger auf ihre FileHandle-Struktur.

Library: -30(dos.library)

Deklaration in C:

```
struct FileHandle *Open(name, mode)
    UBYTE *name;
    LONG mode;
```

Aufruf in C: file = Open(name, mode);

Aufruf in Assembler:

```
move.l name,d1
move.l #mode,d2
move.l DosBase,a6
jsr -30(a6)
```

Parameter:

name:	Name der zu öffnenden Datei
mode:	MODE_OLDFILE (Öffnen einer bereits existierenden Datei), MODE_NEWFILE (Anlegen einer neuen Datei und Löschen einer eventuell unter derselben Bezeichnung vorhandenen Datei), MODE_READWRITE (exklusive Zugriffsberechtigung für den aufrufenden Task)
file (d0):	Zeiger auf Filehandle-Struktur der geöffneten Datei oder NULL bei Abbruch

OpenDevice

Öffnet das bezeichnete Device und initialisiert eine IORequest-Struktur.

Library: -444(exec.library)

Deklaration in C:

```
long OpenDevice
(devname, unit, iorequest, flags)
char *devname;
ULONG unit;
struct IORequest *iorequest;
ULONG flags;
```

Aufruf in C: OpenDevice(devname, unit, iorequest, flags);

Aufruf in Assembler:

```
move.l devname,a0
move.l #unit,d0
move.l iorequest,a1
move.l #flags,d1
move.l ExecBase,a6
jsr -444(a6)
```

Parameter:

devname:	Bezeichnung des gesuchten Devices (nullterminierter String)
unit:	Gerätenummer (device-abhängig)
iorequest:	Adresse einer IORequest-Struktur, die von der Funktion initialisiert wird
flags:	zusätzliche Treiberinformationen
error (d0):	NULL bei erfolgreicher Öffnung, sonst ungleich NULL

OpenDiskFont

Lädt einen diskresidenten Zeichensatz mit den angegebenen Eigenschaften und liefert einen Zeiger auf seine TextFont-Struktur.

Library: -30(diskfont.library)

Deklaration in C:

```
struct TextFont *OpenDiskFont(textattr)
    struct TextAttr *textattr;
```

Aufruf in C: font = OpenDiskFont(textattr);

Aufruf in Assembler:

```
move.l textattr,a0
move.l DiskFontBase,a6
jsr -30(a6)
```

Parameter:

textattr:	Adresse einer TextAttr-Struktur, die die Eigenschaften des gesuchten Zeichensatzes genauer beschreibt.
font (d0):	Zeiger auf TextFont-Struktur oder NULL bei Mißerfolg

OpenFont

Öffnet den Zeichensatz, der den angegebenen Spezifikationen entspricht oder am nächsten kommt.

Library: -72(graphics.library)

Deklaration in C:

```
struct TextFont *OpenFont(textattr)
    struct TextAttr *textattr;
```

Aufruf in C: font = OpenDiskFont(textattr);

Aufruf in Assembler:

```
move.l textattr,a0
move.l DiskFontBase,a6
jsr -72(a6)
```

Parameter:

textattr:	Adresse einer TextAttr-Struktur, die die Eigenschaften des gesuchten Zeichensatzes genauer beschreibt.
font (d0):	Zeiger auf TextFont-Struktur oder NULL bei Mißerfolg

OpenIntuition

Library: -30(intuition.library)

Deklaration in C: void OpenIntuition()

Aufruf in C: OpenIntuition();

OpenLibrary

Öffnet die bezeichnete Library mit der angegebenen Versionsnummer (siehe auch OldOpenLibrary()).

Library: -552(exec.library)

Deklaration in C:

```
struct Library *OpenLibrary(libname, version)
    char *libname;
    long version;
```

Aufruf in C: library = OpenLibrary(libname, version);

Aufruf in Assembler:

```
move.l libname,a1
move.l #version,d0
move.l ExecBase,a6
jsr -552(a6)
```

Parameter:

libname:	Zeiger auf nullterminierten String als Library-Bezeichnung
version:	gesuchte Versionsnummer
library (d0):	Zeiger auf Library-Struktur oder NULL bei Abbruch

OpenResource

Öffnet eine Resource mit Prüfung der Versionsnummer (muß größer oder gleich dem übergebenen Wert sein).

Library: -498(exec.library)

Deklaration in C:

```
struct Node *OpenResource(resname, version)
char *resname;
long version;
```

Aufruf in C:

```
resource = OpenResource(resname, version);
```

Aufruf in Assembler:

```
move.l resname,a1
move.l #version,d0
move.l ExecBase,a6
jsr -498(a6)
```

Parameter:

libname:	Zeiger auf nullterminierten String als Resource-Bezeichnung
version:	gesuchte Versionsnummer
library (d0):	Zeiger auf Library-Struktur oder NULL bei Abbruch

OpenScreen

Öffnet einen Screen mit den Parametern aus der NewScreen-Struktur und gibt den Zeiger auf eine komplett initialisierte Screen-Struktur zurück.

Offset: -198(intuition.library)

Deklaration in C:

```
struct Screen *OpenScreen(newscreen)
    struct NewScreen *newscreen;
```

Aufruf in C:

```
screen = OpenScreen(newscreen);
```

Aufruf in Assembler:

```
move.l newscreen, a0
move.l IntuitionBase, a6
jsr -198(a6)
```

Parameter:

newscreen:	Zeiger auf NewScreen-Struktur mit dem Parameterfeld für den zu öffnenden Screen
screen (d0):	Adresse der angelegten und initialisierten Screen-Struktur (bzw. NULL bei Abbruch)

OpenWindow

Öffnet ein Fenster mit den Parametern aus der NewWindow-Struktur und gibt den Zeiger auf eine komplett initialisierte Window-Struktur zurück.

Library: -204(intuition.library)

Deklaration in C:

```
struct Window *OpenWindow(newwindow)
    struct NewWindow *newwindow;
```

Aufruf in C:

```
window = OpenWindow(newwindow);
```

Aufruf in Assembler:

```
move.l newwindow,a0
move.l IntuitionBase,a6
jsr -204(a6)
```

Parameter:

newwindow:	Zeiger auf NewWindow-Struktur mit dem Parameterfeld für das zu öffnende Fenster
window (d0):	Adresse der angelegten und initialisierten Screen-Struktur (bzw. NULL bei Abbruch)

OpenWorkBench

Öffnet die Workbench-Benutzeroberfläche und liefert einen Zeiger auf den Workbench-Screen.

Library: -210(intuition.library)

Deklaration in C: struct Screen (oder BOOL) *OpenWorkBench()

Aufruf in C: screen = OpenWorkBench()

Parameter: screen (d0): Adresse der Screen-Struktur des Workbench-Fensters oder FALSE bei Abbruch

Anmerkung: Der Rückgabewert ist zwar ein Zeiger auf den Workbench-Screen, sollte aber nicht verwendet werden, da andere Programme die Workbench unbemerkt schließen können, wodurch Zugriffe auf das vermeintlich noch vorhandene Fenster zum Systemabsturz führen. Es bietet sich daher an, den Rückgabeparameter als BOOL-Variable zu betrachten (!0=WB offen, 0=WB zu).

OrRectRegion

Logische ODER-Verknüpfung zwischen einem Clipping-Rechteck und der zugehörigen Region (Ergebnis wird in die Region zurückgeschrieben). Sollte das Rechteck noch nicht vorhanden sein, wird es durch Aufruf der Funktion installiert.

Library: -510(graphics.library)

Deklaration in C:

```

BOOL OrRectRegion(region, rectangle)
    struct Region *region;
    struct Rectangle *rectangle;
  
```

Aufruf in C: status = OrRectRegion(region, rectangle);

Aufruf in Assembler:

```

move.l region,a0
move.l rectangle,a1
move.l ExecBase,a6
jsr -510(a6)
  
```

Parameter:

region:	Zeiger auf Region-Struktur
rectangle:	Adresse einer Rectangle-Struktur
status (d0):	FALSE bei Speicherplatzmangel

OrRegionRegion

Logische ODER-Verknüpfung zweier Regions (Ergebnis liegt in der Ziel-Region).

Library: -612(graphics.library)

Deklaration in C: `BOOL OrRegionRegion(source, destination)`
`struct Region *source, *destination;`

Aufruf in C: `status = OrRegionRegion(source, destination);`

Aufruf in Assembler: `move.l source,a0`
`move.l destination,a1`
`move.l ExecBase,a6`
`jsr -612(a6)`

Parameter:

source:	Zeiger auf Region-Struktur der Quell-Region
destination:	Zeiger auf Region-Struktur der Ziel-Region
status (d0):	FALSE bei Speicherplatzmangel

Output

Ermittelt den Standard-Ausgabekanal und liefert einen Zeiger auf seine FileHandle-Struktur.

Library: -60(dos.library)

Deklaration in C: `struct FileHandle *Output()`

Aufruf in C: `file = Output();`

Parameter: file (d0): Zeiger auf FileHandle-Struktur des aktuellen Ausgabekanals

OwnBlitter

Sichert dem aufrufenden Task Zugriffspriorität auf die Blitter-Hardware (siehe auch DisownBlitter()-Funktion).

Library: -456(graphics.library)

Deklaration in C: `void OwnBlitter()`

Aufruf in C: `OwnBlitter();`

ParentDir

Ermittelt das übergeordnete Verzeichnis.

Library: -210(dos.library)

Deklaration in C:

```
struct FileLock *ParentDir(lock)
    struct FileLock *lock;
```

Aufruf in C:

```
newlock = ParentDir(lock);
```

Aufruf in Assembler:

```
move.l lock,d1
move.l DosBase,a6
jsr -210(a6)
```

Parameter:

lock:	BCPL-Zeiger auf FileLock-Struktur eines Verzeichnisses (Root-Directory = NULL)
newlock (d0):	Adresse der FileLock-Struktur des übergeordneten Verzeichnisses

PeekCLMark

Holt das Byte an der aktuellen CList-Markierung.

Library: -132(clist.library)

Deklaration in C:

```
BYTE PeekCLMark(clist)
        ULONG clist;
```

Aufruf in C:

```
value = PeekCLMark(clist);
```

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a6
jsr -132(a6)
```

Parameter:

clist:	Identifikationsnummer der Character-Liste
value (d0):	ermittelter Byte-Wert

Permit

Aktiviert den Task-Dispatcher und hebt die Sperre des Multitaskingbetriebs wieder auf (siehe auch Forbid()).

Library: -138(exec.library)

Deklaration in C:

```
void Permit();
```

Aufruf in C:

```
Permit();
```

PolyDraw

Zeichnet ein Polygon mit Hilfe einer Vektortabelle.

Library: -336(graphics.library)

Deklaration in C:

```
void PolyDraw(rp, count, polytable)
    struct RastPort *rp;
    short count;
    struct tpoint polytable[count];
```

Aufruf in C: PolyDraw(rp, count, polytable);

Aufruf in Assembler:

```
move.l rp,a1
move.w #count,d0
lea polytable,a0
move.l GfxBase,a6
jsr -336(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Zeichen-RastPorts
count:	Anzahl der Polygoneckpunkte (Vektoren)
polytable:	Vektortabelle

Aufbau der Koordinaten-Struktur:

```
struct tpoint
{
    short x,y
};
```

PrintIText

Schreibt den Zeichenstring der IntuiText-Struktur (bzw. die Strings der verketteten Strukturen) in den spezifizierten RastPort.

Library: -216(intuition.library)

Deklaration in C:

```
void PrintIText(rp, itext, dx, dy)
    struct RastPort *rp;
    struct IntuiText *itext;
    short dx, dy;
```

Aufruf in C: PrintIText(rp, itext, dx, dy);

Aufruf in Assembler:

```
move.l rp,a0
move.l itext,a1
move.w #dx,d0
move.w #dy,d1
move.l IntuitionBase,a6
jsr -216(a6)
```

Parameter:

rp:	Adresse einer RastPort-Struktur
itext:	Zeiger auf IntuiText-Struktur
dx,dy:	x,y-Offset für linke, obere Textecke

PutCLBuf

Hängt Pufferdaten an das Ende der angegebenen Character-Liste.

Library: -108(clist.library)

Deklaration in C:

```
long PutCLBuf(clist, buffer, length)
    long clist;
    UBYTE *buffer;
    long length;
```

Aufruf in C: numbytes = PutCLBuf(clist, buffer, length)

Aufruf in Assembler:

```
move.l clist,a0
move.l buffer,a1
move.l #length,d1
move.l CListBase,a6
jsr -108(a6)
```

Parameter:

clist:	Listen-Identifikation
buffer:	Zeiger auf einen Datenpuffer
length:	seine Länge in Byte
numbytes (d0):	Anzahl der Byte, die aus Speicherplatzmangel nicht mehr angehängt werden konnten (oder NULL bei Abbruch)

PutCLChar

Hängt ein Byte an das Ende der CListe an.

Library: -60(clist.library)

Deklaration in C:

```
long PutCLChar(clist, value)
    long clist;
    BYTE value;
```

Aufruf in C: error = PutCLChar(clist, value);

Aufruf in Assembler:

```
move.l clist,a0
move.b #value,d0
move.l CListBase,a6
jsr -60(a6)
```

Parameter:

clist:	Identifikations-Header der CListe
value:	Wert des einzufügenden Bytes
error (d0):	ungleich NULL bei Mißerfolg

PutCLWord

Hängt ein Wort an das Ende der CListe an.

Library: -84(clist.library)

Deklaration in C:

```
long PutCLWord(clist, value)
    long clist;
    WORD value;
```

Aufruf in C: error = PutCLWord(clist, value);

Aufruf in Assembler:

```
move.l clist,a0
move.w #value,d0
move.l CListBase,a6
jsr -84(a6)
```

Parameter:

clist:	Identifikations-Header der CListe
value:	Wert des einzufügenden Bytes
error (d0):	ungleich NULL bei Mißerfolg

PutDiskObject

Siehe PutIcon().

Library: -84(icon.library)

Deklaration in C:

```
BOOL PutDiskObject(name, diskobj);
        UBYTE *name;
        struct DiskObject *diskobj;
```

Aufruf in C:

```
status = PutDisObject(name, icon);
```

Aufruf in Assembler:

```
move.l name,a0
move.l diskobj,a1
move.l IconBase,a6
jsr -86(a6)
```

Parameter:

name:	Bezeichnung des Disk-Objektes (wird automatisch durch ".info" erweitert)
diskobj:	Zeiger auf die betreffende DiskObject-Struktur
status (d0):	FALSE bei I/O-Fehler

PutIcon

Schreibt die angegebene DiskObject-Struktur auf Diskette.

Library: -48(icon.library)

Deklaration in C:

```
BOOL PutIcon(name, icon)
    UBYTE *name;
    struct DiskObject *icon;
```

Aufruf in C: status = PutIcon(name, icon);

Aufruf in Assembler:

```
move.l name, a0
move.l icon, a1
move.l IconBase, a6
jsr -48(a6)
```

Parameter:

name:	Bezeichnung des Disk-Objektes (wird automatisch durch ".info" erweitert)
icon:	Zeiger auf die betreffende DiskObject Struktur
status (d0):	FALSE bei I/O-Fehler

PutMsg

Sendet eine Nachricht an den spezifizierten Message-Port.

Library: -366(exec.library)

Deklaration in C:

```
void PutMsg(port, message)
    struct MsgPort *port;
    struct Message *message;
```

Aufruf in C: PutMsg(port, message);

Aufruf in Assembler:

```
move.l port, a0
move.l message, a1
move.l ExecBase, a6
jsr -366(a6)
```

Parameter:

port:	Adresse der MsgPort-Struktur des Empfänger-Ports
message:	Zeiger auf Message-Struktur der zu senden- den Nachricht

PutWBOject

Schreibt die angegebene WBOject-Struktur auf Diskette.

Library: -36(icon.library)

Deklaration in C:

```
long PutWBOject(name, object)
    UBYTE *name;
    struct WBOject *object;
```

Aufruf in C: status = PutWBOject(name, object);

Aufruf in Assembler:

```
move.l name,a0
move.l object,a1
move.l IconBase,a6
jsr -36(a6)
```

Parameter:

name:	Bezeichnung des Objektes (der Strings wird automatisch mit ".info" erweitert)
object:	Adresse einer WBOject-Struktur
status:	NULL bei Abbruch

QBlit

Fügt eine BlitNode-Struktur (enthält den Zeiger auf eine Anwenderroutine für den Blitter) in die Liste der Blitter-Jobs ein. Beim Aufruf der Routine aus der Job-Queue erhält der ausführende Task das alleinige Zugriffsrecht auf die Blitter-Hardware.

Library: -276(graphics.library)

Deklaration in C:

```
void QBlit(blit)
    struct BlitNode *blit;
```

Aufruf in C:

```
QBlit(blit);
```

Aufruf in Assembler:

```
move.l blit,a1
move.l GfxBase,a6
jsr -276(a6)
```

Parameter: **blit:** Zeiger auf eine BlitNode-Struktur zur Verwaltung der Blitter-Routine

QBSBlit

Wie QBlit, allerdings geschieht der Aufruf der Blitter-Jobs (Blitter-Routinen) in Synchronisation mit dem Rasterstrahl.

Library: -294(graphics.library)

Deklaration in C:

```
void QBSBlit(blit)
    struct BlitNode *blit;
```

Aufruf in C: `QBSBlit(blit);`

Aufruf in Assembler:

```
move.l blit,a1
move.l GfxBase,a6
jsr -294(a6)
```

Parameter: `blit:` Zeiger auf eine BlitNode-Struktur, die die Einsprungadresse der aufzurufenden Routine enthält

QueuePacket

Übermittelt ein DOS-Paket an einen bestimmten Prozeß (definiert in der DosPacket-Struktur).

Library: -168(dos.library)

Deklaration in C:

```
long QueuePacket(packet)
    struct DosPacket *packet;
```

Aufruf in C:

```
error = QueuePacket(packet);
```

Aufruf in Assembler:

```
move.l packet,d1
move.l DosBase,a6
jsr -168(a6)
```

Parameter:

packet:	Zeiger auf eine DosPacket-Struktur
error (d0):	NULL bei Abbruch

RawDoFmt

Formatierte Ausgabe eines Strings.

Library: -522(exec.library)

Deklaration in C:

```
void RawDoFmt(format, data, putproc, putdata)
    ULONG *format;
    ULONG *data;
    ULONG putproc, putdata;
```

Aufruf in C: RawDoFmt(format, data, putproc, putdata);

Aufruf in Assembler:

```
move.l format,a0
move.l data,a1
lea.l putproc,a2
lea.l putdata,a3
move.l ExecBase,a6
jsr -522(a6)
```

Parameter:	format:	Zeiger auf den Format-String
	data:	Adresse der einzufügenden Daten
	putproc:	Adresse einer Ausgaberroutine
	putdata:	Adresse der Einfügeroutine

RawIOInit

Library: -504(exec.library)

Deklaration in C: void RawIOInit ()

Aufruf in C: RawIOInit ();

Anmerkung: Zur Zeit stellt die Funktion lediglich die Baudrate der seriellen Schnittstelle ein.

RawKeyConvert

Konvertierungs-Routine für RawKey-Codes des Tastaturprozessors in die gewohnte ASCII-Darstellung. Die KeyCodes gelangen über das Keyboard- und Input-Device zum Console-Device, wo sie mittels der aktuellen Keymap in die endgültige Form gebracht werden.

Library: -48(console.library)
 short RawKeyConvert
 (events, buffer, length, keymap)

Deklaration in C: struct InputEvent *events;
 UBYTE *buffer;
 long length;
 struct KeyMap *keymap;

Aufruf in C: actual = RawKeyConvert
 (events, buffer, length, keymap)

Aufruf in Assembler: move.l events,a0
 move.l buffer,a1
 move.l #length,d1
 move.l keymap,a2
 move.l ConsoleBase,a6
 jsr -48(a6)

Parameter:

events:	Zeiger auf InputEvent-Struktur, in der die KeyCodes gepuffert sind
buffer:	Datenpuffer, in dem der konvertierte String abgelegt wird
length:	seine Größe in Byte
keymap:	Adresse der aktuellen Keymap (Konvertierungstabelle)
actual (d0):	Anzahl der effektiv konvertierten Zeichen oder -1, falls Puffer zu klein

Anmerkung: ConsoleDevice in A6.

```
OpenDevice("console.device",-1,IOSDReq,0);
```

Der io_Device-Pointer in IOSDReq liegt dann in A6.

Read

Lesezugriff auf eine zuvor geöffnete Datei.

Library: -42(dos.library)

Deklaration in C:

```
long Read(file, buffer, length)
    struct FileHandle *file;
    UBYTE *buffer;
    long length;
```

Aufruf in C: reallyread = Read(file, buffer, length);

Aufruf in Assembler:

```
move.l file,d1
move.l buffer,d2
move.l #length,d3
move.l DosBase,a6
jsr -42(a6)
```

Parameter:

file:	Zeiger auf FileHandle-Stuktur
buffer:	Adresse des Pufferspeichers, in dem die gelesenen Daten abgelegt werden
length:	Größe dieses Puffers in Byte
reallyread (d0):	Anzahl der effektiv gelesenen DatenByte oder 0, wenn die Daten vollständig in den Puffer passen (-1 bei I/O-Fehler)

ReadExpansionByte

Liest ein Byte aus dem Konfigurationsbereich der Erweiterungskarte an der Basisadresse board.

Library: -96(expansion.library)

Deklaration in C:

```
BYTE ReadExpansionByte(board, offset)
long board, offset;
```

Aufruf in C:

```
value = ReadExpansionByte(board, offset);
```

Aufruf in Assembler:

```
move.l #board,a0
move.l #offset,d0
move.l ExpansionBase,a6
jsr -96(a6)
```

Parameter:

board:	Basisadresse der Erweiterungskarte
offset:	Offset in der ExpansionRom-Struktur (Berechnung mit den Makros EROFFSET/ECOFFSET)
value (d0):	Wert des gelesenen Bytes oder -1 bei Abbruch

ReadExpansionRom

Ermittelt die Konfigurationsparameter einer Erweiterungskarte in der angegebenen ConfigDev-Struktur.

Library: -102(expansion.library)

Deklaration in C:

```
long ReadExpansionRom(board, configdev)
    long board;
    struct ConfigDev *configdev;
```

Aufruf in C: error = ReadExpansionRom(board, configdev);

Aufruf in Assembler:

```
move.l #board, a0
move.l configdev, a1
move.l ExpansionBase, a6
jsr -102(a6)
```

Parameter:

board:	Basisadresse der untersuchten Erweiterungskarte
configdev:	Zeiger auf zugehörige ConfigDev-Struktur
error (d0):	ungleich NULL bei Abbruch

ReadPixel

Ermittelt das Farbregister aus dem die Farbe des angegebenen Bildschirmpunktes stammt.

Library: -318(graphics.library)

Deklaration in C:

```
long ReadPixel(rp, x, y)
    struct RastPort *rp;
    short x, y;
pen = ReadPixel(rp, x, y);
```

Aufruf in C:

Aufruf in Assembler:

```
move.l rp,a1
move.l #x,d0
move.l #y,d1
move.l GfxBase,a6
jsr -318(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Zeichen-RastPorts
x,y:	Bildschirmkoordinate des betroffenen Pixels
pen (d0):	Farbstift (Registernummer) oder -1 bei Mißerfolg (Punkt liegt außerhalb der Rast-Port-Bitmap)

RectFill

Füllt ein Rechteck mit den aktuellen Füll- bzw. Farbparametern des RastPorts.

Library: -306(graphics.library)

Deklaration in C:

```
void RectFill(rp, x1, y2, x1, y2)
    struct RastPort *rp;
    short x1,y1,x2,y2;
```

Aufruf in C: RectFill(rp, x1, y2, x1, y2);

Aufruf in Assembler:

```
move.l rp,a1
move.l #x1,d0
move.l #y1,d1
move.l #x2,d2
move.l #y2,d3
move.l GfxBase,a6
jsr -306(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Zeichen-RastPorts
x1,y1:	linke, obere Ecke des zu füllenden Rechtecks
x2,y2:	rechte, untere Ecke

Anmerkung: Die Funktion hat einen Bug! Wenn im Komplement-Modus mit dem FG-Pen gearbeitet wird, werden alle Bitplanes komplementiert!

RefreshGadgets

Frischt sämtliche Gadgets auf, die auf die angegebene Gadget-Struktur in der Gadget-Liste folgen.

Library: -222(intuition.library)

Deklaration in C:

```
void RefreshGadgets(gadgets, window, requester)
    struct Gadget *gadgets;
    struct Window *window;
    struct Requester *requester;
```

Aufruf in C: RefreshGadgets(gadgets, window, requester);

Aufruf in Assembler:

```
move.l gadgets, a0
move.l window, a1
move.l requester, a2
move.l IntuitionBase, a6
jsr -222(a6)
```

Parameter:

gadgets:	Zeiger auf die Gadget-Struktur (in der Gadget-Liste), ab der die Gadgets neu dargestellt werden sollen
window:	Adresse des zugehörigen Windows
requester:	Zeiger auf eventuell vorhandene Requester-Struktur

RefreshGList

Refreshroutine für die gezielte Auffrischung einer bestimmten Anzahl von Gadgets im angegebenen Window oder Requester.

Library: -32(intuition.library)

Deklaration in C:

```
void RefreshGList
(gadgets, window, requester, numgad)
    struct Gadget *gadgets;
    struct Window *window;
    struct Requester *requester;
    short numgad;
```

Aufruf in C:

```
RefreshGList
(gadgets, window, requester, numgad);
```

Aufruf in Assembler:

```
move.l gadgets,a0
move.l window,a1
move.l requester,a2
move.w #numgad,d0
move.l IntuitionBase,a6
jsr -432(a6)
```

Parameter:

gadgets:	Zeiger auf eine verkettete Gadget-Struktur in der Gadget-Liste, ab der der Refreshzyklus greifen soll
window:	Adresse des zugehörigen Windows
requester:	Zeiger auf eventuell vorhandene Requester-Struktur
numgad:	Anzahl der aufzufrischenden Gadgets (-1 frischt sämtliche Gadgets in der Liste auf => RefreshGadgets())

RefreshWindow

Refresh-Routine für die Fensterdarstellung.

Library: -456(intuition.library)

Deklaration in C:

```
void RefreshWindow(window)
    struct Window *window;
```

Aufruf in C:

```
RefreshWindow(window);
```

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a6
jsr -456(a6)
```

Parameter: window: Zeiger auf eine Window-Struktur

ReleaseConfigBinding

Gibt das Zugriffsrecht auf die Einbindung eines Treibers frei, so daß auch andere Tasks ihre Treiber in die ConfigDev-Struktur integrieren können.

Library: -126(expansion.library)

Deklaration in C:

```
void ReleaseConfigBinding()
```

Aufruf in C:

```
ReleaseConfigBinding();
```

ReleaseSemaphore

Gibt den Zugriff auf eine komplette Semaphore-Liste mit gelinkten SignalSemaphore-Strukturen frei (siehe auch ObtainSemaphore(), ReleaseSemaphoreList()).

Library: -570(exec.library)

Deklaration in C:

```
void ReleaseSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
```

Aufruf in C: ReleaseSemaphore(sigsem);

Aufruf in Assembler:

```
move.l sigsem,a0
move.l ExecBase,a6
jsr -570(a6)
```

Parameter: sigsem: Zeiger auf eine SignalSemaphore-Struktur

ReleaseSemaphoreList

Gibt den Zugriff auf eine komplette Semaphore-Liste mit gelinkten SignalSemaphore-Strukturen frei (siehe auch ObtainSemaphoreList()).

Library: -588(exec.library)

Deklaration in C: void ReleaseSemaphoreList (sigsem)
 struct SignalSemaphore *sigsem;

Aufruf in C: ReleaseSemaphoreList (sigsem);

Aufruf in Assembler: move.l sigsem, a0
 move.l ExecBase, a6
 jsr -588(a6)

Parameter: sigsem: Zeiger auf eine SignalSemaphore-Struktur in der Liste

RemakeDisplay

Neuberechnung sämtlicher Darstellungsparameter.

Library: -384(intuition.library)

Deklaration in C: void RemakeDisplay()

Aufruf in C: RemakeDisplay();

Anmerkung: RemakeDisplay() führt zuerst ein Äquivalent von MakeScreen() durch und ruft dann RethinkDisplay() auf. Wegen der hohen Ausführungszeit und der Multitaskingsperre (Forbid/Permit) sollte man diese Funktion nicht allzu oft verwenden.

RemConfigDev

Entfernt eine ConfigDev-Struktur aus der ConfigDev-Liste.

Library: -108(expansion.library)

Deklaration in C:

```
void RemConfigDev(configdev)
    struct ConfigDev *configdev;
```

Aufruf in C: RemConfigDev(configdev);

Aufruf in Assembler:

```
move.l configdev,a0
move.l ExpansionBase,a6
jsr -108(a6)
```

Parameter: configdev: Adresse einer ConfigDev-Struktur

RemDevice

Entfernt das angegebene Device aus der Systemliste.

Library: -438(exec.library)

Deklaration in C:

```
long RemDevice(device)
    struct Device *device;
```

Aufruf in C:

```
error = RemDevice(device);
```

Aufruf in Assembler:

```
move.l device,a1
move.l ExecBase,a6
jsr -438(a6)
```

Parameter:
device: Zeiger auf eine Device-Struktur
error: NULL bei erfolgreicher Bearbeitung

RemFont

Entfernt einen Zeichensatz aus der System-Fontliste.

Library: -486(graphics.library)

Deklaration in C:

```
void RemFont(textfont)
    struct TextFont *textfont;
```

Aufruf in C: RemFont(textfont);

Aufruf in Assembler:

```
move.l textfont,a1
move.l GfxBase,a6
jsr -486(a6)
```

Parameter:

textfont:	Zeiger auf TextFont-Struktur des zu entfernenden Zeichensatzes.
-----------	---

RemHead

Löscht die erste Node-Struktur (Listenkopf) in einer Liste.

Library: -258(exec.library)

Deklaration in C:

```
struct Node *RemHead(list)
    struct List *list;
```

Aufruf in C: node = RemHead(list);

Aufruf in Assembler:

```
move.l list,a0
move.l ExecBase,a6
jsr -258(a6)
```

Parameter:

list:	Zeiger auf eine List-Struktur
node (d0):	Adresse der entfernten Node-Struktur oder NULL bei Abbruch

RemIBob

RemIBob

Entfernt das spezifizierte Blitter-Objekt (Bob) aus der GEL-Liste und löscht es aus dem Darstellungsbereich (in Synchronisation mit dem Rasterstrahl).

Library: -132(graphics.library)

Deklaration in C:

```
void RemIBob(bob, rp, vp)
    struct Bob *bob;
    struct RastPort *rp;
    struct ViewPort *vp;
```

Aufruf in C: RemIBob(bob, rp, vp);

Aufruf in Assembler:

```
move.l bob,a0
move.l rp,a1
move.l vp,a2
move.l GfxBase,a6
jsr -132(a6)
```

Parameter:

bob:	Adresse einer Bob-Struktur
rp:	Zeiger auf GEL-RastPort
vp:	Adresse des zugehörigen ViewPorts

RemIntServer

Entfernt den angegebenen Interrupt-Server aus der Server-Liste.

Library: -174(exec.library)

Deklaration in C:

```
void RemIntServer(intnum, interrupt)
    long intnum;
    struct Node *interrupt;
```

Aufruf in C: RemIntServer(intnum, interrupt);

Aufruf in Assembler:

```
move.l #intnum,d0
move.l interrupt,a1
move.l ExecBase,a6
jsr -174(a6)
```

Parameter:

intnum:	Nummer des belegten System-Interrupts
interrupt:	Zeiger auf Interrupt-Struktur

RemLibrary

Entfernt eine Library aus der Liste der Systembibliotheken (LibList).

Library: -402(exec.library)

Deklaration in C:

```
long RemLibrary(library)
    struct Library *library;
```

Aufruf in C:

```
error = RemLibrary(library);
```

Aufruf in Assembler:

```
move.l library, a1
move.l ExecBase, a6
jsr -402(a6)
```

Parameter:

library:	Zeiger auf Library-Struktur der zu entfernen- den Library
error (d0):	NULL bei erfolgreicher Entfernung

Remove

Entfernt die angegebene Node-Struktur aus der Liste.

Library: -252(exec.library)

Deklaration in C:

```
void Remove(node)
    struct Node *node;
```

Aufruf in C: Remove (node);

Aufruf in Assembler:

```
move.l node,a1
move.l ExecBase,a6
jsr -252(a6)
```

Parameter: node: Zeiger auf die zu entfernende Node-Struktur

RemoveGadget

Entfernt ein Gadget aus der Gadget-Liste des angegebenen Windows.

Library: -228(intuition.library)

Deklaration in C:

```
short RemoveGadget(window, gadget)
    struct Window *window;
    struct Gadget *gadget;
```

Aufruf in C: RemoveGadget(window, gadget);

Aufruf in Assembler:

```
move.l window, a0
move.l gadget, a1
move.l IntuitionBase, a6
jsr -228(a6)
```

Parameter:

window:	Zeiger auf eine Window-Struktur
gadget:	Adresse des zu entfernenden Gadgets
position (d0):	Ordnungsnummer des entfernten Gadgets oder -1 bei Abbruch

RemPort

Entfernt einen Message-Port aus der PortList des Systems.

Library: -360(exec.library)

Deklaration in C:

```
void RemPort(port)
    struct MsgPort *port;
```

Aufruf in C: `RemPort(port);`

Aufruf in Assembler:

```
move.l port,a1
move.l ExecBase,a6
jsr -360(a6)
```

Parameter: `port:` Adresse der zu entfernenden MsgPort-Struktur

RemResource

Entfernt eine Resource aus der Systemliste.

Library: -92(exec.library)

Deklaration in C:

```
void RemResource(resource)
    struct Resource *resource;
```

Aufruf in C: RemResource(resource);

Aufruf in Assembler:

```
move.l resource,a1
move.l ExecBase,a6
jsr -492(a6)
```

Parameter: resource: Zeiger auf eine Resource-Struktur.

RemSemaphore

Löscht eine SignalSemaphore-Struktur aus der Systemliste.

Library: -606(exec.library)

Deklaration in C:

```
void RemSemaphore(sigsem)
    struct SignalSemaphore *sigsem;
```

Aufruf in C: `RemSemaphore(sigsem);`

Aufruf in Assembler:

```
move.l sigsem,a1
move.l ExecBase,a6
jsr -606(a6)
```

Parameter: sigsem: Adresse der zu entfernenden Signal-Semaphore-Struktur

RemTail

Schneidet die Node-Struktur am Listenende ab.

Library: -264(exec.library)

Deklaration in C:

```
struct Node *RemTail(list)
    struct List *list;
```

Aufruf in C:

```
node = RemTail(list);
```

Aufruf in Assembler:

```
move.l list,a0
move.l ExecBase,a6
jsr -264(a6)
```

Parameter:

list:	Zeiger auf betroffene List-Struktur
node (d0):	Adresse auf Node-Struktur des entfernten Knotens oder NULL bei Abbruch

RemTask

Entfernt den angegebenen Task aus den Systemlisten.

Library: -288(exec.library)

Deklaration in C:

```
void RemTask(task)
    struct Task *task;
```

Aufruf in C: `RemTask(task);`

Aufruf in Assembler:

```
move.l task,a1
move.l ExecBase,a6
jsr -288(a6)
```

Parameter: task: Adresse der zu entfernenden Task-Struktur

RemVSprite

Löscht die angegebene VSprite-Struktur aus der GEL-Liste des Darstellungs-RastPorts.

Library: -138(graphics.library)

Deklaration in C:

```
void RemVSprite(vsprite)
    struct VSprite *vsprite;
```

Aufruf in C: RemVSprite(vsprite);

Aufruf in Assembler:

```
move.l vsprite,a0
move.l GfxBase,a6
jsr -138(a6)
```

Parameter: vsprite: Adresse der VSprite-Struktur des zu entfernenden Virtual-Sprites

Rename

Umbenennung einer Datei oder eines Verzeichnisses.

Library: -78(dos.library)

Deklaration in C: `BOOL Rename (oldname, newname)`
`UBYTE *oldname, *newname;`

Aufruf in C: `success = Rename (oldname, newname);`

Aufruf in Assembler: `move.l oldname,d1`
`move.l newname,d2`
`move.l DosBase,a6`
`jsr -78(a6)`

Parameter:

<code>oldname:</code>	alter Dateiname (vollständiger Suchpfad)
<code>newname:</code>	neuer Dateiname (vollständiger Suchpfad)
<code>success (d0):</code>	FALSE bei Abbruch

ReplyMsg

Bestätigt den Empfang einer Nachricht durch Rücksendung des Message-Pakets.

Library: -378(exec.library)

Deklaration in C:

```
void ReplyMsg(message)
    struct Message *message;
```

Aufruf in C:

```
ReplyMsg(message);
```

Aufruf in Assembler:

```
move.l message,a1
move.l ExecBase,a6
jsr -378(a6)
```

Aufruf in Assembler:

```
move.l message,a1
move.l ExecBase,a6
jsr -378(a6)
```

Parameter:

message:	Zeiger auf Message-Struktur der betreffenden Nachricht
----------	--

ReportMouse

Setzt REPORTMOUSE-Flag (bzw. FOLLOWMOUSE, wenn Mauszeiger über einem aktivierten Gadget) im Window-Message-Port (IDCMP-Flags).

Library: -234(intuition.library)

Deklaration in C:

```
void ReportMouse(boolean, Window)
    BOOL boolean;
    struct Window *window;
```

Aufruf in C: ReportMouse(boolean, Window);

Aufruf in Assembler:

```
move.w #boolean,a0
move.l window,d0
move.l IntuitionBase,a6
jsr -234(a6)
```

Parameter:

boolean:	TRUE, falls Flag gesetzt werden soll, sonst FALSE
window:	Adresse einer Window-Struktur

Request

Blendet den durch die Requester-Struktur definierten Requester in sein zugehöriges Window ein.

Library: -240(intuition.library)

Deklaration in C:

```
BOOL Request(requester, window)
    struct Requester *requester;
    struct Window *window;
```

Aufruf in C:

```
success = Request(requester, window);
```

Aufruf in Assembler:

```
move.l requester,a0
move.l window,a1
move.l IntuitionBase,a6
jsr -240(a6)
```

Parameter:

requester:	Zeiger auf eine Requester-Struktur
window:	Adresse der Window-Struktur des Fensters, in dem der Requester dargestellt werden soll
success:	FALSE bei mißglückter Darstellung

Reschedule

Systemroutine für den Task-Scheduler.

Library: -48(exec.library)

Deklaration in C: void Reschedule()

Aufruf in C: Reschedule();

RethinkDisplay

Überprüft sämtliche Darstellungsparameter und initialisiert die Copperlisten neu.

Library: -390(intuition.library)

Deklaration in C: void RethinkDisplay()

Aufruf in C: RethinkDisplay();

RomBoot

Boot-Routine der RomBoot-Library (ab Kickstart 1.3).

Library: -30(romboot.library)

Deklaration in C: void RomBoot()

Aufruf in C: RomBoot();

Schedule

Systemroutine für den Task-Scheduler.

Library: -42(exec.library)

Deklaration in C: void Schedule()

Aufruf in C: Schedule();

ScreenToBack

Versetzt den angegebene Screen in den Hintergrund (erscheint hinter allen anderen Screens).

Library: -246(intuition.library)

Deklaration in C:

```
void ScreenToBack(screen)
    struct Sreen *screen;
```

Aufruf in C: `ScreenToBack(screen);`

Aufruf in Assembler:

```
move.l screen,a0
move.l IntuitionBase,a6
jsr -246(a6)
```

Parameter: screen: Zeiger auf Screen-Struktur des betroffenen Screens

ScreenToFront

Versetzt den angegebenen Screen in den Vordergrund (erscheint vor allen anderen Screens).

Library: -252(intuition.library)

Deklaration in C:

```
void ScreenToFront(screen)
    struct Screen *screen;
```

Aufruf in C:

```
ScreenToFront(screen);
```

Aufruf in Assembler:

```
move.l screen,a0
move.l IntuitionBase,a6
jsr -252(a6)
```

Parameter: screen: Zeiger auf Screen-Struktur des betroffenen Screens

ScrollLayer

Scroll-Routine zur Verschiebung eines Layers.

Library: -72(layers.library)

Deklaration in C:

```
void ScrollLayer(li, layer, dx, dy)
    struct Layer_Info *li;
    struct Layer *layer;
    short dx, dy;
```

Aufruf in C: ScrollLayer(li, layer, dx, dy);

Aufruf in Assembler:

```
move.l li,a0
move.l layer,a1
move.w #dx,d0
move.w #dy,d1
move.l LayersBase,a6
jsr -72(a6)
```

Parameter:

li:	Zeiger auf eine Layer_Info-Struktur
layer:	Adresse einer Layer-Struktur
dx,dy:	Verschiebungsvektor

ScrollRaster

Scroll-Funktion für rechteckige Bildschirmausschnitte.

Library: -396(graphics.library)

Deklaration in C:

```
void ScrollRaster(rp,dx,dy,minx,miny,maxx,mxy)
struct RastPort *rp;
short dx,dy,minx,miny,maxx,mxy;
```

Aufruf in C: ScrollRaster(rp,dx,dy,minx,miny,maxx,mxy);

Aufruf in Assembler:

```
move.l rp,a1
move.w #dx,d0
move.w #dy,d1
move.w #minx,d2
move.w #miny,d3
move.w #maxx,d4
move.w #mxy,d5
move.l GfxBase,a6
jsr -396(a6)
```

Parameter:	rp:	Adresse einer RastPort-Struktur
	dx,dy:	Verschiebungsvektor
	minx,miny:	linke, obere Ecke des Rasters
	maxx,mxy:	rechte, untere Ecke

ScrollVPort

Zusammenfassung der Aufrufe von MakeVPort(), MrgCop() und LoadView() zur Neuberechnung des dargestellten Rasterausschnitts.

Library: -588(graphics.library)

Deklaration in C:

```
void ScrollVPort(vp)
    struct ViewPort *vp;
```

Aufruf in C: `ScrollVPort(vp);`

Aufruf in Assembler:

```
move.l vp,a0
move.l GfxBase,a6
jsr -588(a6)
```

Parameter: vp: Zeiger auf eine ViewPort-Struktur

Seek

Versetzt den internen Datenzeiger für die Schreib- und Leseoperationen des DOS.

Library: -66(dos.library)

Deklaration in C:

```
long Seek(file, pos, offset)
    struct FileHandle *file;
    long pos, offset;
```

Aufruf in C: oldpos = Seek(file, pos, offset);

Aufruf in Assembler:

```
move.l file,d1
move.l #pos,d2
move.l #offset,d3
move.l DosBase,a6
jsr -66(a6)
```

Parameter:

file:	Zeiger auf FileHandle-Struktur der untersuchten Datei
pos:	Anzahl der Byte, um die der Datenzeiger verschoben werden soll
offset:	Offset-Flag zur Berechnung der absoluten Position

Folgende Modi werden unterstützt:

OFFSET_BEGINNING:
+pos-Byte relativ zum Dateibeginn

OFFSET_END:
-pos-Byte relativ zum Dateende

OFFSET_CURRENT:
+pos-Byte relativ zur aktuellen Zeigerposition

oldpos (d0): alte Position des Dateizeigers als Abstand
zum Dateianfang oder -1 bei Abbruch

Anmerkung: Um die Länge einer Datei zu ermitteln, eignet sich folgende
Sequenz

```
Seek(file, 0L, OFFSET_END); Dateilänge =  
Seek(file, 0L, OFFSET_BEGINNING);
```


SendIO

Übergibt das in der IORequest-Struktur definierte Kommando dem Device zur Ausführung und kehrt sofort wieder zurück. (DoIO() wartet bis zum Ende des I/O-Zyklus.)

Library: -462(exec.library)

Deklaration in C:

```
void SendIO(iorequest)
    struct IORequest *iorequest;
```

Aufruf in C: SendIO(iorequest);

Aufruf in Assembler:

```
move.l iorequest,a1
move.l ExecBase,a6
jsr -462(a6)
```

Parameter:

iorequest:	Zeiger auf zuvor initialisierte IORequest-Struktur
------------	--

SetAPen

Setzt die Zeichenfarbe des Vordergrundstiftes (APen).

Library: -342(graphics.library)

Deklaration in C:

```
void SetAPen(rp, pen)
    struct RastPort *rp;
    short pen;
```

Aufruf in C: SetAPen(rp, pen);

Aufruf in Assembler:

```
move.l rp,a1
move.w #pen,d0
move.l Gfxbase,a6
jsr -342(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Zeichen-RastPorts
pen:	Nummer des Farbregisters, aus dem die Zeichenfarbe stammen soll

SetBPen

Setzt die Zeichenfarbe des Hintergrundstiftes (BPen).

Library: -348(graphics.library)

Deklaration in C:

```
void SetBPen(rp, pen)
    struct RastPort *rp;
    short pen;
```

Aufruf in C: SetBPen(rp, pen);

Aufruf in Assembler:

```
move.l rp,a1
move.w #pen,d0
move.l Gfxbase,a6
jsr -348(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur des Zeichen-RastPorts
pen:	Nummer des Farbregisters, aus dem die Zeichenfarbe stammen soll

SetCollision

Initialisiert den Collision-Handler der GelsInfo-Struktur mit dem Zeiger auf eine benutzerspezifische Routine, die bei erkannter Kollision von Grafikobjekten aufgerufen wird.

Library: -144(graphics.library)

Deklaration in C:

```
void SetCollision(num, routine, gelsinfo)
    ULONG num;
    void (*routine());
    struct GelsInfo *gelsinfo;
```

Aufruf in C: SetCollision(num, routine, gelsinfo);

Aufruf in Assembler:

```
move.l #num,d0
lea.l routine,a0
move.l gelsinfo,a1
move.l Gfxbase,a6
jsr -144(a6)
```

Parameter:	num:	Ordnungsnummer der Kollisionsroutine
	routine:	Adresse dieser Routine
	gelsinfo:	Zeiger auf eine GelsInfo-Struktur

SetComment

Versieht eine Datei mit einem Kommentarstring.

Library: -180(dos.library)

Deklaration in C:

```
BOOL SetComment(name, comment)
        UBYTE *name,*comment;
```

Aufruf in C:

```
success = SetComment(name, comment);
```

Aufruf in Assembler:

```
move.l name,d1
move.l comment,d2
move.l DosBase,a6
jsr -180(a6)
```

Parameter:

name:	Zeiger auf Dateiname
comment:	Adresse des Kommentarstrings
success:	FALSE bei Abbruch

SetCurrentBinding

Setzt Konfigurationsparameter.

Library: -132(expansion.library)

Deklaration in C:

```
void SetCurrentBinding
(currentbinding, bindingsize)
    struct CurrentBindig *currentbinding;
    ushort bindingsize;
```

Aufruf in C: SetCurrentBinding(currentbinding, bindingsize);

Aufruf in Assembler:

```
move.l currentbinding,a0
move.w #bindingsize,d0
move.l ExpansionBase,a6
jsr -132(a6)
```

Parameter:

- currentbinding: Adresse der CurrentBinding-Struktur
- bindingsize: Anzahl der Konfigurations-Byte

SetDMRequest

Macht den angegebenen Requester zu einem Double-Menu-Requester für das zugehörige Fenster.

Library: -258(intuition.library)

Deklaration in C:

```
BOOL SetDMRequest(window, requester)
    struct Window *window;
    struct Requester *requester;
```

Aufruf in C:

```
success = SetDMRequest(window, requester);
```

Aufruf in Assembler:

```
move.l window,a0
move.l requester,a1
move.l IntuitionBase,a6
jsr -258(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur
requester:	Adresse einer Requester-Struktur
success:	TRUE, falls noch kein anderer Requester als DMRequester definiert

SetDrMd

Setzt den Zeichenmodus im angegebenen RastPort.

Library: -354(graphics.library)

Deklaration in C:

```
void SetDrMd(rp, drawmode)
    struct RastPort *rp;
    short drawmode;
```

Aufruf in C: SetDrMd(rp, drawmode);

Aufruf in Assembler:

```
move.l rp,a1
move.w #drawmode,d0
move.l GfxBase,a6
jsr -354(a6)
```

Parameter:

rp:	Zeiger auf betreffenden RastPort
drawmode:	Flags für Zeichenmodus:

#define JAM1 0 => APen-Farbe

#define JAM2 1 => BPen-Farbe

#define

COMPLEMENT 2 => planeweises XOR

#define

INVERSVE 4 => invertierte Text-
darstellung

SetExcept

Setzt Signalbits für die Auslösung einer "Exception".

Library: -312(exec.library)

Deklaration in C: `ULONG SetExcept(newsignals, signalmask);`
`ULONG newsignals, signalmask;`

Aufruf in C: `oldsignals = SetExcept(newsignals, signalmask);`

Aufruf in Assembler: `move.l #newsignals,d0`
`move.l #signalmask,d1`
`move.l ExecBase,a6`
`jsr -312(a6)`

Parameter:

<code>newsignals:</code>	neue Signalbelegung für die Exception-Auslösung
<code>signalmask:</code>	Maskenwert (es werden nur diejenigen Signalbits belegt, die auch in signalmask gesetzt sind => logische UND-Verknüpfung)
<code>oldsignals (d0):</code>	alte Signalbelegung

SetFont

Ordnet dem RastPort einen bereits aktiven Zeichensatz zu.

Library: -66(graphics.library)

Deklaration in C:

```
void SetFont(rp, textfont)
    struct RastPort *rp;
    struct TextFont *textfont;
```

Aufruf in C: SetFont(rp, textfont);

Aufruf in Assembler:

```
move.l rp, a1
move.l textfont, a0
move.l GfxBase, a6
jsr -66(a6)
```

Parameter:

rp:	Adresse einer RastPort-Struktur
textfont:	Zeiger auf TextFont-Struktur des gewählten Zeichensatzes

SetFunction

Biegt den Einsprungvektor einer Library auf eine eigene Routine um.

Library: -420(exec.library)

Deklaration in C:

```
ULONG SetFunction
(library, funcoffset, funcentry)
    struct Library *library;
    long funcoffset;
    void (*funcentry());
```

Aufruf in C:

```
oldfuncentry = SetFunction
(library, funcoffset, funcentry);
```

Aufruf in Assembler:

```
move.l library,a1
move.l #funcoffset,a0
move.l funcentry,d0
move.l ExecBase,a6
jsr -420(a6)
```

Parameter:

library:	Zeiger auf eine Library-Struktur
funcoffset:	Offset der zu modifizierenden Funktion
funcentry:	neuer Einsprungvektor (zeigt auf eigene Routine)
oldfuncentry:	alte Sprungadresse

Anmerkung: SetFunction() berechnet automatisch die Library-Prüfsumme neu.

SetIntVector

Belegt einen System-Interrupt-Vektor.

Library: -162(exec.library)

Deklaration in C:

```
struct Node *SetIntVector(intnum, interrupt)
    long intnum;
    struct Node *interrupt;
```

Aufruf in C: `oldinterrupt = SetIntVector(intnum, interrupt);`

Aufruf in Assembler:

```
move.l #intnum,d0
move.l interrupt,a1
move.l ExecBase,a6
jsr -162(a6)
```

Parameter:

intnum:	Nummer des betreffenden System-Interrupts (Bit 0..4)
interrupt:	Zeiger auf eine Interrupt-Node-Struktur
oldinterrupt (d0):	Adresse der alten Interrupt-Node-Struktur

Anmerkung: Folgende Register dürfen nach den Konventionen der Systemsoftware verändert werden:

D0, D1, A0, A1, A5, A6

Alle anderen Register müssen vom Interrupt-Programm gesichert werden.

SetMenuStrip

Setzt Menüleiste in dem angegebenen Fenster.

Library: -264(intuition.library)

Deklaration in C:

```
BOOL SetMenuStrip(window, menu)
    struct Window *window;
    struct Menu *menu;
```

Aufruf in C:

```
success = SetMenuStrip(window, menu);
```

Aufruf in Assembler:

```
move.l window,a0
move.l menu,a1
move.l IntuitionBase,a6
jsr -264(a6)
```

Parameter:

window:	Zeiger auf eine Window-Struktur
menu:	Zeiger auf eine verkettete Liste von Menu-Strukturen, die die Menüleiste definieren
success:	immer TRUE

Anmerkung: Vor dem Schließen eines Fensters oder der Wahl einer neuen Menüleiste sollte ClearMenuStrip() aufgerufen werden.

SetPointer

Modifiziert die Darstellungsparameter des Mauszeigers.

Library: -270(intuition.library)

Deklaration in C:

```
void SetPointer
(window, pointer, height, width, hotx, hoty)
    struct Window *window;
    USHORT *pointer;
    short height, width, hotx, hoty;
```

Aufruf in C:

```
SetPointer
(window, pointer, height, width, hotx, hoty);
```

Aufruf in Assembler:

```
move.l window, a0
move.l pointer, a1
move.w #height, d0
move.w #width, d1
move.w #hotx, d2
move.w #hoty, d3
move.l IntuitionBase, a6
jsr -270(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur eines Fensters
pointer:	Adresse der Spritedaten des Mauszeigers
height:	Höhe des Sprites in Zeilen
width:	Breite in Pixels (<= 16)
hotx, hoty:	Abstand der sensitiven Punkte ("Hot Spots") von der linken, oberen Ecke des Zeigersprites

SetPrefs

Kopiert die Preference-Struktur ganz oder teilweise in einen vordefinierten Pufferspeicher (System-Preferences-Tabelle).

Library: -324(intuition.library)

Deklaration in C:

```
STRUCT PREFERENCES *SETPREFS  
(prefbuffer, size, flags)  
    struct Preferences *prefbuffer;  
    int size;  
    BOOL flags;
```

Aufruf in C: prefs = SetPrefs(prefbuffer, size, flags);

Aufruf in Assembler:

```
move.l prefbuffer,a0  
move.l #size,d0  
move.w #flags,d1  
move.l IntuitionBase,a6  
jsr -324(a6)
```

Parameter:

prefbuffer:	Zeiger auf System-Preference-Table
size:	Anzahl der zu kopierenden Bytes
flags:	TRUE, falls NEWPREFS gesetzt werden soll
prefs (d0):	= prefbuffer

SetProtection

Setzt Zugriffsschutz einer Datei (bzw. eines Verzeichnisses).

Library: -186(dos.library)

Deklaration in C:

```
BOOL SetProtection(name, mask)
    UBYTE *name;
    long mask;
```

Aufruf in C: success = SetProtection(name, mask);

Aufruf in Assembler:

```
move.l name,d1
move.l mask,d2
move.l DosBase,a6
jsr -186(a6)
```

Parameter:

name:	Bezeichnung der zu schützenden Datei
mask:	Schutzmaske (gesetzte Bits bewirken: Löscheschutz (Bit 0), Schutz, gegen Ausführung (Bit 1), Schreibschutz (Bit 2), Leseschutz (Bit 3), Archivierungsbit (Bit 4))
success (d0):	FALSE bei Abbruch

SetRast

Setzt den kompletten RastPort auf die spezifizierte Farbe.

Library: -234(graphics.library)

Deklaration in C:

```
void SetRast(rp, color)
    struct RastPort *rp;
    UBYTE color;
```

Aufruf in C: SetRast(rp, color);

Aufruf in Assembler:

```
move.l rp,a1
move.b #color,d0
move.l GfxBase,a6
jsr -234(a6)
```

Parameter:

rp:	Zeiger auf betroffenen RastPort
color:	Nummer des Farbregisters, aus dem die Farbparameter geholt werden sollen

SetRGB4

Modifiziert den Inhalt eines bestimmten Farbregisters im ViewPort, trägt die Veränderung in der ColorMap ein und erzwingt die Neuberechnung der Copper-Befehlsliste(n).

Library: -288(graphics.library)

Deklaration in C:

```
void SetRGB4(vp, fregister, r, g, b)
    struct ViewPort *vp;
    short fregister;
    UBYTE r, g, b;
```

Aufruf in C: SetRGB4(vp, fregister, r, g, b);

Aufruf in Assembler:

```
move.l vp, a0
move.w #fregister, d0
move.b #r, d1
move.b #g, d2
move.b #b, d3
move.l GfxBase, a6
jsr -288(a6)
```

Parameter:

vp:	Zeiger auf eine ViewPort-Struktur
fregister:	Nummer des zu modifizierenden Farbregisters
r,g,b:	neue RGB-Farbparameter

SetRGB4CM

Setzt neue Parameter für ein Farbbregister und trägt sie in der ColorMap ein, erzwingt aber (im Gegensatz zu SetRGB4()) keine Neuberechnung der Copper-Befehlsliste(n).

Library: -630(graphics.library)

Deklaration in C:

```
void SetRGB4CM(cm, color, r, g, b)
    struct ColorMap *cm;
    short fregister;
    UBYTE r, g, b;
```

Aufruf in C: SetRGB4CM(vp, fregister, r, g, b);

Aufruf in Assembler:

```
move.l cm,a0
move.w #fregister,d0
move.b #r,d1
move.b #g,d2
move.b #b,d3
move.l GfxBase,a6
jsr -630(a6)
```

Parameter:

cm:	Zeiger auf eine ColorMap-Struktur
fregister:	Nummer des zu modifizierenden Farbbregisters
r,g,b:	neue RGB-Farbparameter

SetSignal

Setzt neue Signalbelegung für den aufrufenden Task.

Library: -306(exec.library)

Deklaration in C:

```
ULONG SetSignal(newsignals, signalmask)
        ULONG newsignals, signalmask;
```

Aufruf in C: `oldsignals = SetSignal(newsignals, signalmask);`

Aufruf in Assembler:

```
move.l #newsignals,d0
move.l #signalmask,d1
move.l ExecBase,a6
jsr -306(a6)
```

Parameter:	newsignals:	neue Signalbelegung
	signalmask:	Maskenwert (es werden nur diejenigen Bits berücksichtigt, die im Maskenlangwort gesetzt sind)
	oldsignals (d0):	alter Signalvektor

SetSoftStyle

Definiert softwaremäßig einstellbare Schriftmodi eines Zeichensatzes.

Library: -90(graphics.library)

Deklaration in C:

```
ULONG SetSoftStyle(rp, styleflags, styleenable)
struct RastPort *rp;
ULONG styleflags, styleenable;
```

Aufruf in C:

```
newstyle = SetSoftStyle
(rp, styleflags, styleenable);
```

Aufruf in Assembler:

```
move.l rp,a1
move.l #styleflags,d0
move.l #styleenable,d1
move.l GfxBase,a6
jsr -90(a6)
```

Parameter:

rp:	Zeiger auf eine RastPort-Struktur
styleflags:	unterstützte Soft-Style-Flags (ermittelt durch Aufruf von AskSoftstyle(), z.B. FFS_UNDERLINED, FFS_BOLD, FFS_ITALIC, FFS_NORMAL)
styleenable:	gewünschte Schriftart
newstyle (d0):	effektiv belegte Schriftart, in der Textausgaben dargestellt werden

SetSR

Modifiziert das Statusregister (kurz SR) der 68000-CPU.

Library: -144(exec.library)

Deklaration in C:

```
long SetSR(news, mask)
    long news, mask;
```

Aufruf in C:

```
oldsr = SetSR(news, mask);
```

Aufruf in Assembler:

```
move.l #news,d0
move.l #mask,d1
move.l ExecBase,a6
jsr -144(a6)
```

Parameter:	news:	neuer Registerwert
	mask:	Maskenwert (gesetzte Bits können verändert werden, alle anderen sind gesperrt)
	oldsr (d0):	alter Wert der SR-Registers

SetTaskPri

Modifiziert die Ausführungspriorität eines Tasks.

Library: -300(exec.library)

Deklaration in C:

```
BYTE SetTaskPri(task, priority)
    struct Task *task;
    BYTE priority;
```

Aufruf in C: `oldpriority = SetTaskPri(task, priority);`

Aufruf in Assembler:

```
move.l task,a1
move.b #priority,d0
move.l ExecBase,a6
jsr -300(a6)
```

Parameter:	task:	Zeiger auf eine Task-Struktur
	priority:	neue Priorität
	oldpriority (d0):	alte Priorität

SetWindowTitles

Modifiziert die Titelleiste eines Fensters und/oder eines Screens.

Library: -276(intuition.library)

Deklaration in C:

```
void SetWindowTitles
(window, windowtitle, *screentitle)
    struct Window *window;
    UBYTE *windowtitle, *screentitle;
```

Aufruf in C:

```
SetWindowTitles
(window, windowtitle, screentitle);
```

Aufruf in Assembler:

```
move.l window, a0
move.l windowtitle, a1
move.l screentitle, a2
move.l IntuitionBase, a6
jsr -276(a6)
```

Parameter:

window:	Zeiger auf eine Window-Struktur
windowtitle:	Zeiger auf den nullterminierten Titelstring des Windows
screentitle:	Zeiger auf nullterminierten Titelstring des Screens

Anmerkung:

Setzt man windowtitle bzw. screentitle auf NULL, so wird eine Blank-Leiste erzeugt, übergibt man den Wert -1, bleibt der ursprüngliche Titel erhalten.

ShowTitle

Blendet Titelleiste ein (eventuell durch ein BACKDROP-Window verdeckt).

Library: -282(intuition.library)

Deklaration in C:

```
void ShowTitle(screen, show)
    struct Screen *screen;
    BOOL show;
```

Aufruf in C: ShowTitle(screen, show);

Aufruf in Assembler:

```
move.l screen,a0
move.w #show,d0
move.l IntuitionBase,a6
jsr -282(a6)
```

Parameter:

screen:	Zeiger auf betroffene Screen-Struktur
show:	Einblenden (ja = TRUE / nein = FALSE)

Signal

Sendet ein Signal an den durch die Task-Struktur definierten Empfänger task.

Library: -324(exec.library)

Deklaration in C:

```
void Signal(task, signalset)
    struct task *task;
    ULONG signalset;
```

Aufruf in C: Signal(task, signalset);

Aufruf in Assembler:

```
move.l task,a1
move.l #signalset,d0
move.l ExecBase,a6
jsr -324(a6)
```

Parameter:

task:	Zeiger auf Task-Struktur des Empfänger tasks
signalset:	Signalvektor

SizeCList

Ermittelt die Größe einer Character-Liste.

Library: -54(clist.library)

Deklaration in C:

```
long SizeCList(clist)
long clist;
```

Aufruf in C: `bytes = SizeCList(clist);`

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a6
jsr -54(a6)
```

Parameter:

clist:	Identifikationsnummer des CList-Headers
bytes (d0):	Speicherbedarf in Byte

SizeLayer

Modifiziert die Größe eines Layers.

Library: -66(layers.library)

Deklaration in C:

```
void SizeLayer(li, layer, x, y)
    struct Layer_info *li;
    struct Layer *layer;
    short x, y;
```

Aufruf in C:

```
move.l li,a0
move.l layer,a1
move.w #x,d0
move.w #y,d1
move.l LayersBase,a6
jsr -66(a6)
```

Parameter:

li:	Adresse einer Layer_Info-Struktur
layer:	Zeiger auf Layer-Struktur
dx,dy:	vertikale und horizontale Offsets, die zur aktuellen Höhe bzw. Breite hinzuaddiert werden

SizeWindow

Verändert die Größe des angegebenen Fensters.

Library: -288(intuition.library)

Deklaration in C:

```
void SizeWindow(window, dx, dy)
    struct Window *window;
    short dx, dy;
```

Aufruf in C: SizeWindow(window, dx, dy);

Aufruf in Assembler:

```
move.l window, a0
move.w #dx, d0
move.w #dy, d1
move.l IntuitionBase, a6
jsr -288(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur des betreffenden Fensters
dx,dy:	Pixelwert, um den das Fenster größer bzw. kleiner (bei negativen Werten) wird

Anmerkung: SizeWindow() nimmt keine Prüfung der Parameter vor.

SortGList

Sortiert sämtliche Elemente einer GEL-Liste.

Library: -150(graphics.library)

Deklaration in C:

```
void SortGList(rp)
    struct RastPort *rp;
```

Aufruf in C: SortGList(rp);

Aufruf in Assembler:

```
move.l rp,a1
move.l GfxBase,a6
jsr -150(a6)
```

Parameter: rp: Zeiger auf RastPort-Struktur des Darstellungs-RastPorts

Anmerkung: SortGList() sollte vor jedem Aufruf von DrawGList() ausgeführt werden.

SplitCList

Teilt eine CListe an der aktuellen Markierung.

Library: -138(clist.library)

Deklaration in C:

```
long SplitCList(clist)
long clist;
```

Aufruf in C:

```
endclist = SplitCList(clist);
```

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a6
jsr -138(a6)
```

Parameter:

clist:	Identifikations-Header einer Character-Liste (enthält nach erfolgreicher Operation den ersten Teil der CListe)
clistend (d0):	zweiter Teil der CListe (einschließlich Markierungsbyte)

SPAbs

Liefert den Betrag einer Fließkommazahl.

Library: -54(mathffp.library)

Deklaration in C:

```
float SPAbs (num)
float num;
```

Aufruf in C:

```
result = SPAbs (num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathBase,a6
jsr -54(a6)
```

Parameter:

num:	einfach-genaue Fließkommazahl
result (d0):	ihr Betrag

SPAcos

Berechnet den arcus cosinus.

Library: -120(mathtrans.library)

Deklaration in C:

```
float SPAcos (num)
float num;
```

Aufruf in C: result = SPAcos (num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -120(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	arcus cosinus von num

SPAdd

Addiert zwei Fließkommazahlen.

Library: -66(mathffp.library)

Deklaration in C:

```
float SPAdd(num1, num2)
float num1, num2;
```

Aufruf in C:

```
result = SPAdd(num1, num2);
```

Aufruf in Assembler:

```
move.l #num1,d1
move.l #num2,d0
move.l MathBase,a6
jsr -66(a6)
```

Parameter:

num1, num2:	zwei einfach-genaue Fließkommazahlen
result (d0):	Ergebnis

SPAsin

Berechnet arcus sinus.

Library: -114(mathtrans.library)

Deklaration in C: float SPAsin(num)
float num;

Aufruf in C: result = SPAsin(num)

Aufruf in Assembler: move.l #num,d0
move.l MathTransBase,a6
jsr -114(a6)

Parameter: num: eine einfach-genaue Fließkommazahl
result (d0): Ergebnis

SPAtan

Berechnet arcus tangens.

Library: -30(mathtrans.library)

Deklaration in C:

```
float SPAtan(num)
    float num;
```

Aufruf in C:

```
result = SPAtan(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -30(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPCeil

Liefert obere Schranke zu num (kleinste Zahl, die größer oder gleich num ist).

Library: -96(mathffp.library)

Deklaration in C:

```
float SPCeil(num)
float num;
```

Aufruf in C: result = SPCeil(num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathBase,a6
jsr -96(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPCmp

Vergleicht zwei Fließkommazahlen.

Library: -42(mathffp.library)

Deklaration in C: long SPCmp(num1, num2)
 float num1, num2;

Aufruf in C: result = SPCmp(num1, num2);

Aufruf in Assembler: move.l #num1,d1
 move.l #num2,d0
 move.l MathBase,a6
 jsr -42(a6)

Parameter: num1,num2: zwei einfach-genaue Fließkommazahlen
 result (d0): Ergebnis

D0: | Bedeutung:

1	arg1 > arg2
0	arg1 = arg2
-1	arg1 < arg2

Anmerkung:

Cond: | Bedeutung:

EQ	$\arg2 = \arg1$
NE	$\arg2 \neq \arg1$
GR	$\arg2 > \arg1$
GE	$\arg2 \geq \arg1$
LT	$\arg2 < \arg1$
LE	$\arg2 \leq \arg1$

SPCos

Berechnet cosinus.

Library: -42(mathtrans.library)

Deklaration in C:

```
float SPCos(num)
float num;
```

Aufruf in C:

```
result = SPCos(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -42(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPCosh

Liefert cosinus hyperbolicus von num.

Library: -66(mathtrans.library)

Deklaration in C: float SPCosh(num)
float num;

Aufruf in C: result = SPCosh(num);

Aufruf in Assembler: move.l #num,d0
move.l MathTransBase,a6
jsr -66(a6)

Parameter: num: eine einfach-genaue Fließkommazahl
result (d0): Ergebnis

SPDiv

Dividiert zwei Fließkommazahlen.

Library: -84(mathffp.library)

Deklaration in C:

```
float SPDiv(num1, num2)
    float num1, num2;
```

Aufruf in C:

```
result = SPDiv(num1, num2);
```

Aufruf in Assembler:

```
move.l #num1,d1
move.l #num2,d0
move.l MathBase,a6
jsr -84(a6)
```

Parameter:

num1, num2:	zwei einfach-genaue Fließkommazahlen
result (d0):	Ergebnis der Division

SPExp

e^{num} .

Library: -78(mathtrans.library)

Deklaration in C:

```
float SPExp(num)
float num;
```

Aufruf in C:

```
result = SPExp(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -78(a6)
```

Parameter:
num: eine einfach-genaue Fließkommazahl
result (d0): Ergebnis

SPFieee

Konvertiert eine Integerzahl in ein einfach-genaues Fließkomma-Format.

Library: -108(mathtrans.library)

Deklaration in C:

```
float SPFieee(intnum)
        LONG intnum;
```

Aufruf in C:

```
result = SPFieee(intnum);
```

Aufruf in Assembler:

```
move.l intnum,d0
move.l MathTransBase,a6
jsr -108(a6)
```

Parameter:

intnum:	Integerzahl
result (d0):	Zahl im FFP-Format

SPFix

Wandelt eine Fließkommazahl in ein Integer-Format um.

Library: -30(mathffp.library)

Deklaration in C:

```
long SPFix(num)
float num;
```

Aufruf in C: result = SPFix(num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathBase,a6
jsr -30(a6)
```

Parameter:

num:	FFP-Zahl
result (d0):	Integerzahl

SPFloor

Bildet untere Schranke zu num (größte Zahl, die kleiner oder gleich num ist).

Library: -90(mathffp.library)

Deklaration in C:

```
float SPFloor(num)
float num;
```

Aufruf in C:

```
result = SPFloor(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathBase,a6
jsr -90(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPFIt

go.J92

Konvertierung vom Integer- ins Fließkommaformat.

Library: -36(mathffp.library)

Deklaration in C: float SPFIt(integer)
long intnum;

Aufruf in C: result = SPFIt(intnum);

Aufruf in Assembler: move.l intnum,d0
move.l MathBase,a6
jsr -36(a6)

Parameter: intnum: Integerzahl
result (d0): Zahl im FFP-Format

SPLog

Berechnet den natürlichen Logarithmus.

Library: -84(mathtrans.library)

Deklaration in C:

```
float SPLog(num)
float num;
```

Aufruf in C:

```
result = SPLog(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -84(a6)
```

Parameter:
num: eine einfach-genaue Fließkommazahl
result (d0): Ergebnis

SPLog10

Berechnet Logarithmus zur Basis 10.

Library: -126(mathtrans.library)

Deklaration in C:

```
float SPLog10(num)
float num;
```

Aufruf in C: `result = SPLog10(num);`

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTansBase,a6
jsr -126(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPMul

Multipliziert zwei Fließkommazahlen.

Library: -78(mathffp.library)

Deklaration in C:

```
float = SPMul(num1, num2)
float num1, num2;
```

Aufruf in C:

```
result = SPMul(num1, num2);
```

Aufruf in Assembler:

```
move.l #num1,d1
move.l #num2,d0
move.l MathBase,a6
jsr -78(a6)
```

Parameter:

num1,num2:	zwei einfach-genaue Fließkommazahlen
result (d0):	Ergebnis

SPNeg

Negation von num (Vorzeichenwechsel).

Library: -60(mathffp.library)

Deklaration in C:

```
float SPNeg (num)
float num;
```

Aufruf in C:

```
result = SPNeg (num) ;
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathBase,a6
jsr -60(a6)
```

Parameter:

num:	eine einfach-genaue Fließkommazahl
result (d0):	Ergebnis

SPPow

$\text{num1}^{\text{num2}}$.

Library: -90(mathtrans.library)

Deklaration in C:

```
float SPPow(num1, num2)
float num1, num2;
```

Aufruf in C:

```
result = SPPow(num1, num2);
```

Aufruf in Assembler:

```
move.l #num2,d0
move.l #num1,d1
move.l MathTransBase,a6
jsr -90(a6)
```

Parameter:

num1,num2:	zwei einfach-genaue Fließkommazahlen
result (d0):	Ergebnis

SPSin

Berechnet sinus von num.

Library: -36(mathtrans.library)

Deklaration in C: float SPSin(num)
float num;

Aufruf in C: result = SPSin(num);

Aufruf in Assembler: move.l #num,d0
move.l MathtransBase,a6
jsr -36(a6)

Parameter: num: Zahl im FFP-Format
result (d0): Ergebnis

SPSincos

Berechnet sinus und cosinus einer Fließkommazahl.

Library: -54(mathtrans.library)

Deklaration in C:

```
float SPSincos(num1, *num2)
float num1, *num2;
```

Aufruf in C:

```
result = SPSincos(num1, num2);
```

Aufruf in Assembler:

```
move.l num2,d0
move.l #num1,d1
move.l MathTransBase,a6
jsr -54(a6)
```

Parameter:

num1:	eine einfach-genaue Fließkommazahl
num2:	Zeiger auf Puffer für cos(num1)
result (d0):	sinus(num1)

SPSinh

Berechnet den sinus hyperbolicus.

Library: -60(mathtrans.library)

Deklaration in C:

```
float SPSinh(num)
float num;
```

Aufruf in C:

```
result = SPSinh(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -60(a6)
```

Parameter:

num:	Zahl im FFP-Format
result (d0):	Ergebnis

SPSqrt

Liefert Quadratwurzel von num.

Library: -96(mathtrans.library)

Deklaration in C:

```
float SPSqrt(num)
float num;
```

Aufruf in C: result = SPSqrt(num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathtansBase,a6
jsr -96(a6)
```

Parameter:

num:	Zahl im FFP-Format
result (d0):	Ergebnis

SPSub

num1-num2.

Library: -72(mathffp.library)

Deklaration in C:

```
float SPSub(num1, num2)
float num1, num2;
```

Aufruf in C: result = SPSub(num1, num2);

Aufruf in Assembler:

```
move.l #num2,d0
move.l #num1,d1
move.l MathBase,a6
jsr -72(a6)
```

Parameter:

num1,num2:	zwei einfach-genaue Fließkommazahlen
result (d0):	Ergebnis

SPTan

Berechnet den tangens von num.

Library: -48(mathtrans.library)

Deklaration in C:

```
float SPTan (num)
float num;
```

Aufruf in C: result = SPTan(num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -48(a6)
```

Parameter:

num:	Zahl im FFP-Format
result (d0):	Ergebnis

SPTanh

Berechnet den tangens hyperbolicus.

Library: -72(mathtrans.library)

Deklaration in C:

```
float SPTanh(num)
float num;
```

Aufruf in C: result = SPTanh(num);

Aufruf in Assembler:

```
move.l #num,d0
move.l MathtransBase,a6
jsr -72(a6)
```

Parameter:

num:	Zahl im FFP-Format
result (d0):	Ergebnis

SPTieee

Wandelt eine einfach-genaue Fließkommazahl in ein doppelt-genaues Format um.

Library: -102(mathtrans.library)

Deklaration in C:

```
IEEEDouble SPTieee(num)
float num;
```

Aufruf in C:

```
result = SPTieee(num);
```

Aufruf in Assembler:

```
move.l #num,d0
move.l MathTransBase,a6
jsr -102(a6)
```

Parameter:

num:	einfach-genaue Fließkommazahl
result (d0/d1):	doppelt-genaues Format

SPTst

Prüft auf Wert 0.

Library: -48(mathffp.library)

Deklaration in C: `LONG SPTst (num)
float num;`

Aufruf in C: `result = SPTst (num);`

Aufruf in Assembler: `move.l #num,d1
move.l MathBase,a6
jsr -48(a6)`

Parameter: `num:` FFP-Zahl, die mit 0 verglichen wird
`result (d0):` Ergebnis der Prüfung

D0: | Bedeutung:

1 | arg > 0.0

0 | arg = 0.0

-1 | arg < 0.0

Anmerkung:

Cond: | Bedeutung:

EQ | arg = 0.0

NE | arg != 0.0

PL | arg >= 0.0

MI | arg < 0.0

SubCList

Kopiert Ausschnitte einer Character-Liste in eine neue CListe.

Library: -150(clist.library)

Deklaration in C:

```
long SubCList(clist, index, length)
    long clist;
    long index, length;
```

Aufruf in C: `neueclist = SubCList(clist, index, length);`

Aufruf in Assembler:

```
move.l clist,a0
move.l #index,d0
move.l #length,d1
move.l CListBase,a6
jsr -150(a6)
```

Parameter:

clist:	CList-Header
index:	Indexmarke ab der kopiert wird
length:	Anzahl der zu kopierenden Byte
neueclist (d0):	Header der neu angelegten Character-Liste

SubTime

Subtrahiert die Zeitangaben zweier timeval-Strukturen.

Library: -48(timer.library)

Deklaration in C:

```
void SubTime(dest, src)
    struct timeval *dest, *src;
```

Aufruf in C: SubTime(dest, src)

Aufruf in Assembler:

```
move.l dest, a0
move.l src, a1
move.l TimerBase, a6
jsr -48(a6)
```

Parameter:

dest:	Zeiger auf Quell-timeval-Struktur
src:	Zeiger auf Ziel-timeval-Struktur (wird mit dem Ergebnis überschrieben)

SumKickData

Ermittelt die Prüfsumme der KickMemPtr-Struktur und des KickTagPtr-Feldes und schreibt das Ergebnis in das KickChecksum-Feld der ExecBase-Struktur.

Library: -612(exec.library)

Deklaration in C: void SumKickData()

Aufruf in C: SumKickData();

SumLibrary

Berechnet erneut die Prüfsumme einer Library. Stimmt die Summe nicht, so wird der Anwender durch eine Alert-Meldung informiert.

Library: -426(exec.library)

Deklaration in C: void SumLibrary(library)
struct Library *library;

Aufruf in C: SumLibrary(library);

Aufruf in Assembler: move.l library,a1
move.l ExecBase,a6
jsr -426(a6)

Parameter: library: Adresse der Library-Struktur.

SuperState

Aktiviert den Supervisor-Mode der CPU.

Library: -150(exec.library)

Deklaration in C: long SuperState();

Aufruf in C: oldSysStack = SuperState();

Parameter: oldSysStack (d0):alter SystemStack-Zeiger (wichtig für die Rückkehr in den Usermode)

Anmerkung: Falls der Prozessor schon im Supervisor-Modus ist, wird als oldSysStack NULL zurückgegeben.

Supervisor

Library: -30(exec.library)

Deklaration in C: void Supervisor()

Aufruf in C: Supervisor();

SwapBitsRastPortClipRect

Vertauscht die Inhalte eines RastPorts und eines Clipping-Rechtecks.

Library: -126(layers.library)

Deklaration in C:

```
void SwapBitsRastPortClipRect(rp, cr)
    struct RastPort *rp;
    struct ClipRect *cr;
```

Aufruf in C: SwapBitsRastPortClipRect(rp, cr);

Aufruf in Assembler:

```
move.l rp,a0
move.l cr,a1
move.l LayersBase,a6
jsr -126(a6)
```

Parameter:

rp:	Zeiger auf eine RastPort-Struktur
cr:	Zeiger auf eine ClipRect-Struktur

Switch

Unterstützungsroutine für den Task-Scheduler.

Library: -54(exec.library)

Deklaration in C: void Switch()

Aufruf in C: Switch();

SyncSBitMap

Kopiert die aktive Bitmap in die Super-Bitmap des LAYERSUPER-Layers.

Library: -444(graphics.library)

Deklaration in C: void SyncSBitMap(layer)
struct Layer *layer;

Aufruf in C: SyncSBitMap(layer);

Aufruf in Assembler: move.l layer,a0
move.l GfxBase,a6
jsr -444(a6)

Parameter: layer: Zeiger auf eine Layer-Struktur

Text

Standard-Textausgaberroutine im RastPort.

Library: -60(graphics.library)

Deklaration in C:

```
void Text(rp, string, count)
    struct RastPort *rp;
    char *string;
    short count;
```

Aufruf in C: Text(rp, string, count);

Aufruf in Assembler:

```
move.l rp,a1
move.l string,a0
move.w #count,d0
move.l GfxBase,a6
jsr -60(a6)
```

Parameter:

rp:	Adresse einer RastPort-Struktur
string:	Zeiger auf einen nullterminierten Textstring
count:	Anzahl der Zeichen, die der String enthält

Anmerkung: Die maximale Pixelbreite des Strings ist auf 1008 begrenzt.

TextLength

Ermittelt die Breite eines Textstrings in Pixeln.

Library: -54(graphics.library)

Deklaration in C:

```
short TextLength(rp, string, count)
struct RastPort *rp;
char *string;
short count;
```

Aufruf in C: length = TextLength(rp, string, count);

Aufruf in Assembler:

```
move.l rp,a1
move.l string,a0
move.l #count,d0
move.l GfxBase,a6
jsr -54(a6)
```

Parameter:

rp:	Zeiger auf RastPort-Struktur
string:	Zeiger auf Stringzeichen
count:	Anzahl der Zeichen, die der String enthält
length (d0):	horizontale Ausdehnung in Pixeln

ThinLayerInfo

Library: -162(layers.library)

Deklaration in C:

```
void ThinLayerInfo(li)
    struct Layers_Info *li;
```

Aufruf in C: `ThinLayerInfo(li);`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a6
jsr -162(a6)
```

Parameter: li: Adresse der Layer_Info-Struktur

Anmerkung: Diese Funktion sollte nicht mehr verwendet werden.
(Alternative: DisposeLayerInfo())

Translate

Transformiert einen englischen Satz in eine Lautschriftdarstellung, wie sie vom Translator-Device verarbeitet wird.

Library: -30(translator.library)

Deklaration in C:

```
long Translate
(input, inputlength, outbuf, bufsize)
    UBYTE *input;
    long inputlength;
    UBYTE *outbuf;
    long bufsize;
```

Aufruf in C:

```
error = Translate
(input, inputlength, outbuf, bufsize);
```

Aufruf in Assembler:

```
move.l input,a0
move.l #inputlength,d0
move.l outbuf,a1
move.l #bufsize,d1
move.l TranslatorBase,a6
jsr -30(a6)
```

Parameter:	input:	Zeiger auf einen nullterminierten Textstring
	inputLength:	Länge dieses Strings
	outBuf:	Zeiger auf den Puffer, in den die Phonem-codes abgelegt werden
	bufSize:	Länge dieses Puffers
	error (d0):	NULL bei erfolgreicher Bearbeitung, sonst negativ (der Betrag dieses Wertes markiert die Stelle, an der der Ausgabepuffer über-gelaufen ist).

TypeOfMem

Ermittelt die Attribute des Speicherbereichs an der angegebenen Adresse.

Library: -534(exec.library)

Deklaration in C:

```
long TypeOfMem(address)
      ULONG address;
```

Aufruf in C: `attributes = TypeOfMem(address);`

Aufruf in Assembler:

```
move.l address,a1
move.l ExecBase,a6
jsr -534(a6)
```

Parameter: `address:` Adresse des Speicherbereichs
`attributes (d0):` Attribute dieses Speicherbereichs

UCopperListInit

Belegt Speicherplatz für eine neue Copper-Befehlsliste oder initialisiert eine bereits erstellte Liste.

Library: -594(graphics.library)

Deklaration in C:

```
struct UCopList *UCopperListInit
(copperlist, num)
    struct UCopList *copperlist;
    short num;
```

Aufruf in C: `ucoplist = UCopperListInit(copperlist, num);`

Aufruf in Assembler:

```
move.l copperlist,a0
move.w #num,d0
move.l GfxBase,a6
jsr -594(a6)
```

Parameter:

<code>copperlist:</code>	Zeiger auf eine bereits vorhandene Copper-List-Struktur oder NULL, falls die Funktion eine neue List erstellen soll
<code>num:</code>	Anzahl der Befehle in der User-Copperliste

UnGetCLChar

Hängt ein Byte an den Anfang der Character-Liste.

Library: -72(clist.library)

Deklaration in C:

```
long UnGetCLChar(clist, value)
    long clist;
    BYTE value;
```

Aufruf in C:

```
error = UnGetCLChar(clist, value);
```

Aufruf in Assembler:

```
move.l clist,a0
move.b #value,d0
move.l CListBase,a6
jsr -72(a6)
```

Parameter:

clist:	CList-Header
value:	Datenbyte
error (d0):	NULL bei erfolgreicher Operation

UnGetCLWord

Fügt ein Wort an den Anfang der angegebenen Character-Liste ein.

Library: -96(clist.library)

Deklaration in C:

```
long UnGetCLWord(clist, value)
long clist;
ULONG word;
```

Aufruf in C: error = UnGetCLWord(clist, word);

Aufruf in Assembler:

```
move.l clist,a0
move.l #value,d0
move.l CListBase,a6
jsr -96(a6)
```

Parameter:

clist:	CList-Header
word:	einzufügendes Datenwort
error (d0):	ungleich NULL bei Abbruch

UnLoadSeg

Gibt ein geladenes Segment frei.

Library: -156(dos.library)

Deklaration in C: `BOOL UnLoadSeg(segment)
BPTR segment;`

Aufruf in C: `error = UnLoadSeg(segment);`

Aufruf in Assembler: `move.l segment,d1
move.l DosBase,a6
jsr -156(a6)`

Parameter:
`segment:` BCPL-Zeiger auf Segment
`error (d0):` FALSE, wenn Fehler aufgetreten ist

UnLock

Lock-Freigabe.

Library: -90(dos.library)

Deklaration in C: `void UnLock(lock)
struct FileLock *lock;`

Aufruf in C: `UnLock(lock);`

Aufruf in Assembler: `move.l lock,d1
move.l DosBase,a6
jsr -90(a6)`

Parameter: `lock:` Zeiger auf FileLock-Struktur des freizugebenden Locks

UnlockIBase

Gibt den IntuitionBase-Lock (LockIBase()) wieder frei.

Library: -420(intuition.library)

Deklaration in C: void UnlockIBase(iblock)
 ULONG iblock;

Aufruf in C: UnlockIBase(iblock);

Aufruf in Assembler: move.l iblock,a0
 move.l IntuitionBase,a6
 jsr -420(a6)

Parameter: iblock: Adresse des IntuitionBase-Locks

Anmerkung: Ein Aufruf dieser Funktion mit falscher Lockadresse führt zu einem Absturz!

UnlockLayer

Gibt das gelockte Layer für den allgemeinen Zugriff frei.

Library: -102(layers.library)

Deklaration in C:

```
void UnlockLayer(layer)
    struct Layer *layer;
```

Aufruf in C: `UnlockLayer(layer);`

Aufruf in Assembler:

```
move.l layer,a0
move.l LayersBase,a6
jsr -102(a6)
```

Parameter: `layer:` Zeiger auf eine Layer-Struktur

UnlockLayerInfo

Hebt den Lock der Layer_Info-Struktur wieder auf.

Library: -138(layers.library)

Deklaration in C:

```
void UnlockLayerInfo(li)
    struct Layer_Info *li;
```

Aufruf in C: `UnlockLayerInfo(li)`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a6
jsr -138(a6)
```

Parameter: li: Adresse der Layer_Info-Struktur

Anmerkung: Die Funktionen der Layers-Library benötigen keinen Aufruf von LockLayerInfo() / UnlockLayerInfo(); lediglich bei eigenen Routinen sollte der kritische Abschnitt auf diese Weise gesichert werden.

UnlockLayerRom

Gibt Layer-Lock wieder frei.

Library: -438(graphics.library)

Deklaration in C:

```
void UnlockLayerRom(layer)
    struct Layer *layer;
```

Aufruf in C: `UnlockLayerRom(layer);`

Aufruf in Assembler:

```
move.l layer, a5
move.l GfxBase, a6
jsr -438(a6)
```

Parameter: `layer:` Zeiger auf eine Layer-Struktur

UnlockLayers

Library: -114(layers.library)

Deklaration in C:

```
void UnlockLayers(li)
    struct Layer_Info *li;
```

Aufruf in C: `UnlockLayers(li)`

Aufruf in Assembler:

```
move.l li,a0
move.l LayersBase,a6
jsr -114(a6)
```

Parameter: `li:` Adresse der Layers_Info-Struktur

UnPutCLChar

Holt ein Byte vom Beginn der Character-Liste.

Library: -78(clist.library)

Deklaration in C: long UnPutCLChar(cList)
 long clist;

Aufruf in C: error = UnPutCLChar(clist);

Aufruf in Assembler: move.l clist,a0
 move.l CListBase,a6
 jsr -78(a6)

Parameter: cList: CList-Header
 error (d0): NULL bei erfolgreicher Operation

UnPutCLWord

Holt ein Wort vom Beginn der Character-Liste.

Library: -102(clist.library)

Deklaration in C:

```
short UnPutCLWord(clist)
long clist;
```

Aufruf in C:

```
word = UnPutCLWord(clist);
```

Aufruf in Assembler:

```
move.l clist,a0
move.l CListBase,a6
jsr -102(a6)
```

Parameter:

clist:	Zeiger auf CList-Struktur
word:	Daten, die gelesen wurden

UpfrontLayer

Bringt das angegebene Layer in den Vordergrund. (Handelt es sich um ein BACKDROP-Layer, so erscheint dieses zwar vor allen BACKDROP-Layern, aber hinter den Layern eines anderen Typs.)

Library: -48(layers.library)

Deklaration in C:

```
BOOL UpfrontLayer(li, layer)
    struct Layer_Info *li;
    struct Layer *layer;
```

Aufruf in C: result = UpfrontLayer(li, layer)

Aufruf in Assembler:

```
move.l li,a0
move.l layer,a1
move.l LayersBase,a6
jsr -48(a6)
```

Parameter:

li:	Zeiger auf Layer_Info-Struktur
layer:	Zeiger auf Layer-Struktur
result (d0):	FALSE bei Abbruch aus Speicherplatzmangel

UserState

Versetzt die CPU wieder in den Usermode.

Library: -156(exec.library)

Deklaration in C:

```
void UserState(OldSysStack)
    ULONG OldSysStack;
```

Aufruf in C: UserState(OldSysStack);

Aufruf in Assembler:

```
move.l OldSysStack,a0
move.l ExecBase,a6
jsr -156(a6)
```

Parameter: OldSysStack: Stackpointer von SuperState()

Anmerkung: Diese Funktion darf nicht im Usermode aufgerufen werden.

VBeamPos

Ermittelt die aktuelle Position des Rasterstrahls.

Library: -384(graphics.library)

Deklaration in C: ULONG VBeamPos()

Aufruf in C: pos = VBeamPos();

Aufruf in Assembler: move.l GfxBase,a6
jsr -384(a6)

Parameter: pos: vertikale Position des Elektronenstrahls

Anmerkung: Im Multitasking-Betrieb ist der zurückgegebene Wert wenig brauchbar (außer vielleicht für einen Zufallsgenerator). Hat der aufrufende Task allerdings die höchste Priorität im System, ist der Wert der ermittelten Strahlposition relativ genau.

ViewAddress

Liefert einen Zeiger auf die View-Struktur der Intuition-Benutzeroberfläche.

Library: -294(intuition.library)

Deklaration in C: `struct view *ViewAddress()`

Aufruf in C: `view = ViewAddress();`

Aufruf in Assembler: `move.l IntuitionBase,a6
jsr -294(a6)`

Parameter: view: Adresse des Intuition-View-Struktur

ViewPortAddress

Ermittelt die ViewPort-Adresse eines Fensters.

Library: -300(intuition.library)

Deklaration in C:

```
struct ViewPort *ViewPortAddress(window)
    struct Window *window;
```

Aufruf in C:

```
vp = ViewPortAddress(window);
```

Aufruf in Assembler:

```
move.l window, a0
move.l IntuitionBase, a6
jsr -300(a6)
```

Parameter:

window:	Zeiger auf Window-Struktur
vp (d0):	Adresse der gesuchten ViewPort-Struktur

Wait

Versetzt den aufrufenden Task so lange in den Wartezustand, bis eines der spezifizierten Signale eintrifft.

Library: -318(exec.library)

Deklaration in C: `ULONG Wait(signalset)`
 `ULONG signalset;`

Aufruf in C: `signals = Wait(signalset);`

Aufruf in Assembler: `move.l signalset,d0`
 `move.l ExecBase,a6`
 `jsr -318(a6)`

Parameter: `signalset:` Maske, die die Signale enthält, auf die gewartet werden soll
 `signals (d0):` empfangene Signale

Anmerkung: Diese Funktion darf nicht im Supervisor-Modus verwendet werden.

WaitBlit

Wartet bis zum Ende der aktuellen Blitter-Operation.

Library: -228(graphics.library)

Deklaration in C: void WaitBlit()

Aufruf in C: WaitBlit();

Aufruf in Assembler: move.l GfxBase,a6
jsr -228(a6)

Anmerkung: Aufgrund eines Fehlers im FAT AGNUS kann es passieren, daß diese Funktion schon zurückkehrt, obwohl der Blitter noch nicht einmal mit der Arbeit angefangen hat. Im Moment wird versucht, diese Problem durch mehrfaches Testen des BBUSY-Bits zu lösen.

WaitBOVP

Wartet, bis der Rasterstrahl die letzte Zeile des spezifizierten ViewPorts gezeichnet hat.

Library: -402(graphics.library)

Deklaration in C:

```
void WaitBOVP(vp)
    struct ViewPort *vp;
```

Aufruf in C: WaitBOVP(vp);

Aufruf in Assembler:

```
move.l vp,a0
move.l GfxBase,a6
jsr -402(a6)
```

Parameter: vp: Adresse des betroffenen ViewPorts

WaitForChar

Wartet für einen festgelegten Zeitraum auf die Eingabe eines Zeichens.

Library: -204(dos.library)

Deklaration in C:

```
BOOL WaitForChar(file, timeout)
    struct FileHandle *file;
    long timeout;
```

Aufruf in C: status = WaitForChar(file, timeout);

Aufruf in Assembler:

```
move.l file,d1
move.l timeout,d2
move.l DosBase,a6
jsr -204(a6)
```

Parameter:

file:	Datei-Identifikation eines virtuellen Terminals
timeout:	Zeit in Mikrosekunden, die die Routine auf ein Zeichen wartet
status:	FALSE, falls innerhalb des angegebenen Zeitintervalls kein Zeichen empfangen wurde

WaitIO

Wartet bis zum Ende des I/O-Kommandos (definiert in der IORequest-Struktur).

Library: -474(exec.library)

Deklaration in C:

```
LONG = WaitIO(iorequest)
      struct IORequest *iorequest;
```

Aufruf in C:

```
error = WaitIO(iorequest);
```

Aufruf in Assembler:

```
move.l iorequest,a1
move.l ExecBase,a6
jsr -474(a6)
```

Parameter:

iorequest:	Zeiger auf IORequest-Struktur
error (d0):	NULL, wenn ok

WaitPort

Wartet bis zum Eintreffen einer Nachricht im angegebenen Message-Port. Sind beim Aufruf bereits Nachrichten vorhanden, wird sofort zurückgekehrt.

Library: -384(exec.library)

Deklaration in C:

```
struct Message *WaitPort(port)
    struct MsgPort *port;
```

Aufruf in C: message = WaitPort(port);

Aufruf in Assembler:

```
move.l port,a0
move.l ExecBase,a6
jsr -384(a6)
```

Parameter:

port:	Zeiger auf Message-Port
Message:	Zeiger auf Message-Struktur der anliegenden Nachricht

Anmerkung: WaitPort() liefert zwar einen Message-Zeiger zurück, die Nachricht muß aber noch mit GetMsg() aus der Message-Queue entfernt werden.

WaitTOF

Wartet auf die vertikale Austastlücke (Effektiv kehrt die Funktion erst zurück, wenn alle Server des Vertical-Blanking-Interrupts bearbeitet worden sind).

Library: -270(graphics.library)

Deklaration in C: void WaitTOF()

Aufruf in C: WaitTOF();

WBenchToBack

Legt die Workbench hinter den Screen-Stapel.

Library: -336(intuition.library)

Deklaration in C: BOOL WBenchToBack()

Aufruf in C: success = WBenchToBack();

Parameter: success: TRUE, falls Workbench geöffnet, sonst FALSE

WBenchToFront

Bringt die Workbench in den Vordergrund.

Library: -342(intuition.library)

Deklaration in C: `BOOL WBenchToFront()`

Aufruf in C: `success = WBenchToFront();`

Aufruf in Assembler: `move.l IntuitionBase,a6
jsr -342(a6)`

Parameter: success: TRUE, falls Workbench geöffnet, sonst
FALSE

WhichLayer

Ermittelt das Layer, zu dem der Punkt (x,y) gehört.

Library: -132(layers.library)

Deklaration in C:

```
struct Layer *WhichLayer(li, x, y)
    struct Layer_Info *li;
    short x,y;
```

Aufruf in C: Layer = WhichLayer(li, x, y)

Aufruf in Assembler:

```
move.l li,a0
move.w #x,d0
move.w #y,d1
move.l LayersBase,a6
jsr -132(a6)
```

Parameter:

li:	Zeiger auf die zu untersuchende Layer_Info-Struktur
x,y:	Punktkoordinaten
Layer (d0):	Zeiger auf Layer-Struktur des gesuchten Layers oder NULL, falls sich der Punkt in keinem Layer befindet

WindowLimits

Modifiziert die Begrenzungsparameter eines Fensters.

Library: -318(intuition.library)

Deklaration in C:

```
BOOL WindowLimits
(window, wmin, hmin, wmax, hmax)
    struct Window *window;
    USHORT wmin, hmin, wmax, hmax;
```

Aufruf in C:

```
success = WindowLimits
(window, wmin, hmin, wmax, hmax);
```

Aufruf in Assembler:

```
move.l window, a0
move.w #wmin, d0
move.w #hmin, d1
move.w #wmax, d2
move.w #hmax, d3
```

Parameter:

window:	Adresse einer Window-Struktur
wmin:	Minimale Breite des Windows
hmin:	Minimale Höhe des Windows
wmax:	Maximale Breite des Windows
hmax:	Maximale Höhe des Windows
success (d0):	TRUE, falls neue Parameter mit der aktuellen Fensterausdehnung korrespondieren

Anmerkung: Soll der ursprüngliche Begrenzungswert erhalten bleiben, setzt man den entsprechenden Parameter auf 0.

WindowToBack

Legt ein Fenster in den Hintergrund.

Library: -306(intuition.library)

Deklaration in C:

```
void WindowToBack(window)
    struct Window *window;
```

Aufruf in C: WindowToBack(window);

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a6
jsr -306(a6)
```

Parameter: window: Zeiger auf eine Window-Struktur

WindowToFront

Bringt ein Fenster in den Vordergrund.

Library: -312(intuition.library)

Deklaration in C:

```
void WindowToFront(window)
    struct Window *window;
```

Aufruf in C: WindowToFront(window);

Aufruf in Assembler:

```
move.l window,a0
move.l IntuitionBase,a6
jsr -312(a6)
```

Parameter: window: Zeiger auf eine Window-Struktur

Write

Standard-Schreibroutine des DOS (schreibt length Bytes in den spezifizierten Ausgabe-kanal).

Library: -48(dos.library)

Deklaration in C:

```
long Write(file, buffer, length)
    struct FileHandle *file;
    UBYTE *buffer;
    long length;
```

Aufruf in C: `writtenout = Write(file, buffer, length);`

Aufruf in Assembler:

```
move.l file,d1
move.l buffer,d2
move.l #length,d3
move.l DosBase,a6
jsr -48(a6)
```

Parameter:

file:	Zeiger auf FileHandle-Struktur
buffer:	Zeiger auf Datenpuffer
length:	Anzahl der Byte, die geschrieben werden sollen
writtenOut (d0):	gibt an, wie viele Byte tatsächlich geschrieben worden sind (negativer Wert zeigt Fehler an; die genaue Fehlermeldung liefert IoErr())

WriteExpansionByte

Schreibt ein Byte (nibbleweise) in den Konfigurationsbereich einer Erweiterungskarte.

Library: -114(expansion.library)

Deklaration in C:

```
long WriteExpansionByte(board, offset, value)
    long board, offset;
    BYTE value;
```

Aufruf in C:

```
error = WriteExpansionByte
(board, offset, value);
```

Aufruf in Assembler:

```
move.l #board, a0
move.l #offset, d0
move.b #value, d1
move.l ExpansionBase, a6
jsr -114(a6)
```

Parameter:

board:	Zeiger auf die Basisadresse einer Erweiterungskarte
offset:	Offset relativ zur ExpansionRom-Struktur
value:	Bytewert
error (d0):	bei Fehler ungleich NULL

WritePixel

Setzt einen Punkt an der Koordinate (x,y) im angegebenen RastPort.

Library: -324(graphics.library)

Deklaration in C:

```
long WritePixel(rp, x, y)
    struct Rastport *rp;
    short x,y;
```

Aufruf in C: error = WritePixel(rp, x, y)

Aufruf in Assembler:

```
move.l rp,a1
move.w #x,d0
move.w #y,d1
move.l GfxBase,a6
jsr -324(a6)
```

Parameter:

rp:	Adresse der RastPort-Struktur
x,y:	Koordinate des Punktes
error (d0):	-1, wenn Punkt außerhalb des RastPorts liegt

Anmerkung: Der Rückgabewert -1 sollte nur auftreten, wenn der Punkt (x,y) außerhalb des RastPorts liegt.

WritePotGo

Setzt Bits im POTGO-Register.

Library: -18(potgo.library)

Deklaration in C:

```
void WritePotGo(word, mask)
    long word,mask;
```

Aufruf in C:

```
void WritePotGo(word, mask)
```

Aufruf in Assembler:

```
move.l #word,d0
move.l #mask,d1
move.l PotgoBase,a6
jsr -18(a6)
```

Parameter:

word:	Daten
mask:	Maskenwort (nur gesetzte Bits beeinflussen das Zielregister)

XorRectRegion

Library: -558(graphics.library)

Deklaration in C:

```
BOOL XorRectRegion(rgn, rect)
    struct Region *rgn;
    struct Rectangle *rect;
```

Aufruf in C: status = XorRectRegion(rgn, rect);

Aufruf in Assembler:

```
move.l rgn,a0
move.l rect,a1
move.l GfxBase,a6
jsr -558(a6)
```

Parameter:

rgn:	Adresse der Region-Struktur
rect:	Adresse der Rectangle-Struktur
status (d0):	FALSE bei Abbruch aus Speicherplatzmangel

XorRegionRegion

Library: -618(graphics.library)

Deklaration in C: `BOOL XorRegionRegion(src, dest)
struct Region *src, *dest;`

Aufruf in C: `status = XorRegionRegion(src, dest)`

Aufruf in Assembler: `move.l src,a0
move.l dest,a1
move.l GfxBase,a6
jsr -618(a6)`

Parameter:	<code>src:</code>	Zeiger auf Region-Struktur der Quell-Region
	<code>dest:</code>	Zeiger auf Region-Struktur der Ziel Region
	<code>status (d0):</code>	FALSE bei Abbruch aus Speicherplatzmangel

Schlußwort

Der Wind bläst in Richtung Zukunft oder: Wie geht's weiter mit Commodore und dem Amiga?

Zu der Zeit, da ich diesen Text als Schlußwort eingebe, hat die Zukunft für den Amiga zumindest in den Entwickler-Arbeitsstellen schon begonnen. Die Hardware wurde um das ECS (Enhanced Chip Set) erweitert, das zahlreiche neue Grafik-Modi zur Verfügung stellt; das Chip-Ram hat sich verdoppelt, und als Monitor muß jetzt ein "ordentlicher" Multisync erhalten. Des weiteren wird die 14MHz 68020 Karte schon fast zum Standard.

Die Software verspricht auch einiges: So enthalten die ersten Kickstart/Jumpstart V1.4-Disketten schon eine Reihe interessanter Verbesserungen, weitere Funktionen sind angekündigt:

- Die Workbench kann praktisch auf alle möglichen Darstellungsarten gebracht werden, die Anzahl der Bitplanes ist auch frei wählbar (alles per Preferences).
- Das ARexx-System ist in die Kickstart eingebunden.
- FastFilesystem auch in Verbindung mit Disketten (läuft jetzt schon!)
- Überarbeitung vieler Intuition-Routinen:

StringGadgets mit vielen neuen Funktionen, Proportional-Fonts, frei programmierbaren Eingabe-Filtern.

Multiple Workbench-Screens, sogenannte Public-Screens; Visitor-Windows.

Verschieben eines Screens in der horizontalen und von jeder Position des Bildschirms aus (also nicht nur Titel-Zeile; läuft ebenfalls schon).

LeftAmiga-M blättert jeweils die Screens durch und bringt nicht mehr nur die Workbench nach hinten.

ZipWindows, d.h. man kann zwischen zwei Window-Größen umschalten.

Abbrechen von WindowSize/Drag-Funktion durch Drücken der rechten Maustaste.

DeadEnd-Alerts werden weiterhin in rot dargestellt, Recoverable sind grün. Beide werden allerdings den anderen Screen nicht mehr nach unten schieben, sondern mit einem schwarzen Screen erscheinen.

"Gurus" sollen in Kürze "verbannt" werden. Kommentar von J. Mackraz (jimm): "Rest In Peace."

Wenn Autorequester beendet werden, kommt der Screen wieder nach vorne, der vor ihrem Auftauchen zu sehen war.

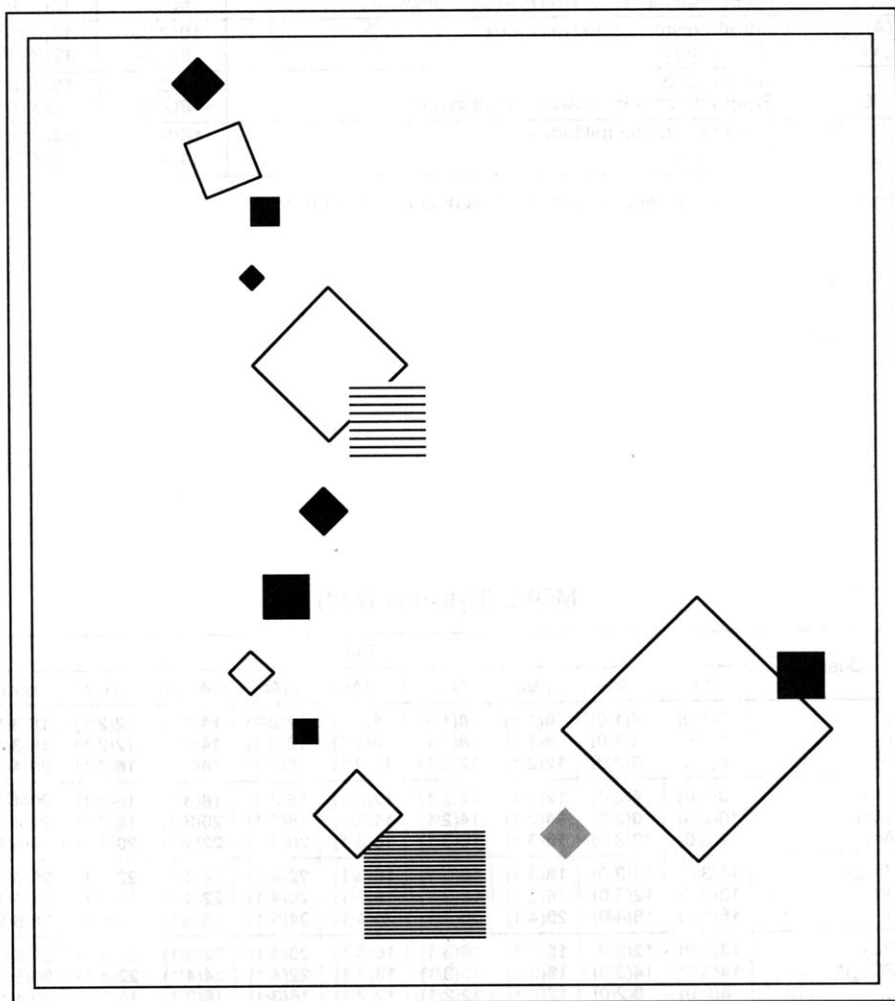
Die Workbench mußte aus Platzgründen auf die Diskette verbannt werden. Somit dürfte für eigene "Hacks" Tür und Tor geöffnet sein.

Zwar sind das zur Zeit teilweise noch unbestätigte Gerüchte, doch konnten im Verlag schon einige Tests auf die oben aufgeführten Funktionen gemacht werden, die alle positiv ausfielen. Doch da sich die Kickstart 1.4 noch in der Alpha-Phase befindet, sollte man noch nicht zu sehr auf diese Funktionen hoffen, da weitgehende Änderungen durchaus noch 'drin sind.

Bleibt also abzuwarten, wann die neue Kickstart offiziell erhältlich sein wird, und womit uns Dale, R.J., Andy und all die anderen guten Geister von Commodore Amiga noch überraschen werden.

Anhang A

Ausführungszeiten der Befehle des MC 68000



Berechnung der effektiven Adresse

	Adressierungart	Byte, Wort	Doppelwort
	Register		
Dn	Datenregister direkt	0(0/0)	0(0/0)
An	Adreßregister direkt	0(0/0)	0(0/0)
	Speicher		
(An)	Adreßregister indirekt	4(1/0)	8(2/0)
(An) +	Adreßregister indirekt mit Postinkrementierung	4(1/0)	8(2/0)
– (An)	Adreßregister indirekt mit Predekrementierung	6(1/0)	10(2/0)
d(An)	Adreßregister indirekt mit Adreßdistanzwert	8(2/0)	12(3/0)
d(An, ix)*	Adreßregister indirekt mit Index	10(2/0)	14(3/0)
xxx W	absolut kurz	8(2/0)	12(3/0)
xxx L	absolut lang	12(3/0)	16(4/0)
d(PC)	Programmzähler mit Adreßdistanzwert	8(2/0)	12(3/0)
d(PC, ix)*	Programmzähler mit Index	10(2/0)	14(3/0)
#xxx	unmittelbar	4(1/0)	8(2/0)

* Die Länge des Indexregisters (ix) beeinflusst nicht die Ausführungszeit

MOVE (Byte und Wort)

Quelle	Ziel								
	Dn	An	(An)	(An) +	– (An)	d(An)	d(An, ix)*	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
(An) +	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
– (An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(3/1)	18(3/1)	22(4/1)
d(An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(An, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
xxx W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
xxx L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	28(6/1)
d(PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(PC, ix)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#xxx	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)

* Die Länge des Indexregisters beeinflusst nicht die Ausführungszeit

MOVE (Doppelwort)

Quelle	Ziel								
	Dn	An	(An)	(An) +	– (An)	d(An)	d(An, ix)*	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	12(1/2)	12(1/2)	14(1/2)	16(2/2)	18(2/2)	16(2/2)	20(3/2)
An	4(1/0)	4(1/0)	12(1/2)	12(1/2)	14(1/2)	16(2/2)	18(2/2)	16(2/2)	20(3/2)
(An)	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
(An) +	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
– (An)	14(3/0)	14(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)
d(An)	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	28(5/2)	30(5/2)	28(5/2)	32(6/2)
d(An, ix)*	18(4/0)	18(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
xxx.W	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	28(5/2)	30(5/2)	28(5/2)	32(6/2)
xxx.L	20(5/0)	20(5/0)	28(5/2)	28(5/2)	28(5/2)	32(6/2)	34(6/2)	32(6/2)	36(7/2)
d(PC)	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	28(5/2)	30(5/2)	28(5/2)	32(5/2)
d(PC, ix)*	18(4/0)	18(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
#xxx	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)

* Die Größe des Indexregisters beeinflusst nicht die Ausführungszeit

Gebräuchliche Befehle

Befehl	Größe	op <ea>, Ant	op <ea>, Dn	op Dn, <M>
ADD	Byte, Wort	8(1/0) +	4(1/0) +	8(1/1) +
	Doppelwort	6(1/0) +**	6(1/0) +**	12(1/2) +
AND	Byte, Wort	–	4(1/0) +	8(1/1) +
	Doppelwort	–	6(1/0) +**	12(1/2) +
CMP	Byte, Wort	6(1/0) +	4(1/0) +	–
	Doppelwort	6(1/0) +	6(1/0) +	–
DIVS	–	–	158(1/0) +*	–
DIVU	–	–	140(1/0) +*	–
EOR	Byte, Wort	–	4(1/0)***	8(1/1) +
	Doppelwort	–	8(1/0)***	12(1/2) +
MULS	–	–	70(1/0) +*	–
MULU	–	–	70(1/0) +*	–
OR	Byte, Wort	–	4(1/0) +	8(1/1) +
	Doppelwort	–	6(1/0) +**	12(1/2) +
SUB	Byte, Wort	8(1/0) +	4(1/0) +	8(1/1) +
	Doppelwort	6(1/0) +**	6(1/0) +**	12(1/2) +

Beachten Sie:

+ Addieren Sie die Berechnungszeit für die effektive Adresse

t Nur Wort und Doppelwort

* Zeigt Maximalwert an

** Die Basiszeit von 6 Taktzyklen erhöht sich auf 8, wenn die Adressierungsart Register direkt oder unmittelbar ist (Zeit zur Berechnung der effektiven Adresse muß zusätzlich addiert werden).

*** Die einzige verfügbare effektive Adressierungsart ist Datenregister direkt.

DIVS, DIVU – Der Dividier-Algorithmus, den der MC68000 verwendet, bewirkt, daß sich die Zeiten für den günstigsten und den ungünstigsten Fall nur um 10% unterscheiden.

MULS, MULU – Der Multiplizier-Algorithmus benötigt $38 + 2n$ Taktzyklen, wobei n definiert ist als $MULU\ n = \text{Anzahl der Einsen in der } \langle EA \rangle$

MULS n = Zusammenfügen der $\langle EA \rangle$ und einer 0 als LSB. n ist die Anzahl der 10- oder 01-Muster in der 17-Bit-Quelle. Der ungünstigste Fall tritt dann auf, wenn die Quelle \$5555 ist.

Unmittelbare Befehle

Befehl	Größe	op #, Dn	op #, An	op #, M
ADDI	Byte, Wort	8(2/0)	—	12(2/1) +
	Doppelwort	16(3/0)	—	20(3/2) +
ADDQ	Byte, Wort	4(1/0) +	8(1/0)*	8(1/1) +
	Doppelwort	8(1/0)	8(1/0)	12(1/2) +
ANDI	Byte, Wort	8(2/0)	—	12(2/1) +
	Doppelwort	16(3/0)	—	20(3/1) +
CMPI	Byte, Wort	8(2/0)	—	8(2/0) +
	Doppelwort	14(3/0)	—	12(3/0) +
EORI	Byte, Wort	8(2/0)	—	12(2/1) +
	Doppelwort	16(3/0)	—	20(3/2) +
MOVEQ	Doppelwort	4(1/0)	—	—
ORI	Byte, Wort	8(21/0)	—	12(2/1) +
	Doppelwort	16(3/0)	—	20(3/2) +
SUBI	Byte, Wort	8(2/0)	—	12(2/1) +
	Doppelwort	16(3/0)	—	20(3/2) +
SUBQ	Byte, Wort	4(1/0)	8(1/0)*	8(1/1) +
	Doppelwort	8(1/0)	8(1/0)	12(1/2) +

+ Addieren Sie die Berechnungszeit für die effektive Adresse

* Nur Wort

Befehle, die nur einen einzigen Operanden verwenden

Befehl	Größe	Register	Speicher
CLR	Byte, Wort	4(1/0)	8(1/1) +
	Doppelwort	6(1/0)	12(1/2) +
NBCD	Byte	6(1/0)	8(1/1) +
NEG	Byte, Wort	4(1/0)	8(1/1) +
	Doppelwort	6(1/0)	12(1/2) +
NEGX	Byte, Wort	4(1/0)	8(1/1) +
	Doppelwort	6(1/0)	12(1/2) +
NOT	Byte, Wort	4(1/0)	8(1/1) +
	Doppelwort	6(1/0)	12(1/2) +
SCC	Byte, unwahr	4(1/0)	8(1/1) +
	Byte, wahr	6(1/0)	8(1/1) +
TAS	Byte	4(1/0)	10(1/1) +
TST	Byte, Wort	4(1/0)	4(1/0) +
	Doppelwort	4(1/0)	4(1/0)

+ Addieren Sie die Berechnungszeit für die effektive Adresse

Schiebe- und Rotierbefehle

Befehl	Größe	Register	Speicher
ASR, ASL	Byte, Wort	$6 + 2n(1/0)$	$8(1/1) +$
	Doppelwort	$8 + 2n(1/0)$	–
LSR, LSL	Byte, Wort	$6 + 2n(1/0)$	$8(1/1) +$
	Doppelwort	$8 + 2n(1/0)$	–
ROR, ROL	Byte, Wort	$6 + 2n(1/0)$	$8(1/1) +$
	Doppelwort	$8 + 2n(1/0)$	–
ROXR, ROXL	Byte, Wort	$6 + 2n(1/0)$	$8(1/1) +$
	Doppelwort	$8 + 2n(1/0)$	–

+ Addieren Sie die zur Berechnung der effektiven Adresse benötigte Zeit
 n ist die Anzahl der Bitpositionen, um die verschoben wird

Bitmanipulation

Befehl	Größe	Dynamisch		Statisch	
		Register	Speicher	Register	Speicher
BCHG	Byte	–	$8(1/1) +$	–	$12(2/1) +$
	Doppelwort	$8(1/0)^*$	–	$12(2/0)^*$	–
BCLR	Byte	–	$8(1/1) +$	–	$12(2/1) +$
	Doppelwort	$10(1/0)^*$	–	$14(2/0)^*$	–
BSET	Byte	–	$8(1/1) +$	–	$12(2/1) +$
	Doppelwort	$8(1/0)^*$	–	$12(2/0)^*$	–
BTST	Byte	–	$4(1/1) +$	–	$8(2/1) +$
	Doppelwort	$6(1/0)^*$	–	$10(2/0)^*$	–

+ Addieren Sie die zur Berechnung der effektiven Adresse benötigte Zeit

* Zeigt Maximalwert an

Bedingte Befehle

Befehl	Adreßdistanzwert	Verzweigung ausgeführt	Verzweigung nicht ausgeführt
B_{CC}	Byte	$10(2/0)$	$8(1/0)$
	Wort	$10(2/0)$	$12(2/0)$
BRA	Byte	$10(2/0)$	–
	Wort	$10(2/0)$	–
BSR	Byte	$18(2/2)$	–
	Wort	$18(2/2)$	–
DB_{CC}	CC wahr	–	$12(2/0)$
	CC falsch	$10(2/0)$	$14(2/0)$

+ Addieren Sie die zur Berechnung der effektiven Adresse benötigte Zeit

* Zeigt Maximalwert an

JMP, JSR, LEA, PEA und MOVEM

Befehl	Größe	(An)	(An)+	-(An)	d(An)	d(An, ix)+	xxx.W	xxx.L	d(PC)	d(PC, ix)*
JMP	-	8(2/0)	-	-	10(2/0)	14(3/0)	10(2/0)	12(3/0)	10(2/0)	14(3/0)
JSR	-	16(2/2)	-	-	18(2/2)	22(2/2)	18(2/2)	20(3/2)	18(2/2)	22(2/2)
LEA	-	4(1/0)	-	-	8(2/0)	12(2/0)	8(2/0)	12(3/0)	8(2/0)	12(2/0)
PEA	-	12(1/2)	-	-	16(2/2)	20(2/2)	16(2/2)	20(3/2)	16(2/2)	20(2/2)
MOVEM M→R	Wort	12+4n (3+n/0)	12+4n (3+n/0)	-	16+4n (4+n/0)	18+4n (4+n/0)	16+4n (4+n/0)	20+4n (5+n/0)	16+4n (4+n/0)	18+4n (4+n/0)
	Doppelwort	12+8n (3+2n/0)	12+8n (3+2n/0)	-	16+8n (4+2n/0)	18+8n (4+2n/0)	16+8n (4+2n/0)	20+8n (5+2n/0)	16+8n (4+2n/0)	18+8n (4+2n/0)
MOVEM R→M	Wort	8+4n (2/n)	-	8+4n (2/n)	12+4n (3/n)	14+4n (3/n)	12+4n (3/n)	16+4n (4/n)	-	-
	Doppelwort	8+8n (2/2n)	-	8+8n (2/2n)	12+8n (3/2n)	14+8n (3/2n)	12+8n (3/2n)	16+8 (4/2n)	-	-

n ist die Anzahl zu bewogender Registerinhalte

* Die Größe des Indexregisters (ix) beeinflusst nicht die Ausführungszeit

Mehrfachgenaue Befehle

Befehl	Größe	Op Dn, Dn	Op M, M
ADDX	Byte, Wort	4(1/0)	18(3/1)
	Doppelwort	8(1/0)	30(5/2)
CMPM	Byte, Wort	-	12(3/0)
	Doppelwort	-	20(5/0)
SUBX	Byte, Wort	4(1/0)	18(3/1)
	Doppelwort	8(1/0)	30(5/2)
ABCD	Byte	6(1/0)	18(3/1)
SBCD	Byte	6(1/0)	18(3/1)

Verschiedene Befehle

Befehl	Größe	Register	Speicher
ANDI to CCR	Byte	20(3/0)	–
ANDI to SR	Wort	20(3/0)	–
CHK	–	10(1/0)+	–
EORI to CCR	Byte	20(3/0)	–
EORI to SR	Wort	20(3/0)	–
ORI to CCR	Byte	20(3/0)	–
ORI to SR	Wort	20(3/0)	–
MOVE from SR	–	6(1/0)	8(1/1)+
MOVE to CCR	–	12(2/0)	12(2/0)+
MOVE to SR	–	12(2/0)	12(2/0)+
EXG	–	6(1/0)	–
EXT	Wort	4(1/0)	–
	Doppelwort	4(1/0)	–
LINK	–	16(2/2)	–
MOVE from USP	–	4(1/0)	–
MOVE to USP	–	4(1/0)	–
NOP	–	4(1/0)	–
RESET	–	132(1/0)	–
RTE	–	20(5/0)	–
RTR	–	20(5/0)	–
RTS	–	16(4/0)	–
STOP	–	4(0/0)	–
SWAP	–	4(1/0)	–
TRAPV	–	4(1/0)	–
UNLK	–	12(3/0)	–

+ Addieren Sie die zur Berechnung der effektiven Adresse benötigte Zeit

MOVEP

Befehl	Größe	Register → Speicher	Speicher → Register
MOVEP	Wort	16(2/2)	16(4/0)
	Doppelwort	24(2/4)	24(6/0)

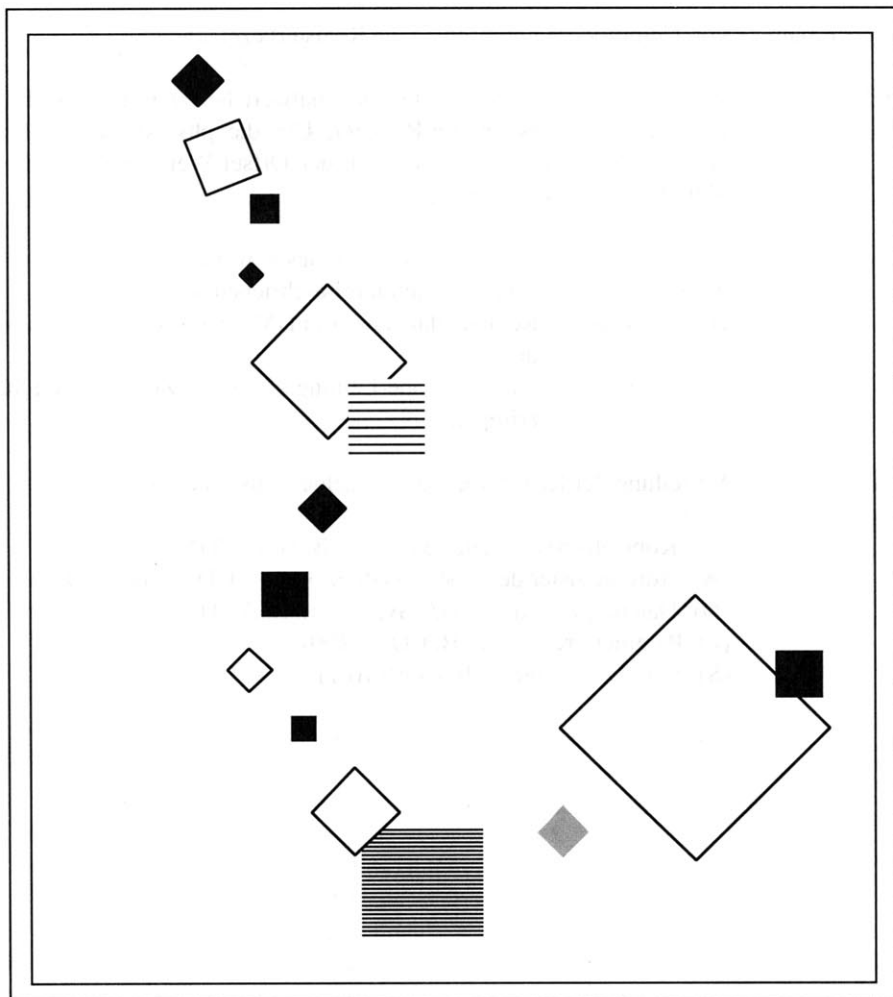
Ausnahme-Prozeduren

Ausnahme	Zyklen
Adreßfehler	50(4/7)
Busfehler	50(4/7)
CHK-Befehl	44(5/4) +
Division durch Null	42(5/4)
Illegaler Befehl	34(4/3)
Unterbrechung	44(5/3)*
Privilegverletzung	34(4/3)
RESET**	40(6/0)
Trace	34(4/3)
TRAP-Befehl	38(4/4)
TRAPV-Befehl	34(4/3)

- + Addieren Sie die zur Berechnung der effektiven Adresse benötigte Zeit
- * Es wird angenommen, daß der Unterbrechungserkennungszyklus vier Taktzyklen dauert.
- ** Zeit vor der Erkennung, daß RESET und HALT aktiv sind, bis zum Beginn der Befehlsausführung

Anhang B

Alphabetische Auflistung der Customchip - Register



Alle Register werden nach folgendem Schema beschrieben:

<Registername> <Offset> <Zugriff> <Chip> <Typ>

- ausführliche englische Schreibweise
- deutsche Übersetzung
- Beschreibung der Registerfunktion
- Bitbelegung

<Registername>: von Commodore vorgeschlagene Registerbezeichnung

<Offset>: Relative Adreßangabe als Hexadezimalwert bezogen auf den Tabellenanfang der Customchip-Register. Um die physikalische Adresse eines Registers zu berechnen, muß der Offset-Wert zur Basisadresse \$DFF000 addiert werden.

<Zugriff>:

r (read):	Register kann nur gelesen werden
w (write):	Register kann nur beschrieben werden
er (early read):	Register darf nur vom DMA-Controller gelesen werden
s (strobe):	interne Strobe-Leitung, löst beim Zugriff einen Steuerimpuls aus

<Chip>: Verteilung der Register auf die einzelnen Customchips.

<Typ>:

- (K): Kontroll- bzw. Steuerregister (z.B. DMACON)
- (A): Adreßregister des DMA-Systems (z.B. AUDxLCH/AUDxLCL)
- (D): Datenregister des DMA-Systems (z.B. AUDxDAT)
- (P): Parameterregister (z.B. COLOR00)
- (S): Strobe-Register (z.B. COPJMP1)

Die Customchip-Register

ADKCON \$09E w PAULA (K)

(Audio/Disk CONTROL Write = Audio- und Disk-Kontrollregister)

Bitbelegung:

Nr.	Label	Funktion										
15	SET/CLEAR	Set/Clear-Bit, schaltet zwischen Setz- (= 1) und Löschzugriff (= 0) um.										
14/13	PRECOMP	Mit diesen beiden Bits lassen sich die vier unterstützten Präkompensationszeiten des Disk-Controllers anwählen: <table><tr><th>Bitkombination</th><th>Zeit</th></tr><tr><td>00</td><td>—</td></tr><tr><td>01</td><td>140 ns</td></tr><tr><td>10</td><td>280 ns</td></tr><tr><td>11</td><td>560 ns</td></tr></table>	Bitkombination	Zeit	00	—	01	140 ns	10	280 ns	11	560 ns
Bitkombination	Zeit											
00	—											
01	140 ns											
10	280 ns											
11	560 ns											
12	MFMPREC	Aufzeichnungsformat: 1 = MFM, 0 = GCR.										
11	UARTBRK	unterbricht die Ausgabe über die serielle Schnittstelle. Die TXD-Leitung geht auf 0.										
10	WORDSYNC	Synchronisations-Marke (MFM) setzen. Die Datenübertragung startet erst dann, wenn die Markierung gefunden wurde.										
09	MSBSYNC	schaltet GCR-Synchronisationsmarke ein (Wert: \$FF).										
08	FAST	Taktrate des Disk-Controllers. 1 = 2 Mikrosekunden pro Bit (MFM-Format). 2 = 4 Mikrosekunden pro Bit (CGR-Format).										
07	USE3PN	Tonkanal 3 moduliert nichts.										
06	USE2P3	Tonkanal 2 moduliert die Frequenz von Tonkanal 3.										
05	USE1P2	Tonkanal 1 moduliert die Frequenz von Tonkanal 2.										
04	USE0P1	Tonkanal 0 moduliert die Frequenz von Tonkanal 1.										
03	USE3VN	siehe Bit 07.										
02	USE2V3	Tonkanal 2 moduliert die Lautstärke von Tonkanal 3.										
01	USE1V2	Tonkanal 1 moduliert die Lautstärke von Tonkanal 2.										
00	USE0V1	Tonkanal 0 moduliert die Lautstärke von Tonkanal 1. Bei einer kombinierten Frequenz/Amplituden-Modulation werden die Datenworte abwechselnd in das AUDxVOL- (Bits 0 bis 6 enthalten den Amplitudenwert) und AUDxPER-Register (Bits 0 bis 15) geschrieben.										

ADKCONR	\$010	r	PAULA	(K)
----------------	--------------	----------	--------------	------------

(AUDio/Disk CONTRol Read = Audio- und Disk-Kontrollregister)

Leseadresse von ADKCON (Bitbelegung identisch).

AUDxDAT	\$0AA+x*\$00A	w	PAULA	(D)
----------------	----------------------	----------	--------------	------------

(AUDio Channel x DATa = Datenwort für Tonkanal x)

DMA-Puffer zur Aufnahme des aktuellen Datenwortes für die Tonausgabe. Das Register wird normalerweise durch den DMA-Controller beschrieben, kann aber auch von der CPU geladen werden. (Sample-Byte müssen im Zweierkomplement vorliegen.)

AUDxLCH	\$0A0+x*\$00A	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(AUDio Channel x LoCation High = Speicherposition der Sampledaten von Audiokanal x [0..3], höherwertige Bits 16 bis 18)

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
x x x x x x x x x x x x x x	[18-16] = höherwertige drei Bits der 19-Bit-Speicheradresse

AUDxLCL	\$0A2+x*\$0A0	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(AUDio channel x LoCation Low = Speicherposition der Sampledaten von Audiokanal x [0..3], niederwertige Bits 0 bis 15)

Die Register AUDxLCH und AUDxLCL bilden ein 19-Bit-Adreßregister für die Startadresse der Sampledaten. Ihr Wert wird während des DMA-Betriebs nicht verändert, so daß man sie in den Interrupt-Phasen auf beliebige Speicherblöcke umbiegen kann.

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[niederwertiger Teil der 19-Bit-Speicheradresse]

AUDxLEN **\$0A4+x*\$0A0** **w** **PAULA** **(P)**

(AUDio channel x LENgth = Länge der Sampledaten von Audiokanal x)

Gibt die Länge des Speicherblocks für Tonkanal x an (maximal 128 KByte). Nach Übertragung der durch AUDxLEN festgelegten Anzahl von Datenworten, wird ein AUDx-Interrupt ausgelöst (Bit 7 bis 10 im INTENA-Register).

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 [Länge der Sampledaten]

AUDxPER **\$0A6+x*\$0A0** **w** **PAULA** **(P)**

(AUDio channel x PERiod = Ausgabeperiode von Audiokanal x)

Dieses Register bestimmt den Zeitabstand zweier Ausgabezyklen. Ist die Samplingfrequenz vorgegeben, ergibt sich die erforderliche Anzahl von Buszyklen aus dem Term: $AUDxVOL = (3,55 \text{ MHz} / \text{Samplingrate in KHz})$. Das DMA-System begrenzt die maximale Ausgabefrequenz auf 28,6 KHz (entspricht 124 Buszyklen).

AUDxVOL **\$0A8+x*\$0A0** **w** **PAULA** **(P)**

(AUDio channel x VOLume = Lautstärke-Einstellung für Tonkanal x)

AUDxVOL bietet die Auswahl zwischen 64 (Bits 0 bis 5) verschiedenen Lautstärkestufen.

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 x x x x x x x x x x x [Volume-Wert]

BLTxDAT (x = A,B,C) A=\$074/B=\$072/C=\$070 w **AGNUS (D)**

(BLiTter DATa register A/B/C= Blitter-DMA-Datenregister für Quellbereich A, B, C)

Diese Register enthalten die aktuellen Daten-Wörter des Quellbereichs einer Blitter-Operation. Ihre Datenversorgung läuft über das DMA-System, wahlweise können sie aber auch direkt vom Prozessor angesprochen werden.

Besonderheit im LINE-DRAW-Modus:

BLTADAT dient als Index-Register und sollte auf den Hex-Wert \$8000 gesetzt werden. BLTBDAT enthält das Linienmuster. Die Initialisierung \$FFFF erzeugt eine durchgezogene Linie.

BLTDDAT \$000 er **AGNUS (D)**

(BLiTter DATa register D = Blitter-DMA-Datenregister für Zielbereich D)

BLTDDAT puffert das Verknüpfungsergebnis der Quelldaten von A, B und C, sein Inhalt wird vom DMA-Controller an die Zieladresse D im Chip-RAM-Speicher geschrieben. Es handelt sich dabei um ein sogenanntes Early-Read-Register, auf das der Prozessor keinen Zugriff hat.

BLTxMOD (x=A/B/C/D) A=\$064/B=\$062/C=\$060/D=\$066 w **AGNUS (K)**

(BLiTter MODulo x = Blitter-Modulo-Wert für Bitmap x)

Mit dem BLTxMOD-Register lassen sich verschiedene Modulo-Werte für die drei Quell-Bitmaps (A bis C) und die Ziel-Bitmap D einstellen. Der Modulo-Wert erlaubt die gezielte Bearbeitung eines Bitmap-Ausschnitts, da er nach jeder Zeile des Blitter-Fensters zu den Adreßzeigern (BPLxPTH/L) hinzuaddiert wird und sie auf den horizontalen Beginn des bearbeiteten Blitter-Fensters umlenkt.

Besonderheit im LINE-DRAW:

BLTAMOD und BLTBMOD spezifizieren die Liniensteigung (X/Y). BLTAMOD enthält den Ausdruck 4Y-4X, BLTBMOD den Wert 4Y. Die Byte-Breite der zum Linienzeichnen verwendeten Bitmap muß in den Registern BLTCMOD und BLTDMOD stehen.

BLTxPTH(x=A/B/C/D) A=\$050/B=\$04C/C=\$048/D=\$054 w AGNUS (A)

(BLiTter Pointer to x High = Zeiger auf Bitmap x, höherwertige Bits 16 bis 18)

Höherwertiger Teil des Adreßregisters für die Quell- (A bis C) bzw. Ziel-Bitmaps (D).

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

[18–16] = höherwertige drei Bits der 19-Bit-Speicheradresse

BLTxPTL(x=A/B/C/D) A=\$052/B=\$04E/C=\$04A/D=\$056 w AGNUS (A)

(BLiTter Pointer to x Low = Zeiger auf Bitmap x, niederwertige Bits)

Das Registerpaar BLTxPTH und BLTxPTL bildet einen Zeiger auf die Speicheradresse der Quell- bzw. Ziel-Bitmaps, die vom DMA-Controller bearbeitet werden. Während der Blitter-Operation wird dieser Zeiger laufend erhöht und zeigt nach Beendigung des Bearbeitungszyklus auf den Nachfolger des letzten Datenwortes (+2) plus Modulo-Wert.

LINE-DRAW-Modus:

In diesem Modus wird BLTAPTL als Arbeitsregister verwendet und muß mit dem Wert 2Y–X geladen werden, wobei Y/X wieder die Liniensteigung angibt. Die Adreßregister BLTCPT(H/L) sowie BLTDPT(H/L) enthalten die Startadresse der Linie.

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

[niederwertiger Teil der 19-Bit-Speicheradresse]

BLTAFWM \$044 w AGNUS (P)

(BLiTter First Word Mask for source A = Blitter-Maskierung für das erste Datenwort der Quelle A)

Beschreibung unter BLTALWM

BLTALWM **\$046** **w** **AGNUS** **(P)**

(BLiTter Last Word Mask for source A = Blitter-Maskierung für das letzte Datenwort der Quelle A)

Die Werte dieser Register werden mit dem ersten und letzten Datenwort einer Zeile der Quellbitmap A logisch UND-verknüpft. Dadurch wird es möglich, Bits am Rande des Blitter-Fensters gezielt auszublenden und die Beschränkung auf Wort-Grenzen zu umgehen. Leider ist der Maskierungsmechanismus nur für den Quellbereich A vorgesehen. Im Fill- und Line-Draw-Modus empfiehlt es sich, die Register auf \$FFFF zu setzen.

BLTCON0 **\$040** **w** **AGNUS** **(K)**

(BLiTter CONTROL register 0 = Blitter-Kontrollregister 0)

Zentrales Steuerregister der Blitter-Hardware (siehe auch BLTCON1)

Bitbelegung:

Nr.	Label	Funktion
15-12	ASHx	4-Bit-Shift für Quelle A (0 bis 15 Binärstellen)Verschiebung durch den internen Barrel-Shifter
11	USEA	Quelle A aktivieren
10	USEB	Quelle B aktivieren
09	USEC	Quelle C aktivieren
08	USED	Ziel D aktivieren
07-00	LFx	Auswahl der logischen Verknüpfungen (Minterms, 8-Bit-Wert) für die Blitter-Operation.

Im LINE-DRAW-Modus sind die Bits teilweise anders belegt:

15-12	START	4-Bit-Startposition der Linie im ersten Datenwort.
11-08		müssen %1011 sein.
07-00		Die Minterms LFX sollten mit \$4A initialisiert werden, was dem logischen Ausdruck $D=(aC+ABc)$ entspricht (erlaubt die Verwendung von Linienmustern).

BLTCON1 \$042 w AGNUS (K)

(BLiTter CONTROL register 1 = Blitter-Kontrollregister 1)

Zweites Blitter-Kontrollregister.

Bitbelegung:

Nr.	Label	Funktion
15–12	BSHx	Shift-Wert für Quelle B (siehe BLTCON0).
11–05	–	nicht verwendet.
04	EFE	Exclusive-Fill-Enable-Modus einschalten.
03	IFE	Inclusive-Fill-Enable-Modus aktivieren.
02	FCI	Das Fill-Carry-Input-Bit bestimmt, ob der Blitter beim Füllen von Flächen die folgenden Datenbits setzen (= 1) oder löschen soll (= 0).
01	DESC	Descending-Modus. Blitter arbeitet mit absteigenden Adressen
00	LINE	LINE-Modus einschalten (= 1).

Auch hier weicht die Bitbelegung im LINE-DRAW-Modus ab:

15–12	TEXTUREx	Mit diesen vier Bits kann man das Startbit des Linienmusters festlegen.
11–07		müssen auf %00000 gesetzt werden.
06	SIGN	Vorzeichenbit.
05	–	nicht belegt (sollte auf 0 gesetzt werden).
04	SUD	SUD, SUL und AUL ergeben zusammen den Bit-Code für den Oktanten, in dem die Linie liegt.
03	SUL	
02	AUL	
01	SING	Der Blitter darf nur einen Punkt pro Zeile setzen – eine sinnvolle Beschränkung, wenn die zu zeichnende Linie die Begrenzung eines Polygons darstellt und dieses anschließend gefüllt werden soll.

BLTSIZE	\$058	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(BLiTter SIZE = Größe des Blitter-Fensters)

BLTSIZE legt die Breite und Höhe des Blitter-Fensters fest, gleichzeitig liefert es das Startsignal zum Beginn der Blitter-Operation. Deshalb sollte das Register immer zuletzt beschrieben werden.

Bitbelegung:

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
h9 h8 h7 h6 h5 h4 h3 h2 h1 h0 b5 b4 b3 b2 b1 b0
[Fensterhöhe] [Breite]

h =vertikale Ausdehnung in Zeilen (10-Bit-Wertebereich, max. 1024 Zeilen)

b =Anzahl der Worte pro Zeile (6-Bit-Wertebereich, max. 1024 Pixel)

Der Wert 0 steht für die Maximal-Werte in der Horizontalen bzw. Vertikalen (Fenstergröße: 1024x1024).

Besonderheit des LINE-DRAW-Modus:

Bits 15 bis 6 [h9..h0] enthalten die Längenausdehnung der Linie (max. 1024 Punkte), die Breite (Bits 0 bis 5) muß immer 2 sein.

BPLxDAT	\$110+x*\$002	w	DENISE	(D)
----------------	----------------------	----------	---------------	------------

(Bit PLane x DATa = DMA-Datenregister für Bitplane x, Zugriff durch CPU erlaubt)

BPLxDAT arbeitet als Puffer für die Parallel/Seriell-Konvertierung der Grafikdaten (Schieberegister auf DENISE), die nach dem Zugriff auf das BPL1DAT-Register stattfindet (beginnt mit dem MSB des Datenwortes).

BPL1MOD	\$108	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Bit PLane MODulo = Modulo-Wert der ungeraden Bitplanes)

siehe BPL2MOD.

BPL2MOD	\$10A	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Bit PLane MODulo = Modulo-Wert für alle geraden Bitplanes)

BPL1MOD und BPL2MOD erlauben die getrennte Einstellung eines Modulo-Werts für gerade und ungerade Bitplanes. Ihr Inhalt wird nach der Darstellung einer Rasterzeile zu den Bitplane-Adreßzeigern BPLxPT(H/L) hinzuaddiert, um an den korrekten Anfang der nächsten Zeile zu gelangen. Dadurch können auch überbreite Bitplanes verarbeitet werden, die nur ausschnittsweise auf dem Bildschirm zu sehen sind. Im DUAL-PLAY-FIELD-Modus dürfen gerade und ungerade Bitplanes unterschiedliche Breite haben.

BPLxPTH	\$0E0+x*\$004	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(Bit PLane x PoinTer High = Adreßzeiger auf Bitplane x, obere drei Bits)

Dieses Register enthält die höherwertigen drei Bits der 19-Bit-Chip-RAM-Adresse einer der sechs darstellbaren Bitplanes (x = 1...6).

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
x x x x x x x x x x x x x x	[18-16] = höherwertige drei Bits der 19-Bit-Speicheradresse

BPLxPTL	\$0E2+x*\$004	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(Bit PLane x PoinTer Low = Adreßzeiger auf Bitplane x, untere 16 Bits)

BPLxPTL bildet zusammen mit BPLxPTH einen 19-Bit-Adreßregister für die Speicherposition von Bitplane x. Die Zeiger werden synchron zum Bildaufbau erhöht und müssen deshalb während der vertikalen Austastlücke durch den Copper neu initialisiert werden.

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[niederwertiger Teil der 19-Bit-Speicheradresse]

BPLCON0 \$100 w AGNUS/DENISE (K)

(Bit PLane CONtrol register 0 = Bitplane-Kontrollregister 0)

Zentrales Steuerregister für Bildschirmauflösung, Darstellungsmodus, Farb-Betriebsart und diverse andere Funktionen.

Bitbelegung:

Nr.	Label	Funktion
15	HIRES	High-Resolution = hohe Auflösung mit einer Pixelfrequenz von 14,2 MHz und maximal vier Bitplanes.
14-12	BPU _x	Anzahl der verwendeten Bitplanes (max. 6).
11	HOMOD	Dieses Bit selektiert den HAM-Modus (= Hold-And-Modify).
10	DBLPF	Einschalten des DUAL-PLAYFIELD-Modus.
09	COLOR	Phasenlage des Farbträger (Colorburst) zur Erzeugung des FBAS-Signals einblenden.
08	GAUD	Genlock-Audio-Bit.
07-04	–	nicht verwendet.
03	LPEN	Light-Pen-Abfrage aktivieren (zurückgesetzt nach Neustart!).
02	LACE	Interlace-Modus einschalten (zurückgesetzt nach Neustart!).
01	ERSY	Externe Synchronisation erlauben (HSYNC- und VSYNC-Leitungen der RGB-Buchse sind als Eingänge geschaltet).
00	–	nicht verwendet.

BPLCON1 \$102 w DENISE (K)

(Bit PLane CONtrol register 1 = Bitplane-Kontrollregister 1)

Steuerregister für Fein-Scrolling (Verschiebung des sichtbaren Bildbereiches).

Bitbelegung:

Nr.	Label	Funktion
15-08	–	nicht verwendet.
07-04	PF2Hx	4-Bit-Verzögerung der Darstellung ungerader Playfields.
03-00	PF1Hx	Gleiche Funktion wie PF2Hx für gerade Playfields.

BPLCON2 \$104 w DENISE (K)

(Bit PLane CONTrol register 2 = Bitplane-Kontrollregister 2)

Dieses Register steuert die Darstellungs-Priorität der verschiedenen Grafikobjekte (Playfields, Bobs, Sprites).

Bitbelegung:

Nr.	Label	Funktion
15-07	–	nicht verwendet.
06	PF2PRI	Playfield 2 (ungerade Bitplanes) besitzt erhöhte Priorität gegenüber Playfield 1 (gerade Playfields) und erscheint immer vor den geraden Bitplanes.
05-03	PF2Px	Diese Bits beschreiben die Position der ungeraden Bitplanes zu den vier Sprite-Paaren. Es sind Werte von 0 (Bitplanes vorne) bis 4 (Bitplanes ganz hinten) möglich.
02-00	PF1Px	Analoges für Playfield 1 (gerade Bitplanes).

CLXCON \$098 w DENISE (K)

(CoLLision CONTrol = Kollisions-Kontrollregister)

Kontrollregister der Kollisions-Erkennungs-Logik. Aus der Verwaltung der Grafikobjekte ergeben sich einige Einschränkungen für die Programmierung. So können lediglich Kollisionen der vier Sprite-Paare untereinander, aber nicht zwischen den Sprites eines Paares registriert werden. Mit den ENBPx-Bits lassen sich alle sechs Bitplanes einzeln in den Kollisions-Erkennungsprozeß einblenden. Zusätzlich kann man mit den MVBPx-Bits die Bitkombinationen (Match-Value) der Planes festlegen, die Kollisionen auslösen dürfen.

Bitbelegung:

Nr.	Label	Funktion
15	ENSP7	Kollisionskontrolle von Sprite 7 einschalten.
14	ENSP5	Kollisionskontrolle von Sprite 5 einschalten.
13	ENSP3	Kollisionskontrolle von Sprite 3 einschalten.
12	ENSP1	Kollisionskontrolle von Sprite 1 einschalten.
11-06	ENBPx	Kollisionskontrolle von Bitplane x (Bit 11 für Plane 6, Bit 10 für Plane 5 usw.) aktivieren.

- 05-00 MVBPx Vergleichswert für Bitplane x setzen. Nur wenn Match-Bit und Bitplane an der Kollisionstelle übereinstimmen, wird die Kollision registriert.

CLXDAT **\$00E** **r** **DENISE** **(P)**

(CoLlision DATa register = Datenregister der Kollisions-Logik)

Dieses Register reflektiert den aktuellen Kollisions-Status. Nach dem Zugriff wird es automatisch zurückgesetzt.

Besonderheit:

Zu Playfield 1 gehören alle aktiven ungeraden Bitplanes, zu Playfield 2 alle eingeschalteten geraden Bitplanes.

Bitbelegung:

Nr. Funktion

- | | |
|----|---|
| 15 | nicht verwendet. |
| 14 | Sprite 4 (oder 5) kollidiert mit Sprite 6 (oder 7). |
| 13 | Sprite 2 (oder 3) kollidiert mit Sprite 6 (oder 7). |
| 12 | Sprite 2 (oder 3) kollidiert mit Sprite 4 (oder 5). |
| 11 | Sprite 0 (oder 1) kollidiert mit Sprite 6 (oder 7). |
| 10 | Sprite 0 (oder 1) kollidiert mit Sprite 4 (oder 5). |
| 09 | Sprite 0 (oder 1) kollidiert mit Sprite 2 (oder 3). |
| 08 | Sprite 6 (oder 7) kollidiert mit Playfield 2. |
| 07 | Sprite 4 (oder 5) kollidiert mit Playfield 2. |
| 06 | Sprite 2 (oder 3) kollidiert mit Playfield 2. |
| 05 | Sprite 0 (oder 1) kollidiert mit Playfield 2. |
| 04 | Sprite 6 (oder 7) kollidiert mit Playfield 1. |
| 03 | Sprite 4 (oder 5) kollidiert mit Playfield 1. |
| 02 | Sprite 2 (oder 3) kollidiert mit Playfield 1. |
| 01 | Sprite 0 (oder 1) kollidiert mit Playfield 1. |
| 00 | Playfield 1 kollidiert mit Playfield 2. |

Während des Strahlrücklaufs wird COP1LC(H/L) automatisch in den internen Programm-Zähler übertragen.

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

[niederwertiger Teil der 19-Bit-Speicheradresse]

COP2LCH	\$084	w	AGNUS	(A)
----------------	--------------	----------	--------------	------------

(COPper second LoCation register High = Adreßzeiger für die zweite Copper-Befehlsliste, obere drei Bits)

Bitbelegung:

x x x x x x x x x x x x x x [18-16] = höherwertige drei Bits der 19-Bit-Speicheradresse

COP2LfCL	\$086	w	AGNUS	(K)
-----------------	--------------	----------	--------------	------------

(COPper second LoCation register Low = Adreßzeiger für die zweite Copper-Befehlsliste, untere 16 Bits)

COP2LCH und COP2LCL bilden wieder ein Adreßregister. Es enthält die Startadresse der zweiten Copper-Befehlsliste, die für den Aufbau einer Interlace-Darstellung benötigt wird, da jedes Halbbild eine eigene Copper-Liste verwendet.

COPCON	\$02E	w	AGNUS	(K)
---------------	--------------	----------	--------------	------------

(COPper CONTROL register = Copper-Kontrollregister)

Dieses Register enthält zur Zeit nur ein relevantes Bit (Nr. 0). Es erlaubt den Copper-Zugriff auf die Register der Blitter-Hardware (Offset \$040 bis \$07E), die normalerweise lediglich von DMA-Controller und CPU angesprochen werden. Nach einem Neustart ist das Bit gelöscht.

BIT 00 CDANG Copper-DANger-mode.

COPINS	\$08C	w	AGNUS	(D)
---------------	--------------	----------	--------------	------------

(COPper INStruction fetch identify = Befehlsregister des Coppers)

DMA-Pufferregister für die Befehle der Copper-Listen.

Der Copper kennt drei Befehle:

- MOVE:** Übertrage ein unmittelbares Datenwort in ein Customchip-Register (ab \$040, sofern das CDANG-Bit in COPCON gesetzt ist).
- WAIT:** Warte bis der Elektronenstrahl eine bestimmte Bildzeile erreicht oder überschritten hat.
- SKIP:** Hat der Elektronenstrahl bereits die angegebene Position überschritten, dann überspringe den nächsten Befehl.

Copper-Befehle bestehen aus zwei Datenworten. Durch die RISC-Struktur der Schaltung überlagern sich Befehlsausführung und Holmechanismus, so daß der Copper lediglich zwei DMA-Zyklen zum Einladen der beiden Befehlsörter benötigt.

Der WAIT-Befehl verbraucht zwar einen weiteren Zyklus beim Erreichen der angegebenen Strahlposition, belastet die Busse aber nicht während der Wartephase. Da der Copper nur gerade Buszyklen verwendet (dabei hat er Vorrang gegenüber Blitter und CPU), dauert ein kompletter Befehlszyklus vier bzw. sechs (WAIT) Prozessor-Taktzyklen.

Struktur des MOVE-Befehls:

Erstes Instruktions-Wort (1. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x	x	x	x	x	x	x	x	[	Registeradresse						]	I

Registeradresse: Tabellenoffset (Nummer) des Zielregisters der Customchip-Hardware.
I: Kennungsbit für den MOVE-Befehl (Wert: 0).

Zweites Instruktions-Wort (2. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[16-Bit-Datenwort (unmittelbar)]															

Struktur des WAIT-Befehls:

Erstes Instruktions-Wort (1. DMA-Zyklus):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[VP7 VP6 VP5 VP4 VP3 VP2 VP1 VP0] [HP8 HP7 HP6 HP5 HP4 HP3 HP2] I															

VP7 bis VP0: vertikale Strahlposition, auf die gewartet werden soll.

HP8 bis HP2: horizontale Strahlposition.

I: 1. Kennungsbit des WAIT-Befehls (Wert: 1)

Zweites Instruktions-Wort (2. DMA-Zyklus):

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
BFD [VM6VM5 VM4 VM3 VM2 VM1 VM0][HM8HM7 HM6 HM5 HM4 HM3 HM2]I

BFD: Blitter-Finished-Disable-Bit.

0 = Warten auf Ende der Blitter-Operation

1 = Registerzugriff ohne Beachtung des Blitter-Status.

VP7 bis VP0: Maske für die vertikale Rasterstrahlposition.

HP8 bis HP2: Maske für die horizontale Strahlposition.

I: 2. Kennungsbit des WAIT-Befehls (Wert: 0)

Es werden nur die VP- bzw. HP-Bits mit der Strahlposition verglichen, deren korrespondierende Maskenbits im zweiten Befehlswort gesetzt sind.

Die Struktur des SKIP-Befehls:

Die funktionale Ähnlichkeit des SKIP-Befehls mit der WAIT-Instruktion zeigt sich auch im Befehlsaufbau; er ist – bis auf das Kennungsbit des zweiten Instruktions-Wortes (Wert: 1) – identisch.

COPJMP1	\$088	s	AGNUS	(S)
----------------	--------------	----------	--------------	------------

(COPper JuMP register 1 = Strobe-Signal zur Initialisierung des Programmzählers auf die erste Copper-Befehlsliste COP1LC(H/L)).

Ein Zugriff auf die Adresse \$DFF088 initialisiert den Programmzähler der Copper-Hardware auf den Beginn (COP1LC[H/L]) der ersten Befehlsliste im Speicher.

Der Copper kann dieses Register selbst beschreiben, so daß sich auch weite Sprünge in der Copperlist realisieren lassen (in Ergänzung zum SKIP-Befehl).

COPJMP2	\$08A	s	AGNUS	(S)
----------------	--------------	----------	--------------	------------

(COPper JuMP register 2 = Strobe-Signal zur Initialisierung des Programmzählers auf die zweite Copper-Liste COP2LC(H+L)).

Bitbelegung und Funktion (für die zweite Befehlsliste) wie COPJMP1.

DIWSTOP	\$090	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Display Window STOP = Endposition der Bildschirm-Darstellung)

Dieses Register legt die Position der rechten, unteren Ecke der Bildschirm-Darstellung fest. Sie ist vertikal auf die unteren zwei Drittel und horizontal auf das rechte Viertel des Bildschirms beschränkt.

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[V7							V0]	[H7							H0]

Vx: Vertikale Startposition des Darstellungsbereichs mit einer Auflösung von acht Bits.

Hx: Horizontale Startposition [8-Bit-Wertebereich].

Der horizontale Stoppwert (Hx-Bits) des Bildschirmfensters wird intern um ein gesetztes neuntes Bit erweitert, so daß sein Wertebereich effektiv bei 256 (\$100) beginnt – schließlich sind kleinere Werte auch kaum sinnvoll. Die vertikale Position wird ebenfalls durch ein imaginäres 9. Bit ergänzt, das immer auf einen anderen Pegel liegt als das V7-Bit. Dadurch sind vertikale Positionen zwischen 128 (=\$80) und 383 (=\$17F) möglich, die vertikale Endposition kann sich also auch im oberen Bildschirmbereich befinden.

DIWSTRT	\$08E	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Display Window STaRT = Startposition der Bildschirmdarstellung)

DIWSTRT bestimmt die Position der oberen, linken Ecke der Bilddarstellung. Dieser Punkt muß in den oberen zwei Drittel bzw. linken drei Viertel des Darstellungsbereichs liegen. Die Bitbelegung ist mit DIWSTOP identisch.

DDFSTOP	\$094	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Display Data Fetch STOP = Ende des Display-DMA in einer Rasterzeile)

DDFSTOP markiert den letzten Buszyklus der Display-DMA.

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	H7	H6	H5	H4	H3	H2	x	x

x: nicht verwendet (sollte auf 0 gesetzt werden).

Hx: Startposition der Display-DMA. Es stehen sechs Bits zur Verfügung, die Auflösung beträgt acht Buszyklen bei niedrigauflösender Darstellung und vier Buszyklen im High-Resolution-Modus.

DDFSTRT	\$092	w	AGNUS	(P)
----------------	--------------	----------	--------------	------------

(Display Data Fetch STaRT = Start des Display-DMA in einer Rasterzeile)

Bitbelegung wie DDFSTOP.

Berechnung für Low-Resolution:

DDFSTRT = (horizontale Startposition/2 – 8,5) AND \$FFF8

Im HIRES-Mode:

DDFSTRT = (horizontale Startposition/2 – 4,5) AND \$FFFC

DMACON	\$096	w	AGNUS/DENISE/PAULA	(K)
---------------	--------------	----------	---------------------------	------------

(DMA CONTRol write = DMA-Kontrollregister)

Zentrales Steuerregister des DMA-Systems zur Aktivierung der einzelnen DMA-Kanäle.

Bitbelegung:

Nr.	Label	Funktion
15	SET/CLR	Set/Clear-Bit 1 = setzen, 0 = löschen.
14	BBUSY	Blitter-Status-Bit (nur lesen). Dieses Bit signalisiert, daß der Blitter noch mit der aktuellen Aufgabe beschäftigt ist.
13	BZERO	Ebenfalls ein Status-Bit des Blitters. Es wird gesetzt, wenn das Ergebnis einer Blitter-Operation Null ist.
12-11	–	nicht verwendet.
10	BLTPRI	Ist dieses Bit gesetzt, so hat der Blitter eine höhere Bus-Priorität als die CPU. Der Prozessor muß warten, bis der Blitter den Chip-RAM-Bus wieder freigibt.
09	DMAEN	DMA-Hauptschalter, erlaubt das schnelle ein- (= 1) bzw. ausschalten (= 0) der DMA-Kanäle. Ist es gesetzt, so werden alle DMA-Kanäle entsprechend ihrer Status-Bits aktiviert, ist es gelöscht, findet (unabhängig von der Initialisierung der Kanäle) keine DMA-Übertragung statt.
08	BPLEN	Bitplane-DMA aktivieren.
07	COPEN	Copper-DMA einschalten.
06	BLTEN	Blitter-DMA ein.
05	SPREN	Sprite-DMA ein.
04	DSKEN	Disk-DMA ein.
03-00	AUDxEN	Audio-DMA ein für Tonkanal x (Bit 3 = Kanal3, Bit 2 = Kanal2).

DMACONR \$002 r AGNUS/DENISE/PAULA (K)

(DMA CONTROL Read = Statusregister des DMA-Systems)

Belegung und Funktion der einzelnen Bits entsprechen weitgehend dem Schreibregister (DMACON). BBUSY und BZERO (nur lesen) reflektieren den aktuellen Blitter-Status.

DSKBYTR \$01A r PAULA (P)

(DiSK data BYTe and status Read = Daten- und Status-Register der Disk-Hardware)

Status- und Arbeitsregister

Bitbelegung:

Nr.	Label	Funktion
15	BYTEREADY	signalisiert ein gültiges Lesebyte von der Diskette (bleibt lediglich bis zum nächsten DMA-Zugriff gültig: zwei oder vier Mikrosekunden).
14	DMAON	zeigt aktuellen DMA-Status.
13	DSKWRITE	reflektiert WRITE-Bit im DSKLEN-Register
12	WORDEQUAL	Sync-Wort gefunden.
11-8	-	nicht belegt.
7-0	DATA	niederwertiges Byte der Disk-Lesedaten.

DSKDAT	\$026	w	PAULA	(D)
---------------	--------------	----------	--------------	------------

(DiSK DATa write = Disk-Datenpuffer für Schreibzugriff)

16-Bit-Datenregister des Floppy-Controllers als Datenpuffer zwischen Chip-RAM und Disketten-Laufwerk. Das Register wird normalerweise vom DMA-Controller beschrieben, es ist aber auch ein direkter Prozessorzugriff möglich.

DSKDATR	\$008	er	PAULA	(D)
----------------	--------------	-----------	--------------	------------

(DiSK DATa Read = Disk-Datenpuffer für Lesezugriff)

Datenregister für die Übertragungsrichtung Laufwerk → Speicher. Da es sich um ein Early-Read-Register handelt, sind CPU-Zugriffe nicht erlaubt.

DSKLEN	\$024	w	PAULA	(K)
---------------	--------------	----------	--------------	------------

(DiSK LENgth = Länge der DMA-Daten und Startimpuls des Disk-Zugriffs)

Kontrollregister für Speicherblock-Länge, Zugriffsrichtung und DMA-Aktivierung.

Bitbelegung:

Nr.	Label	Funktion
15	DMAEN	Disk-DMA-einschalten (in Verbindung mit DMACON).
14	WRITE	Datenflußrichtung: 1 = schreiben / 0 = lesen.
13-0	LENGTH	Wortlänge des Datenpuffers im Speicher (max. 16 KByte).

DSKPTH **\$020** **w** **AGNUS** **(A)**

(DiSK PoinTer High = Adreßregister für die höchsten drei Bits der Disk-DMA-Daten)

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 x x x x x x x x x x x x x x [18-16] = höherwertige drei Bits der 19-Bit-Speicheradresse

DSKPTL **\$022** **w** **AGNUS** **(A)**

(DiSK PoinTer Low = Adreßzeiger auf Disk-DMA-Daten, untere 16 Bits)

DSKPTH und DSKPTL enthalten die 19-Bit-Adresse der Disk-Daten im Chip-RAM. Der Zeiger muß vor der Aktivierung der Disk-DMA initialisiert werden.

Bitbelegung:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 [niederwertiger Teil der 19-Bit-Speicheradresse]

DSKSYNC **\$07E** **w** **PAULA** **(P)**

(DiSK SYNC register = Datenwort für Disk-Synchronisation)

Die Datenübertragung vom Laufwerk zum Chip-RAM-Speicher startet erst dann, wenn der Floppy-Controller das Synchronisationswort (DSKSYNC-Register) im Datenstrom gefunden hat. Dadurch wird der gezielte Zugriff auf bestimmte Teile einer Spur möglich.

INTENA **\$09A** **w** **PAULA** **(K)**

(INTerrupt ENABle bits = Interrupt-Kontroll-Bits)

Zentrales Kontrollregister zum Ein- bzw. Ausblenden der einzelnen Interrupt-Quellen. Der Zugriff auf das Register erfolgt (wie bei DMACON) durch einen Set/Clear-Mechanismus. Ist das korrespondierende Bit einer Interrupt-Quelle gesetzt, wird ihre Anforderung akzeptiert und zum Prozessor weitergeleitet.

Bitbelegung: (Die eingeklammerten Zahlen stellen den zugewiesenen Prozessor-Interrupt-Level dar.)

Nr.	Label	Funktion
15	SET/CLR	Set/Clear-Bit. Schaltet zwischen Setz- (1) und Lösch-Modus um. Es werden nur die Bits beeinflusst, die an den Positionen der Einer-Bits im Maskenwort liegen.
14	INTEN	Interrupt-Hauptschalter (siehe Bit 9, DMACON). Unterbrechungsanforderungen werden erst dann bearbeitet, wenn dieses Bit gesetzt ist. Durch Löschen des Bits läßt sich das gesamte Interrupt-System sperren, ohne jede Quelle einzeln verbieten zu müssen.
13	EXTER	(6) Externer Interrupt von CIA-B oder Expansion-Port.
12	DSKSYN	(5) Disketten-Synchronisationsmarkierung gefunden.
11	RBF	(5) Empfangspuffer der seriellen Schnittstelle voll.
10-07	AUDx	(4) Audio-DMA-Zyklus beendet (AUDLEN heruntergezählt). (Bit 10 = Tonkanal 3, Bit 9 = Tonkanal 2 usw.)
06	BLIT	(3) Blitter-Operation beendet.
05	VERTB	(3) Beginn der vertikalen Austastlücke.
04	COPER	(3) Copper-Interrupt (durch Copper-Liste selbst ausgelöst!).
03	PORTS	(2) Interrupt von CIA-A.
02	SOFT	(1) für Software-Interrupt reserviert.
01	DSKBLK	(1) Disk-DMA fertig.
00	TBE	(1) Sendepuffer der seriellen Schnittstelle leer.

INTENAR **\$01C** **r** **PAULA** **(K)**

(INTerrupt ENABle bits Read = Lesen der Interrupt-Kontroll-Bits)

Bitbelegung wie INTENA.

INTREQ	\$09C	w	PAULA	(P)
---------------	--------------	----------	--------------	------------

(INTerrupt REQuest bits = Interrupt-Statusbits (setzen und löschen))

Die Bitbelegung ist mit der des INTENA-Registers identisch. Sämtliche Interrupt-Anforderungen des Systems laufen in diesem Register zusammen. Der Vergleich mit INTENA entscheidet über die Annahme oder Ablehnung der Anforderung. Das INTREQ-Register ist für die CPU voll transparent, so daß sich System-Interrupts auch durch Setzen der korrespondierenden Status-Bits simulieren lassen (Bit 2 ist sogar speziell für Software-Interrupts reserviert).

Nach der Bearbeitung einer Unterbrechungs-Routine muß der Programmierer das auslösende Interrupt-Bit löschen, da das Register nicht automatisch zurückgesetzt wird.

INTREQR	\$01E	r	PAULA	(P)
----------------	--------------	----------	--------------	------------

(INTerrupt REQuest bits Read = Leseadresse des Interrupt-Anforderungs-Registers)

Das Register reflektiert den aktuellen Status des Interrupt-Systems und erlaubt die exakte Lokalisierung der auslösenden Interrupt-Quelle. Dies ist notwendig, da die CPU lediglich über 7 Interrupt-Ebenen verfügt, das System aber 14 verschiedene Unterbrechungs-Quellen unterstützt. Die Handler des Betriebssystems werten das INTREQR-Register aus (polling) und springen in die korrespondierende Unterbrechungs-Routine.

JOY0DAT	\$00A	r	DENISE	(P)
----------------	--------------	----------	---------------	------------

(JOYstick mouse 0 DATa = Datenregister für Maus und Joystick am Game-Port 0)

Das Register enthält den aktuellen Stand der beiden 8-Bit-Zähler für vertikale bzw. horizontale Mausposition.

Bitbelegung: (Der vertikale Zähler ist mit Pins 1 und 3, der horizontale Zähler mit den Pins 2 und 4 des Game-Ports verbunden)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0]	[X7	X6	X5	X4	X3	X2	X1	X0]

Die vier Maussignale liegen an den Mauseingängen des DENISE-Chips. Die Anschlüsse der Game-Ports sind gemultiplext und werden abwechselnd auf die Eingänge

von DENISE gelegt, wo sie gesammelt und auf die internen Zähler verteilt werden. Die Joystickleitungen sind ebenfalls mit diesen Eingängen verbunden, so daß die Hebelstellung aus dem Pegelniveau der beiden niederwertigsten Bits (X0, X1, Y0, Y1) eines Zählregisters hervorgeht. Da die Signale aber zur Generierung des Zähltaktes modifiziert werden, muß ihr Ursprungspegel durch logische Verknüpfung bzw. Invertierung wiederhergestellt werden:

Richtung	logische Operation
oben	$Y1 \text{ XOR } Y0 = 1$
links	$Y1 = 1$
unten	$X1 \text{ XOR } X0 = 1$
rechts	$X1 = 1$

JOY1DAT	\$00C	r	DENISE (P)
----------------	--------------	----------	-------------------

(JOYstick mouse 1 DATA = Joystick/Maus-Kontrollregister für den rechten Anschluß 1)

Bitbelegung und Funktion wie JOY0DAT.

JOYTEST	\$036	w	DENISE (K)
----------------	--------------	----------	-------------------

(JOYstick mouse TEST = Testmodus durch gleichzeitiges Beschreiben sämtlicher Mauszähler)

Ein Schreibzugriff auf dieses Register initialisiert alle vier Mauszähler.

Bitbelegung:

15 14 13 12 11 10 9 8 7	6 5 4 3 2 1 0
[Y7 Y6 Y5 Y4 Y3 Y2 xx xx] [X7	X6 X5 X4 X3 X2 xx xx]

Die mit xx bezeichneten Bits können nicht beeinflußt werden, da sie direkt durch die Eingangspegel der Mausleitungen gesteuert werden und nicht gepuffert sind.

POT0DAT **\$012** **r** **PAULA** **(P)**

(POT counter DATA left pair = Datenregister des Potentiometer-Zählers am POT-Port 0 [linker Paddle-Eingang])

In diesem Register befinden sich die beiden 8-Bit-Zähler des linken Analog-Ports. Die Widerstandsmessung wird durch Setzen des START-Bits in POTGO gestartet.

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
[vertikaler Zähler]								[horizontaler Zähler]							

POT1DAT **\$014** **r** **PAULA** **(P)**

(POT counter DATA right pair = Datenregister des Potentiometer-Zählers am Potentiometer-Zähler des rechten [1] Paddle-Paares)

Dieses Register ist in seiner Belegung identisch mit POT0DAT.

POTGO **\$034** **w** **PAULA** **(K/P)**

(POT port data write and start [GO] = Register zum Start der Potentiometer-Messung und Steuerung der Schnittstelle-Funktionen)

Das POTGO-Register erlaubt die Nutzung der Analog-Ports als bidirektionale 2-Bit-Digital-Ports (jede Leitung kann wahlweise als Ein- bzw. Ausgang programmiert werden).

Bitbelegung: (Bei gelöschten OUTxx-Bits sind die entsprechenden Leitungen als Eingänge geschaltet.)

Nr.	Label	Funktion
15	OUTRY	POTY-Leitung des rechten Gameports als Ausgang schalten.
14	DATRY	Datenbit der POTY-Leitung des rechten Gameports.
13	OUTRX	POTX-Leitung des rechten Gameports als Ausgang schalten.
12	DATRX	Datenbit der POTX-Leitung des rechten Gameports.
11	OUTLY	POTY-Leitung des linken Gameports als Ausgang schalten.
10	DATLY	Datenbit der POTY-Leitung des linken Gameports.

09	OUTLX	POTX-Leitung des linken Gameports als Ausgang schalten.
08	DATLX	Datenbit der POTX-Leitung des linken Gameports.
07-01	0	reserviert für Chip-Identifikationsnummer (zur Zeit 0).
00	START	Startbit der Widerstandsmessung.

POTGOR **\$016** **r** **PAULA** **(K)**

(POT port data Read = Leseadresse von POTGO)

POTGOR reflektiert die aktuellen Pegelzustände der Potentiometereingänge im Schnittstellen-Modus. Die Bitbelegung entspricht POTGO.

REFPTR **\$028** **w** **AGNUS** **(Spezialfunktion)**

(REFresh PoinTeR = Refreshzeiger für die zyklische Aufrichtung der dynamischen RAM-Chips)

Zu Beginn jeder Rasterzeile legt der DMA-Controller 4 Refreshadressen auf den Chip-RAM-Adreßbus, die den Zeilenrefresh-Mechanismus der DRAMs aktivieren und die Speicherdaten stabilisieren.

SERDAT **\$030** **w** **PAULA** **(P)**

(SERial port DATa and stop bits write = Sendepuffer der seriellen Schnittstelle)

Sendedatenpuffer der seriellen Schnittstelle.

unterstützte Formate Datenformat

LONG: 9 Daten-, 1 Stoppbit (Bit 9)
normal: 8 Daten-, 1 Stoppbit (Bit 8)
 8 Daten-, 2 Stoppbits (Bit 8 und 9)

Bitbelegung:

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
0 0 0 0 0 0 S D8 D7 D6 D5 D4 D3 D2 D1 D0

S: Stoppmarkierung

Dx: Datenbits

SERDATR \$018 r PAULA (P)

(SERial port DATa and status Read = Lese- und Status-Register der seriellen Schnittstelle)

SERDATR dient als Empfangsdatenpuffer der UART-Schaltung. Neben Datenwert und Stoppmarkierung enthält es noch einige Statusbits.

Bitbelegung:

Nr.	Label	Funktion
15	OVRUN	Überlauf des Empfangsschieberegisters. Das Bit wird automatisch gelöscht, sobald das RBF-Bit im INTREQ-Register zurückgesetzt ist.
14	RBF	Receive-Buffer-Full-Bit (= Empfangsdatenpuffer voll). Ist mit dem RBF-Bit des Interrupt-Systems gekoppelt.
13	TBE	Transmit Buffer Empty. Dieses Bit wird gesetzt, wenn alle Daten des SERDAT-Registers gesendet wurden (siehe TBE-Bit von INTREQ).
12	TSRE	Transmit Shift Register Empty = Sendeschieberegister leer.
11	RXD	Dieses Bit reflektiert den aktuellen Zustand der RXD-Leitung.
10	–	nicht verwendet.
09	STP	1. Stoppbit (optional, LONG-Bit in SERPER).
08	STP/D8	Daten- bzw. 1./2. Stoppbit (je nach Initialisierung).
07–00	Dx	Datenbits 7 bis 0.

SERPER \$032 w PAULA (K)

(SERial port PERiod and control = Ausgabeperiode und Datenformat der seriellen Schnittstelle)

SERPER bestimmt den Zeitabstand (in Buszyklen) zwischen zwei Übertragungszyklen der UART-Schaltung und legt die Länge der empfangenen Daten (neun oder acht Bits) fest.

Bitbelegung:

Nr.	Label	Funktion
15	LONG	Dieses Bit bestimmt, ob ein gelesenes Datenbyte acht (= 0) oder neun Bits (= 1) umfaßt.

Nr.	Label	Funktion
14-00	RATE	Übertragungsperiode in Buszyklen. Bei einer effektiven Baud-Rate von n Bits/s, resultiert die erforderliche Initialisierung des SERPER-Registers aus dem Term:

$$\text{RATE} = \frac{1}{((n+1)*0,2817)}$$

SPRxDATA	\$144+x*\$008	w	DENISE	(D)
-----------------	----------------------	----------	---------------	------------

(SPRite x image DATA register A = DMA-Datenregister A für Sprite x)

SPRxDATA enthält zusammen mit SPRxDATB die Bildinformation eines Sprites. Ein Zugriff auf das Register aktiviert die Sprite-Darstellung des DENISE-Chips, der daraufhin die horizontale Position des Rasterstrahls mit der Spaltenangabe im SPRxPOS-Register (SHx-Bits) vergleicht. Stimmen beide Werte überein, werden die Spritedaten entsprechend ihrer Priorität in die Videoausgabe gemischt.

SPRxDATB	\$146+x*\$008	w	DENISE	(D)
-----------------	----------------------	----------	---------------	------------

(SPRite x image DATA register B = DMA-Datenregister B für Sprite x)

Zweiter 16-Bit-Datenpuffer für die Sprite-Darstellung (siehe SPRxDATA).

SPRxCTL	\$142+x*\$008	w	AGNUS/DENISE	(K)
----------------	----------------------	----------	---------------------	------------

(SPRite x vertical stop position and ConTroL data = Kontrollregister für vertikale Stopposition und Darstellungsparameter von Sprite x)

Mit der Aktivierung der Sprite-DMA durch Setzen des SPREN-Bits im DMACON-Register überträgt der DMA-Controller die ersten beiden Worte der Sprite-Datenliste in die Kontrollregister SPRxCTL und SPRxPOS. Sie enthalten die horizontale (Auflösung: 1 Low-Res-Pixel) bzw. vertikale (Auflösung: 1 Rasterzeile) Bildschirmposition des Sprites.

Der Controller wartet nun bis der Rasterstrahl den vertikalen Startwert (SVx-Bits) erreicht hat und beginnt damit, die Pufferregister SPRDATA bzw. SPRDATAB zu füllen. Der vertikale Stoppwert (EVx-Bits, letzte Spritezeile +1) markiert das Ende der Datenliste und signalisiert dem Controller, daß die folgenden zwei Worte als Kontrollinformationen aufzufassen sind. Haben sie den Wert Null, so wird der DMA-Kanal bis zum Beginn des nächsten Bildschirmrasters inaktiv.

Jeder Zugriff auf die Kontrollregister schaltet die Sprite-Darstellung des Videochips ab. Auf diese Weise wird verhindert, daß ungültige Daten auf den Bildschirm gelangen. Erst wenn die eigentliche Datenübertragung durch Schreiben in die DMA-Datenregister startet, wird DENISE wieder aktiv und wartet auf die horizontale Startposition (SH0 bis SH8) des Sprites.

Bitbelegung:

Nr.	Label	Funktion
15-08	EVx	Untere acht Bits des vertikalen Stoppwerts; wird vom DMA-Controller ständig mit der aktuellen Rasterzeile verglichen, um bei Gleichstand die Sprite-Darstellung abzuschalten.
07	ATT	Attach-Bit (gilt nur für Sprites mit ungeraden Nummern). Ist es gesetzt, so werden die ungeraden und geraden Sprites zu Sprite-Paaren mit 16-farbiger Darstellung kombiniert.
06-03	—	nicht verwendet.
02	SV8	9. Bit des vertikalen Startwerts. Die übrigen acht Bits befinden sich im SPRxPOS-Register.
01	EV8	9. Bit des vertikalen Stoppwerts.
00	SH0	Unterstes Bit des horizontalen Startwerts. Die höherwertigen acht Bits liegen im SPRxPOS-Register.

SPRxPOS	\$140+x*\$008	w	AGNUS/DENISE (K)
----------------	----------------------	----------	-------------------------

(SPRite x vertical and horizontal start POSition data = Bildschirmposition von Sprite x [0...7])

Zweites Kontrollwort für die Bildschirmposition von Sprite x (x = 0-7).

Bitbelegung:

Nr.	Label	Funktion
-----	-------	----------

15-08	SVx	Vertikaler Startwert (untere acht Bits) der Sprite-Darstellung. Das 9. Bit steht in SPRxCTL-Register.
07-00	SHx	Horizontaler Startwert (obere acht Bits) der Sprite-Darstellung. Das untere 0. Bit liegt in SPRxCTL.

SPRxPTH	\$120+x*\$004	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(SPRite x PoinTer High = Adreßzeiger auf Datenliste von Sprite 0 bis 7, obere drei Bits)

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

[18-16] = höherwertige drei Bits der 19-Bit-Speicheradresse

SPRxPTL	\$122+x*\$004	w	AGNUS	(A)
----------------	----------------------	----------	--------------	------------

(SPRite x PoinTer Low = Adreßregister (untere 16 Bits) für die Speicherposition der Datenliste von Sprite 0 bis 7)

Zusammen mit SPRxPTH enthält SPRxPTL die 19-Bit-Adresse der Sprite-Daten im Chip-RAM-Speicher. Da die Adreßzeiger vom DMA-Controller laufend erhöht werden, müssen sie in jeder vertikalen Austastlücke neu initialisiert werden (durch Prozessor oder Copper).

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

[niederwertiger Teil der 19-Bit-Speicheradresse]

STREQU	\$038	s	DENISE	(S)
---------------	--------------	----------	---------------	------------

(STrobe for horizontal sync with vertical blank and EQU = Strobe-Adresse zur Zeilensynchronisation des Videoshifters mit der vertikalen Austastlücke)

STRHOR	\$03C	s	DENISE/PAULA	(S)
---------------	--------------	----------	---------------------	------------

(STRobe for horizontal sync = Strobe-Adresse für horizontale Synchronisation)

Synchronisationsimpuls des horizontalen Rasterstrahl-Zählers (AGNUS), signalisiert den Beginn einer neuen Bildschirmzeile.

STRLONG	\$03E	s	DENISE	(S)
----------------	--------------	----------	---------------	------------

(STRobe for identification of LONG horizontal line = Strobe-Leitung zur Identifizierung einer langen Rasterzeile)

Wird bei der Darstellung einer langen Rasterzeile von 228 Buszyklen aktiviert.

STRVBL	\$03A	s	DENISE	(S)
---------------	--------------	----------	---------------	------------

(STRobe for horizontal sync with Vertical BLank = Strobe-Adresse zur horizontalen Synchronisation mit der vertikalen Austastlücke)

VPOSr	\$004	r	AGNUS	(P)
--------------	--------------	----------	--------------	------------

(Read Vertical most significant bit and frame flop = Register für vertikale Rasterstrahlposition (höchstes Bit) und Bildkennung)

Dieses Register enthält das höchstwertige Bit der vertikalen Strahlposition und markiert die Darstellung eines langen Rasters (Frame) mit 313 Zeilen (ungerade Zeilen).

Bitbelegung:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOF	x	x	x	x	x	x	x	x	x	x	x	x	x	x	V8

LOF = Long-Frame-Erkennungsbit

V8 = 9. Bit der vertikalen Rasterstrahlposition.

VPOSW	\$02A	w	AGNUS (P)
--------------	--------------	----------	------------------

(Write vertical most significant bit and frame flop = Schreibadresse des Zählers für vertikale Rasterstrahlposition und Bildkennung)

siehe VPOSR.

VHPOSR	\$006	w	AGNUS (P)
---------------	--------------	----------	------------------

(Read Vertical and Horizontal POSition of beam and lightpen = Lese-Register vertikalen und horizontalen Rasterstrahlposition)

VHPOSR enthält die aktuelle Rasterstrahlposition bzw. die eingefrorene Bildschirmposition des Lightpens.

Bitbelegung:

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
[V7 V0] [H7 H0]

Vx = niederwertige acht Bits der vertikalen Strahlposition
(Auflösung: Eine Non-Interlace-Zeile, zwei Interlace-Zeilen).

Hx = 8-Bit-Wert der horizontalen Strahlposition
(Auflösung: zwei niedrigauflösende Pixel, vier hochauflösende Pixel)

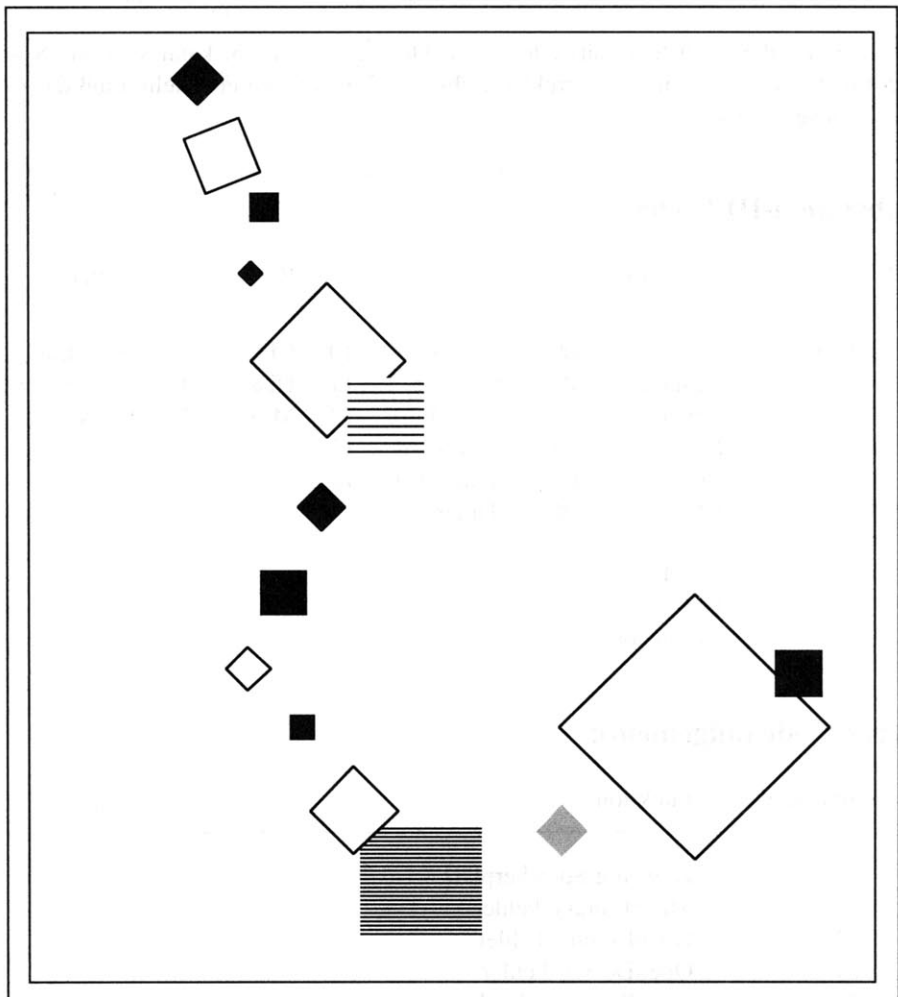
VHPOSW	\$02C	w	AGNUS (P)
---------------	--------------	----------	------------------

(Write Vertical and Horizontal POSition of beam and lightpen = Schreibadresse des vertikalen und horizontalen Rasterstrahl-Zählers)

Bitbelegung wie VHPOSR.

Anhang C

Die Guru - Meditation - Fehlermeldungen



Die Guru-Meditation-Fehlermeldungen

Aufbau des Fehler-Vektors:

Subsystem ID	Error (allgemein)	Error(speziell)	Task-Adresse
0/0	00	0000	00000000

Wichtig: Ist die höchste Binärstelle des Vektors größer als 3, kann sich das System nicht mehr erholen. Um den korrekten Subsystem-ID-Code zu ermitteln, muß das oberste Bit ausgeblendet werden.

Subsystem-ID-Codes:

CPU	Libraries	Devices	Resource	weitere
CPU-Traps	00 Exec	00 Audio	10 CIA	00 Bootstrap
	Graphics	02 Console	11 Disk	21 Workbench
	Layers	03 GamePort	12 Misc	22 DiskCopy
	Intuition	04 Keyboard	13	
	Math	05 TrackDisk	14	
	Clist	06 Timer	15	
	DOS	07		
	RAM	08		
	Icon	09		
	Expansion	0A		

Error Code (allgemein):

Bitkombination	Funktion
----------------	----------

01	zu wenig Speicherplatz
02	MakeLibrary-Fehler
03	OpenLibrary-Fehler
04	OpenDevice-Fehler
05	OpenResource-Fehler
06	E/A-Error
07	kein Signal empfangen

Error Code (speziell):**Exec Library**

81 00 0001	Checksumme des 68000-Exception-Vektors falsch
81 00 0002	Checksumme von ExecBase falsch
81 00 0003	Checksumme der Library falsch
81 00 0004	kein Speicherplatz für die Erstellung der Library
81 00 0005	zerstörte Memory-List
81 00 0006	kein Speicherplatz für Interrupt-Server
81 00 0007	InitAPtr
81 00 0008	Semaphore zerstört
81 00 0009	bereits delokierter Speicher erneut freigegeben
81 00 000A	Bogus-Exception

Graphics Library

82 01 0001	kein Speicherplatz für Copper-Display-Liste
82 01 0002	kein Speicherplatz für Copper-Befehls-Liste
82 00 0003	Überlauf der Copper-List
82 00 0004	Überlauf der provisorischen Copper-Liste
82 01 0005	kein Speicherplatz für den Kopf der Copper-Liste
82 01 0006	kein Speicherplatz für Long-Frame
82 01 0007	kein Speicherplatz für Short-Frame
82 01 0009	kein Speicherplatz für Flood-Fill-Operation
82 01 000A	kein Speicherplatz für BlitBitMap-Funktion
82 01 000B	kein Speicherplatz für Region
82 01 0030	kein Speicherplatz für MakeVPort-Funktion
82 01 1234	GfxNoLCM

Layers Library

83 01 0001	kein Speicherplatz für Layer-Bereich
------------	--------------------------------------

Intuition Library

84 00 0001	unbekannter Gadget-Typ
04 00 0001	"Recoverable"-Form der letzten Fehlermeldung
84 01 0002	kein Speicherplatz für Message-Port
84 01 0003	kein Speicherplatz für Item-Plane-Allocation
84 01 0004	kein Speicherplatz für Sub-Allocation
84 01 0005	kein Speicherplatz für Plane-Allocation
84 00 0006	Top der Item-Box < RelZero
84 01 0007	kein Speicherplatz für Screen
84 00 0008	OpenScreen: kein Speicherplatz für Raster-Allocation
84 00 0009	unbekannter System-Screen-Typ
84 01 000A	kein Speicherplatz für System-Window-Gadgets
84 01 000B	kein Speicherplatz für Window
84 00 000C	"BAD State" beim Eintritt in Intuition
84 00 000D	"BAD Message" von IDCMP empfangen
84 00 000E	unbekannte Meldung
84 00 000F	Console-Device läßt sich nicht öffnen

DOS Library

07 01 0001	kein Speicherplatz in der Startphase
07 00 0002	Fehler beim Beenden eines Tasks
07 00 0003	Qpkt Error
07 00 0004	unerwartetes Packet empfangen
07 00 0005	Freigabe-Vektor-Fehler
07 00 0006	Disk-Block-Sequenz-Fehler
07 00 0007	Bitmap zerstört
07 00 0008	Key schon freigegeben
07 00 0009	falsche Checksumme
07 00 000A	Disk Error
07 00 000B	Key außerhalb des erlaubten Bereichs
07 00 000C	bad overlay

RAM Library

08 00 0001	bad segment list
------------	------------------

Expansion Library

0A 00 0001 bad expansion free

TrackDisk Device

14 00 0001 Such-Fehler
14 00 0002 Fehler beim Warten auf das Timer-Signal

Timer Device

15 00 0001 falsche Anfrage
15 00 0002 bad supply

Disk Resource

21 00 0001 GetUnit hat bereits Disk Acces
21 00 0002 Interrupt-Fehler: kein aktives Unit

BootStrap

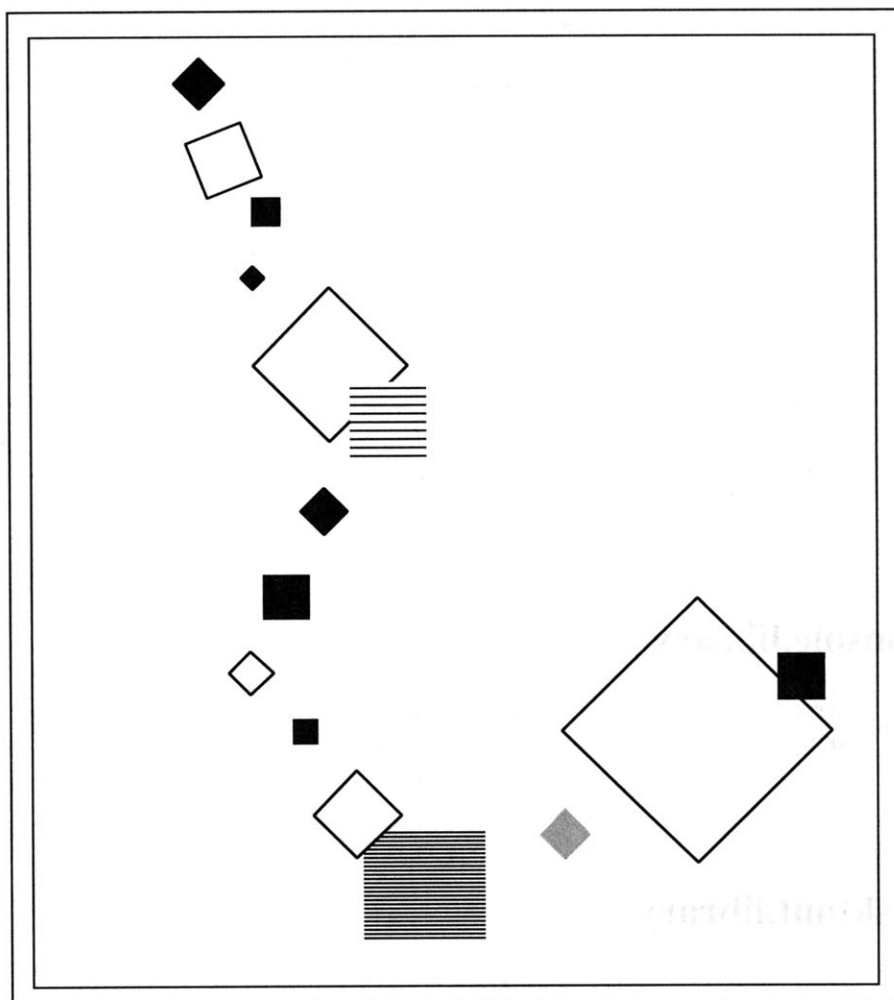
30 00 0001 Error-Meldung des Boot-Code

Prozessor-Traps

00 00 0002 Busfehler
00 00 0003 Adreßfehler
00 00 0004 Illegale Instruktion
00 00 0005 Division durch Null
00 00 0006 CHK-Befehl
00 00 0007 TRAPV-Befehl
00 00 0008 Privileg-Verletzung
00 00 0009 Trace-Modus
00 00 000A OpCode: 1010
00 00 000B OpCode: 1111

Anhang D

Library - Routinen nach Offset sortiert



clist.library

```

-30  0xFFE2  -0x01E  InitCLPool (cLPool, size) (a0, d0)
-36  0xFFDC  -0x024  AllocCLList (cLPool) (a1)
-42  0xFFD6  -0x02A  FreeCLList (cList) (a0)
-48  0xFFD0  -0x030  FlushCLList (cList) (a0)
-54  0xFFCA  -0x036  SizeCLList (cList) (a0)
-60  0xFFC4  -0x03C  PutCLChar (cList, byte) (a0, d0)
-66  0xFFBE  -0x042  GetCLChar (cList) (a0)
-72  0xFFB8  -0x048  UnGetCLChar (cList, byte) (a0, d0)
-78  0xFFB2  -0x04E  UnPutCLChar (cList) (a0)
-84  0xFFAC  -0x054  PutCLWord (cList, word) (a0, d0)
-90  0xFFA6  -0x05A  GetCLWord (cList) (a0)
-96  0xFFA0  -0x060  UnGetCLWord (cList, word) (a0, d0)
-102 0xFF9A  -0x066  UnPutCLWord (cList) (a0)
-108 0xFF94  -0x06C  PutCLBuf (cList, buffer, length) (a0, a1, d1)
-114 0xFF8E  -0x072  GetCLBuf (cList, buffer, length) (a0, a1, d1)
-120 0xFF88  -0x078  MarkCLList (cList, offset) (a0, d0)
-126 0xFF82  -0x07E  IncrCLMark (cList) (a0)
-132 0xFF7C  -0x084  PeekCLMark (cList) (a0)
-138 0xFF76  -0x08A  SplitCLList (cList) (a0)
-144 0xFF70  -0x090  CopyCLList (cList) (a0)
-150 0xFF6A  -0x096  SubCLList (cList, index, length) (a0, d0, d1)
-156 0xFF64  -0x09C  ConcatCLList (sourceCLList, destCLList) (a0, a1)

```

console.library

```

-42  0xFFD6  -0x02A  CDInputHandler (events, device) (a0, a1)
-48  0xFFD0  -0x030  RawKeyConvert (events, buffer, length, keyMap)
                        (a0, a1, d1, a2)

```

diskfont.library

```

-30  0xFFE2  -0x01E  OpenDiskFont (textAttr) (a0)
-36  0xFFDC  -0x024  AvailFonts (buffer, bufBytes, flags) (a0, d0, d1)
-42  0xFFD6  -0x02A  NewFontContents (fontsLock, fontName) (a0, a1)
-48  0xFFD0  -0x030  DisposeFontContents (fontContentsHeader) (a1)

```

dos.library

-30	0xFFE2	-0x01E	Open(name,mode) (d1,d2)
-36	0xFFDC	-0x024	Close(file) (d1)
-42	0xFFD6	-0x02A	Read(file,buffer,length) (d1,d2,d3)
-48	0xFFD0	-0x030	Write(file,buffer,length) (d1,d2,d3)
-54	0xFFCA	-0x036	Input() ()
-60	0xFFC4	-0x03C	Output() ()
-66	0xFFBE	-0x042	Seek(file,pos,offset) (d1,d2,d3)
-72	0xFFB8	-0x048	DeleteFile(name) (d1)
-78	0xFFB2	-0x04E	Rename(oldname,newname) (d1,d2)
-84	0xFFAC	-0x054	Lock(name,type) (d1,d2)
-90	0xFFA6	-0x05A	UnLock(lock) (d1)
-96	0xFFA0	-0x060	DupLock(lock) (d1)
-102	0xFF9A	-0x066	Examine(lock,fileInfoBlock) (d1,d2)
-108	0xFF94	-0x06C	ExNext(lock,fileInfoBlock) (d1,d2)
-114	0xFF8E	-0x072	Info(lock,parameter) (d1,d2)
-120	0xFF88	-0x078	CreateDir(name) (d1)
-126	0xFF82	-0x07E	CurrentDir(lock) (d1)
-132	0xFF7C	-0x084	IoErr() ()
-138	0xFF76	-0x08A	CreateProc(name,pri,segList,stack) (d1,d2,d3,d4)
-144	0xFF70	-0x090	Exit(returnCode) (d1)
-150	0xFF6A	-0x096	LoadSeg(filename) (d1)
-156	0xFF64	-0x09C	UnLoadSeg(segment) (d1)
-162	0xFF5E	-0x0A2	GetPacket(wait) (d1)
-168	0xFF58	-0x0A8	QueuePacket(packet) (d1)
-174	0xFF52	-0x0AE	DeviceProc(name) (d1)
-180	0xFF4C	-0x0B4	SetComment(name,comment) (d1,d2)
-186	0xFF46	-0x0BA	SetProtection(name,mask) (d1,d2)
-192	0xFF40	-0x0C0	DateStamp(date) (d1)
-198	0xFF3A	-0x0C6	Delay(timeout) (d1)
-204	0xFF34	-0x0CC	WaitForChar(file,timeout) (d1,d2)
-210	0xFF2E	-0x0D2	ParentDir(lock) (d1)
-216	0xFF28	-0x0D8	IsInteractive(file) (d1)
-222	0xFF22	-0x0DE	Execute(string,file,file) (d1,d2,d3)

exec.library

-30	0xFFE2	-0x01E	Supervisor() ()
-36	0xFFDC	-0x024	ExitIntr() ()
-42	0xFFD6	-0x02A	Schedule() ()
-48	0xFFD0	-0x030	Reschedule() ()
-54	0xFFCA	-0x036	Switch() ()
-60	0xFFC4	-0x03C	Dispatch() ()
-66	0xFFBE	-0x042	Exception() ()
-72	0xFFB8	-0x048	InitCode(startClass,version) (d0,d1)
-78	0xFFB2	-0x04E	InitStruct(initTable,memory,size) (a1,a2,d0)
-84	0xFFAC	-0x054	MakeLibrary(funcInit,structInit,libInit,data,code) (a0,a1,a2,d0,d1)
-90	0xFFA6	-0x05A	MakeFunctions(target,functionArray,funcDispBase) (a0,a1,a2)
-96	0xFFA0	-0x060	FindResident(name) (a1)
-102	0xFF9A	-0x066	InitResident(resident,segList) (a1,d1)
-108	0xFF94	-0x06C	Alert(alertNum,parameters) (d7,a5)
-114	0xFF8E	-0x072	Debug() ()
-120	0xFF88	-0x078	Disable() ()
-126	0xFF82	-0x07E	Enable() ()
-132	0xFF7C	-0x084	Forbid() ()
-138	0xFF76	-0x08A	Permit() ()
-144	0xFF70	-0x090	SetSR(newSR,mask) (d0,d1)
-150	0xFF6A	-0x096	SuperState() ()
-156	0xFF64	-0x09C	UserState() ()
-162	0xFF5E	-0x0A2	SetIntVector(intNumber,interrupt) (d0,a1)
-168	0xFF58	-0x0A8	AddIntServer(intNumber,interrupt) (d0,a1)
-174	0xFF52	-0x0AE	RemIntServer(intNumber,interrupt) (d0,a1)
-180	0xFF4C	-0x0B4	Cause(interrupt) (a1)
-186	0xFF46	-0x0BA	Allocate(freelist,byteSize) (a0,d0)
-192	0xFF40	-0x0C0	Deallocate(freeList,memoryBlock,byteSize) (a0,a1,d0)
-198	0xFF3A	-0x0C6	AllocMem(byteSize,requirements) (d0,d1)
-204	0xFF34	-0x0CC	AllocAbs(byteSize,location) (d0,a1)
-210	0xFF2E	-0x0D2	FreeMem(memoryBlock,byteSize) (a1,d0)
-216	0xFF28	-0x0D8	AvailMem(requirements) (d1)
-222	0xFF22	-0x0DE	AllocEntry(entry) (a0)
-228	0xFF1C	-0x0E4	FreeEntry(entry) (a0)
-234	0xFF16	-0x0EA	Insert(list,node,pred) (a0,a1,a2)
-240	0xFF10	-0x0F0	AddHead(list,node) (a0,a1)
-246	0xFF0A	-0x0F6	AddTail(list,node) (a0,a1)
-252	0xFF04	-0x0FC	Remove(node) (a1)
-258	0xFEFE	-0x102	RemHead(list) (a0)

-264	0xFE8	-0x108	RemTail(list) (a0)
-270	0xFE2	-0x10E	Enqueue(list,node) (a0,a1)
-276	0xFEEC	-0x114	FindName(list,name) (a0,a1)
-282	0xFEE6	-0x11A	AddTask(task,initialPC,finalPC) (a1,a2,a3)
-288	0xFEE0	-0x120	RemTask(task) (a1)
-294	0xFEDA	-0x126	FindTask(name) (a1)
-300	0xFED4	-0x12C	SetTaskPri(task,priority) (a1,d0)
-306	0xFECE	-0x132	SetSignal(newSignals,signalSet) (d0,d1)
-312	0xFEC8	-0x138	SetExcept(newSignals,signalSet) (d0,d1)
-318	0xFEC2	-0x13E	Wait(signalSet) (d0)
-324	0xFEBC	-0x144	Signal(task,signalSet) (a1,d0)
-330	0xFEB6	-0x14A	AllocSignal(signalNum) (d0)
-336	0xFEB0	-0x150	FreeSignal(signalNum) (d0)
-342	0xFEAA	-0x156	AllocTrap(trapNum) (d0)
-348	0xFE4	-0x15C	FreeTrap(trapNum) (d0)
-354	0xFE9E	-0x162	AddPort(port) (a1)
-360	0xFE98	-0x168	RemPort(port) (a1)
-366	0xFE92	-0x16E	PutMsg(port,message) (a0,a1)
-372	0xFE8C	-0x174	GetMsg(port) (a0)
-378	0xFE86	-0x17A	ReplyMsg(message) (a1)
-384	0xFE80	-0x180	WaitPort(port) (a0)
-390	0xFE7A	-0x186	FindPort(name) (a1)
-396	0xFE74	-0x18C	AddLibrary(library) (a1)
-402	0xFE6E	-0x192	RemLibrary(library) (a1)
-408	0xFE68	-0x198	OldOpenLibrary(libName) (a1)
-414	0xFE62	-0x19E	CloseLibrary(library) (a1)
-420	0xFE5C	-0x1A4	SetFunction(library,funcOffset,funcEntry) (a1,a0,d0)
-426	0xFE56	-0x1AA	SumLibrary(library) (a1)
-432	0xFE50	-0x1B0	AddDevice(device) (a1)
-438	0xFE4A	-0x1B6	RemDevice(device) (a1)
-444	0xFE44	-0x1BC	OpenDevice(devName,unit,ioRequest,flags) (a0,d0,a1,d1)
-450	0xFE3E	-0x1C2	CloseDevice(ioRequest) (a1)
-456	0xFE38	-0x1C8	DoIO(ioRequest) (a1)
-462	0xFE32	-0x1CE	SendIO(ioRequest) (a1)
-468	0xFE2C	-0x1D4	CheckIO(ioRequest) (a1)
-474	0xFE26	-0x1DA	WaitIO(ioRequest) (a1)
-480	0xFE20	-0x1E0	AbortIO(ioRequest) (a1)
-486	0xFE1A	-0x1E6	AddResource(resource) (a1)
-492	0xFE14	-0x1EC	RemResource(resource) (a1)
-498	0xFE0E	-0x1F2	OpenResource(resName,version) (a1,d0)
-504	0xFE08	-0x1F8	RawIOInit() ()
-510	0xFE02	-0x1FE	RawMayGetChar() ()

-516	0xFDFC	-0x204	RawPutChar(char) (d0)
-522	0xFDF6	-0x20A	RawDoFmt(format,data,putProc,putData) (a0,a1,a2,a3)
-528	0xFDF0	-0x210	GetCC() ()
-534	0xFDEA	-0x216	TypeOfMem(address) (a1)
-540	0xFDE4	-0x21C	Procure(semaphore,bidMsg) (a0,a1)
-546	0xFDDE	-0x222	Vacate(semaphore) (a0)
-552	0xFDD8	-0x228	OpenLibrary(libName,version) (a1,d0)
-558	0xFDD2	-0x22E	InitSemaphore(sigSem) (a0)
-564	0xFDCC	-0x234	ObtainSemaphore(sigSem) (a0)
-570	0xFDC6	-0x23A	ReleaseSemaphore(sigSem) (a0)
-576	0xFDC0	-0x240	AttemptSemaphore(sigSem) (a0)
-582	0xFDBA	-0x246	ObtainSemaphoreList(sigSem) (a0)
-588	0xFDB4	-0x24C	ReleaseSemaphoreList(sigSem) (a0)
-594	0xFDAE	-0x252	FindSemaphore(sigSem) (a1)
-600	0xFDA8	-0x258	AddSemaphore(sigSem) (a1)
-606	0xFDA2	-0x25E	RemSemaphore(sigSem) (a1)
-612	0xFD9C	-0x264	SumKickData() ()
-618	0xFD96	-0x26A	AddMemList(size,attributes,pri,base,name) (d0,d1,d2,a0,a1)
-624	0xFD90	-0x270	CopyMem(source,dest,size) (a0,a1,d0)
-630	0xFD8A	-0x276	CopyMemQuick(source,dest,size) (a0,a1,d0)

expansion.library

-30	0xFFE2	-0x01E	AddConfigDev (configDev) (a0)
-42	0xFFD6	-0x02A	AllocBoardMem (slotSpec) (d0)
-48	0xFFD0	-0x030	AllocConfigDev () ()
-54	0xFFCA	-0x036	AllocExpansionMem (numSlots, SlotAlign, SlotOffset) (d0, d1, d2)
-60	0xFFC4	-0x03C	ConfigBoard (board, configDev) (a0, a1)
-66	0xFFBE	-0x042	ConfigChain (baseAddr) (a0)
-72	0xFFB8	-0x048	FindConfigDev (oldConfigDev, manufacturer, product) (a0, d0, d1)
-78	0xFFB2	-0x04E	FreeBoardMem (startSlot, slotSpec) (d0, d1)
-84	0xFFAC	-0x054	FreeConfigDev (configDev) (a0)
-90	0xFFA6	-0x05A	FreeExpansionMem (startSlot, numSlots) (d0, d1)
-96	0xFFA0	-0x060	ReadExpansionByte (board, offset) (a0, d0)
-102	0xFF9A	-0x066	ReadExpansionRom (board, configDev) (a0, a1)
-108	0xFF94	-0x06C	RemConfigDev (configDev) (a0)
-114	0xFF8E	-0x072	WriteExpansionByte (board, offset, byte) (a0, d0, d1)
-120	0xFF88	-0x078	ObtainConfigBinding () ()
-126	0xFF82	-0x07E	ReleaseConfigBinding () ()
-132	0xFF7C	-0x084	SetCurrentBinding (currentBinding, bindingSize) (a0, d0)
-138	0xFF76	-0x08A	GetCurrentBinding (currentBinding, bindingSize) (a0, d0)
-144	0xFF70	-0x090	MakeDosNode (parmPacket) (a0)
-150	0xFF6A	-0x096	AddDosNode (bootPri, flags, dosNode) (d0, d1, a0)

graphics.library

```

-30  0xFFE2  -0x01E  BltBitMap
                        (srcBM,srcX,srcY,destBM,destX,destY,sizeX,sizeY,minterm)
                        (a0,d0,d1,a1,d2,d3,d4,d5,d6)
                        (mask,tempA)
                        (d7,a2)

-36  0xFFDC  -0x024  BltTemplate
                        (source,srcX,srcMod,destRP,destX,destY,sizeX,sizeY)
                        (a0,d0,d1,a1,d2,d3,d4,d5)

-42  0xFFD6  -0x02A  ClearEOL(rastPort) (a1)

-48  0xFFD0  -0x030  ClearScreen(rastPort) (a1)

-54  0xFFCA  -0x036  TextLength(rastPort,string,count) (a1,a0,d0)

-60  0xFFC4  -0x03C  Text(rastPort,string,count) (a1,a0,d0)

-66  0xFFBE  -0x042  SetFont(rastPortID,textFont) (a1,a0)

-72  0xFFB8  -0x048  OpenFont(textAttr) (a0)

-78  0xFFB2  -0x04E  CloseFont(textFont) (a1)

-84  0xFFAC  -0x054  AskSoftStyle(rastPort) (a1)

-90  0xFFA6  -0x05A  SetSoftStyle(rastPort,style,enable) (a1,d0,d1)

-96  0xFFA0  -0x060  AddBob(bob,rastPort) (a0,a1)

-102 0xFF9A  -0x066  AddVSprite(vSprite,rastPort) (a0,a1)

-108 0xFF94  -0x06C  DoCollision(rastPort) (a1)

-114 0xFF8E  -0x072  DrawGList(rastPort,viewPort) (a1,a0)

-120 0xFF88  -0x078  InitGels(dummyHead,dummyTail,gelsInfo) (a0,a1,a2)

-126 0xFF82  -0x07E  InitMasks(vsprite) (a0)

-132 0xFF7C  -0x084  RemIBob(bob,rastPort,viewPort) (a0,a1,a2)

-138 0xFF76  -0x08A  RemVSprite(vsprite) (a0)

-144 0xFF70  -0x090  SetCollision(type,routine,gelsInfo) (d0,a0,a1)

-150 0xFF6A  -0x096  SortGList(rastPort) (a1)

-156 0xFF64  -0x09C  AddAnimOb(obj,animationKey,rastPort) (a0,a1,a2)

-162 0xFF5E  -0x0A2  Animate(animationKey,rastPort) (a0,a1)

-168 0xFF58  -0x0A8  GetGBuffers(animationObj,rastPort,doubleBuffer) (a0,a1,d0)

-174 0xFF52  -0x0AE  InitGMasks(animationObj) (a0)

-180 0xFF4C  -0x0B4  DrawEllipse(rastPort,centerX,centerY,a,b) (a1,d0,d1,d2,d3)

-186 0xFF46  -0x0BA  AreaEllipse(rastPort,centerX,centerY,a,b) (a1,d0,d1,d2,d3)

-192 0xFF40  -0x0C0  LoadRGB4(viewPort,colors,count) (a0,a1,d0)

-198 0xFF3A  -0x0C6  InitRastPort(rastPort) (a1)

-204 0xFF34  -0x0CC  InitVPort(viewPort) (a0)

-210 0xFF2E  -0x0D2  MrgCop(view) (a1)

-216 0xFF28  -0x0D8  MakeVPort(view,viewPort) (a0,a1)

-222 0xFF22  -0x0DE  LoadView(view) (a1)

-228 0xFF1C  -0x0E4  WaitBlit() ()

```

-234	0xFF16	-0x0EA	SetRast(rastPort,color)(a1,d0)
-240	0xFF10	-0x0F0	Move(rastPort,x,y)(a1,d0,d1)
-246	0xFF0A	-0x0F6	Draw(rastPort,x,y)(a1,d0,d1)
-252	0xFF04	-0x0FC	AreaMove(rastPort,x,y)(a1,d0,d1)
-258	0xFEFE	-0x102	AreaDraw(rastPort,x,y)(a1,d0,d1)
-264	0xFEF8	-0x108	AreaEnd(rastPort)(a1)
-270	0xFEF2	-0x10E	WaitTOF()()
-276	0xFEEC	-0x114	QBlit(blit)(a1)
-282	0xFEE6	-0x11A	InitArea(areaInfo,vectorTable,tableSize)(a0,a1,d0)
-288	0xFEE0	-0x120	SetRGB4(viewPort,index,r,g,b)(a0,d0,d1,d2,d3)
-294	0xFEDA	-0x126	QBSBlit(blit)(a1)
-300	0xFED4	-0x12C	BltClear(memory,size,flags)(a1,d0,d1)
-306	0xFECE	-0x132	RectFill(rastPort,x1,y1,xu,yu)(a1,d0,d1,d2,d3)
-312	0xFEC8	-0x138	BltPattern(rastPort,ras,x1,y1,maxX,maxY,fillBytes) (a1,a0,d0,d1,d2,d3,d4)
-318	0xFEC2	-0x13E	ReadPixel(rastPort,x,y)(a1,d0,d1)
-324	0xFEBE	-0x144	WritePixel(rastPort,x,y)(a1,d0,d1)
-330	0xFEB6	-0x14A	Flood(rastPort,mode,x,y)(a1,d2,d0,d1)
-336	0xFEB0	-0x150	PolyDraw(rastPort,count,polyTable)(a1,d0,a0)
-342	0xFEAA	-0x156	SetAPen(rastPort,pen)(a1,d0)
-348	0xFEAA	-0x15C	SetBPen(rastPort,pen)(a1,d0)
-354	0xFE9E	-0x162	SetDrMd(rastPort,drawMode)(a1,d0)
-360	0xFE98	-0x168	InitView(view)(a1)
-366	0xFE92	-0x16E	CBump(copperList)(a1)
-372	0xFE8C	-0x174	CMove(copperList,destination,data)(a1,d0,d1)
-378	0xFE86	-0x17A	CWait(copperList,v,h)(a1,d0,d1)
-384	0xFE80	-0x180	VBeamPos()()
-390	0xFE7A	-0x186	InitBitMap(bitMap,depth,width,height)(a0,d0,d1,d2)
-396	0xFE74	-0x18C	ScrollRaster(rastPort,dX,dY,minX,minY,maxX,maxY) (a1,d0,d1,d2,d3,d4,d5)
-402	0xFE6E	-0x192	WaitBOVP(viewPort)(a0)
-408	0xFE68	-0x198	GetSprite(simpleSprite,num)(a0,d0)
-414	0xFE62	-0x19E	FreeSprite(num)(d0)
-420	0xFE5C	-0x1A4	ChangeSprite(viewPort,simpleSprite,data)(a0,a1,a2)
-426	0xFE56	-0x1AA	MoveSprite(viewPort,simpleSprite,x,y)(a0,a1,d0,d1)
-432	0xFE50	-0x1B0	LockLayerRom(layer)(a5)
-438	0xFE4A	-0x1B6	UnlockLayerRom(layer)(a5)
-444	0xFE44	-0x1BC	SyncSBitMap(l)(a0)
-450	0xFE3E	-0x1C2	CopySBitMap(l1,l2)(a0,a1)
-456	0xFE38	-0x1C8	DisownBlitter()()
-456	0xFE38	-0x1C8	OwnBlitter()()
-468	0xFE2C	-0x1D4	InitTmpRas(tmpras,buffer,size)(a0,a1,d0)

-474	0xFE26	-0x1DA	AskFont (rastPort, textAttr) (a1, a0)
-480	0xFE20	-0x1E0	AddFont (textFont) (a1)
-486	0xFE1A	-0x1E6	RemFont (textFont) (a1)
-492	0xFE14	-0x1EC	AllocRaster (width, height) (d0, d1)
-498	0xFE0E	-0x1F2	FreeRaster (planePtr, width, height) (a0, d0, d1)
-504	0xFE08	-0x1F8	AndRectRegion (rgn, rect) (a0, a1)
-510	0xFE02	-0x1FE	OrRectRegion (rgn, rect) (a0, a1)
-516	0xFDFC	-0x204	NewRegion () ()
-528	0xFDF0	-0x210	ClearRegion (rgn) (a0)
-534	0xFDEA	-0x216	DisposeRegion (rgn) (a0)
-540	0xFDE4	-0x21C	FreeVPortCopLists (viewPort) (a0)
-546	0xFDDE	-0x222	FreeCopList (coplist) (a0)
-552	0xFDD8	-0x228	ClipBlit (srcRP, srcX, srcY, destRP, destX, destY, sizeX, sizeY) (a0, d0, d1, a1, d2, d3, d4, d5) (minterm) (d6)
-552	0xFDD8	-0x228	ClearRectRegion (rgn, rect) (a0, a1)
-558	0xFDD2	-0x22E	XorRectRegion (rgn, rect) (a0, a1)
-564	0xFDCC	-0x234	FreeCprList (cprList) (a0)
-570	0xFDC6	-0x23A	GetColorMap (entries) (d0)
-576	0xFDC0	-0x240	FreeColorMap (colormap) (a0)
-582	0xFDBA	-0x246	GetRGB4 (colormap, entry) (a0, d0)
-588	0xFDB4	-0x24C	ScrollVPort (viewPort) (a0)
-594	0xFDAE	-0x252	UCopperListInit (copperList, num) (a0, d0)
-600	0xFDA8	-0x258	FreeGBuffers (animObj, rastPort, doubleBuffer) (a0, a1, d0)
-606	0xFDA2	-0x25E	BlitBitMapRastPort (srcBM, srcX, srcY, destRP, destX, destY, sizeX, sizeY, minterm) (a0, d0, d1, a1, d2, d3, d4, d5, d6)
-612	0xFD9C	-0x264	OrRegionRegion (src, dest) (a0, a1)
-618	0xFD96	-0x26A	XorRegionRegion (src, dest) (a0, a1)
-624	0xFD90	-0x270	AndRegionRegion (src, dest) (a0, a1)
-630	0xFD8A	-0x276	SetRGB4CM (colorMap, color, r, g, b) (a0, d0, d1, d2, d3)
-636	0xFD84	-0x27C	BlitMaskBitMapRastPort (srcBM, srcX, srcY, destRP, destX, destY, sizeX, sizeY, minterm) (a0, d0, d1, a1, d2, d3, d4, d5, d6) (bltMask) (a2)
-654	0xFD72	-0x28E	AttemptLockLayerRom (layer) (a5)

icon.library

-30	0xFFE2	-0x01E	GetWBOobject (name) (a0)
-36	0xFFDC	-0x024	PutWBOobject (name,object) (a0,a1)
-42	0xFFD6	-0x02A	GetIcon (name,icon,freelist) (a0,a1,a2)
-48	0xFFD0	-0x030	PutIcon (name,icon) (a0,a1)
-54	0xFFCA	-0x036	FreeFreeList (freelist) (a0)
-60	0xFFC4	-0x03C	FreeWBOobject (WBOobject) (a0)
-66	0xFFBE	-0x042	AllocWBOobject () ()
-72	0xFFB8	-0x048	AddFreeList (freelist,mem,size) (a0,a1,a2)
-78	0xFFB2	-0x04E	GetDiskObject (name) (a0)
-84	0xFFAC	-0x054	PutDiskObject (name,diskobj) (a0,a1)
-90	0xFFA6	-0x05A	FreeDiskObject (diskobj) (a0)
-96	0xFFA0	-0x060	FindToolType (toolTypeArray,typeName) (a0,a1)
-102	0xFF9A	-0x066	MatchToolValue (typeString,value) (a0,a1)
-108	0xFF94	-0x06C	BumpRevision (newname,oldname) (a0,a1)

intuition.library

-30	0xFFE2	-0x01E	OpenIntuition() ()
-36	0xFFDC	-0x024	Intuition(ievent) (a0)
-42	0xFFD6	-0x02A	AddGadget (AddPtr, Gadget, Position) (a0, a1, d0)
-48	0xFFD0	-0x030	ClearDMRequest (Window) (a0)
-54	0xFFCA	-0x036	ClearMenuStrip (Window) (a0)
-60	0xFFC4	-0x03C	ClearPointer (Window) (a0)
-66	0xFFBE	-0x042	CloseScreen (Screen) (a0)
-72	0xFFB8	-0x048	CloseWindow (Window) (a0)
-78	0xFFB2	-0x04E	CloseWorkBench () ()
-84	0xFFAC	-0x054	CurrentTime (Seconds, Micros) (a0, a1)
-90	0xFFA6	-0x05A	DisplayAlert (AlertNumber, String, Height) (d0, a0, a1)
-96	0xFFA0	-0x060	DisplayBeep (Screen) (a0)
-102	0xFF9A	-0x066	DoubleClick (sseconds, smicros, cseconds, cmicros) (d0, d1, d2, d3)
-108	0xFF94	-0x06C	DrawBorder (RPort, Border, LeftOffset, TopOffset) (a0, a1, d0, d1)
-114	0xFF8E	-0x072	DrawImage (RPort, Image, LeftOffset, TopOffset) (a0, a1, d0, d1)
-120	0xFF88	-0x078	EndRequest (requester, Window) (a0, a1)
-126	0xFF82	-0x07E	GetDefPrefs (preferences, size) (a0, d0)
-132	0xFF7C	-0x084	GetPrefs (preferences, size) (a0, a1)
-138	0xFF76	-0x08A	InitRequester (req) (a0)
-144	0xFF70	-0x090	ItemAddress (MenuStrip, MenuNumber) (a0, d0)
-150	0xFF6A	-0x096	ModifyIDCMP (Window, Flags) (a0, d0)
-156	0xFF64	-0x09C	ModifyProp (Gadget, Win, Req, Flags, HPos, VPos, HBody, VBody) (a0, a1, a2, d0, d1, d2, d3, d4)
-162	0xFF5E	-0x0A2	MoveScreen (Screen, dx, dy) (a0, d0, d1)
-168	0xFF58	-0x0A8	MoveWindow (Window, dx, dy) (a0, d0, d1)
-174	0xFF52	-0x0AE	OffGadget (Gadget, Win, Req) (a0, a1, a2)
-180	0xFF4C	-0x0B4	OffMenu (Window, MenuNum) (a0, d0)
-186	0xFF46	-0x0BA	OnGadget (Gadget, Win, Req) (a0, a1, a2)
-192	0xFF40	-0x0C0	OnMenu (Window, MenuNum) (a0, d0)
-198	0xFF3A	-0x0C6	OpenScreen (Args) (a0)
-204	0xFF34	-0x0CC	OpenWindow (Args) (a0)
-210	0xFF2E	-0x0D2	OpenWorkBench () ()
-216	0xFF28	-0x0D8	PrintIText (rp, IText, LeftOffset, TopOffset) (a0, a1, d0, d1)
-222	0xFF22	-0x0DE	RefreshGadgets (Gadgets, Win, Req) (a0, a1, a2)
-228	0xFF1C	-0x0E4	RemoveGadget (RemPtr, Gadget) (a0, a1)
-234	0xFF16	-0x0EA	ReportMouse (Boolean, Window) (a0, d0)
-240	0xFF10	-0x0F0	Request (Req, Win) (a0, a1)
-246	0xFF0A	-0x0F6	ScreenToBack (Screen) (a0)
-252	0xFF04	-0x0FC	ScreenToFront (Screen) (a0)
-258	0xFFFE	-0x102	SetDMRequest (Win, Req) (a0, a1)

-264	0xFEFE8	-0x108	SetMenuStrip(Win, Menu) (a0, a1)
-270	0xFEFE2	-0x10E	SetPointer(Win, Pointer, Height, Width, HotX, HotY) (a0, a1, d0, d1, d2, d3)
-276	0xFEEC	-0x114	SetWindowTitles(window, winTitle, scrTitle) (a0, a1, a2)
-282	0xFEE6	-0x11A	ShowTitle(Screen, ShowIt) (a0, d0)
-288	0xFEE0	-0x120	SizeWindow(Window, dx, dy) (a0, d0, d1)
-294	0xFEDA	-0x126	ViewAddress() ()
-300	0xFED4	-0x12C	ViewPortAddress(Window) (a0)
-306	0xFECE	-0x132	WindowToBack(Window) (a0)
-312	0xFEC8	-0x138	WindowToFront(Window) (a0)
-318	0xFEC2	-0x13E	WindowLimits(Window, WMin, HMin, WMax, HMax) (a0, d0, d1, d2, d3)
-324	0xFEEB	-0x144	SetPrefs(preferences, size, flags) (a0, d0, d1)
-330	0xFEB6	-0x14A	IntuiTextLength(IText) (a0)
-336	0xFEB0	-0x150	WBenchToBack() ()
-342	0xFEAA	-0x156	WBenchToFront() ()
-348	0xFEAA4	-0x15C	AutoRequest(Win, Body, PText, NText, PFlag, NFlag, W, H) (a0, a1, a2, a3, d0, d1, d2, d3)
-354	0xFE9E	-0x162	BeginRefresh(Window) (a0)
-360	0xFE98	-0x168	BuildSysRequest(Win, Body, PText, NText, Flags, W, H) (a0, a1, a2, a3, d0, d1, d2)
-366	0xFE92	-0x16E	EndRefresh(Window, Complete) (a0, d0)
-372	0xFE8C	-0x174	FreeSysRequest(Window) (a0)
-378	0xFE86	-0x17A	MakeScreen(Screen) (a0)
-384	0xFE80	-0x180	RemakeDisplay() ()
-390	0xFE7A	-0x186	RethinkDisplay() ()
-396	0xFE74	-0x18C	AllocRemember(RememberKey, Size, Flags) (a0, d0, d1)
-402	0xFE6E	-0x192	AlohaWorkbench(wbport) (a0)
-408	0xFE68	-0x198	FreeRemember(RememberKey, ReallyForget) (a0, d0)
-414	0xFE62	-0x19E	LockIBase(dontknow) (d0)
-420	0xFE5C	-0x1A4	UnlockIBase(IBLock) (a0)
-426	0xFE56	-0x1AA	GetScreenData(buffer, size, type, Screen) (a0, d0, d1, a1)
-432	0xFE50	-0x1B0	RefreshGList(Gadgets, Win, Req, NumGad) (a0, a1, a2, d0)
-438	0xFE4A	-0x1B6	AddGList(AddPtr, Gadget, Pos, NumGad, Req) (a0, a1, d0, d1, a2)
-444	0xFE44	-0x1BC	RemoveGList(RemPtr, Gadget, NumGad) (a0, a1, d0)
-450	0xFE3E	-0x1C2	ActivateWindow(Window) (a0)
-456	0xFE38	-0x1C8	RefreshWindow(Window) (a0)
-462	0xFE32	-0x1CE	ActivateGadget(Gadget, Window, Req) (a0, a1, a2)
-468	0xFE2C	-0x1D4	NewModifyProp(Gad, Win, Req, Flag, HPos, VPos, HBod, VBod, NumGad) (a0, a1, a2, d0, d1, d2, d3, d4, d5)

layers.library

-30	0xFFE2	-0x01E	InitLayers(li) (a0)
-36	0xFFDC	-0x024	CreateUpfrontLayer(li,bm,x0,y0,x1,y1,flags,bm2) (a0,a1,d0,d1,d2,d3,d4,a2)
-42	0xFFD6	-0x02A	CreateBehindLayer(li,bm,x0,y0,x1,y1,flags,bm2) (a0,a1,d0,d1,d2,d3,d4,a2)
-48	0xFFD0	-0x030	UpfrontLayer(li,layer) (a0,a1)
-54	0xFFCA	-0x036	BehindLayer(li,layer) (a0,a1)
-60	0xFFC4	-0x03C	MoveLayer(li,layer,dx,dy) (a0,a1,d0,d1)
-66	0xFFBE	-0x042	SizeLayer(li,layer,dx,dy) (a0,a1,d0,d1)
-72	0xFFB8	-0x048	ScrollLayer(li,layer,dx,dy) (a0,a1,d0,d1)
-78	0xFFB2	-0x04E	BeginUpdate(layer) (a0)
-84	0xFFAC	-0x054	EndUpdate(layer,flag) (a0,d0)
-90	0xFFA6	-0x05A	DeleteLayer(li,layer) (a0,a1)
-96	0xFFA0	-0x060	LockLayer(li,layer) (a0,a1)
-102	0xFF9A	-0x066	UnlockLayer(layer) (a0)
-108	0xFF94	-0x06C	LockLayers(li) (a0)
-114	0xFF8E	-0x072	UnlockLayers(li) (a0)
-120	0xFF88	-0x078	LockLayerInfo(li) (a0)
-126	0xFF82	-0x07E	SwapBitsRastPortClipRect(rp,cr) (a0,a1)
-132	0xFF7C	-0x084	WhichLayer(li,x,y) (a0,d0,d1)
-138	0xFF76	-0x08A	UnlockLayerInfo(li) (a0)
-144	0xFF70	-0x090	NewLayerInfo() ()
-150	0xFF6A	-0x096	DisposeLayerInfo(li) (a0)
-156	0xFF64	-0x09C	FattenLayerInfo(li) (a0)
-162	0xFF5E	-0x0A2	ThinLayerInfo(li) (a0)
-168	0xFF58	-0x0A8	MoveLayerInFrontOf(movelayer,backlayer) (a0,a1)
-174	0xFF52	-0x0AE	InstallClipRegion(layer,region) (a0,a1)

mathffp.library

-30	0xFFE2	-0x01E	SPFix(float) (d0)
-36	0xFFDC	-0x024	SPFlt(integer) (d0)
-42	0xFFD6	-0x02A	SPCmp(leftFloat,rightFloat) (d1,d0)
-48	0xFFD0	-0x030	SPTst(float) (d1)
-54	0xFFCA	-0x036	SPAbs(float) (d0)
-60	0xFFC4	-0x03C	SPNeg(float) (d0)
-66	0xFFBE	-0x042	SPAdd(leftFloat,rightFloat) (d1,d0)
-72	0xFFB8	-0x048	SPSub(leftFloat,rightFloat) (d1,d0)
-78	0xFFB2	-0x04E	SPMul(leftFloat,rightFloat) (d1,d0)
-84	0xFFAC	-0x054	SPDiv(leftFloat,rightFloat) (d1,d0)
-90	0xFFA6	-0x05A	SPFloor(float) (d0)
-96	0xFFA0	-0x060	SPCeil(float) (d0)

mathieedoubbas.library

-30	0xFFE2	-0x01E	IEEEDPFix(double) (d0/d1)
-36	0xFFDC	-0x024	IEEEDPFlt(integer) (d0)
-42	0xFFD6	-0x02A	IEEEDPCmp(double,double) (d0/d1,d2/d3)
-48	0xFFD0	-0x030	IEEEDPTst(double) (d0/d1)
-54	0xFFCA	-0x036	IEEEDPAbs(double) (d0/d1)
-60	0xFFC4	-0x03C	IEEEDPNeg(double) (d0/d1)
-66	0xFFBE	-0x042	IEEEDPAdd(double,double) (d0/d1,d2/d3)
-72	0xFFB8	-0x048	IEEEDPMul(double,double) (d0/d1,d2/d3)
-72	0xFFB8	-0x048	IEEEDPSub(double,double) (d0/d1,d2/d3)
-84	0xFFAC	-0x054	IEEEDPDiv(double,double) (d0/d1,d2/d3)
-90	0xFFA6	-0x05A	IEEEDPFloor(double) (d0/d1)
-96	0xFFA0	-0x060	IEEEDPCeil(double) (d0/d1)

mathieeedoubtrans.library

-30	0xFFE2	-0x01E	IEEEDPATan(double) (d0/d1)
-36	0xFFDC	-0x024	IEEEDPSin(double) (d0/d1)
-42	0xFFD6	-0x02A	IEEEDPCos(double) (d0/d1)
-48	0xFFD0	-0x030	IEEEDPTan(double) (d0/d1)
-54	0xFFCA	-0x036	IEEEDPSincos(double, pf2) (a0, d0/d1)
-60	0xFFC4	-0x03C	IEEEDPSinh(double) (d0/d1)
-66	0xFFBE	-0x042	IEEEDPCosh(double) (d0/d1)
-72	0xFFB8	-0x048	IEEEDPTanh(double) (d0/d1)
-78	0xFFB2	-0x04E	IEEEDPExp(double) (d0/d1)
-84	0xFFAC	-0x054	IEEEDPLog(double) (d0/d1)
-90	0xFFA6	-0x05A	IEEEDPPow(exp, arg) (d2/d3, d0/d1)
-96	0xFFA0	-0x060	IEEEDPSqrt(double) (d0/d1)
-102	0xFF9A	-0x066	IEEEDPTieee(double) (d0/d1)
-108	0xFF94	-0x06C	IEEEDPFieee(single) (d0)
-114	0xFF8E	-0x072	IEEEDPAsin(double) (d0/d1)
-120	0xFF88	-0x078	IEEEDPAcos(double) (d0/d1)
-126	0xFF82	-0x07E	IEEEDPLog10(double) (d0/d1)

mathtrans.library

-30	0xFFE2	-0x01E	SPAtan(float) (d0)
-36	0xFFDC	-0x024	SPSin(float) (d0)
-42	0xFFD6	-0x02A	SPCos(float) (d0)
-48	0xFFD0	-0x030	SPTan(float) (d0)
-54	0xFFCA	-0x036	SPSincos(leftFloat, rightFloat) (d1, d0)
-60	0xFFC4	-0x03C	SPSinh(float) (d0)
-66	0xFFBE	-0x042	SPCosh(float) (d0)
-72	0xFFB8	-0x048	SPTanh(float) (d0)
-78	0xFFB2	-0x04E	SPExp(float) (d0)
-84	0xFFAC	-0x054	SPLog(float) (d0)
-90	0xFFA6	-0x05A	SPPow(leftFloat, rightFloat) (d1, d0)
-96	0xFFA0	-0x060	SPSqrt(float) (d0)
-102	0xFF9A	-0x066	SPTieee(float) (d0)
-108	0xFF94	-0x06C	SPFieee(integer) (d0)
-114	0xFF8E	-0x072	SPAsin(float) (d0)
-120	0xFF88	-0x078	SPAcos(float) (d0)
-126	0xFF82	-0x07E	SPLog10(float) (d0)

potgo.library

-6	0xFFFA	-0x006	AllocPotBits(bits) (d0)
-12	0xFFF4	-0x00C	FreePortBits(bits) (d0)
-18	0xFFEE	-0x012	WritePotGo(word,mask) (d0,d1)

romboot.library

-30	0xFFE2	-0x01E	RomBoot() ()
-----	--------	--------	--------------

timer.library

-42	0xFFD6	-0x02A	AddTime(dest,src) (a0,a1)
-48	0xFFD0	-0x030	SubTime(dest,src) (a0,a1)
-54	0xFFCA	-0x036	CmpTime(dest,src) (a0,a1)

translator.library

-30	0xFFE2	-0x01E	Translate(input,inputLength,outBuf,bufSize) (a0,d0,a1,d1)
-----	--------	--------	---

clist.library

-36	0xFFDC	-0x024	AllocCList (cLPool) (a1)
-156	0xFF64	-0x09C	ConcatCList (sourceCList, destCList) (a0, a1)
-144	0xFF70	-0x090	CopyCList (cList) (a0)
-48	0xFFD0	-0x030	FlushCList (cList) (a0)
-42	0xFFD6	-0x02A	FreeCList (cList) (a0)
-114	0xFF8E	-0x072	GetCLBuf (cList, buffer, length) (a0, a1, d1)
-66	0xFFBE	-0x042	GetCLChar (cList) (a0)
-90	0xFFA6	-0x05A	GetCLWord (cList) (a0)
-126	0xFF82	-0x07E	IncrCLMark (cList) (a0)
-30	0xFFE2	-0x01E	InitCLPool (cLPool, size) (a0, d0)
-120	0xFF88	-0x078	MarkCList (cList, offset) (a0, d0)
-132	0xFF7C	-0x084	PeekCLMark (cList) (a0)
-108	0xFF94	-0x06C	PutCLBuf (cList, buffer, length) (a0, a1, d1)
-60	0xFFC4	-0x03C	PutCLChar (cList, byte) (a0, d0)
-84	0xFFAC	-0x054	PutCLWord (cList, word) (a0, d0)
-54	0xFFCA	-0x036	SizeCList (cList) (a0)
-138	0xFF76	-0x08A	SplitCList (cList) (a0)
-150	0xFF6A	-0x096	SubCList (cList, index, length) (a0, d0, d1)
-72	0xFFB8	-0x048	UnGetCLChar (cList, byte) (a0, d0)
-96	0xFFA0	-0x060	UnGetCLWord (cList, word) (a0, d0)
-78	0xFFB2	-0x04E	UnPutCLChar (cList) (a0)
-102	0xFF9A	-0x066	UnPutCLWord (cList) (a0)

console.library

-42	0xFFD6	-0x02A	CDInputHandler (events, device) (a0, a1)
-48	0xFFD0	-0x030	RawKeyConvert (events, buffer, length, keyMap) (a0, a1, d1, a2)

diskfont.library

-36	0xFFDC	-0x024	AvailFonts (buffer, bufBytes, flags) (a0, d0, d1)
-48	0xFFD0	-0x030	DisposeFontContents (fontContentsHeader) (a1)
-42	0xFFD6	-0x02A	NewFontContents (fontsLock, fontName) (a0, a1)
-30	0xFFE2	-0x01E	OpenDiskFont (textAttr) (a0)

dos.library

-36	0xFFDC	-0x024	Close(file) (d1)
-120	0xFF88	-0x078	CreateDir(name) (d1)
-138	0xFF76	-0x08A	CreateProc(name,pri,segList,stack) (d1,d2,d3,d4)
-126	0xFF82	-0x07E	CurrentDir(lock) (d1)
-192	0xFF40	-0x0C0	DateStamp(date) (d1)
-198	0xFF3A	-0x0C6	Delay(timeout) (d1)
-72	0xFFB8	-0x048	DeleteFile(name) (d1)
-174	0xFF52	-0x0AE	DeviceProc(name) (d1)
-96	0xFFA0	-0x060	DupLock(lock) (d1)
-102	0xFF9A	-0x066	Examine(lock,fileInfoBlock) (d1,d2)
-222	0xFF22	-0x0DE	Execute(string,file,file) (d1,d2,d3)
-144	0xFF70	-0x090	Exit(returnCode) (d1)
-108	0xFF94	-0x06C	ExNext(lock,fileInfoBlock) (d1,d2)
-162	0xFF5E	-0x0A2	GetPacket(wait) (d1)
-114	0xFF8E	-0x072	Info(lock,parameter) (d1,d2)
-54	0xFFCA	-0x036	Input() ()
-132	0xFF7C	-0x084	IoErr() ()
-216	0xFF28	-0x0D8	IsInteractive(file) (d1)
-150	0xFF6A	-0x096	LoadSeg(filename) (d1)
-84	0xFFAC	-0x054	Lock(name,type) (d1,d2)
-30	0xFFE2	-0x01E	Open(name,mode) (d1,d2)
-60	0xFFC4	-0x03C	Output() ()
-210	0xFF2E	-0x0D2	ParentDir(lock) (d1)
-168	0xFF58	-0x0A8	QueuePacket(packet) (d1)
-42	0xFFD6	-0x02A	Read(file,buffer,length) (d1,d2,d3)
-78	0xFFB2	-0x04E	Rename(oldname,newname) (d1,d2)
-66	0xFFBE	-0x042	Seek(file,pos,offset) (d1,d2,d3)
-180	0xFF4C	-0x0B4	SetComment(name,comment) (d1,d2)
-186	0xFF46	-0x0BA	SetProtection(name,mask) (d1,d2)
-156	0xFF64	-0x09C	UnLoadSeg(segment) (d1)
-90	0xFFA6	-0x05A	UnLock(lock) (d1)
-204	0xFF34	-0x0CC	WaitForChar(file,timeout) (d1,d2)
-48	0xFFD0	-0x030	Write(file,buffer,length) (d1,d2,d3)

exec.library

-480	0xFE20	-0x1E0	AbortIO(ioRequest) (a1)
-432	0xFE50	-0x1B0	AddDevice(device) (a1)
-240	0xFF10	-0x0F0	AddHead(list,node) (a0,a1)
-168	0xFF58	-0x0A8	AddIntServer(intNumber,interrupt) (d0,a1)
-396	0xFE74	-0x18C	AddLibrary(library) (a1)
-618	0xFD96	-0x26A	AddMemList(size,attributes,pri,base,name) (d0,d1,d2,a0,a1)
-354	0xFE9E	-0x162	AddPort(port) (a1)
-486	0xFE1A	-0x1E6	AddResource(resource) (a1)
-600	0xFDA8	-0x258	AddSemaphore(sigSem) (a1)
-246	0xFF0A	-0x0F6	AddTail(list,node) (a0,a1)
-282	0xFEE6	-0x11A	AddTask(task,initialPC,finalPC) (a1,a2,a3)
-108	0xFF94	-0x06C	Alert(alertNum,parameters) (d7,a5)
-204	0xFF34	-0x0CC	AllocAbs(byteSize,location) (d0,a1)
-186	0xFF46	-0x0BA	Allocate(freelist,byteSize) (a0,d0)
-222	0xFF22	-0x0DE	AllocEntry(entry) (a0)
-198	0xFF3A	-0x0C6	AllocMem(byteSize,requirements) (d0,d1)
-330	0xFEB6	-0x14A	AllocSignal(signalNum) (d0)
-342	0xFEAA	-0x156	AllocTrap(trapNum) (d0)
-576	0xFDC0	-0x240	AttemptSemaphore(sigSem) (a0)
-216	0xFF28	-0x0D8	AvailMem(requirements) (d1)
-180	0xFF4C	-0x0B4	Cause(interrupt) (a1)
-468	0xFE2C	-0x1D4	CheckIO(ioRequest) (a1)
-450	0xFE3E	-0x1C2	CloseDevice(ioRequest) (a1)
-414	0xFE62	-0x19E	CloseLibrary(library) (a1)
-624	0xFD90	-0x270	CopyMem(source,dest,size) (a0,a1,d0)
-630	0xFD8A	-0x276	CopyMemQuick(source,dest,size) (a0,a1,d0)
-192	0xFF40	-0x0C0	Deallocate(freeList,memoryBlock,byteSize) (a0,a1,d0)
-114	0xFF8E	-0x072	Debug() ()
-120	0xFF88	-0x078	Disable() ()
-60	0xFFC4	-0x03C	Dispatch() ()
-456	0xFE38	-0x1C8	DoIO(ioRequest) (a1)
-126	0xFF82	-0x07E	Enable() ()
-270	0xFE72	-0x10E	Enqueue(list,node) (a0,a1)
-66	0xFFBE	-0x042	Exception() ()
-36	0xFFDC	-0x024	ExitIntr() ()
-276	0xFEEC	-0x114	FindName(list,name) (a0,a1)
-390	0xFE7A	-0x186	FindPort(name) (a1)
-96	0xFFA0	-0x060	FindResident(name) (a1)
-594	0xFDAE	-0x252	FindSemaphore(sigSem) (a1)
-294	0xFEDA	-0x126	FindTask(name) (a1)

-132	0xFF7C	-0x084	Forbid() ()
-228	0xFF1C	-0x0E4	FreeEntry(entry) (a0)
-210	0xFF2E	-0x0D2	FreeMem(memoryBlock,byteSize) (a1,d0)
-336	0xFEB0	-0x150	FreeSignal(signalNum) (d0)
-348	0xFEA4	-0x15C	FreeTrap(trapNum) (d0)
-528	0xFDF0	-0x210	GetCC() ()
-372	0xFE8C	-0x174	GetMsg(port) (a0)
-72	0xFFB8	-0x048	InitCode(startClass,version) (d0,d1)
-102	0xFF9A	-0x066	InitResident(resident,segList) (a1,d1)
-558	0xFDD2	-0x22E	InitSemaphore(sigSem) (a0)
-78	0xFFB2	-0x04E	InitStruct(initTable,memory,size) (a1,a2,d0)
-234	0xFF16	-0x0EA	Insert(list,node,pred) (a0,a1,a2)
-90	0xFFA6	-0x05A	MakeFunctions(target,functionArray,funcDispBase) (a0,a1,a2)
-84	0xFFAC	-0x054	MakeLibrary(funcInit,structInit,libInit,data,code) (a0,a1,a2,d0,d1)
-564	0xFDCC	-0x234	ObtainSemaphore(sigSem) (a0)
-582	0xFDBA	-0x246	ObtainSemaphoreList(sigSem) (a0)
-408	0xFE68	-0x198	OldOpenLibrary(libName) (a1)
-444	0xFE44	-0x1BC	OpenDevice(devName,unit,ioRequest,flags) (a0,d0,a1,d1)
-552	0xFDD8	-0x228	OpenLibrary(libName,version) (a1,d0)
-498	0xFE0E	-0x1F2	OpenResource(resName,version) (a1,d0)
-138	0xFF76	-0x08A	Permit() ()
-540	0xFDE4	-0x21C	Procure(semaphore,bidMsg) (a0,a1)
-366	0xFE92	-0x16E	PutMsg(port,message) (a0,a1)
-522	0xFDF6	-0x20A	RawDoFmt(format,data,putProc,putData) (a0,a1,a2,a3)
-504	0xFE08	-0x1F8	RawIOInit() ()
-510	0xFE02	-0x1FE	RawMayGetChar() ()
-516	0xFDFC	-0x204	RawPutChar(char) (d0)
-570	0xFDC6	-0x23A	ReleaseSemaphore(sigSem) (a0)
-588	0xFDB4	-0x24C	ReleaseSemaphoreList(sigSem) (a0)
-438	0xFE4A	-0x1B6	RemDevice(device) (a1)
-258	0xFEFE	-0x102	RemHead(list) (a0)
-174	0xFF52	-0x0AE	RemIntServer(intNumber,interrupt) (d0,a1)
-402	0xFE6E	-0x192	RemLibrary(library) (a1)
-252	0xFF04	-0x0FC	Remove(node) (a1)
-360	0xFE98	-0x168	RemPort(port) (a1)
-492	0xFE14	-0x1EC	RemResource(resource) (a1)
-606	0xFDA2	-0x25E	RemSemaphore(sigSem) (a1)
-264	0xFE78	-0x108	RemTail(list) (a0)
-288	0xFEE0	-0x120	RemTask(task) (a1)
-378	0xFE86	-0x17A	ReplyMsg(message) (a1)
-48	0xFFD0	-0x030	Reschedule() ()

-42	0xFFD6	-0x02A	Schedule() ()
-462	0xFE32	-0x1CE	SendIO(ioRequest) (a1)
-312	0xFEC8	-0x138	SetExcept(newSignals, signalSet) (d0, d1)
-420	0xFE5C	-0x1A4	SetFunction(library, funcOffset, funcEntry) (a1, a0, d0)
-162	0xFF5E	-0x0A2	SetIntVector(intNumber, interrupt) (d0, a1)
-306	0xFECE	-0x132	SetSignal(newSignals, signalSet) (d0, d1)
-144	0xFF70	-0x090	SetSR(newSR, mask) (d0, d1)
-300	0xFED4	-0x12C	SetTaskPri(task, priority) (a1, d0)
-324	0xFEBC	-0x144	Signal(task, signalSet) (a1, d0)
-612	0xFD9C	-0x264	SumKickData() ()
-426	0xFE56	-0x1AA	SumLibrary(library) (a1)
-150	0xFF6A	-0x096	SuperState() ()
-30	0xFFE2	-0x01E	Supervisor() ()
-54	0xFFCA	-0x036	Switch() ()
-534	0xFDEA	-0x216	TypeOfMem(address) (a1)
-156	0xFF64	-0x09C	UserState() ()
-546	0xFDDE	-0x222	Vacate(semaphore) (a0)
-318	0xFEC2	-0x13E	Wait(signalSet) (d0)
-474	0xFE26	-0x1DA	WaitIO(ioRequest) (a1)
-384	0xFE80	-0x180	WaitPort(port) (a0)

expansion.library

-30	0xFFE2	-0x01E	AddConfigDev(configDev) (a0)
-150	0xFF6A	-0x096	AddDosNode(bootPri, flags, dosNode) (d0, d1, a0)
-42	0xFFD6	-0x02A	AllocBoardMem(slotSpec) (d0)
-48	0xFFD0	-0x030	AllocConfigDev() ()
-54	0xFFCA	-0x036	AllocExpansionMem(numSlots, SlotAlign, SlotOffset) (d0, d1, d2)
-60	0xFFC4	-0x03C	ConfigBoard(board, configDev) (a0, a1)
-66	0xFFBE	-0x042	ConfigChain(baseAddr) (a0)
-72	0xFFB8	-0x048	FindConfigDev(oldConfigDev, manufacturer, product) (a0, d0, d1)
-78	0xFFB2	-0x04E	FreeBoardMem(startSlot, slotSpec) (d0, d1)
-84	0xFFAC	-0x054	FreeConfigDev(configDev) (a0)
-90	0xFFA6	-0x05A	FreeExpansionMem(startSlot, numSlots) (d0, d1)
-138	0xFF76	-0x08A	GetCurrentBinding(currentBinding, bindingSize) (a0, d0)
-144	0xFF70	-0x090	MakeDosNode(parmPacket) (a0)
-120	0xFF88	-0x078	ObtainConfigBinding() ()
-96	0xFFA0	-0x060	ReadExpansionByte(board, offset) (a0, d0)
-102	0xFF9A	-0x066	ReadExpansionRom(board, configDev) (a0, a1)
-126	0xFF82	-0x07E	ReleaseConfigBinding() ()
-108	0xFF94	-0x06C	RemConfigDev(configDev) (a0)
-132	0xFF7C	-0x084	SetCurrentBinding(currentBinding, bindingSize) (a0, d0)
-114	0xFF8E	-0x072	WriteExpansionByte(board, offset, byte) (a0, d0, d1)

graphics.library

-156	0xFF64	-0x09C	AddAnimOb(obj, animationKey, rastPort) (a0, a1, a2)
-96	0xFFA0	-0x060	AddBob(bob, rastPort) (a0, a1)
-480	0xFE20	-0x1E0	AddFont(textFont) (a1)
-102	0xFF9A	-0x066	AddVSprite(vSprite, rastPort) (a0, a1)
-492	0xFE14	-0x1EC	AllocRaster(width, height) (d0, d1)
-504	0xFE08	-0x1F8	AndRectRegion(rgn, rect) (a0, a1)
-624	0xFD90	-0x270	AndRegionRegion(src, dest) (a0, a1)
-162	0xFF5E	-0x0A2	Animate(animationKey, rastPort) (a0, a1)
-258	0xFEFE	-0x102	AreaDraw(rastPort, x, y) (a1, d0, d1)
-186	0xFF46	-0x0BA	AreaEllipse(rastPort, centerX, centerY, a, b) (a1, d0, d1, d2, d3)
-264	0xFEF8	-0x108	AreaEnd(rastPort) (a1)
-252	0xFF04	-0x0FC	AreaMove(rastPort, x, y) (a1, d0, d1)
-474	0xFE26	-0x1DA	AskFont(rastPort, textAttr) (a1, a0)
-84	0xFFAC	-0x054	AskSoftStyle(rastPort) (a1)
-654	0xFD72	-0x28E	AttemptLockLayerRom(layer) (a5)
-30	0xFFE2	-0x01E	BlitBitMap(srcBM, srcX, srcY, destBM, destX, destY, sizeX, sizeY) (a0, d0, d1, a1, d2, d3, d4, d5) (minterm, mask, tempA) (d6, d7, a2)
-606	0xFDA2	-0x25E	BlitBitMapRastPort(srcBM, srcX, srcY, destRP, destX, destY) (a0, d0, d1, a1, d2, d3) (sizeX, sizeY, minterm) (d4, d5, d6)
-300	0xFED4	-0x12C	BlitClear(memory, size, flags) (a1, d0, d1)
-636	0xFD84	-0x27C	BlitMaskBitMapRastPort (srcBM, srcX, srcY, destRP, destX, destY, sizeX, sizeY, minterm) (a0, d0, d1, a1, d2, d3, d4, d5, d6) (bltMask) (a2)
-312	0xFEC8	-0x138	BlitPattern(rastPort, ras, x1, y1, maxX, maxY, fillBytes) (a1, a0, d0, d1, d2, d3, d4)
-36	0xFFDC	-0x024	BlitTemplate (source, srcX, srcMod, destRP, destX, destY, sizeX, sizeY) (a0, d0, d1, a1, d2, d3, d4, d5)
-366	0xFE92	-0x16E	CBump(copperList) (a1)
-420	0xFE5C	-0x1A4	ChangeSprite(viewPort, simpleSprite, data) (a0, a1, a2)
-42	0xFFD6	-0x02A	ClearEOL(rastPort) (a1)
-552	0xFDD8	-0x228	ClearRectRegion(rgn, rect) (a0, a1)
-528	0xFDF0	-0x210	ClearRegion(rgn) (a0)

-48	0xFFD0	-0x030	ClearScreen(rastPort) (a1)
-552	0xFDD8	-0x228	ClipBlit (srcRP,srcX,srcY,destRP,destX,destY,sizeX,sizeY,minterm) (a0,d0,d1,a1,d2,d3,d4,d5,d6)
-78	0xFFB2	-0x04E	CloseFont(textFont) (a1)
-372	0xFE8C	-0x174	CMove(copperList,destination,data) (a1,d0,d1)
-450	0xFE3E	-0x1C2	CopySBitMap(l1,l2) (a0,a1)
-378	0xFE86	-0x17A	CWait(copperList,v,h) (a1,d0,d1)
-456	0xFE38	-0x1C8	DisownBlitter() ()
-534	0xFDEA	-0x216	DisposeRegion(rgn) (a0)
-108	0xFF94	-0x06C	DoCollision(rastPort) (a1)
-246	0xFF0A	-0x0F6	Draw(rastPort,x,y) (a1,d0,d1)
-180	0xFF4C	-0x0B4	DrawEllipse(rastPort,centerX,centerY,a,b) (a1,d0,d1,d2,d3)
-114	0xFF8E	-0x072	DrawGList(rastPort,viewPort) (a1,a0)
-330	0xFEB6	-0x14A	Flood(rastPort,mode,x,y) (a1,d2,d0,d1)
-576	0xFDC0	-0x240	FreeColorMap(colormap) (a0)
-546	0xFDDE	-0x222	FreeCopList(coplist) (a0)
-564	0xFDCC	-0x234	FreeCprList(cprList) (a0)
-600	0xFDA8	-0x258	FreeGBuffers(animObj,rastPort,doubleBuffer) (a0,a1,d0)
-498	0xFE0E	-0x1F2	FreeRaster(planeptr,width,height) (a0,d0,d1)
-414	0xFE62	-0x19E	FreeSprite(num) (d0)
-540	0xFDE4	-0x21C	FreeVPortCopLists(viewPort) (a0)
-570	0xFDC6	-0x23A	GetColorMap(entries) (d0)
-168	0xFF58	-0x0A8	GetGBuffers(animationObj,rastPort,doubleBuffer) (a0,a1,d0)
-582	0xFDBA	-0x246	GetRGB4(colormap,entry) (a0,d0)
-408	0xFE68	-0x198	GetSprite(sprite,num) (a0,d0)
-282	0xFEE6	-0x11A	InitArea(areaInfo,vectorTable,tableSize) (a0,a1,d0)
-390	0xFE7A	-0x186	InitBitMap(bitMap,depth,width,height) (a0,d0,d1,d2)
-120	0xFF88	-0x078	InitGels(dummyHead,dummyTail,gelsInfo) (a0,a1,a2)
-174	0xFF52	-0x0AE	InitGMasks(animationObj) (a0)
-126	0xFF82	-0x07E	InitMasks(vsprite) (a0)
-198	0xFF3A	-0x0C6	InitRastPort(rastPort) (a1)
-468	0xFE2C	-0x1D4	InitTmpRas(tmpRas,buff,size) (a0,a1,d0)
-360	0xFE98	-0x168	InitView(view) (a1)
-204	0xFF34	-0x0CC	InitVPort(viewPort) (a0)
-192	0xFF40	-0x0C0	LoadRGB4(viewPort,colors,count) (a0,a1,d0)
-222	0xFF22	-0x0DE	LoadView(view) (a1)
-432	0xFE50	-0x1B0	LockLayerRom(layer) (a5)
-216	0xFF28	-0x0D8	MakeVPort(view,viewPort) (a0,a1)
-240	0xFF10	-0x0F0	Move(rastPort,x,y) (a1,d0,d1)
-426	0xFE56	-0x1AA	MoveSprite(viewPort,sprite,x,y) (a0,a1,d0,d1)
-210	0xFF2E	-0x0D2	MrgCop(view) (a1)

-516	0xFDFC	-0x204	NewRegion() ()
-72	0xFFB8	-0x048	OpenFont(textAttr) (a0)
-510	0xFE02	-0x1FE	OrRectRegion(rgn, rect) (a0, a1)
-612	0xFD9C	-0x264	OrRegionRegion(src, dest) (a0, a1)
-456	0xFE38	-0x1C8	OwnBlitter() ()
-336	0xFEB0	-0x150	PolyDraw(rastPort, count, polyTable) (a1, d0, a0)
-276	0xFEEC	-0x114	QBlit(blit) (a1)
-294	0xFEDA	-0x126	QBSBlit(blit) (a1)
-318	0xFEC2	-0x13E	ReadPixel(rastPort, x, y) (a1, d0, d1)
-306	0xFECE	-0x132	RectFill(rastPort, xl, yl, xu, yu) (a1, d0, d1, d2, d3)
-486	0xFE1A	-0x1E6	RemFont(textFont) (a1)
-132	0xFF7C	-0x084	RemIBob(bob, rastPort, viewPort) (a0, a1, a2)
-138	0xFF76	-0x08A	RemVSprite(vsprite) (a0)
-396	0xFE74	-0x18C	ScrollRaster(rastPort, dX, dY, minX, minY, maxX, maxY) (a1, d0, d1, d2, d3, d4, d5)
-588	0xFDB4	-0x24C	ScrollVPort(viewPort) (a0)
-342	0xFEAA	-0x156	SetAPen(rastPort, pen) (a1, d0)
-348	0xFE44	-0x15C	SetBPen(rastPort, pen) (a1, d0)
-144	0xFF70	-0x090	SetCollision(type, routine, gelsInfo) (d0, a0, a1)
-354	0xFE9E	-0x162	SetDrMd(rastPort, drawMode) (a1, d0)
-66	0xFFBE	-0x042	SetFont(rastPortID, textFont) (a1, a0)
-234	0xFF16	-0x0EA	SetRast(rastPort, color) (a1, d0)
-288	0xFEE0	-0x120	SetRGB4(viewPort, index, r, g, b) (a0, d0, d1, d2, d3)
-630	0xFD8A	-0x276	SetRGB4CM(colorMap, color, r, g, b) (a0, d0, d1, d2, d3)
-90	0xFFA6	-0x05A	SetSoftStyle(rastPort, style, enable) (a1, d0, d1)
-150	0xFF6A	-0x096	SortGList(rastPort) (a1)
-444	0xFE44	-0x1BC	SyncSBitMap(l) (a0)
-60	0xFFC4	-0x03C	Text(rastPort, string, count) (a1, a0, d0)
-54	0xFFCA	-0x036	TextLength(rastPort, string, count) (a1, a0, d0)
-594	0xFDAE	-0x252	UCopperListInit(copperList, num) (a0, d0)
-438	0xFE4A	-0x1B6	UnlockLayerRom(layer) (a5)
-384	0xFE80	-0x180	VBeamPos() ()
-228	0xFF1C	-0x0E4	WaitBlit() ()
-402	0xFE6E	-0x192	WaitBOVP(viewPort) (a0)
-270	0xFE72	-0x10E	WaitTOF() ()
-324	0xFEBC	-0x144	WritePixel(rastPort, x, y) (a1, d0, d1)
-558	0xFDD2	-0x22E	XorRectRegion(rgn, rect) (a0, a1)
-618	0xFD96	-0x26A	XorRegionRegion(src, dest) (a0, a1)

icon.library

-72	0xFFB8	-0x048	AddFreeList (freelist, mem, size) (a0, a1, a2)
-66	0xFFBE	-0x042	AllocWBOBJECT () ()
-108	0xFF94	-0x06C	BumpRevision (newname, oldname) (a0, a1)
-96	0xFFA0	-0x060	FindToolType (toolTypeArray, typeName) (a0, a1)
-90	0xFFA6	-0x05A	FreeDiskObject (diskobj) (a0)
-54	0xFFCA	-0x036	FreeFreeList (freelist) (a0)
-60	0xFFC4	-0x03C	FreeWBOBJECT (WBOBJECT) (a0)
-78	0xFFB2	-0x04E	GetDiskObject (name) (a0)
-42	0xFFD6	-0x02A	GetIcon (name, icon, freelist) (a0, a1, a2)
-30	0xFFE2	-0x01E	GetWBOBJECT (name) (a0)
-102	0xFF9A	-0x066	MatchToolValue (typeString, value) (a0, a1)
-84	0xFFAC	-0x054	PutDiskObject (name, diskobj) (a0, a1)
-48	0xFFD0	-0x030	PutIcon (name, icon) (a0, a1)
-36	0xFFDC	-0x024	PutWBOBJECT (name, object) (a0, a1)

intuition.library

-462	0xFE32	-0x1CE	ActivateGadget (Gadget, Window, Req) (a0, a1, a2)
-450	0xFE3E	-0x1C2	ActivateWindow (Window) (a0)
-42	0xFFD6	-0x02A	AddGadget (AddPtr, Gadget, Position) (a0, a1, d0)
-438	0xFE4A	-0x1B6	AddGList (AddPtr, Gadget, Pos, NumGad, Req) (a0, a1, d0, d1, a2)
-396	0xFE74	-0x18C	AllocRemember (RememberKey, Size, Flags) (a0, d0, d1)
-402	0xFE6E	-0x192	AlohaWorkbench (wbport) (a0)
-348	0xFE44	-0x15C	AutoRequest (Win, Body, PText, NText, PFlag, NFlag, W, H) (a0, a1, a2, a3, d0, d1, d2, d3)
-354	0xFE9E	-0x162	BeginRefresh (Window) (a0)
-360	0xFE98	-0x168	BuildSysRequest (Win, Body, PText, NText, Flags, W, H) (a0, a1, a2, a3, d0, d1, d2)
-48	0xFFD0	-0x030	ClearDMRequest (Window) (a0)
-54	0xFFCA	-0x036	ClearMenuStrip (Window) (a0)
-60	0xFFC4	-0x03C	ClearPointer (Window) (a0)
-66	0xFFBE	-0x042	CloseScreen (Screen) (a0)
-72	0xFFB8	-0x048	CloseWindow (Window) (a0)
-78	0xFFB2	-0x04E	CloseWorkBench () ()
-84	0xFFAC	-0x054	CurrentTime (Seconds, Micros) (a0, a1)
-90	0xFFA6	-0x05A	DisplayAlert (AlertNumber, String, Height) (d0, a0, a1)
-96	0xFFA0	-0x060	DisplayBeep (Screen) (a0)
-102	0xFF9A	-0x066	DoubleClick (sseconds, smicros, cseconds, cmicros) (d0, d1, d2, d3)
-108	0xFF94	-0x06C	DrawBorder (RPort, Border, LeftOffset, TopOffset) (a0, a1, d0, d1)
-114	0xFF8E	-0x072	DrawImage (RPort, Image, LeftOffset, TopOffset) (a0, a1, d0, d1)
-366	0xFE92	-0x16E	EndRefresh (Window, Complete) (a0, d0)
-120	0xFF88	-0x078	EndRequest (requester, Window) (a0, a1)
-408	0xFE68	-0x198	FreeRemember (RememberKey, ReallyForget) (a0, d0)
-372	0xFE8C	-0x174	FreeSysRequest (Window) (a0)
-126	0xFF82	-0x07E	GetDefPrefs (preferences, size) (a0, d0)
-132	0xFF7C	-0x084	GetPrefs (preferences, size) (a0, a1)
-426	0xFE56	-0x1AA	GetScreenData (buffer, size, type, Screen) (a0, d0, d1, a1)
-138	0xFF76	-0x08A	InitRequester (req) (a0)
-330	0xFEB6	-0x14A	IntuiTextLength (IText) (a0)
-36	0xFFDC	-0x024	Intuition (ievent) (a0)
-144	0xFF70	-0x090	ItemAddress (MenuStrip, MenuNumber) (a0, d0)
-414	0xFE62	-0x19E	LockIBase (dontknow) (d0)
-378	0xFE86	-0x17A	MakeScreen (Screen) (a0)
-150	0xFF6A	-0x096	ModifyIDCMP (Window, Flags) (a0, d0)
-156	0xFF64	-0x09C	ModifyProp (Gadget, Win, Req, Flags, HPos, VPos, HBody, VBody) (a0, a1, a2, d0, d1, d2, d3, d4)
-162	0xFF5E	-0x0A2	MoveScreen (Screen, dx, dy) (a0, d0, d1)

-168	0xFF58	-0x0A8	MoveWindow(Window,dx,dy) (a0,d0,d1)
-468	0xFE2C	-0x1D4	NewModifyProp(Gad,Win,Req,Flag,HPos,VPos,HBod,VBod,NumGad) (a0,a1,a2,d0,d1,d2,d3,d4,d5)
-174	0xFF52	-0x0AE	OffGadget(Gadget,Win,Req) (a0,a1,a2)
-180	0xFF4C	-0x0B4	OffMenu(Window,MenuNum) (a0,d0)
-186	0xFF46	-0x0BA	OnGadget(Gadget,Win,Req) (a0,a1,a2)
-192	0xFF40	-0x0C0	OnMenu(Window,MenuNum) (a0,d0)
-30	0xFFE2	-0x01E	OpenIntuition() ()
-198	0xFF3A	-0x0C6	OpenScreen(Args) (a0)
-204	0xFF34	-0x0CC	OpenWindow(Args) (a0)
-210	0xFF2E	-0x0D2	OpenWorkBench() ()
-216	0xFF28	-0x0D8	PrintIText(rp,IText,LeftOffset,TopOffset) (a0,a1,d0,d1)
-222	0xFF22	-0x0DE	RefreshGadgets(Gadgets,Win,Req) (a0,a1,a2)
-432	0xFE50	-0x1B0	RefreshGList(Gadgets,Win,Req,NumGad) (a0,a1,a2,d0)
-456	0xFE38	-0x1C8	RefreshWindow(Window) (a0)
-384	0xFE80	-0x180	RemakeDisplay() ()
-228	0xFF1C	-0x0E4	RemoveGadget(RemPtr,Gadget) (a0,a1)
-444	0xFE44	-0x1BC	RemoveGList(RemPtr,Gadget,NumGad) (a0,a1,d0)
-234	0xFF16	-0x0EA	ReportMouse(Boolean,Window) (a0,d0)
-240	0xFF10	-0x0F0	Request(Req,Win) (a0,a1)
-390	0xFE7A	-0x186	RethinkDisplay() ()
-246	0xFF0A	-0x0F6	ScreenToBack(Screen) (a0)
-252	0xFF04	-0x0FC	ScreenToFront(Screen) (a0)
-258	0xFEFE	-0x102	SetDMRequest(Win,Req) (a0,a1)
-264	0xFEF8	-0x108	SetMenuStrip(Win,Menu) (a0,a1)
-270	0xFEF2	-0x10E	SetPointer(Win,Pointer,Height,Width,HotX,HotY) (a0,a1,d0,d1,d2,d3)
-324	0xFEBC	-0x144	SetPrefs(preferences,size,flags) (a0,d0,d1)
-276	0xFEEC	-0x114	SetWindowTitles(window,winTitle,scrTitle) (a0,a1,a2)
-282	0xFEE6	-0x11A	ShowTitle(Screen,ShowIt) (a0,d0)
-288	0xFEE0	-0x120	SizeWindow(Window,dx,dy) (a0,d0,d1)
-420	0xFE5C	-0x1A4	UnlockIBase(IBLock) (a0)
-294	0xFEDA	-0x126	ViewAddress() ()
-300	0xFED4	-0x12C	ViewPortAddress(Window) (a0)
-336	0xFEB0	-0x150	WBenchToBack() ()
-342	0xFEAA	-0x156	WBenchToFront() ()
-318	0xFEC2	-0x13E	WindowLimits(Window,WMin,HMin,WMax,HMax) (a0,d0,d1,d2,d3)
-306	0xFECE	-0x132	WindowToBack(Window) (a0)
-312	0xFEC8	-0x138	WindowToFront(Window) (a0)

layers.library

-78	0xFFB2	-0x04E	BeginUpdate(layer) (a0)
-54	0xFFCA	-0x036	BehindLayer(li, layer) (a0, a1)
-42	0xFFD6	-0x02A	CreateBehindLayer(li, bm, x0, y0, x1, y1, flags, bm2) (a0, a1, d0, d1, d2, d3, d4, a2)
-36	0xFFDC	-0x024	CreateUpfrontLayer(li, bm, x0, y0, x1, y1, flags, bm2) (a0, a1, d0, d1, d2, d3, d4, a2)
-90	0xFFA6	-0x05A	DeleteLayer(li, layer) (a0, a1)
-150	0xFF6A	-0x096	DisposeLayerInfo(li) (a0)
-84	0xFFAC	-0x054	EndUpdate(layer, flag) (a0, d0)
-156	0xFF64	-0x09C	FattenLayerInfo(li) (a0)
-30	0xFFE2	-0x01E	InitLayers(li) (a0)
-174	0xFF52	-0x0AE	InstallClipRegion(layer, region) (a0, a1)
-96	0xFFA0	-0x060	LockLayer(li, layer) (a0, a1)
-120	0xFF88	-0x078	LockLayerInfo(li) (a0)
-108	0xFF94	-0x06C	LockLayers(li) (a0)
-60	0xFFC4	-0x03C	MoveLayer(li, layer, dx, dy) (a0, a1, d0, d1)
-168	0xFF58	-0x0A8	MoveLayerInFrontOf(movelayer, backlayer) (a0, a1)
-144	0xFF70	-0x090	NewLayerInfo() ()
-72	0xFFB8	-0x048	ScrollLayer(li, layer, dx, dy) (a0, a1, d0, d1)
-66	0xFFBE	-0x042	SizeLayer(li, layer, dx, dy) (a0, a1, d0, d1)
-126	0xFF82	-0x07E	SwapBitsRastPortClipRect(rp, cr) (a0, a1)
-162	0xFF5E	-0x0A2	ThinLayerInfo(li) (a0)
-102	0xFF9A	-0x066	UnlockLayer(layer) (a0)
-138	0xFF76	-0x08A	UnlockLayerInfo(li) (a0)
-114	0xFF8E	-0x072	UnlockLayers(li) (a0)
-48	0xFFD0	-0x030	UpfrontLayer(li, layer) (a0, a1)
-132	0xFF7C	-0x084	WhichLayer(li, x, y) (a0, d0, d1)

mathffp.library

-54	0xFFCA	-0x036	SPAbs(float) (d0)
-66	0xFFBE	-0x042	SPAdd(leftFloat,rightFloat) (d1,d0)
-96	0xFFA0	-0x060	SPCeil(float) (d0)
-42	0xFFD6	-0x02A	SPCmp(leftFloat,rightFloat) (d1,d0)
-84	0xFFAC	-0x054	SPDiv(leftFloat,rightFloat) (d1,d0)
-30	0xFFE2	-0x01E	SPFix(float) (d0)
-90	0xFFA6	-0x05A	SPFloor(float) (d0)
-36	0xFFDC	-0x024	SPFlt(integer) (d0)
-78	0xFFB2	-0x04E	SPMul(leftFloat,rightFloat) (d1,d0)
-60	0xFFC4	-0x03C	SPNeg(float) (d0)
-72	0xFFB8	-0x048	SPSub(leftFloat,rightFloat) (d1,d0)
-48	0xFFD0	-0x030	SPTst(float) (d1)

mathieeedoubbas.library

-54	0xFFCA	-0x036	IEEEDPAbs(double) (d0/d1)
-66	0xFFBE	-0x042	IEEEDPAdd(double,double) (d0/d1,d2/d3)
-96	0xFFA0	-0x060	IEEEDPCeil(double) (d0/d1)
-42	0xFFD6	-0x02A	IEEEDPCmp(double,double) (d0/d1,d2/d3)
-84	0xFFAC	-0x054	IEEEDPDiv(double,double) (d0/d1,d2/d3)
-30	0xFFE2	-0x01E	IEEEDPFix(double) (d0/d1)
-90	0xFFA6	-0x05A	IEEEDPFloor(double) (d0/d1)
-36	0xFFDC	-0x024	IEEEDPFlt(integer) (d0)
-72	0xFFB2	-0x048	IEEEDPMul(double,double) (d0/d1,d2/d3)
-60	0xFFC4	-0x03C	IEEEDPNeg(double) (d0/d1)
-72	0xFFB8	-0x048	IEEEDPSub(double,double) (d0/d1,d2/d3)
-48	0xFFD0	-0x030	IEEEDPTst(double) (d0/d1)

athieeedoubtrans.library

-120	0xFF88	-0x078	IEEEDPAcos(double) (d0/d1)
-114	0xFF8E	-0x072	IEEEDPAsin(double) (d0/d1)
-30	0xFFE2	-0x01E	IEEEDPATan(double) (d0/d1)
-42	0xFFD6	-0x02A	IEEEDPCos(double) (d0/d1)
-66	0xFFBE	-0x042	IEEEDPCosh(double) (d0/d1)
-78	0xFFB2	-0x04E	IEEEDPExp(double) (d0/d1)
-108	0xFF94	-0x06C	IEEEDPFieee(single) (d0)
-84	0xFFAC	-0x054	IEEEDPLog(double) (d0/d1)
-126	0xFF82	-0x07E	IEEEDPLog10(double) (d0/d1)
-90	0xFFA6	-0x05A	IEEEDPPow(exp, arg) (d2/d3, d0/d1)
-36	0xFFDC	-0x024	IEEEDPSin(double) (d0/d1)
-54	0xFFCA	-0x036	IEEEDPSincos(double, pf2) (a0, d0/d1)
-60	0xFFC4	-0x03C	IEEEDPSinh(double) (d0/d1)
-96	0xFFA0	-0x060	IEEEDPSqrt(double) (d0/d1)
-48	0xFFD0	-0x030	IEEEDPTan(double) (d0/d1)
-72	0xFFB8	-0x048	IEEEDPTanh(double) (d0/d1)
-102	0xFF9A	-0x066	IEEEDPTieee(double) (d0/d1)

mathtrans.library

-120	0xFF88	-0x078	SPAcos(float) (d0)
-114	0xFF8E	-0x072	SPAsin(float) (d0)
-30	0xFFE2	-0x01E	SPAtan(float) (d0)
-42	0xFFD6	-0x02A	SPCos(float) (d0)
-66	0xFFBE	-0x042	SPCosh(float) (d0)
-78	0xFFB2	-0x04E	SPExp(float) (d0)
-108	0xFF94	-0x06C	SPFieee(integer) (d0)
-84	0xFFAC	-0x054	SPLog(float) (d0)
-126	0xFF82	-0x07E	SPLog10(float) (d0)
-90	0xFFA6	-0x05A	SPPow(leftFloat, rightFloat) (d1, d0)
-36	0xFFDC	-0x024	SPSin(float) (d0)
-54	0xFFCA	-0x036	SPSincos(leftFloat, rightFloat) (d1, d0)
-60	0xFFC4	-0x03C	SPSinh(float) (d0)
-96	0xFFA0	-0x060	SPSqrt(float) (d0)
-48	0xFFD0	-0x030	SPTan(float) (d0)
-72	0xFFB8	-0x048	SPTanh(float) (d0)
-102	0xFF9A	-0x066	SPTieee(float) (d0)

potgo.library

```
-6  0xFFFA  -0x006  AllocPotBits(bits) (d0)
-12 0xFFFF4 -0x00C  FreePortBits(bits) (d0)
-18 0xFFEE  -0x012  WritePotGo(word,mask) (d0,d1)
```

romboot.library

```
-30 0xFFE2  -0x01E  RomBoot() ()
```

timer.library

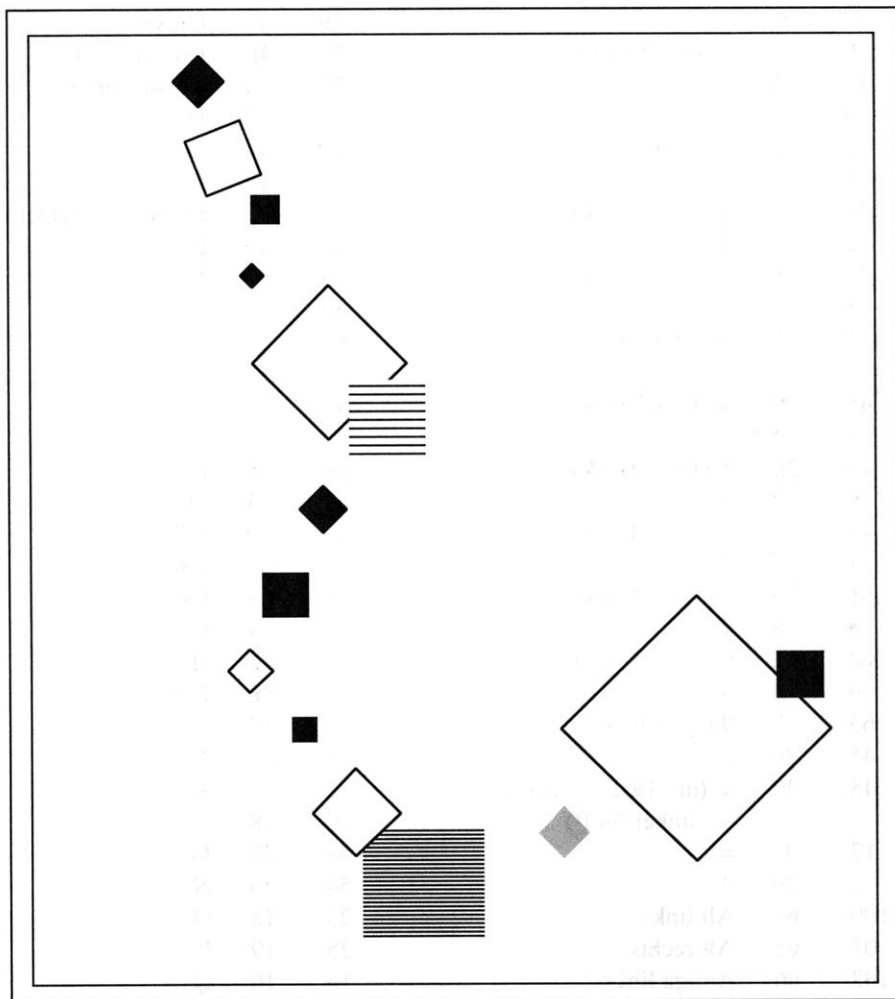
```
-42 0xFFD6  -0x02A  AddTime(dest,src) (a0,a1)
-54 0xFFCA  -0x036  CmpTime(dest,src) (a0,a1)
-48 0xFFD0  -0x030  SubTime(dest,src) (a0,a1)
```

translator.library

```
-30 0xFFE2  -0x01E  Translate(input,inputLength,outBuf,bufSize) (a0,d0,a1,d1)
```


Anhang E

Liste der Tastencodes



Liste der Tasten-Codes (alphabetisch sortiert)

42	2A	'	51	33	C
56	38	,	98	62	CapsLock
11	B	-	99	63	Ctrl
74	4A	- (10er-Block)	76	4C	Cursor hoch
57	39	.	79	4F	Cursor links
60	3C	. (10er-Block)	78	4E	Cursor rechts
58	3A	/	77	4D	Cursor runter
10	A	0	34	22	D
15	F	0 (10er-Block)	70	46	Del
1	1	1	18	12	E
29	1D	1 (10er-Block)	67	43	Enter (10er-Block)
2	2	2	69	45	Esc
30	1E	2 (10er-Block)	35	23	F
3	3	3	80	50	F1
31	1F	3 (10er-Block)	89	59	F10
4	4	4	81	51	F2
45	2D	4 (10er-Block)	82	52	F3
5	5	5	83	53	F4
46	2E	5 (10er-Block)	84	54	F5
6	6	6	85	55	F6
47	2F	6 (10er-Block)	86	56	F7
7	7	7	87	57	F8
61	3D	7 (10er-Block)	88	58	F9
8	8	8	36	24	G
62	3E	8 (10er-Block)	37	25	H
9	9	9	95	5F	Help
63	3F	9 (10er-Block)	23	17	I
41	29	;	38	26	J
48	30	< (dt. Tastatur neben linker Shift)	39	27	K
12	C	=	40	28	L
32	20	A	55	37	M
100	64	Alt links	54	36	N
101	65	Alt rechts	24	18	O
102	66	Amiga links	25	19	P
103	67	Amiga rechts	16	10	Q
53	35	B	19	13	R
65	41	Backspace	68	44	Return
			33	21	S

96	60	Shift links
97	61	Shift rechts
64	40	Space
20	14	T
66	42	Tab
22	16	U
52	34	V
17	11	W
50	32	X
21	15	Y
49	31	Z
26	1A	[
13	D	\
27	1B	]
0	0	^

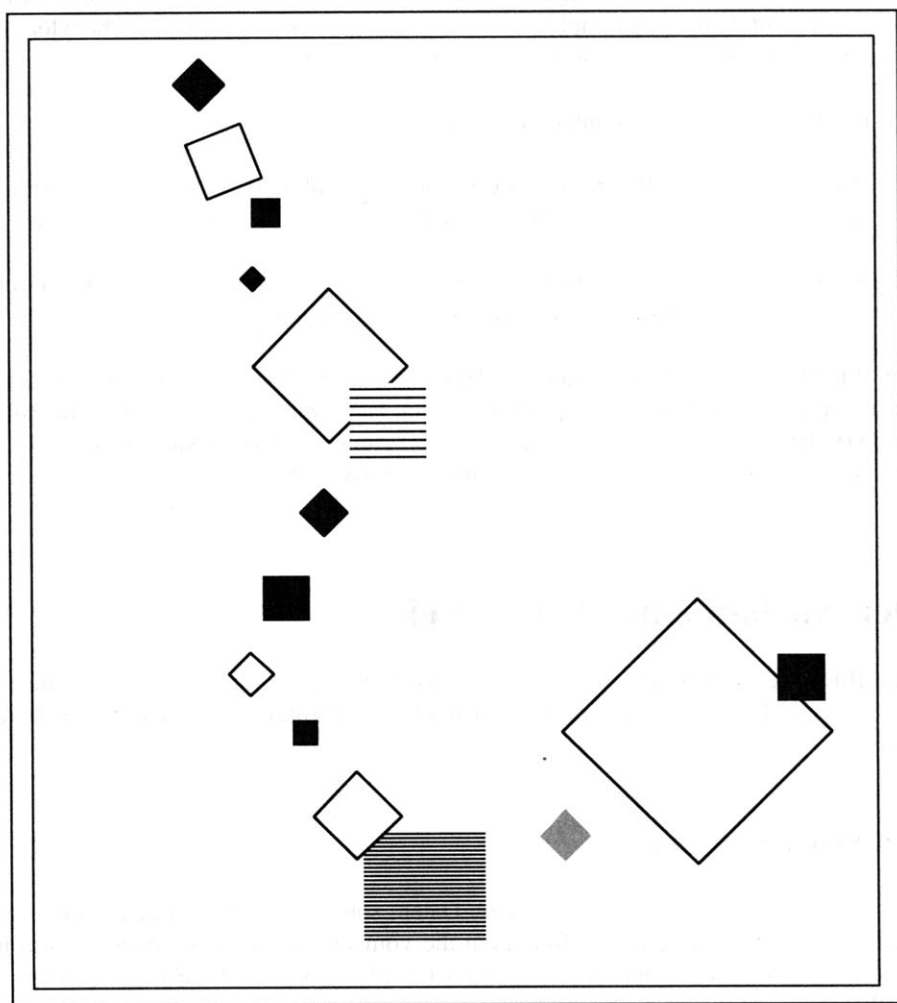
Liste der Tasten-Codes (numerisch sortiert)

0	0	`	40	28	L
1	1	1	41	29	;
2	2	2	42	2A	'
3	3	3	45	2D	4 (10er-Block)
4	4	4	46	2E	5 (10er-Block)
5	5	5	47	2F	6 (10er-Block)
6	6	6	48	30	(dt. Tastatur neben linker Shift)
7	7	7			
8	8	8	49	31	Z
9	9	9	50	32	X
10	A	0	51	33	C
11	B	-	52	34	V
12	C	=	53	35	B
13	D	\	54	36	N
15	F	0 (10er-Block)	55	37	M
16	10	Q	56	38	,
17	11	W	57	39	.
18	12	E	58	3A	/
19	13	R	61	3D	7 (10er-Block)
20	14	T	62	3E	8 (10er-Block)
21	15	Y	63	3F	9 (10er-Block)
22	16	U	64	40	Space
23	17	I	65	41	Backspace
24	18	O	67	43	Enter (10er-Block)
25	19	P			
26	1A	[	68	44	Return
27	1B	]	69	45	Esc
29	1D	1 (10er-Block)	70	46	Del
30	1E	2 (10er-Block)	66	42	Tab
31	1F	3 (10er-Block)	74	4A	- (10er-Block)
32	20	A	60	3C	. (10er-Block)
33	21	S	76	4C	Cursor hoch
34	22	D	77	4D	Cursor runter
35	23	F	78	4E	Cursor rechts
36	24	G	79	4F	Cursor links
37	25	H	80	50	F1
38	26	J	81	51	F2
39	27	K			

82	52	F3
83	53	F4
84	54	F5
85	55	F6
86	56	F7
87	57	F8
88	58	F9
89	59	F10
95	5F	Help
96	60	Shift links
97	61	Shift rechts
98	62	CapsLock
99	63	Ctrl
100	64	Alt links
101	65	Alt rechts
102	66	Amiga links
103	67	Amiga rechts
249	F9	Letzter Code unzulässig, nächstes Zeichen =Wiederholung
250	FA	Tastatur-interner Puffer-Überlauf
252	FC	Initialisierung fehlerhaft
253	FD	Einschalt- Meldung
254	FE	Ende der Einschalt- Meldung

Anhang F

Das IFF - Grafikformat



Das IFF-Format (Interchange File Format)

Der IFF-Standard wurde geschaffen, um einen reibungslosen Datenaustausch zwischen verschiedenen Programmen und Computersystemen zu realisieren und die Inkompatibilität vieler Softwareprodukte zu überwinden.

Das durch Electronic-Arts (C) 1985 definierte IFF-Format steht für einen flexiblen Datenaustausch ohne Konvertierungen und läßt sich beliebig erweitern. Da dieses Konzept von Anfang auf dem Amiga implementiert wurde, blieb dem System eine "babylonische Format-Verwirrung" wie auf vielen anderen Computer erspart.

Das IFF-Datenformat leistet unter anderem:

- den Austausch von Bilderdateien verschiedener Mal- und Zeichenprogramme
- die Bearbeitung eines Musikstückes durch unterschiedliche Musikprogramme

Da sämtliche Daten transparent sind, müssen IFF-fähige Programme nicht die gesamte Datei analysieren, sondern lediglich den für sie relevanten Teil.

Der IFF-Standard wurde so allgemein definiert, daß praktisch jede denkbare Dateninformation in einer IFF-Datei gespeichert werden kann. Zu den Konventionen für Bilder (ILBM), Text (FTXT), Musik (SMUS) und Instrumente (8SVX) kamen im Laufe der Zeit auch Vereinbarungen über Animationsdateien (ANIM).

Der Aufbau einer IFF-Datei

Eine IFF-Datei besteht aus mehreren Datenblöcken (engl. Chunks), die hintereinander gesetzt sind. Es gibt zwei Arten von Chunks: Iff-Struktur-Chunks und IFF-Daten-Chunks.

IFF-Struktur-Chunks

Diese genormten Chunks enthalten keine Daten, sondern bestimmen den strukturellen Aufbau einer IFF-Datei; eine Information die vom Lese-Programm benötigt wird, um die Datei korrekt zu interpretieren. Struktur-Chunks geben auch Auskunft darüber, ob die IFF-Datei überhaupt die benötigten Daten enthält (Datei-Typ: z.B. Bilderdatei).

Der Chunk beginnt mit einem Identifikator (ID), bestehend aus vier ASCII-Zeichen (byte-codiert), die sich zu einem Langwort ergänzen. Der ID-Code ist bei Struktur-Chunks genormt. Es dürfen keine eigenmächtigen Erweiterungen eingebaut werden, da sie die Kompatibilität mit anderen Programmen zerstören würden. Die reservierten IDs sind "FORM", "LIST", "CAT " und "PROP"; ihre Bedeutung wird unter der Überschrift "Struktur einer IFF-Datei" beschrieben.

Auf den Identifikator folgt die Länge des Chunks in Byte. Diese Angabe wird ebenfalls in einem Langwort zusammengefaßt.

Bei der Speicherung sind folgende Dinge zu beachten:

- Langworte (32 Bits) und Worte (16 Bits) müssen mit dem High-Byte (höchstwertiges Byte) zuerst abgespeichert werden (MC68000 kompatibel).
- Worte und Langworte müssen an geraden Adressen beginnen, andernfalls wird ihre Position durch ein Füll-Byte umgelenkt.
- Die vier ID-Zeichen entstammen dem ASCII-Zeichensatz im Bereich " " (SPACE, \$20) bis "~" (tilde, \$7E). Führende Leerzeichen sind nicht erlaubt.
- Die Datenformate werden alle als benutzerdefinierte Typen der Sprache "C" formuliert. Konvertierungen in andere (höhere) Programmiersprachen dürften kein Problem darstellen.

Die verwendeten Datentypen

```
typedef unsigned char  UBYTE;
                        /* 8-Bit-Ganzzahl ohne Vorzeichen. */
typedef short          WORD;
                        /* 16-Bit-Ganzzahl mit Vorzeichen. */
typedef unsigned short UWORD;
                        /* 16-Bit-Ganzzahl ohne Vorzeichen. */
typedef long           LONG;
                        /* 32-Bit-Ganzzahl mit Vorzeichen. */
typedef char           ID[4];
                        /* Zeichen-Feld mit vier Elementen. */
```

Sollte der Chunk eine ungerade Anzahl von Byte enthalten, so muß ein Füll-Byte an sein Ende angehängt werden, das allerdings nicht in der Längenangabe vermerkt wird.

Es folgt der Typ-Identifikator, der die Art der Chunk-Daten beschreibt. Er ist im Prinzip frei wählbar, sollte aber sorgfältig ausgesucht sein, da einige Identifikatoren schon vergeben sind und nicht für andere Zwecke verwendet werden dürfen.

Zu den bereits belegten Typ-Identifikatoren gehören:

ILBM: InterLeaved BitMap	= Format für Bilderdateien (Bitmaps)
FTXT: Formatted TeXT	= formatierte Textdateien
SMUS: Simple MU(s)ical Score	= Notenblatt-Information
8SVX: 8-Bit Sampled Voice	= digitalisierte Geräusche und Musikinstrumente

Nach dem Typ-Identifikator folgt eine beliebige Anzahl weiterer IFF-Struktur -bzw. Daten-Chunks (werden noch beschrieben), deren Gesamtlänge im Längenfeld vermerkt ist.

Der Aufbau eines IFF-Struktur-Chunks

```
typedef struct
{
    ID      ckID;          /* Struktur-Identifikator */
    LONG    ckSize;        /* Gesamtlänge des Struktur-Chunks */
    ID      grpSubID;      /* Typ-Kennung */
} GroupHeader;
```

Beispiel eines IFF-Struktur-Chunks:

Datei-Offset	Label	Funktion
00	"FORM"	Struktur-Identifikator (siehe unten)
04	4934	Langwort mit Längenangabe des FORM-Chunks
08	"ILBM"	Typ-Identifikator (Es handelt sich um eine ILBM-Datei)
12	??	beliebige Chunks, die Summe der Chunk-Längen und der Länge des Typ-Identifikators sei 4934 Byte. Der Kopf eines Chunks wird nicht mitberechnet.

IFF-Daten-Chunks

Der Aufbau eines IFF-Daten-Chunks ähnelt dem eines IFF-Struktur-Chunks. Genau wie dieser beginnt er mit einem Identifikator, der den Namen des Daten-Chunks angibt. Im

Gegensatz zum Struktur-Identifikator darf der Name allerdings frei gewählt werden, solange er nicht mit schon bestehenden IDs kollidiert.

Anschließend folgt die Längenangabe als Langwort. Auch hier muß auf die besondere Behandlung ungeradzahlig langer Chunks hingewiesen werden.

Danach kommen die eigentlichen Daten, deren Länge bereits eingetragen wurde. Die Struktur des Datenfeldes ist völlig offen.

Wichtig ist nur, daß Daten-Chunks sequentiell angeordnet werden, Schachtelungen sind nicht erlaubt.

Auch für Daten-Chunks ist eine Darstellung als C-Struktur gebräuchlich:

```
typedef struct
{
    ID    ckID;        /* Chunk-Identifikator */
    LONG ckSize;       /* Länge des Daten-Chunks */
} ChunkHeader;
```

Beispiel eines IFF-Daten-Chunks:

Chunk-Offset	Label	Funktion
00	"CMAP"	Namen-Identifikator (CMAP steht für Color-MAP)
04	12	Länge der Daten beträgt 12 Byte.
08-19	??	12 Byte, die als 4*3 Byte Farbinformation interpretiert werden (drei Byte für die RGB-Werte von Farbe 0, weitere drei Byte für Farbe 1, usw.)

Im folgenden Beispiel ist die Chunk-Länge ungerade:

Chunk-Offset	Label	Funktion
00	"CMAP"	Namen-Identifikator
04	15	Der Daten-Chunk enthält 15 Byte!
08-22	??	15 Daten-Byte (5*3 Byte)
23	00	Füll-Byte (ist nicht in der Längenangabe enthalten!)

2. IFF-Datei mit mehreren FORM-Chunks zum Teil unterschiedlichen Typs (nicht unbedingt empfehlenswert, aber als Beispiel geeignet!)

```

FORM Länge TEST      :siehe oben
  CHD1 Länge         :1. Daten-Chunk
    Daten
  CHD2 Länge         :2. Daten-Chunk
    Daten
  CHD2 Länge         :3. Daten-Chunk mit gleichem Aufbau wie
                     2. Daten-Chunk
FORM Länge TEST      :Ende des ersten FORM-Chunks,
                     Beginn des zweiten
  CHD1 Länge         :1. Daten-Chunk des 2. FORMs
    Daten
  CHD2 Länge         :2. Daten-Chunk des 2. FORMs
FORM Länge DEMO      :3. FORM-Chunk (anderer Typ)
  CHD4 Länge         :1. Daten-Chunk des 3. FORMs
    Daten
  CHD1 Länge         :Gleicher Daten-Chunk, wie in FORM-Chunk
                     TEST.
    Daten
<EOF>

```

3. Verschachtelte FORM-Chunks zur Strukturierung. (In FORM-Chunks dürfen auch die unten beschriebenen CAT- oder LIST-Chunks auftreten, nicht aber der PROP-Chunk, der lediglich dem LIST-Chunk vorbehalten ist.)

```

FORM Länge ALLE      :Umfassender Struktur-Chunk (Länge gilt
                     für gesamte Datei inklusive aller
                     geschachtelten FORMs!)
  FORM Länge TEST    :Integrierter FORM-Chunk (Länge gilt nur
                     für diesen FORM-Chunk)
    CHD1 Länge       :1. Daten-Chunk des integrierten Chunks
      Daten
    CHD2 Länge       :2. Daten-Chunk
      Daten
  FORM Länge TEST    :Hier ist die sequentielle Anordnung von
                     FORM-Chunks sinnvoll!
    CHD1 Länge       :
      Daten
  FORM Länge DEMO    :Auch andere FORM-Chunks können einge-
                     baut werden.

```

```

      CHD4 Länge
      Daten
CHD5 Länge      :Privater Daten-Chunk des umfassenden
                  FORMs
      Daten
CHD6 Länge      :2. privater Daten-Chunk
      Daten
<EOF>

```

"CAT "

Der CAT-Chunk wird immer dann verwendet, wenn die unterschiedlichsten Daten in einer Datei zusammengefaßt werden sollen, auch wenn sie logisch nicht zueinander in Beziehung stehen.

Ein CAT-Chunk ist im wesentlichen eine Ansammlung unterschiedlichster FORM- und LIST-Chunks. Eine Schachtelung von CAT-Chunks ist nicht zu empfehlen, da es sich um eine unstrukturierte Datenverbindung handelt. Vielmehr ist es sinnvoll, den Inhalt unterschiedlicher CAT-Chunks zu vermischen.

Lokale Daten-Chunks sind in CAT-Chunks allerdings nicht erlaubt. Die Einstreuung dieses Typs ist dem Aneinanderreihen von FORM-Chunks auf jeden Fall vorzuziehen, da die Gesamtlänge der Daten schon am Anfang feststeht.

Der Typ-Identifikator eines CAT-Chunks hat aufgrund des unterschiedlichen Inhalts einer solchen Datei nur geringe Bedeutung, er sollte lediglich einen Hinweis auf den möglichen Inhalt geben.

Beispiel:

```

CAT Länge XXXX      :Struktur-Identifikator "CAT " mit
                    Typ-ID "XXXX"
  FORM Länge TEST    :1. FORM-Chunk
    CHD1 Länge
    Daten
  FORM Länge DEMO    :2. FORM-Chunk
    CHD4 Länge
    Daten
    CHD4 Länge
    Daten
  FORM Länge YYYY    :3. FORM-Chunk

```

```

        CHD7 Länge
        Daten
<EOF>

```

"LIST"

Der LIST-Chunk ist vergleichbar mit dem CAT-Chunk, allerdings ist er wesentlich strukturierter. Meistens enthält ein LIST-Chunk nur FORM- oder LIST-Chunks gleichen Typs, vergleichbar etwa mit einem eindimensionalen Feld (Array).

Durch Schachtelung lassen sich auch mehrdimensionale Felder simulieren, so daß ein Auflösen der Verschachtelung nicht zu empfehlen ist. LISTs, sowie CATs dürfen zwar keine Daten-Chunks enthalten, doch bietet die Verwendung von PROP-Chunks besondere Möglichkeiten. Der Typ-Identifikator ist aufgrund der strengen Ordnung ein wichtiger Bestandteil des LIST-Chunks.

Beispiel:

```

LIST Länge TEST           :Struktur-Identifikator "LIST" mit
                           Typ-ID "TEST"
                           (Der Typ-ID besagt, daß vorwiegend
                           oder sogar ausschließlich
                           FORM- oder LIST-Chunks vom Typ
                           TEST vorkommen!)
FORM Länge TEST           :1. FORM-Chunk
    CHD1 Länge
    Daten
    ...
FORM Länge TEST           :2. FORM-Chunk
    ...
FORM Länge TEST           :3. FORM-Chunk
    ...
LIST Länge TEST           :Warum nicht?
                           Schachtelung ist sinnvoll!
    FORM Länge TEST       :1. FORM-Chunk des integrierten LIST-
                           Chunks
    ...
    FORM Länge TEST
    ...
<EOF>

```


"PROP"

PROP-Chunks dürfen nur innerhalb von LIST-Chunks vorkommen. Sie erlauben eine Komprimierung der Datei, da gleichartige Daten-Chunks durch Verwendung der PROP-Chunks nur einmal aufgeführt werden müssen.

Der PROP-Chunk enthält ausschließlich Daten-Chunks, die als Vorgabe- oder Standardwert für nachfolgende FORM- oder LIST-Chunks interpretiert werden. Gleichlautende Daten-Chunks innerhalb der korrespondierenden FORM-Chunks überschreiben diesen Vorgabewert.

Der Typ-Identifikator eines PROP-Chunks bestimmt, welcher FORM-Typ von der Vorgabe beeinflusst wird. Es ist wichtig, daß die PROP-Chunks vor den weiteren FORM- und LIST-Chunks stehen.

Beispiel:

LIST Länge TEST	:Hauptsächlich FORMs vom Typ TEST treten auf.
PROP Länge TEST	:Dieser PROP-Chunk beeinflusst nur FORMs vom Typ TEST.
CHD1 Länge	:Vorgabe für CHD1-Chunks in FORMs vom Typ TEST.
Daten	
CHD2 Länge	:Vorgabe für CHD2-Chunks.
Daten	
PROP Länge DEMO	:Folgende Vorgaben gelten nur für FORMs vom Typ DEMO.
CHD4 Länge	:Vorgabe für CHD4-Chunks (gilt nur für CHD4-Chunks innerhalb von FORMs des Typs DEMO, nicht jedoch für FORMs vom Typ TEST!)
Daten	
FORM Länge TEST	:FORM-Chunk vom Typ TEST, wird durch PROP-Chunk beeinflusst.
CHD3 Länge	:Lokaler Daten-Chunk.
Daten	
FORM Länge TEST	:Weiterer FORM-Chunk vom Typ TEST.
CHD1 Länge	:Dieser Daten-Chunk überschreibt den Vorgabewert des CHD1-Chunks im PROP-Chunk.

Daten	
CHD4 Länge	:Dieser CHD4-Chunk wird nicht durch den CHD4-Chunk im PROP-Chunk beeinflusst!
Daten	
LIST Länge TEST	:Lokaler LIST-Chunk vom Typ TEST.
PROP Länge TEST	:Neuer Vorgabewert. Der alte muß gespeichert werden, da auf diese Werte später wieder zurückgegriffen werden kann.
CHD1 Länge	:Neuer Vorgabewert für CHD1-Chunk. Der alte Wert des CHD2-Chunks wird nicht verändert.
Daten	
FORM Länge TEST	:Hier kommt die neue Vorgabe zum Tragen!
CHD3 Länge	:Lokaler CHD3-Chunk.
Daten	
...	
FORM Länge TEST	:Der alte Wert des CHD2-Chunks muß wieder gelten, obwohl er zwischen- durch überschrieben wurde.
CHD3 Länge	
Daten	
FORM Länge DEMO	:Anderer FORM-Typ. Die Vorgaben des vorgesehenen PROP-Chunks gelten nun.
CHD1 Länge	:Der Daten-Chunk hat innerhalb von FORM-Chunks dieses Typs keine Vorgabe!
Daten	
<EOF>	

Das IFF-Grafikformat (ILBM)

Der Aufbau einer kompletten IFF-Datei soll exemplarisch anhand des IFF-Grafikformats gezeigt werden. Gewöhnlich verwendet es lediglich einen einzigen FORM-Chunk (Animationsprogramme greifen teilweise auf die anderen Struktur-Chunks zurück) mit dem Typ-Identifikator "ILBM", was für "InterLeaved BitMap" (= gepackte Bitmap, beschreibt die Präsentationsform der Bitmap-Daten) steht.

Es folgen die einzelnen Daten-Chunks, deren Bezeichnungen reserviert sind.

Die Reihenfolge der Chunks ist normalerweise statisch und sollte nicht verändert werden, obwohl der IFF-Standard dies zuläßt. Es erleichtert aber das Lesen der Datei, wenn die Daten-Chunks in der richtigen Reihenfolge auftreten.

Folgende Daten-Chunks sind definiert:

Standardisierter Daten-Chunk "BMHD" (BitMapHeaDer)

Dieser Daten-Chunk hat eine Länge von 20 Byte. Er ist fester Bestandteil einer ILBM-Datei und beschreibt den grundlegenden Aufbau der Grafik (Anzahl der Farben, Größe der Bitmap usw.). BMHD steht grundsätzlich an erster Stelle, da die Interpretation sämtlicher anderen Daten-Chunks von seinen Informationen abhängt.

Ein BMHD-Chunk hat folgendes Aussehen:

```
typedef UBYTE Masking;
#define mskNone                0
#define mskHasMask             1
#define mskHasTransparentColor 2
#define mskLasso               3
typedef UBYTE Compression;
#define cmpNone                0
#define cmpByteRun1            1
typedef struct
{
    UWORD w,h;                 /* Breite und Höhe der Bitmap */
    WORD  x,y;                 /* Position der Grafik */
    UBYTE nPlanes;             /* Anzahl der Bitplanes in der Bitmap */
    UBYTE masking;             /* Verwendetes Maskierungsverfahren */
    UBYTE compression;        /* Verwendetes Komprimierungsverfahren */
    UBYTE pad1;                /* Füll-Byte als Ausgleich */
    UWORD transparentColor;    /* Bitmuster der transparenten Farbe */
    UBYTE xAspect,yAspect;    /* Seiten-Verhältnis des Bildschirms */
    WORD  pageWidth,pageHeight; /* Größe der Quell-Bitmap */
} BitmapHeader;
```

Die Variablen `w` und `h` legen die Breite und Höhe der Bitmap (in Pixeln) fest. Die Bitmap wird zeilenweise (aufgeteilt in Worte) abgespeichert. Sollte die Breite nicht ein Vielfaches von 16 betragen, so muß ein zusätzliches Datenwort hinzuaddiert werden. Die Anzahl der Worte pro Zeile folgt aus dem Term: $\text{Wortbreite} = ((w+15)/16)$. `x` und `y` bestimmen die Position der Grafik innerhalb der Ziel-Bitmap. Der Standardwert beträgt für beide Werte 0. Dies entspricht der linken, oberen Ecke der Ziel-Bitmap.

Mit `nPlanes` wird die Anzahl der verwendeten Bitplanes angegeben. Je größer dieser Wert, desto mehr Farben können gleichzeitig dargestellt werden. Gilt `nPlanes` = 0, so enthält die ILBM-Datei keine Bitmap. Das ist sinnvoll, wenn nur die Farben (Palette) eines Bildes abgespeichert werden soll.

Das Feld `masking` wählt eine von zur Zeit vier verschiedenen Maskierungstechniken. Die Maske legt den Verknüpfungsmodus der Grafik mit dem Hintergrund fest. An den Stellen, deren korrespondierende Masken-Bits gelöscht sind, bleibt die alte Grafik bestehen; dort, wo die Bits gesetzt sind, wird die neue Grafik übernommen. Auf diese Weise lassen sich auch Grafiken ohne strenge Rechteckform einbinden.

Der Wert `mskNone` zeigt an, daß keine Maskierung vorgesehen ist. Die Grafikdaten werden also direkt in die Ziel-Bitmap eingeblendet.

`mskHasMask` signalisiert, daß die Maske in den Grafikdaten integriert ist und deswegen berücksichtigt werden muß – egal, ob sie verwendet wird oder nicht.

Der Wert `mskHasTransparentColor` bestimmt, welches Bitmuster der Bitplanes als transparent angesehen wird. Es handelt sich dabei nicht direkt um ein Farbregister, da die Ziel-Bitmap eventuell mehr Bitplanes umfaßt. Durch die transparente Farbe geht eine Farbe der Ausgangs-Bitmap verloren.

Die letzte Maskierungsoption ist `mskLasso`. Die Grafik ist von einem 1 Pixel breiten Rahmen mit dem Bitmuster aus `transparentColor` eingehüllt und wird anschließend vom Rand aus mit dieser Farbe gefüllt. Alle auf diese Weise gefärbten Pixel sind transparent.

Der Struktureintrag `compression` bezeichnet das verwendete Komprimierungsverfahren. Der Wert `Null` (`cmpNone`) signalisiert, daß die Daten nicht komprimiert sind. Das einzige zur Zeit unterstützte Verfahren wird mit `cmpByteRun1` (= 1) eingeschaltet (Beschreibung folgt weiter unten).

Das `pad1`-Feld ist ein Füll-Byte.

transparentColor spezifiziert ein Bitmuster, das transparent dargestellt werden soll. Dieses Feld hat nur dann eine Funktion, wenn die Maskierung vom Typ mskLasso oder mskHasTransparentColor benutzt wird.

Die Felder xAspect und yAspect geben das Größenverhältnis der Pixel an. Es entspricht direkt der Breite und Höhe eines Pixels. Damit kann man die Pixeldarstellung der unterschiedlichen Bildschirmauflösungen ausgleichen, so daß sich Verzerrungen korrigieren lassen.

Die letzten beiden Felder pageWidth und pageHeight enthalten die Bildschirmauflösung, die bei der Erstellung der Grafik verwendet wurde (beispielsweise 320x256 für ein Low-Resolution-Bild). Dieser Wert kann dann für die Darstellung der Grafik übernommen werden.

Standardisierter Daten-Chunk "CMAP" (ColorMAP)

Dieser Chunk ist optional, befindet sich aber gewöhnlich innerhalb einer ILBM-Datei. Er enthält die Farbinformation für das Bild. Je nach Anzahl der verwendeten Farben ist die Länge dieses Chunks variabel.

Der Aufbau des CMAP-Chunks:

```
typedef struct
{
    UBYTE red, green, blue; /* Farbanteile für Rot, Grün und Blau */
} ColorRegister;
typedef ColorRegister ColorMap[n];
/* Farbtabelle mit n*3 Elementen */
```

Auf den Chunk-Header folgt die eigentliche Farbtabelle, deren einzelne Einträge die Inhalte der Farbgregister darstellen mit den drei Komponenten für Rot-, Grün- und Blauanteil der additiven Farbmischung. Für jeden Wert steht ein UBYTE zur Verfügung. Sein Wertebereich reicht von 0 bis 255, wobei 0 für keine und 255 für volle Intensität der entsprechenden Grundfarbe steht.

Sollte der Computer nicht den vollen Intensitätsumfang verarbeiten können, so sind lediglich die oberen Bits der Byte-Werte relevant. Beim Amiga ist das der Fall. Er kann lediglich 16 Intensitäten (vier Bits pro Grundfarbe) unterscheiden, so daß die unteren vier Bits der Werte ignoriert werden können. Dies erreicht man durch Verschieben der Werte um vier Bits nach rechts.

Die Anzahl der Farbreger folgt normalerweise direkt aus der Darstellungstiefe (verwendete Bitplanes). Bei vier Bitplanes sind es $2^4 = 16$ Farben, die Farbtabelle enthält 16 Farbeinträge. Das entspricht einer Chunk-Länge von $16 \times 3 = 48$ Byte.

Standardisierter Daten-Chunk "GRAB"

Dieser Chunk enthält die Koordinaten des sogenannten "Hot-Spots" (wirksame Punkte des Mauszeiger-Sprites). Wegen seiner Spezialverwendung ist der Chunk nicht immer vorhanden.

Datenstruktur des GRAB-Chunks:

```
typedef struct
{
    WORD x,y;          /* Koordinaten des Hot-Spots */
} Point2D;
```

Die Werte x und y geben die relative Verschiebung des Hot-Spots gegenüber der linken, oberen Ecke der Grafik in Pixeln an.

Standardisierter Daten-Chunk "DEST" (DESTination)

Dieser optionale Chunk informiert das Lese-Programm darüber, wie die Grafik in die Ziel-Bitmap eingefügt werden soll und bestimmt die Interpretation der transparenten Farbe sowie der Maskierungsfunktion.

Inhalt des DEST-Chunks:

```
typedef struct
{
    UBYTE depth;        /* Anzahl der Bitplanes in der urspr.
                        Quell-Bitmap */
    UBYTE pad1;         /* Füll-Byte zum Ausgleich */
    UWORD planePick;     /* Verwendete Quell-Bitplanes */
    UWORD planeOnOff;    /* Standard-Wert für fehlende Bitplanes */
    UWORD planeMask;     /* Verwendete Ziel-Bitplanes */
} DestMerge;
```

Das Feld depth gibt die ursprüngliche Anzahl der Bitplanes in der Quell-Bitmap an, aus der die Grafik "ausgeschnitten" wurde.

Die Vektoren planePick, planeOnOff, planeMask:

Bitposition:	5	4	3	2	1	0
Bitplane:	Plane6	Plane5	Plane4	Plane3	Plane2	Plane1

planePick legt fest, welche Bitplanes der Quell- in die Ziel-Bitplane geschrieben werden. Nur die korrespondierenden Bitplanes der Einer-Bits der planePick-Maske finden Verwendung. planeMask wirkt in der gleichen Weise auf die Ziel-Bitmaps. Null-Bits sperren die Bitplanes, Einer-Bits geben sie frei.

planeOnOff bestimmt den Wert der Bitplanes, die nicht von der Quelle überschrieben wurden.

Standardisierter Daten-Chunk "SPRT" (SPRiTe)

Der SPRT-Chunk legt die Darstellungspriorität einer eventuell vorhandenen Sprite-Grafik fest.

Inhalt des Chunks:

```
typedef UWORD SpritePrecedence;    /* Priorität des Sprites */
```

Das Feld SpritePrecedence darf Werte von 0 bis 7 enthalten. Dem Wert 0 ist die höchste Priorität zugeordnet, das zugehörige Sprite erscheint vor allen anderen.

Bei der Interpretation von Sprite-Daten ist zu beachten, daß eine maximale Breite von 16 Pixeln erlaubt ist. Sprite-Grafiken enthalten zwei Bitplanes, was einer Farbanzahl von vier entspricht.

Die Farbwerte müssen allerdings noch den entsprechenden Farbregistern zugeordnet werden, in dem der Tabellenanfang verschoben wird.

Standardisierter Daten-Chunk "CAMG"

Dieser Chunk ist speziell für den Amiga entwickelt worden und korrespondiert mit den verschiedenen Grafikmodi des Systems. Er braucht nicht ausgewertet zu werden, erleichtert aber die Initialisierung von HAM-, Dual-PlayField- und Extra-Half-Bright-Modi.

Die Definitionen für den CAMG-Chunk sind folgende:

```
#include <graphics/view.h>
#define BADFLAGS (SPRITES|VP_HIDE|GENLOCK_AUDIO|GENLOCK_VIDEO)
/* Nicht erlaubte Grafikmodi */
#define FLAGMASK (~BADFLAGS) /* Alle erlaubten Grafikmodi */
#define CAMGMASK (FLAGMASK & 0000FFFFL)
/* Tatsächlich benutzbare Grafikmodi */
typedef unsigned long ViewModes;
/* Verwendete Grafikmodi */
```

Diese Definitionen erlauben die Aktivierung der speziellen Grafikmodi.

Der Wert des ViewModes-Feldes wird mit CAMGMASK maskiert und kann dann direkt in das Modes-Feld eines ViewPorts übertragen werden. Die mit BADFLAGS bezeichneten Grafikmodi werden durch den CAMG-Chunk nicht unterstützt und sollten daher auch nicht verwendet werden.

FLAGMASK stellt den Maskenwert der erlaubten Modi an. CAMGMASK blendet die (von Commodore reservierten) oberen 16 Bits des ViewModes-Feldes aus.

Die korrekte Initialisierung des ViewModes-Eintrages sieht so aus:

```
Lesen:      NewScreen.ViewModes = ViewModes & CAMGMASK;
Schreiben:  ViewModes           = Screen->ViewPort.Modes &
                                         CAMGMASK;
```

Standardisierter Daten-Chunk "BODY"

Dieser Daten-Chunk enthält die eigentliche Bitmap-Information als gepackte (komprimierte) oder ungepackte Daten.

Da er den meisten Speicherplatz benötigt, sollte er auf jeden Fall am Ende der ILBM-Datei stehen, um nicht unnötig zwischenspeichern zu müssen.

Die Bitmap-Daten werden zeilenweise in einer langen Kette (nicht bitplaneweise) abgespeichert. Dabei wird mit der ersten Bitplane der obersten Rasterzeile in Richtung aufsteigender Adressen (also von links nach rechts) begonnen. Sollte die Breite der Bitmap nicht das Vielfache eines Wortes (16 Bits) betragen, so müssen die überzähligen Bits in ein zusätzliches Wort gepackt werden.

Die Datenübertragung zu den Bitplanes erfolgt byteweise, anschließend werden auf gleiche Weise die übrigen Bitplanes, sowie das optionale Masken-Bitplane übertragen. Erst danach wird die Rasterzeilennummer erhöht. Dieses Verfahren erlaubt es dem Lese-Programm, nach einer bestimmten Anzahl von Zeilen den Lesevorgang zu beenden. Auch ist es sehr leicht möglich, die Anzahl der Bitplanes zu reduzieren, indem einfach die nachfolgenden Bitplanes einer Zeile überlesen werden.

Ein optionaler Packalgorithmus (zur Zeit nur `cmpByteRun1` verwendet), erlaubt die zeilenweise Komprimierung der Rasterdaten. Dabei dürfen allerdings nie die Grenzen der Zeilenbreite überschritten werden.

Der `cmpByteRun1`-Algorithmus verwendet folgendes Verfahren:

Packen

Wenn in der Bitplane der aktuellen Rasterzeile mehr als zwei gleiche Byte aufeinanderfolgen, so werden diese n Byte als zwei Byte mit folgendem Inhalt abgespeichert:

1. Byte: Anzahl der gleichen Byte gespeichert als $1-n$.
2. Byte: Byte, welches n mal wiederholt wird.

Wichtig: Sollten mehr als 128 gleiche Byte aufeinanderfolgen, so müssen die Daten in mehrere Gruppen zu je 128 Byte aufgeteilt werden.

Sollten weniger als zwei gleiche Byte aufeinanderfolgen, so wird die längste Kette aufeinanderfolgender, unterschiedlicher Byte gesucht.

Diese Byte-Folge wird dann folgendermaßen abgelegt:

1. Byte: Ist n die Anzahl der unterschiedlichen Byte, so enthält dieses Byte den Wert $n-1$.
2. Byte: Nun folgen ab dem zweiten Byte die n unterschiedlichen Byte. Das ergibt eine Gesamtlänge von $n+1$ Byte.

Wichtig: Sollten mehr als 128 unterschiedliche Byte hintereinanderliegen, so sind die Daten in zwei oder mehrere 128-Byte-Blöcke aufzuteilen.

Der Byte-Wert $-128 = \$80$ ist als Null-Operator vorgesehen. Ein solcher Wert hat keine Funktion und muß überlesen werden. Es empfiehlt sich, ab einer Folge von zwei Byte zu packen, da hier schon ein Byte gespart wird. Nur wenn diese Zwei-Byte-Folge von

nicht-komprimierten Daten umgeben ist, sollten diese Abschnitte in einem großen ungepackten Block zusammengefaßt werden.

Entpacken

Nach dem Lesen des ersten Bytes der gepackten Daten wird sein Vorzeichen geprüft. Ist der Wert n positiv, so folgen auf dieses Byte $n+1$ unterschiedliche Daten-Byte. Sie können direkt in die aktuelle Bitplane übernommen werden.

War der Wert n dagegen negativ, so folgt ein Daten-Byte, das $(-n+1)$ mal in die aktuelle Bitplane kopiert werden muß. Sollte der Wert $n-128$ ($=\$80$) betragen, so hat das Byte keine Funktion und muß überlesen werden.

Dieser Vorgang wird solange wiederholt, bis die komplette Zeile der aktuellen Bitplane versorgt ist. Anschließend ist die gleiche Zeile der folgenden Bitplane an der Reihe. Ist auch sie verarbeitet, so wird mit der nächsten Rasterzeile begonnen.

Nichtstandardisierter Daten-Chunk "CRNG" (ColorRaNGe)

Dieser von Deluxe Paint benutzte Chunk beschreibt einen Color-Cycling-Bereich der Farbtabelle. Deluxe Paint verwendet (je nach Version) vier bis sechs dieser Chunks in einer ILBM-Datei.

Unter Color-Cycling versteht man eine zyklische Vertauschung der Farbgisterinhalte, wodurch sich eindrucksvolle Bewegungseffekte simulieren lassen.

Der Aufbau des Chunks:

```
#define RNG_ACTIVE 1      /* Color-Cycling-Bereich angeschaltet */
#define RNG_REVERSE 2    /* Die Farben werden rückwärts
                           "gecyclet" */

typedef struct
{
    WORD    pad1;          /* Feld für zukünftige Erweiterungen,
                           aktuell 0 */
    WORD    rate;          /* Cycle-Geschwindigkeit */
    WORD    flags;         /* siehe oben! */
    UBYTE   low,high;      /* Cycle-Bereich innerhalb der
                           Farbpalette */
} CRange;
```

Das Feld pad1 ist für spätere Erweiterungen reserviert.

Im Feld rate steht die Geschwindigkeit, mit der die Farben des Bereichs rotiert werden sollen. Dabei entspricht ein Wert von 214 = 16384 einer Umschaltgeschwindigkeit von 60 Cycles pro Sekunde. Der Wert für 30 Cycles pro Sekunde beträgt $x = 30 / 60 * 16384 = 8192$. Die für die PAL-Norm übliche Frequenz von 50 Cycles/s bzw. 25 Cycles/s ergibt einen Wert von $x = 50 / 60 * 16384 = 13653$ (oder: $x = 25 / 60 * 16384 = 6827$).

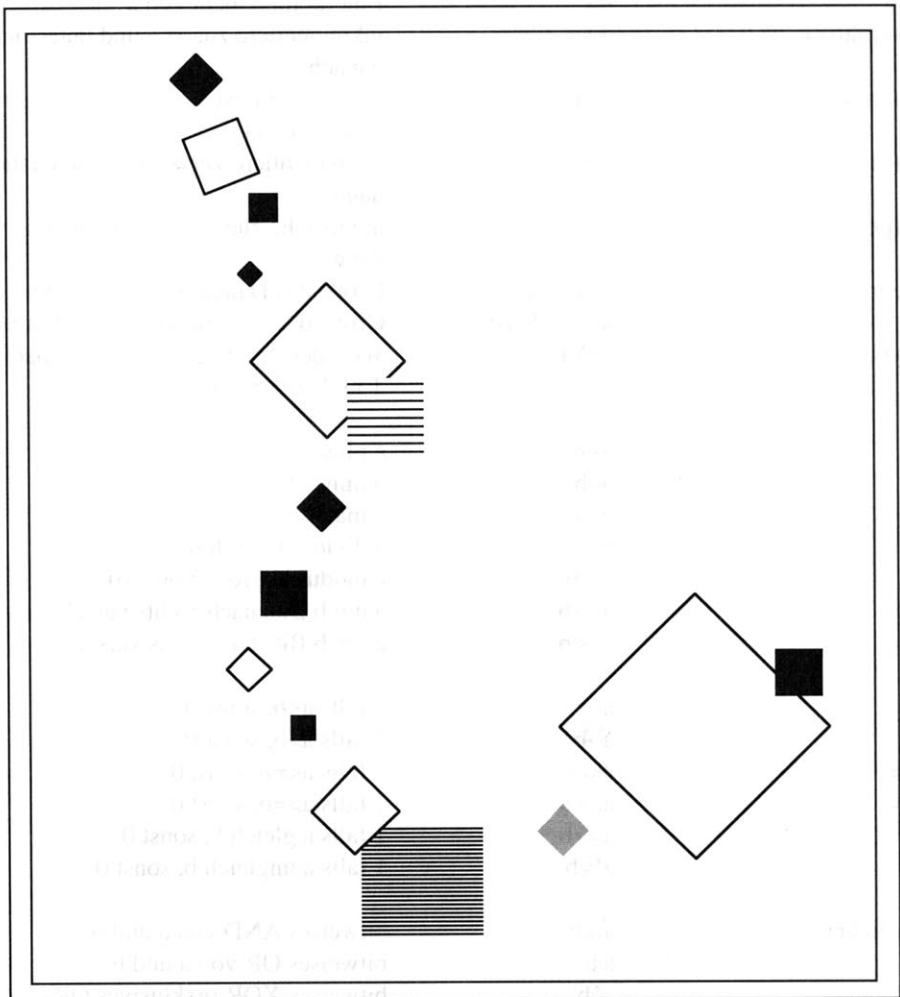
Das Feld flags kann zur Zeit die Werte RNG_ACTIVE und RNG_REVERSE enthalten. RNG_ACTIVE bedeutet, daß der Cycle-Bereich zur Zeit der Abspeicherung des Bildes aktiv war.

RNG_REVERSE kehrt die Rotationsrichtung um. Normalerweise werden die Farbregister in aufsteigender Reihenfolge vertauscht, also Farbregister n erhält den Wert von Register n-1. Ist RNG_REVERSE gesetzt, erhält Register n den Wert von Register n+1.

Die beiden Felder low und high markieren den beeinflussten Farbbereich. Der Wert low gibt die Nummer des unteren Farbregisters an, high bezeichnet das obere Farbregister. (Der high-Wert sollte immer größer als low sein.)

Anhang G

Die C - Operatoren und ihre Prioritätsverteilung



Liste der C-Operatoren:

Operator	Beispiel	Funktion
-(einstellig)	-a	Negation von a
*(einstellig)	*a	Inhalt des Zeigers a
&(einstellig)	&a	Adresse der Variablen a
~	~a	Einerkomplement von a
++(präfix)	++a	inkrementiere zuerst a und untersuche danach
++(postfix)	a++	betrachte zuerst a und inkrementiere anschließend
--(präfix)	--a	dekrementiere zuerst a und untersuche dann
--(postfix)	a--	untersuche zuerst und dekrementiere dann
sizeof	sizeof(TYP)	Größe des Datentyps TYP in Byte
	sizeof(EXP)	Größe des Ausdrucks EXP in Byte
(typ-name)	(TYP)EXP	Wert des Ausdrucks EXP wird nach TYP konvertiert
+	a+b	a plus b
-	a-b	a minus b
*	a*b	a mal b
/	a/b	a dividiert durch b
%	a%b	a modulo b (Rest von a/b)
>>	a>>b	a um b Bits nach rechts verschoben
<<	a<<b	a um b Bits nach links verschoben
<	a<b	1 falls a<b, sonst 0
>	a>b	1 falls a>b, sonst 0
<=	a<=b	1 falls a<=b, sonst 0
>=	a>=b	1 falls a>=b, sonst 0
==	a==b	1 falls a gleich b, sonst 0
!=	a!=b	1 falls a ungleich b, sonst 0
&(binär)	a&b	bitweises AND von a und b
	a b	bitweises OR von a und b
^	a^b	bitweises XOR (exklusives OR) von a und b

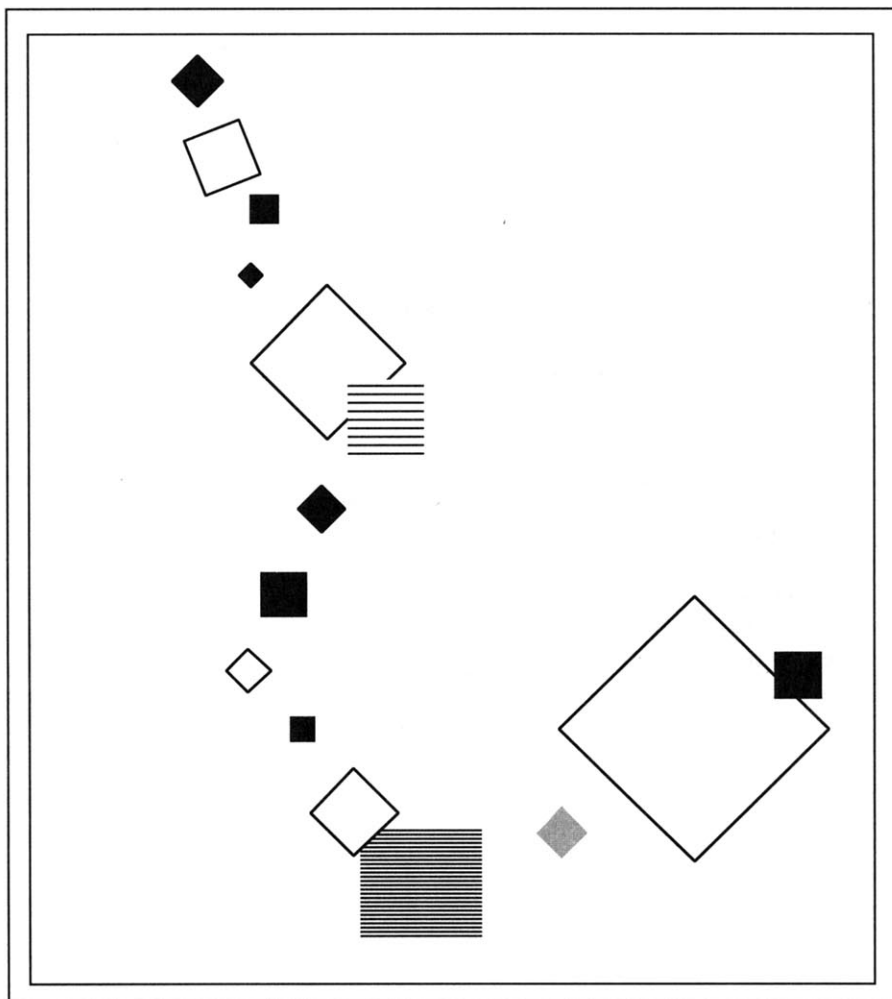
Operator	Beispiel	Funktion
&&	a&&b	logisch a AND b (liefert 1 oder 0)
	a b	logisch a OR b (liefert 1 oder 0)
!	!a	NOT a (liefert 1 oder 0)
?:	a?EXP1:EXP2	Ausdruck EXP1 falls a ungleich Null, sonst Ausdruck EXP2
=	a=b	weise a den Wert von b zu
+=	a+=b	a=a+b (a plus b, Ergebnis a zugewiesen)
-=	a-=b	a=a-b (a minus b, Ergebnis a zugewiesen)
=	a=b	a=a*b (a mal b, Ergebnis a zugewiesen)
%=	a%=b	a=a%b (a modulo b, Ergebnis a zugewiesen)
>>=	a>>=b	a=a>>b (a um b Stellen nach rechts geschoben, Ergebnis a zugewiesen)
<<=	a<<=b	a=a<<b (a um b Stellen nach links geschoben, Ergebnis a zugewiesen)
&=	a&=b	a=a&b (a AND b, Ergebnis a zugewiesen)
=	a =b	a=a b (a OR b, Ergebnis a zugewiesen)
^=	a^=b	a=a^b (a XOR b, Ergebnis a zugewiesen)
,	EXP1,EXP2	Resultat EXP2 (Ausdruck EXP1 zuerst ausgewertet)
[]	a[3]	drittes Element des a-Feldes (Feld-Selektor)
->		Struktur-Selektor (Zeiger)
.		Struktur-Selektor (Variablen)
()		Funktions-Deklarator

Prioritätsverteilung der C-Operatoren:

Priorität	Operator	Assoziativität
15	() [] -> .	von links nach rechts
14	! ~ ++ -- (TYP) * & sizeof	von links nach rechts
13	* / %	von links nach rechts
12	+ -	von links nach rechts
11	<< >>	von links nach rechts
10	< <= > >=	von links nach rechts
9	== !=	von links nach rechts
8	&	von links nach rechts
7	^	von links nach rechts
6		von links nach rechts
5	&&	von links nach rechts
4		von links nach rechts
3	?:	von rechts nach links
2	= += -= /= *= %= >>= <<= &= = ~=	von rechts nach links
1		von links nach rechts

Anhang H

Die Diskette zum Buch – Programme



clibtest.c

```
/*  Testprogramm für die Library  */
/*  c 1989 by G. Glendown      */

#include      <exec/types.h>
#include      <exec/libraries.h>
#include      <libraries/dos.h>

#include      "samplebase.h"

struct SampleBase *SampleBase,*OpenLibrary();

void main()
{
    LONG n;
    struct Library *slib;
    if (!(SampleBase=OpenLibrary("sample.library",0L))) {
        printf("Couldn't open...\n");
        exit(RETURN_FAIL);
    }
    slib=&SampleBase->LibNode;
    printf("%s Version %d Revision %d\n",slib->lib_Node.ln_Name,
        slib->lib_Version,slib->lib_Revision);
    n=Double(-7L);
    printf("Double(-7) returned %ld\n",n);
    n=AddThese(21L,4L);
    printf("AddThese(21,4) returned %ld\n",n);
    CloseLibrary(SampleBase);
    exit(RETURN_OK);
}
```

hunk.c

```

/*   DisplayHunk      23.3.1989 G.Glendown */
/* Usage:  hunk File [H] (=Display up to 1024 LWords of data) */

#define MAXBUF  1024
#define CHECK if (t) goto ende

#define PrintF printf

struct FileHandle *fh,*Open();
long l,g,j;
BOOL INFOONLY,PrintNum;

char *hunk[]={
    "unit","name","code","data","bss","reloc32","reloc16","reloc8",
    "ext","symbol","debug","end","header","illegal!","overlay","break"};

char *sym[]={"symb","def","abs","res","ref32","common","ref16","ref8"};

char *mem[]={"","(MEMF_CHIP)","(MEMF_FAST)","(NOT SUPPORTED)"};

long buf[MAXBUF];
UBYTE *bu;

PLen(l)
long l;
{
    int g;
    l<=2;
    PrintF("%ld / $%lx\n",l,l);
}

int Get(g)
long g;
{
    long h,i=0;
    if (g>(MAXBUF<<2))
        i=g-(MAXBUF<<2),
        g=(MAXBUF<<2);
    h=Read(fh,buf,g);
    if (i) Seek(fh,i,OFFSET_CURRENT);
}

```

```

    if (g!=h) return(1);
    else return(0);
}

```

```

Display(l)
long l;
{
    long t;
    int c=0;
    if (INFOONLY) return();
    if (t>(MAXBUF<<2)) t=MAXBUF<<2;
    for (t=0;t<(l>>2);t++) {
        if (c==8) c=0,PrintF("\n");
        PrintF("%08lx ",buf[t]);
        c++;
    }
    if (c!=0) PrintF("\n");
}

```

```

main(argc,argv)
int argc;
char *argv[];
{
    int t,ty,mt,hn;
    bu=(UBYTE *)buf;
    if (argc<2)
        PrintF("Usage: %s Filename [H(unk-Info)]\n",argv[0]),
        exit();
    if (!(fh=Open(argv[1],MODE_OLDFILE)))
        PrintF("File not found: %s\n",argv[1]),
        exit();
    t=0;
    hn=0;
    if (argv[2][0]=='H') INFOONLY=FALSE;
    else INFOONLY=TRUE;
    PrintNum=TRUE;
    do {
        t=Get(4L);
        CHECK;
        mt=(buf[0]>>28);
        buf[0]&=0xfffffff;

```

```
if ((buf[0]<999) || (buf[0]>1014))
    t=1,
    Printf("Illegal Hunk: %ld / $%lx\n",buf[0],buf[0]);
else {
    if (PrintNum) PrintNum=FALSE,Printf("Hunk #d:\n",hn);
    Printf("hunk_ %s:\n",hunk[buf[0]-999]);
    switch(buf[0]) {
        case 999:      /* hunk_unit */
            t=Get(4L);
            CHECK;
            l=buf[0]<<2;
            Printf(" HunkLen: ");
            PLen(buf[0]);
            t=Get(1);
            CHECK;
            Display(1);
            break;
        case 1000:     /* hunk_name */
            t=Get(4L);
            l=buf[0]<<2;
            CHECK;
            t=Get(1);
            CHECK;
            buf[l>>2]=0L;
            Printf(" Name of Hunk: %s\n",buf);
            break;
        case 1001:     /* hunk_code */
        case 1002:     /* hunk_data */
            t=Get(4L);
            CHECK;
            l=buf[0]<<2;
            Printf(" HunkLen: ");
            PLen(buf[0]);
            t=Get(1);
            CHECK;
            Display(1);
            break;
        case 1003:     /* hunk_bss */
            t=Get(4L);
            CHECK;
            Printf(" HunkLen: ");
            PLen(buf[0]);
            break;
```

```

case 1004:      /* hunk_reloc32 */
case 1005:      /* hunk_reloc16 */
case 1006:      /* hunk_reloc8 */
    t=Get(4L);
    while (buf[0]!=0) {
        l=buf[0]<<2;
        t=Get(4L);
        CHECK;
        Printf("    %ld references to hunk %ld\n",l>>2,buf[0]);
        t=Get(1);
        CHECK;
        Display(1);
        t=Get(4L);
        CHECK;
    }
    break;
case 1007:      /* ext */
case 1008:      /* symbol */
    t=Get(4L);
    CHECK;
    while (buf[0]!=0) {
        ty=bu[0];
        if (ty!=0) Printf(" Symbol/Reference type: ");
        if (((ty>4)&&(ty<129))||(ty>132))
            Printf("unknown (%d)\n",ty);
        else {
            if (ty>4) ty-=125;
            if (ty!=0) Printf("ext_%s\n",sym[ty]);
            switch(ty) {
                case 0:      /* symbol */
                case 1:
                case 2:
                case 3:      /* def/abs/res */
                    l=(buf[0]&0xffffffff)<<2;
                    t=Get(1);
                    CHECK;
                    buf[l>>2]=0;
                    Printf(" Symbol name: %s",buf);
                    t=Get(4L);
                    CHECK;
                    Printf(" = %ld / %lx\n",buf[0],buf[0]);
                    t=Get(4L);

```

```
CHECK;
break;
case 4:
case 6:
case 7:    /* ref32/16/8 */
    l=(buf[0]&0xfffff)<<2;
    t=Get(l);
    CHECK;
    buf[l>>2]=0;
    Printf("  Reference name: %s\n",buf);
    t=Get(4L);
    CHECK;
    l=buf[0]<<2;
    t=Get(l);
    CHECK;
    Display(l);
    t=Get(4L);
    CHECK;
    break;
case 5:    /* common */
    l=(buf[0]&0xfffff)<<2;
    t=Get(l);
    CHECK;
    buf[l>>2]=0;
    Printf("  Reference name: %s\n",buf);
    t=Get(8L);
    CHECK;
    Printf("  Size of common block: %ld / $%lx\n",
        buf[0],buf[0]);
    l=buf[1]<<2;
    t=Get(l);
    CHECK;
    Display(l);
    t=Get(4L);
    CHECK;
    break;
}
}
break;
case 1009:    /* debug */
    t=Get(4L);
```

```

CHECK;
l=buf[0]<<2;
t=Get(1);
CHECK;
Display(1);
break;
case 1010:          /* end */
    Printf(" (End of Hunk)\n\n");
    PrintNum=TRUE;
    hn++;
    break;
case 1011:          /* header */
    t=Get(4L);
    CHECK;
    while (t!=0) {
        l=buf[0]<<2;
        t=Get(1);
        CHECK;
        buf[l>>2]=0;
        Printf("    Name: %s\n",buf);
        t=Get(4L);
        CHECK;
    }
    t=Get(12L);
    CHECK;
    l=buf[0];
    j=buf[1];
    g=buf[2];
    Printf(" Number of hunks: %ld\n
           First hunk: %ld\n
           Last hunk: %ld\n
           Sizes of hunks:\n",l,buf[1],buf[2]);
    t=Get(((g-j)+1)<<2);
    CHECK;
    for (ty=j;ty<=g;ty++)
        mt=(buf[ty-j]>>30),
        buf[ty-j]&=0x0fffffff,
        Printf("    Hunk %3d: %8ld / %08lx Bytes %s\n",ty,
               buf[ty-j]<<2,buf[ty-j]<<2,mem[mt]);
    break;

```

```
case 1012:
    Printf("Illegal Hunk: 1012 / $3F5\n");
    t=1;
case 1013:      /* overlay */
    t=Get(4L);
    CHECK;
    l=buf[0];
    Printf(" Overlay table size: %ld / $%lx\n",l,l);
    t=Get(4L);
    CHECK;
    Printf(" Maximum depth of overlay tree: %ld\n",buf[0]);
    t=Get(1<<2);
    CHECK;
    Printf("    Overlay-Table:\n");
    for (t=0;t<l;t+=4) {
        Printf("          %10ld / $%08lx  %10ld / $%08lx\n",
            buf[t],buf[t],buf[t+1],buf[t+1]),
        Printf("          %10ld / $%08lx  %10ld / $%08lx\n",
            buf[t+2],buf[t+2],buf[t+3],buf[t+3]);
    }
    t=0;
    break;
case 1014:      /* break */
    Printf(" (End of overlay node)\n");
    break;
}
}
ende:;
}
while (!t);
Close(fh);
}
```


makefile.

```
# Achtung ! Anstelle von '+ilib:all.pre' den Pfad zu den vorcompilierten  
# Includes (siehe Aztec-Handbuch) einfügen.
```

```
.c.o:  
    cc  $*.c +ijd0:lib/all.pre  
  
all: clibtest sl  
  
clibtest: clibtest.o samplelib.o  
    ln  +q clibtest.o samplelib.o lib:c.lib  
  
sl.o:  
    as sl.asm  
  
sl: sl.o  
    ln +q sl.o  
    copy sl libs:sample.library
```

makefile.hnk

```
# Achtung ! Anstelle von '+ilib:all.pre' den Pfad zu den vorcompilierten  
# Includes (siehe Aztec-Handbuch) einfügen.
```

```
.c.o:  
    cc $*.c +ilib:all.pre  
  
hunk: hunk.o  
    ln +q -o $* $*.o lib:c.lib
```

makefile.win

```
# Achtung ! Anstelle von '+ilib:all.pre' den Pfad zu den vorcompilierten  
# Includes (siehe Aztec-Handbuch) einfügen.
```

```
.C.o:  
    cc $*.c +ilib:all.pre  
  
win:  win.o  
    ln +q -o $* $*.o lib:c.lib
```

mkassl.v

```
' ConvertFDAssInc (C) 1989 by G. Glendown

' path$:   Pfad zu den FD-Files (z.B.: "df0:fd/" oder "FD:")
' lb$:     Name der Library, ohne irgendwas (also z.B.: clist)
' lbname$: Name der Library (für Überschrift)
' lbf$:    Name des FD-Files,      Kanal: 1
' lbout$:  Name des Ausgabe-Files, Kanal: 2
' iline$:  Eingabe-Zeile vom FD-File
' oline$:  Ausgabe-Zeile fr Include-File
' CurOff:  Aktueller Offset in der Library

INPUT "Wo stehen die FD-Files (mit '/' bzw. ':'):";path$
INPUT "Welche Library:";lb$
INPUT "Name der Library (Return=Library):";lbname$
IF lbname$="" THEN lbname$=lb$
lbf$=path$+lb$+"_lib.fd"
lbout$="ram:"+lb$+"_lib.i"
OPEN lbf$ FOR INPUT AS #1
OPEN lbout$ FOR OUTPUT AS #2
CurOff=0

PRINT#2," IFND ";UCASE$(lbname$);"_LIB_I"
PRINT#2,UCASE$(lbname$);"_LIB_I SET 1"

WHILE ((NOT(EOF(1))) AND (CurOff<>1))

    GOSUB ReadALine
    GOSUB ParseLine
    GOSUB OutputLine

WEND

PRINT#2," ENDC !";UCASE$(lbname$);"_LIB_I"
CLOSE #2
CLOSE #1
END ' das war's
```

```

ReadALine:
  iline$=""
  WHILE iline$=""
    IF EOF(1) THEN
      iline$="##THEEND"
      GOTO NoRead
    ELSE
      LINE INPUT#1,iline$
    END IF
  END IF
NoRead:
  WEND
  IF LEFT$(iline$,1)="*" THEN GOTO ReadALine

  RETURN

ParseLine:
  oline$=""
  IF LEFT$(iline$,2)="##" THEN
    IF UCASE$(MID$(iline$,3,5))="BIAS " THEN
      IF CurOff=0 THEN
        CurOff=-VAL(MID$(iline$,8))
      ELSE
        PRINT "Library-Offset doppelt definiert!"
      END IF
    ELSE
      IF MID$(iline$,3,6)="THEEND" THEN
        CurOff=1
      END IF
    END IF
  ELSE
    oline$="_LVO"
    g=-1
    FOR t=1 TO LEN(iline$)
      IF MID$(iline$,t,1)="(" THEN g=t:t=255
    NEXT t
    IF (g>0) THEN
      oline$=oline$+LEFT$(iline$,g-1)+" EQU "+STR$(CurOff)
      CurOff=CurOff-6
    ELSE
      PRINT "Illegales Format!"
      GOTO EndScan
    END IF
  END IF
END IF

```

EndScan:

 RETURN

OutputLine:

 IF oline\$<>" THEN

 PRINT#2,oline\$

 END IF

 RETURN

sampleba.seh

```
/* SampleBase.h
   Include-File für C-Programme */
/* (C) 1989 by G. Glendown */

#ifndef SAMPLE_BASE_H
#define SAMPLE_BASE_H

#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif EXEC_TYPES_H

#ifndef EXEC_LISTS_H
#include <exec/lists.h>
#endif EXEC_LISTS_H

#ifndef EXEC_LIBRARIES_H
#include <exec/libraries.h>
#endif EXEC_LIBRARIES_H

struct SampleBase {
    struct Library LibNode;
    UBYTE Flags;
    UBYTE pad;
    ULONG SysLib;
    ULONG DosLib;
    ULONG SegList;
};

#define SAMPLENAME    "sample.library"

#endif EXEC_SAMPLEBASE_H
```

sampleba.sei

```
; SampleBase.i
; Include-File für Assembler-Programme bzw. die Library
; (C) 1989 by G. Glendown

        IFND      SAMPLE_BASE_I
SAMPLE_BASE_I    SET        1

        IFND EXEC_TYPES_I
        INCLUDE "exec/types.i"
        ENDC

        IFND EXEC_LISTS_I
        INCLUDE "exec/lists.i"
        ENDC

        IFND EXEC_LIBRARIES_I
        INCLUDE "exec/libraries.i"
        ENDC

        STRUCTURE SampleBase, LIB_SIZE
        UBYTE    sb_Flags
        UBYTE    sb_pad
        ULONG    sb_SysLib
        ULONG    sb_DosLib
        ULONG    sb_SegList
        LABEL    SampleBase_SIZEOF

SAMPLENAME    MACRO
        DC.B    'sample.library', 0
        ENDM

        ENDC
```

samplelb.asm

```
; SampleLib.asm
; Code zum Einbinden mit C/Ass-Programmen
; (C) 1989 by G. Glendown
```

```
INCLUDE "exec/types.i"
INCLUDE "exec/libraries.i"
```

```
SECTION _LVO
DSEG
LIBINIT
LIBDEF _LVODouble
XDEF _LVODouble
LIBDEF _LVOAddThese
XDEF _LVOAddThese
```

```
SECTION CSTUB
CSEG
```

```
XREF _SampleBase
```

```
; XREF _LVODouble
; XREF _LVOAddThese
```

```
XDEF _Double
XDEF _AddThese
```

```
_Double:
```

```
MOVE.l a6,-(sp)
move.l 8(sp),d0
move.l _SampleBase,a6
jsr _LVODouble(a6)
move.l (sp)+,a6
rts
```

```
_AddThese:
```

```
move.l a6,-(sp)
movem.l 8(sp),d0/d1
move.l _SampleBase,a6
jsr _LVOAddThese(a6)
move.l (sp)+,a6
rts
END
```


sl.asm

```
; sl.asm
; Source der samplelib.library
; (C) 1989 by G. Glendown

SECTION Section
NOLIST
include "exec/types.i"
include "exec/libraries.i"
include "exec/lists.i"
include "exec/alerts.i"
include "exec/initializers.i"
include "exec/resident.i"
include "libraries/dos.i"

include "samplebase.i"

LIST

XDEF      Init
XDEF      Open
XDEF      Close
XDEF      Expunge
XDEF      Null
XDEF      sampleName
XDEF      Double
XDEF      AddThese

include "libvecs/exec_lib.i"

Start:
    moveq   #-1,d0
    rts

MYPRI     EQU      0

initDDDescrip:
    dc.w    RTC_MATCHWORD
    DC.l    initDDDescrip
    DC.L    EndCode
```

```

DC.B    RTF_AUTOINIT
DC.B    VERSION
DC.B    NT_LIBRARY
DC.b    MYPRI
DC.L    sampleName
DC.L    idString
DC.L    Init

sampleName:    SAMPLENAME

VERSION:      EQU    34
REVISION:     EQU    1

idString:     dc.b    'samplelib 34.1 (9. Aug 89)',13,10,0

dosName:      DOSNAME

ds.w    0

Init:
dc.l    SampleBase_SIZEOF
dc.l    funcTable
dc.l    dataTable
dc.l    initRoutine

funcTable:

dc.l    Open
dc.l    Close
dc.l    Expunge
dc.l    Null

dc.l    Double
dc.l    AddThese

dc.l    -1

dataTable:
INITBYTE    LN_TYPE,NT_LIBRARY
INITLONG    LN_NAME,sampleName
INITBYTE    LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
INITWORD    LIB_VERSION,VERSION

```

```

INITWORD      LIB_REVISION,REVISION
INITLONG      LIB_IDSTRING,idString
dc.l         0

```

```
initRoutine:
```

```

    move.l    a5,-(sp)
    move.l    d0,a5

    move.l    a6,sb_SysLib(a5)

    move.l    a0,sb_SegList(a5)

    lea       dosName(pc),a1
    moveq     #0,d0
    CALLLIB   OpenLibrary

    move.l    d0,sb_DosLib(a5)
    bne       1$

    ALERT     AG_OpenLib!AO_DOSLib

```

```
1$:
```

```

;
    move.l    a5,d0
    move.l    (sp)+,a5
    rts

```

```
Open:
```

```

    addq.w    #1,LIB_OPENCNT(a6)
    bclr      #LIBB_DELEXP,sb_Flags(a6)
    move.l    a6,d0
    rts

```

```
Close:
```

```

    moveq     #0,d0
    subq.w    #1,LIB_OPENCNT(a6)
    bne.s     1$
    btst      #LIBB_DELEXP,sb_Flags(a6)
    beq.s     1$
    bsr       Expunge

```

```
1$:
```

```
    rts

```

Expunge:

```
movem.l d2/a5/a6, -(sp)
move.l  a6, a5
move.l  sb_SysLib(a5), a6

tst.w   LIB_OPENCNT(a5)
beq.s   1$

bset    #LIBB_DELEXP, sb_Flags(a5)
moveq   #0, d0
bra.s   Expunge_End
```

1\$:

```
move.l  sb_SegList(a5), d2
move.l  a5, a1
CALLLIB Remove

move.l  sb_DosLib(a5), a1
CALLLIB CloseLibrary

moveq   #0, d0
move.l  a5, a1
move.w  LIB_NEGSIZE(a5), d0

sub.l   d0, a1
add.w   LIB_POSSIZE(a5), d0

CALLLIB FreeMem
move.l  d2, d0
```

Expunge_End:

```
movem.l (sp)+, d2/a5/a6
rts
```

Null:

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

Double:

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

AddThese:

```
add.l    d1,d0
rts
```

EndCode:

```
END
```

win.c

```

#include "win.h"

#define RefGad Gadget50
#define Printf printf
#define SPrintf sprintf

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
VOID *OpenLibrary();

struct Window *win,*OpenWindow(),*win2;

/* Verwaltung der REFRESH-Flag: */
int rf;      /* Zähler von 0 bis 2 */
UBYTE reftxt[15]={SMART_REFRESH,"SIMPLE_REFRESH","SUPER_BITMAP"};
/* Texte für's Gadget */
ULONG refflg[]={SMART_REFRESH,SIMPLE_REFRESH,SUPER_BITMAP};

struct IntuiMessage *msg,*GetMsg(),*msg2;

char titel[81]={"ÓÓÓ Test-Fenster ! %44\0"}; /* Titel des Fensters */

struct NewWindow window = { /* Struktur für Demo-Window */
    0,0,
    200,100,
    0,1,
    0L,
    0L,
    0L,
    NULL,
    (UBYTE *)titel,
    NULL,
    NULL,
    40,20,
    640,256,
    WBENCHSCREEN};

char *IDCMP[]={ /* IDCMP-Message */
    "SIZEVERIFY","NEWSIZE","REFRESHWINDOW","MOUSEBUTTONS",
    "MOUSEMOVE","GADGETDOWN","GADGETUP","REQSET",
    "MENUPIK","CLOSEWINDOW","RAWKEY","REQVERIFY",

```

```

"REQCLEAR", "MENUVERIFY", "NEWPREFS", "DISKINSERTED",
"DISKREMOVED", "WBENCHMESSAGE", "ACTIVEWINDOW", "INACTIVEWINDOW",
"DELTAMOVE", "VANILLAKEYS", "INTUITICKS"};

char *NFlags[]={ /* Flags */
    "WINDOWSIZING", "WINDOWDRAG", "WINDOWDEPTH", "WINDOWCLOSE",
    "SIZEBRIGHT", "SIZEBBOTTOM", "SIMPLE_REFRESH", "SUPER_BITMAP",
    "BACKDROP", "REPORTMOUSE", "GIMMEZEROZERO", "BORDERLESS",
    "ACTIVATE",
    "WINDOWACTIVE", /* werden von */
    "INREQUEST", /* Intuition */
    "MENUSTATE", /* gesetzt ! */
    "RMBTRAP", "NOCAREREFRESH"};

struct Gadget *gad;
int id;
long fl, flag[3];
long class;
struct StringInfo *si;
int v;
char *s, *d;
char mes[]={ "Intui-Message vom Fenster: "};
char pmes[100];
struct RastPort *rp;

char FileName[]={ "ram:WindowA.h"};
#define CPos 10
struct FileHandle *fh, *Open();
long Write(), l;
char mesg[]={ "/***** Window-Source *****/\n\nstruct NewWindow windef ={\n"};
char out[256];

main()
{
    int t;

    rf=0; /* Refresh-Counter */
    if (!(IntuitionBase=OpenLibrary("intuition.library", 0L))
        Printf("Hilfe! Intuition!\n"), ex());
    if (!(GfxBase=OpenLibrary("graphics.library", 0L))
        Printf("Hilfe! Graphics!\n"), ex());
    if (!(win=OpenWindow(&windef)))
        Printf("Kein Speicher für Fenster!\n"), ex());

```

```

rp=win->RPort;
SetDrMd(rp, JAM2);
SetRefresh();
for (t=0;t==0;) {
    Wait((1L<<win->UserPort->mp_SigBit)|(1L<<win2->UserPort->mp_SigBit));

    msg=GetMsg(win->UserPort);
    if (win2&&win2->UserPort) msg2=GetMsg(win2->UserPort);
    while ((long)msg|(long)msg2) {
        if (msg) {
            class=msg->Class;
            switch (class) {
                case GADGETUP:
                    gad=(struct Gadget *)msg->IAddress;
                    id=gad->GadgetID;
                    fl=(long)gad->UserData;
                    si=(struct StringInfo *)gad->SpecialInfo;
                    /* Da Werte nur gelesen werden */
                    v=(int)(si->LongInt); /* kann es ohne Prüfung verw. */
                    switch (id) { /* werden... */
                        case 20: /* LeftEdge */
                            window.LeftEdge=v;
                            break;
                        case 21: /* TopEdge */
                            window.TopEdge=v;
                            break;
                        case 22: /* Width */
                            window.Width=v;
                            break;
                        case 23: /* Height */
                            window.Height=v;
                            break;
                        case 24: /* DPen */
                            window.DetailPen=v;
                            break;
                        case 25: /* BPen */
                            window.BlockPen=v;
                            break;
                        case 26: /* Title */
                            s=(char *)si->Buffer;
                            d=titel;
                            while (*d++=*s++);
                            break;

```



```

case 27:    /* MinWidth */
    window.MinWidth=v;
    break;
case 28:    /* MinHeight */
    window.MinHeight=v;
    break;
case 29:    /* MaxWidth */
    window.MaxWidth=v;
    break;
case 30:    /* MaxHeight */
    window.MaxHeight=v;
    break;
case 40:    /* OpenWindow */
case 98:    /* CreateSource */
    if (win2) CloseWindow(win2);
    gad=win->FirstGadget;
    SetAPen(rp,0L);
    RectFill(rp,5L,180L,450L,195L);
    flag[1]=0;
    flag[2]=0;
    while (gad) {
        if ((gad->GadgetID==1) || (gad->GadgetID==2)) {
            if (gad->Flags&SELECTED)
                flag[gad->GadgetID] |= (ULONG)
                    (gad->UserData);
        }
        gad=gad->NextGadget;
    }
    window.IDCMPFlags=flag[1];
    window.Flags=flag[2] | refflg[rf];
    win2=OpenWindow(&window);
    if (id==98) {    /* CreateSource ? */
        if (!(fh=Open(FileName,MODE_NEWFILE)))
            PrintLine("Couldn't open file!");
        else {
            Write(fh,mesg,(long)strlen(mesg));
            SPrintf(out,"%d,%d,%d,%d,\n\0",
                window.LeftEdge,window.TopEdge,
                window.Width,window.Height);
            Write(fh,out,(long)strlen(out));
        }
    }

```

```

Sprintf(out, "%d,%d,\n\0",
        window.DetailPen, window.BlockPen);
Write(fh, out, (long) strlen(out));
out[0]='\0';
if (window.IDCMPFlags==NULL)
    Write(fh, "NULL,\n", 6L);
else {
    l=window.IDCMPFlags;
    for (t=0;t<24;t++) {
        if (l&(long)(1L<<t)) {
            if (strlen(out)>0)
                strcat(out, "|");
            strcat(out, IDCMP[t]);
        }
    }
    Write(fh, out, (long) strlen(out));
    Write(fh, "\n", 2L);
}
out[0]='\0';
if (window.Flags==NULL)
    Write(fh, "NULL,\n", 6L);
else {
    l=window.Flags;
    for (t=0;t<18;t++) {
        if (l&(long)(1L<<t)) {
            if (strlen(out)>0)
                strcat(out, "|");
            strcat(out, NFlags[t]);
        }
    }
    Write(fh, out, (long) strlen(out));
    Write(fh, "\n", 2L);
}
Write(fh, "NULL,NULL,\n(UBYTE *)\n", 21L);
Write(fh, titel, (long) strlen(titel));
Write(fh, "\n\nNULL,NULL,\n", 14L);
Sprintf(out, "%d,%d,%d,%d,
        WBSCHSCREEN);\n",
        window.MinWidth, window.MinHeight,
        window.MaxWidth, window.MaxHeight);
l=Write(fh, out, (long) strlen(out));
Close(fh);

```

```

        if (l!=(long)strlen(out))
            PrintLine("Error writing file!");
        else FileName[CPos]++;
    }
    t=0;
}
break;
case 99: /* RefreshMode */
    rf++;
    if (rf==3) rf=0;
    SetRefresh();
    break;
}
}
ReplyMsg(msg);
if (class==CLOSEWINDOW) ex();
}
if (msg2) {
    class=msg2->Class;
    for (t=0;t<24;t++) {
        if (class&(long)(1L<<t)) {
            strcpy(pmes,mes);
            strcat(pmes,IDCMP[t]);
            PrintLine(pmes);
        }
    }
    ReplyMsg(msg2);
    t=0;
}
msg=GetMsg(win->UserPort);
if (win2&&win2->UserPort) msg2=GetMsg(win2->UserPort);
}
}

PrintLine(ln)
char *ln;
{
    SetAPen(rp,0L);
    RectFill(rp,5L,180L,450L,195L);
    SetAPen(rp,1L);
    Move(rp,10L,190L);
    Text(rp,ln,(long)strlen(ln));
}

```

```
ex()
{
    if (win2) CloseWindow(win2);
    if (win) CloseWindow(win);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    exit();
}
```

```
SetRefresh()
{
    IText1.IText=reftxt[rf];
    RefreshGList(&RefGad,win,0L,1L);
}
```

win.h

```

UBYTE UNDOBUFFER[81];
SHORT BorderVectors1[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border1 = {-2,-1,1,0,JAM1,5,BorderVectors1,NULL};

struct TextAttr TOPAZ80 = {
    (STRPTR)"topaz.font",
    TOPAZ_EIGHTY,0,0
};

struct IntuiText IText2 = {1,0,JAM2,-71,1,&TOPAZ80,
    (UBYTE *)"Refresh:",NULL};

struct IntuiText IText1 = {1,0,JAM2,5,1,&TOPAZ80,
    NULL,/*(UBYTE *)"SMART_REFRESH",*/&IText2};

struct Gadget Gadget50 = {NULL,386,101,120,9,NULL,
    RELVERIFY,BOOLGADGET,(APTR)&Border1,NULL,&IText1,
    NULL,NULL,99,NULL};

SHORT BorderVectors2[] = {0,0,161,0,161,9,0,9,0,0};
struct Border Border2 = {-1,-1,1,0,JAM1,5,BorderVectors2,NULL};

struct IntuiText IText3 = {1,0,JAM2,15,0,&TOPAZ80,
    (UBYTE *)"Source erstellen",NULL};

struct Gadget Gadget49 = {&Gadget50,463,184,160,8,NULL,
    RELVERIFY,BOOLGADGET,(APTR)&Border2,NULL,&IText3,
    NULL,NULL,98,NULL};

SHORT BorderVectors3[] = {0,0,92,0,92,66,0,66,0,0};
struct Border Border3 = {-1,-1,1,0,JAM1,5,BorderVectors3,NULL};

struct IntuiText IText5 = {1,0,JAM2,6,38,&TOPAZ80,
    (UBYTE *)"erstellen!",NULL};

struct IntuiText IText4 = {1,0,JAM2,18,14,&TOPAZ80,
    (UBYTE *)"Fenster",&IText5};

struct Gadget Gadget48 = {&Gadget49,533,94,91,65,GADGHBOX,
    RELVERIFY,BOOLGADGET,(APTR)&Border3,NULL,&IText4,
    NULL,NULL,40,NULL};

```

```

UBYTE Gadget47SIBuff[5] =
    "256";
struct StringInfo Gadget47SInfo = {Gadget47SIBuff, NULL,
    0, 5, 0, 0, 0, 0, 0, 0, 0, 0, NULL};

SHORT BorderVectors4[] = {0, 0, 57, 0, 57, 9, 0, 9, 0, 0};
struct Border Border4 = {-1, -1, 1, 0, JAM2, 5, BorderVectors4, NULL};

struct Gadget Gadget47 = {&Gadget48, 463, 168, 56, 8, NULL,
    RELVERIFY|STRINGCENTER|LONGINT, STRGADGET, (APTR)&Border4, NULL, NULL,
    NULL, (APTR)&Gadget47SInfo, 30, NULL};

UBYTE Gadget46SIBuff[5] =
    "640";
struct StringInfo Gadget46SInfo = {Gadget46SIBuff, NULL,
    0, 5, 0, 0, 0, 0, 0, 0, 0, 0, NULL};

SHORT BorderVectors5[] = {0, 0, 57, 0, 57, 9, 0, 9, 0, 0};
struct Border Border5 = {-1, -1, 1, 0, JAM2, 5, BorderVectors5, NULL};

struct IntuiText IText8 = {1, 0, JAM2, 128, 0, &TOPAZ80,
    (UBYTE *) " ", NULL};

struct IntuiText IText7 = {1, 0, JAM2, 59, 1, &TOPAZ80,
    (UBYTE *) " ", &IText8};

struct IntuiText IText6 = {1, 0, JAM2, -108, 0, &TOPAZ80,
    (UBYTE *) "Gr  e Max.: ", &IText7};

struct Gadget Gadget46 = {&Gadget47, 394, 168, 56, 8, NULL,
    RELVERIFY|STRINGCENTER|LONGINT, STRGADGET, (APTR)&Border5, NULL, &IText6,
    NULL, (APTR)&Gadget46SInfo, 29, NULL};

UBYTE Gadget45SIBuff[5] =
    "20";
struct StringInfo Gadget45SInfo = {Gadget45SIBuff, NULL,
    0, 5, 0, 0, 0, 0, 0, 0, 0, 0, NULL};

SHORT BorderVectors6[] = {0, 0, 57, 0, 57, 9, 0, 9, 0, 0};
struct Border Border6 = {-1, -1, 1, 0, JAM2, 5, BorderVectors6, NULL};

```

```

struct Gadget Gadget45 = {&Gadget46,182,168,56,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border6,NULL,NULL,
    NULL,(APTR)&Gadget45SInfo,28,NULL};

UBYTE Gadget44SIBuff[5] =
    "40";
struct StringInfo Gadget44SInfo = {Gadget44SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors7[] = {0,0,57,0,57,9,0,9,0,0};
struct Border Border7 = {-1,-1,1,0,JAM2,5,BorderVectors7,NULL};

struct IntuiText IText11 = {1,0,JAM2,128,0,&TOPAZ80,
    (UBYTE *)" ",NULL};

struct IntuiText IText10 = {1,0,JAM2,59,1,&TOPAZ80,
    (UBYTE *)" ",&IText11};

struct IntuiText IText9 = {1,0,JAM2,-108,0,&TOPAZ80,
    (UBYTE *)"Gr e Min.: ",&IText10};

struct Gadget Gadget44 = {&Gadget45,113,168,56,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border7,NULL,&IText9,
    NULL,(APTR)&Gadget44SInfo,27,NULL};

UBYTE Gadget43SIBuff[81] =
    "ÓÓÓ Test-Fenster ! ¼¼¼";
struct StringInfo Gadget43SInfo = {Gadget43SIBuff,UNDOBUFFER,
    0,81,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors8[] = {0,0,270,0,270,9,0,9,0,0};
struct Border Border8 = {-1,-1,1,0,JAM1,5,BorderVectors8,NULL};

struct IntuiText IText12 = {1,0,JAM2,-59,0,&TOPAZ80,
    (UBYTE *)"Titel:",NULL};

struct Gadget Gadget43 = {&Gadget44,68,154,269,8,NULL,
    RELVERIFY|STRINGCENTER,STRGADGET,(APTR)&Border8,NULL,&IText12,
    NULL,(APTR)&Gadget43SInfo,26,NULL};

SHORT BorderVectors9[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border9 = {-2,-1,3,0,JAM1,5,BorderVectors9,NULL};

```

```
struct IntuiText IText13 = {3,0,JAM2,13,1,&TOPAZ80,
    (UBYTE *)"WINDOWSIZING",NULL};

struct Gadget Gadget42 = {&Gadget43,386,140,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border9,NULL,&IText13,
    NULL,NULL,2,WINDOWSIZING};

SHORT BorderVectors10[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border10 = {-2,-1,3,0,JAM1,5,BorderVectors10,NULL};

struct IntuiText IText14 = {3,0,JAM2,21,1,&TOPAZ80,
    (UBYTE *)"WINDOWDRAG",NULL};

struct Gadget Gadget41 = {&Gadget42,260,140,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border10,NULL,&IText14,
    NULL,NULL,2,WINDOWDRAG};

SHORT BorderVectors11[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border11 = {-2,-1,3,0,JAM1,5,BorderVectors11,NULL};

struct IntuiText IText15 = {3,0,JAM2,17,1,&TOPAZ80,
    (UBYTE *)"WINDOWDEPTH",NULL};

struct Gadget Gadget40 = {&Gadget41,134,140,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border11,NULL,&IText15,
    NULL,NULL,2,WINDOWDEPTH};

SHORT BorderVectors12[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border12 = {-2,-1,3,0,JAM1,5,BorderVectors12,NULL};

struct IntuiText IText16 = {3,0,JAM2,17,1,&TOPAZ80,
    (UBYTE *)"WINDOWCLOSE",NULL};

struct Gadget Gadget39 = {&Gadget40,8,140,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border12,NULL,&IText16,
    NULL,NULL,2,WINDOWCLOSE};

SHORT BorderVectors13[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border13 = {-2,-1,3,0,JAM1,5,BorderVectors13,NULL};

struct IntuiText IText17 = {3,0,JAM2,21,1,&TOPAZ80,
    (UBYTE *)"SIZEBRIGHT",NULL};
```



```

struct Gadget Gadget38 = {&Gadget39,386,128,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border13,NULL,&IText17,
    NULL,NULL,2,SIZEBRIGHT};

SHORT BorderVectors14[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border14 = {-2,-1,3,0,JAM1,5,BorderVectors14,NULL};

struct IntuiText IText18 = {3,0,JAM2,16,1,&TOPAZ80,
    (UBYTE *)"SIZEBBOTTOM",NULL};

struct Gadget Gadget37 = {&Gadget38,260,128,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border14,NULL,&IText18,
    NULL,NULL,2,SIZEBBOTTOM};

SHORT BorderVectors15[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border15 = {-2,-1,3,0,JAM1,5,BorderVectors15,NULL};

struct IntuiText IText19 = {3,0,JAM2,33,1,&TOPAZ80,
    (UBYTE *)"RMBTRAP",NULL};

struct Gadget Gadget36 = {&Gadget37,134,128,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border15,NULL,&IText19,
    NULL,NULL,2,RMBTRAP};

SHORT BorderVectors16[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border16 = {-2,-1,3,0,JAM1,5,BorderVectors16,NULL};

struct IntuiText IText20 = {3,0,JAM2,16,1,&TOPAZ80,
    (UBYTE *)"REPORTMOUSE",NULL};

struct Gadget Gadget35 = {&Gadget36,8,128,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border16,NULL,&IText20,
    NULL,NULL,2,REPORTMOUSE};

SHORT BorderVectors17[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border17 = {-2,-1,3,0,JAM1,5,BorderVectors17,NULL};

struct IntuiText IText21 = {3,0,JAM2,9,1,&TOPAZ80,
    (UBYTE *)"NOCAREREFRESH",NULL};

struct Gadget Gadget34 = {&Gadget35,260,116,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border17,NULL,&IText21,
    NULL,NULL,2,NOCAREREFRESH};

SHORT BorderVectors18[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border18 = {-2,-1,3,0,JAM1,5,BorderVectors18,NULL};

```

```
struct IntuiText IText22 = {3,0,JAM2,9,1,&TOPAZ80,
    (UBYTE *) "GIMMEZEROZERO",NULL};

struct Gadget Gadget33 = {&Gadget34,386,116,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR) &Border18,NULL,&IText22,
    NULL,NULL,2,GIMMEZEROZERO};

SHORT BorderVectors19[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border19 = {-2,-1,3,0,JAM1,5,BorderVectors19,NULL};

struct IntuiText IText23 = {3,0,JAM2,20,1,&TOPAZ80,
    (UBYTE *) "BORDERLESS",NULL};

struct Gadget Gadget32 = {&Gadget33,386,152,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR) &Border19,NULL,&IText23,
    NULL,NULL,2,BORDERLESS};

SHORT BorderVectors20[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border20 = {-2,-1,3,0,JAM1,5,BorderVectors20,NULL};

struct IntuiText IText24 = {3,0,JAM2,28,1,&TOPAZ80,
    (UBYTE *) "BACKDROP",NULL};

struct Gadget Gadget31 = {&Gadget32,134,116,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR) &Border20,NULL,&IText24,
    NULL,NULL,2,BACKDROP};

SHORT BorderVectors21[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border21 = {-2,-1,3,0,JAM1,5,BorderVectors21,NULL};

struct IntuiText IText26 = {1,0,JAM2,-1,-10,&TOPAZ80,
    (UBYTE *) "Flags:",NULL};

struct IntuiText IText25 = {3,0,JAM2,28,1,&TOPAZ80,
    (UBYTE *) "ACTIVATE",&IText26};

struct Gadget Gadget30 = {&Gadget31,8,116,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR) &Border21,NULL,&IText25,
    NULL,NULL,2,ACTIVATE};

SHORT BorderVectors22[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border22 = {-2,-1,3,0,JAM1,5,BorderVectors22,NULL};
```

```

struct IntuiText IText27 = {3,0,JAM2,9,1,&TOPAZ80,
    (UBYTE *) "WBENCHMESSAGE",NULL};

struct Gadget Gadget29 = {&Gadget30,260,80,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border22,NULL,&IText27,
    NULL,NULL,1,WBENCHMESSAGE};

SHORT BorderVectors23[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border23 = {-2,-1,3,0,JAM1,5,BorderVectors23,NULL};

struct IntuiText IText28 = {3,0,JAM2,20,1,&TOPAZ80,
    (UBYTE *) "VANILLAKEY",NULL};

struct Gadget Gadget28 = {&Gadget29,134,80,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border23,NULL,&IText28,
    NULL,NULL,1,VANILLAKEY};

SHORT BorderVectors24[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border24 = {-2,-1,3,0,JAM1,5,BorderVectors24,NULL};

struct IntuiText IText29 = {3,0,JAM2,32,1,&TOPAZ80,
    (UBYTE *) "NEWSIZE",NULL};

struct Gadget Gadget27 = {&Gadget28,8,80,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border24,NULL,&IText29,
    NULL,NULL,1,NEWSIZE};

SHORT BorderVectors25[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border25 = {-2,-1,3,0,JAM1,5,BorderVectors25,NULL};

struct IntuiText IText30 = {3,0,JAM2,20,1,&TOPAZ80,
    (UBYTE *) "SIZEVERIFY",NULL};

struct Gadget Gadget26 = {&Gadget27,512,68,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border25,NULL,&IText30,
    NULL,NULL,1,SIZEVERIFY};

SHORT BorderVectors26[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border26 = {-2,-1,3,0,JAM1,5,BorderVectors26,NULL};

struct IntuiText IText31 = {3,0,JAM2,25,1,&TOPAZ80,
    (UBYTE *) "REQVERIFY",NULL};

```

```
struct Gadget Gadget25 = {&Gadget26, 386, 68, 120, 9, NULL,
    TOGGLESELECT, BOOLGADGET, (APTR) &Border26, NULL, &IText31,
    NULL, NULL, 1, REQVERIFY};

SHORT BorderVectors27[] = {0, 0, 123, 0, 123, 10, 0, 10, 0, 0};
struct Border Border27 = {-2, -1, 3, 0, JAM1, 5, BorderVectors27, NULL};

struct IntuiText IText32 = {3, 0, JAM2, 37, 1, &TOPAZ80,
    (UBYTE *) "REQSET", NULL};

struct Gadget Gadget24 = {&Gadget25, 260, 68, 120, 9, NULL,
    TOGGLESELECT, BOOLGADGET, (APTR) &Border27, NULL, &IText32,
    NULL, NULL, 1, REQSET};

SHORT BorderVectors28[] = {0, 0, 123, 0, 123, 10, 0, 10, 0, 0};
struct Border Border28 = {-2, -1, 3, 0, JAM1, 5, BorderVectors28, NULL};

struct IntuiText IText33 = {3, 0, JAM2, 27, 1, &TOPAZ80,
    (UBYTE *) "REQCLEAR", NULL};

struct Gadget Gadget23 = {&Gadget24, 134, 68, 120, 9, NULL,
    TOGGLESELECT, BOOLGADGET, (APTR) &Border28, NULL, &IText33,
    NULL, NULL, 1, REQCLEAR};

SHORT BorderVectors29[] = {0, 0, 123, 0, 123, 10, 0, 10, 0, 0};
struct Border Border29 = {-2, -1, 3, 0, JAM1, 5, BorderVectors29, NULL};

struct IntuiText IText34 = {3, 0, JAM2, 9, 1, &TOPAZ80,
    (UBYTE *) "REFRESHWINDOW", NULL};

struct Gadget Gadget22 = {&Gadget23, 8, 68, 120, 9, NULL,
    TOGGLESELECT, BOOLGADGET, (APTR) &Border29, NULL, &IText34,
    NULL, NULL, 1, REFRESHWINDOW};

SHORT BorderVectors30[] = {0, 0, 123, 0, 123, 10, 0, 10, 0, 0};
struct Border Border30 = {-2, -1, 3, 0, JAM1, 5, BorderVectors30, NULL};

struct IntuiText IText35 = {3, 0, JAM2, 37, 1, &TOPAZ80,
    (UBYTE *) "RAWKEY", NULL};

struct Gadget Gadget21 = {&Gadget22, 512, 56, 120, 9, NULL,
    TOGGLESELECT, BOOLGADGET, (APTR) &Border30, NULL, &IText35,
    NULL, NULL, 1, RAWKEY};
```

```

SHORT BorderVectors31[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border31 = {-2,-1,3,0,JAM1,5,BorderVectors31,NULL};

struct IntuiText IText36 = {3,0,JAM2,28,1,&TOPAZ80,
    (UBYTE *)"NEWPREFS",NULL};

struct Gadget Gadget20 = {&Gadget21,386,56,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border31,NULL,&IText36,
    NULL,NULL,1,NEWPREFS};

SHORT BorderVectors32[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border32 = {-2,-1,3,0,JAM1,5,BorderVectors32,NULL};

struct IntuiText IText37 = {3,0,JAM2,26,1,&TOPAZ80,
    (UBYTE *)"MOUSEMOVE",NULL};

struct Gadget Gadget19 = {&Gadget20,260,56,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border32,NULL,&IText37,
    NULL,NULL,1,MOUSEMOVE};

SHORT BorderVectors33[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border33 = {-2,-1,3,0,JAM1,5,BorderVectors33,NULL};

struct IntuiText IText38 = {3,0,JAM2,14,1,&TOPAZ80,
    (UBYTE *)"MOUSEBUTTONS",NULL};

struct Gadget Gadget18 = {&Gadget19,134,56,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border33,NULL,&IText38,
    NULL,NULL,1,MOUSEBUTTONS};

SHORT BorderVectors34[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border34 = {-2,-1,3,0,JAM1,5,BorderVectors34,NULL};

struct IntuiText IText39 = {3,0,JAM2,20,1,&TOPAZ80,
    (UBYTE *)"MENUVERIFY",NULL};

struct Gadget Gadget17 = {&Gadget18,8,56,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border34,NULL,&IText39,
    NULL,NULL,1,MENUVERIFY};

SHORT BorderVectors35[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border35 = {-2,-1,3,0,JAM1,5,BorderVectors35,NULL};

```

```
struct IntuiText IText40 = {3,0,JAM2,29,1,&TOPAZ80,
    (UBYTE *) "MENUPICK",NULL};

struct Gadget Gadget16 = {&Gadget17,512,44,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border35,NULL,&IText40,
    NULL,NULL,1,MENUPICK};

SHORT BorderVectors36[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border36 = {-2,-1,3,0,JAM1,5,BorderVectors36,NULL};

struct IntuiText IText41 = {3,0,JAM2,19,1,&TOPAZ80,
    (UBYTE *) "INTUITICKS",NULL};

struct Gadget Gadget15 = {&Gadget16,386,44,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border36,NULL,&IText41,
    NULL,NULL,1,INTUITICKS};

SHORT BorderVectors37[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border37 = {-2,-1,3,0,JAM1,5,BorderVectors37,NULL};

struct IntuiText IText42 = {3,0,JAM2,4,1,&TOPAZ80,
    (UBYTE *) "INACTIVIEWINDOW",NULL};

struct Gadget Gadget14 = {&Gadget15,260,44,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border37,NULL,&IText42,
    NULL,NULL,1,INACTIVIEWINDOW};

SHORT BorderVectors38[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border38 = {-2,-1,3,0,JAM1,5,BorderVectors38,NULL};

struct IntuiText IText43 = {3,0,JAM2,21,1,&TOPAZ80,
    (UBYTE *) "GADGETDOWN",NULL};

struct Gadget Gadget13 = {&Gadget14,134,44,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border38,NULL,&IText43,
    NULL,NULL,1,GADGETDOWN};

SHORT BorderVectors39[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border39 = {-2,-1,3,0,JAM1,5,BorderVectors39,NULL};

struct IntuiText IText44 = {3,0,JAM2,29,1,&TOPAZ80,
    (UBYTE *) "GADGETUP",NULL};
```

```

struct Gadget Gadget12 = {&Gadget13,8,44,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border39,NULL,&IText44,
    NULL,NULL,1,GADGETUP};

SHORT BorderVectors40[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border40 = {-2,-1,3,0,JAM1,5,BorderVectors40,NULL};

struct IntuiText IText45 = {3,0,JAM2,17,1,&TOPAZ80,
    (UBYTE *)"DISKREMOVED",NULL};

struct Gadget Gadget11 = {&Gadget12,512,32,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border40,NULL,&IText45,
    NULL,NULL,1,DISKREMOVED};

SHORT BorderVectors41[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border41 = {-2,-1,3,0,JAM1,5,BorderVectors41,NULL};

struct IntuiText IText46 = {3,0,JAM2,13,1,&TOPAZ80,
    (UBYTE *)"DISKINSERTED",NULL};

struct Gadget Gadget10 = {&Gadget11,386,32,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border41,NULL,&IText46,
    NULL,NULL,1,DISKINSERTED};

SHORT BorderVectors42[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border42 = {-2,-1,3,0,JAM1,5,BorderVectors42,NULL};

struct IntuiText IText47 = {3,0,JAM2,24,1,&TOPAZ80,
    (UBYTE *)"DELTAMOVE",NULL};

struct Gadget Gadget9 = {&Gadget10,260,32,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border42,NULL,&IText47,
    NULL,NULL,1,DELTAMOVE};

SHORT BorderVectors43[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border43 = {-2,-1,3,0,JAM1,5,BorderVectors43,NULL};

struct IntuiText IText48 = {3,0,JAM2,17,1,&TOPAZ80,
    (UBYTE *)"CLOSEWINDOW",NULL};

struct Gadget Gadget8 = {&Gadget9,134,32,120,9,NULL,
    TOGGLESELECT,BOOLGADGET, (APTR)&Border43,NULL,&IText48,
    NULL,NULL,1,CLOSEWINDOW};

```

```
SHORT BorderVectors44[] = {0,0,123,0,123,10,0,10,0,0};
struct Border Border44 = {-2,-1,3,0,JAM1,5,BorderVectors44,NULL};

struct IntuiText IText50 = {1,0,JAM2,-3,-10,&TOPAZ80,
    (UBYTE *)"IDCMP-Flags:",NULL};

struct IntuiText IText49 = {3,0,JAM2,13,1,&TOPAZ80,
    (UBYTE *)"ACTIVEWINDOW",&IText50};

struct Gadget Gadget7 = {&Gadget8,8,32,120,9,NULL,
    TOGGLESELECT,BOOLGADGET,(APTR)&Border44,NULL,&IText49,
    NULL,NULL,1,ACTIVEWINDOW};

UBYTE Gadget6SIBuff[5] =
    "1";
struct StringInfo Gadget6SInfo = {Gadget6SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors45[] = {0,0,35,0,35,9,0,9,0,0};
struct Border Border45 = {-1,-1,1,0,JAM2,5,BorderVectors45,NULL};

struct Gadget Gadget6 = {&Gadget7,561,12,34,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border45,NULL,NULL,
    NULL,(APTR)&Gadget6SInfo,25,NULL};

UBYTE Gadget5SIBuff[5] =
    "0";
struct StringInfo Gadget5SInfo = {Gadget5SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors46[] = {0,0,35,0,35,9,0,9,0,0};
struct Border Border46 = {-1,-1,1,0,JAM2,5,BorderVectors46,NULL};

struct IntuiText IText53 = {1,0,JAM2,85,0,&TOPAZ80,
    (UBYTE *)" ",NULL};

struct IntuiText IText52 = {1,0,JAM2,38,1,&TOPAZ80,
    (UBYTE *)" ",&IText53};

struct IntuiText IText51 = {1,0,JAM2,-51,0,&TOPAZ80,
    (UBYTE *)"Pen: (",&IText52};
```



```

struct Gadget Gadget5 = {&Gadget6,513,12,34,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border46,NULL,&IText51,
    NULL,(APTR)&Gadget5SInfo,24,NULL};

UBYTE Gadget4SIBuff[5] =
    "100";
struct StringInfo Gadget4SInfo = {Gadget4SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors47[] = {0,0,57,0,57,9,0,9,0,0};
struct Border Border47 = {-1,-1,1,0,JAM2,5,BorderVectors47,NULL};

struct Gadget Gadget4 = {&Gadget5,382,12,56,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border47,NULL,NULL,
    NULL,(APTR)&Gadget4SInfo,23,NULL};

UBYTE Gadget3SIBuff[5] =
    "200";
struct StringInfo Gadget3SInfo = {Gadget3SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,0,0,NULL};

SHORT BorderVectors48[] = {0,0,57,0,57,9,0,9,0,0};
struct Border Border48 = {-1,-1,1,0,JAM2,5,BorderVectors48,NULL};

struct IntuiText IText56 = {1,0,JAM2,128,0,&TOPAZ80,
    (UBYTE *)" ",NULL};

struct IntuiText IText55 = {1,0,JAM2,59,1,&TOPAZ80,
    (UBYTE *)" ",",",&IText56};

struct IntuiText IText54 = {1,0,JAM2,-68,0,&TOPAZ80,
    (UBYTE *)"Gr `e: (",&IText55};

struct Gadget Gadget3 = {&Gadget4,313,12,56,8,NULL,
    RELVERIFY|STRINGCENTER|LONGINT,STRGADGET,(APTR)&Border48,NULL,&IText54,
    NULL,(APTR)&Gadget3SInfo,22,NULL};

UBYTE Gadget2SIBuff[5] =
    "0";
struct StringInfo Gadget2SInfo = {Gadget2SIBuff,NULL,
    0,5,0,0,0,0,0,0,0,0,0,0,NULL};

```


Stichwortverzeichnis

8520, Blockdiagramm 96

A1000 14

A2000 16

A500 15

Abtastfrequenz 203

Adreß-

bus 27

einbindung 125 ff.

register 50

Agnus 11, 67

Akustik 194

ALARM-Bit 107

Alert 322

AmigaDos 344

Amplituden

-modulation 215

-verlauf 221

Analog-/Digitalwandlung 200

Analogeingang 166

Animation 253

Asynchrone Datenübertragung 150

AttnFlags 41

Audio-

Hardware 206

Register 210

System 194

AUDxDAT-Register 214

Aufzeichnungsformate 186

Ausgaben 300

Ausgabezyklus 213

Ausnahme

-vektoren 39

-verarbeitung 38

-zustand 39

Aussteuerung 205

Barrelshifter 65

Befehls-cache 45

BERR* 35, 36

BG* 31

BGACK* 31, 32

Bildschirm

-position, Definition der 247

-priorität 254

Binärfiles, Struktur der 359

Bitplane

-Adressierung

-DMA 120

Bits 78

Blitter 63

Border-Struktur 322, 338

BR* 31

Bus 12

-steuerung 29

-zugriffssteuerung 31

C-Operatoren 954

Prioritäten 956

Cache 45

CAT-Chunk 940 f.

CD-Player 219

Centronics-Schnittstelle 142 ff.

Aufbau 144 f.

Pinbelegung 142 f.

Technische Realisierung 144 f.

Chip SELECT 95

CHIP-RAM 12, 50, 65

-Bus 119

CIA

- 8520 13
- Systemeinbindung 111 ff.
- Bausteine, Adreßvektor der 128
- Chips 94
 - Registerarchitektur 97
- Hardware, Pinbelegung 95
- Interrupts 108

clist.library 894, 910

Color

- Cycling 241
- Lookup-Table 78

Condition-Code-Register 554

console.library 894, 910

CONTINUOUS-Mode 102

Copper 62, 228

- Befehle 228
- Befehlsliste 235
- Hardware, Initialisierung der 235
- Liste, Modifikation der 242
- Programmierung 238

CRA 100

CRB 100

CS* 95

CTS 147

Customchip-Register 54, 855 ff.

D/A-Wandler 206

Data-Block 349

Daten

- bus 26
- organisation auf Disk 357
- register 51
- übertragung 177
 - asynchrone 150

Denise 74

Blockdiagramm 76

Pinbelegung 75, 79

Digitalisierung 202

Directory-Block 346 f.

Disketten-Format 344

Diskettenwechsel 185

diskfont.library 894, 910

DMA 10

- Controller 62 ff.
- System 116
- System 84
- Zyklus 213 f.

DMACON-Register 50 ff., 251

dos.library 895, 911

DosPacket-Struktur 351

DRAM 131 f.

DTACK* 29, 30, 135

Dual-Playfield-Modus 254

Eingaben 300

Entpacken 951

Ereigniszähler 107 f.

Event-Counter 106

exec.library 896, 912

ExecBase 262

expansion.library 899, 915

Extra-Half-Bright-Modus 79

Funktion

- AbortIO 303, 372
- ActivateGadget 373
- ActivateWindow 375
- AddAnimOb 376
- AddBob 377
- AddConfigDev 378
- AddDevice 379
- AddDosNode 380
- AddFont 381
- AddFreeList 382
- AddGadget 383
- AddGList 384
- AddHead 269, 385
- AddIntServer 386
- AddLibrary 388
- AddMemList 389
- AddPort 297, 391
- AddResource 392

- AddSemaphore 393
- AddTail 394
- AddTask 395
- AddTime 397
- AddVSprite 398
- Alert 399
- AllocAbs 400
- Allocate 307, 401
- AllocBoardMem 402
- AllocCList 403
- AllocConfigDev 404
- AllocEntry 405
- AllocExpansionMem 406
- AllocMem 304, 407
- AllocPotBits 409
- AllocRaster 410
- AllocRemember 341, 411
- AllocSignal 413
- AllocTrap 414
- AllocWBOobject 415
- AlohaWorkbench 342, 416
- AndRectRegion 417
- AndRegionRegion 418
- Animate 419
- AreaDraw 420
- AreaEllipse 421
- AreaEnd 422
- AreaMove 423
- AskFont 424
- AskSoftStyle 425
- AttemptLockLayerRom 426
- AttemptSemaphore 427
- AutoRequest 332, 428
- AvailFonts 430
- AvailMem 431
- BeginIO 303
- BeginRefresh 432
- BeginUpdate 433
- BehindLayer 434
- BlitBitMap 435
- BlitBitMapRastPort 437
- BlitClear 439
- BlitMaskBitMapRastPort 441
- BlitPattern 443
- BlitTemplate 444
- BuildSysRequest 446
- BumpRevision 448
- Cause 449
- CBump 450
- CDInputHandler 451
- ChangeSprite 452
- CheckIO 302, 454
- ClearDMRequest 455
- ClearEOL 456
- ClearMenuStrip 457
- ClearPointer 458
- ClearRectRegion 459
- ClearRegion 460
- ClearScreen 461
- ClipBlit 462
- Close 272, 464
- CloseDevice 465
- CloseFont 466
- CloseLibrary 467
- CloseScreen 468
- CloseWindow 469
- CloseWorkBench 470
- CMove 471
- CmpTime 472
- ConcatCList 473
- ConfigBoard 474
- ConfigChain 475
- CopyCList 476
- CopyMem 477
- CopyMemQuick 478
- CopySBitMap 479
- CreateBehindLayer 480
- CreateDir 382
- CreatePort 297
- CreateProc 483
- CreateUpfrontLayer 484
- CurrentDir 486
- CurrentTime 487
- CWait 488

- DateStamp 489
- Deallocate 490
- Debug 491
- Delay 492
- DeleteFile 493
- DeleteLayer 494
- DeviceProc 495
- Disable 496
- DisownBlitter 497
- Dispatch 497
- DisplayAlert 322, 498
- DisplayBeep 499
- DisposeLayerInfo 500
- DisposeRegion 501
- DoCollision 255, 502
- DoIO 302, 503
- DoubleClick 504
- Draw 505
- DrawBorder 506
- DrawEllipse 507
- DrawGLList 508
- DrawImage 509
- DupLock 510
- Enable 511
- EndRefresh 512
- EndRequest 513
- EndUpdate 514
- Enqueue 270, 515
- Examine 516
- Execute 518
- Exeption 517
- Exit 519
- ExitIntr 520
- ExNext 521
- Expunge 272
- ExtFunc 272
- FattenLayerInfo 522
- FindConfigDev 523
- FindName 270, 524
- FindPort 299, 525
- FindResident 526
- FindSemaphore 527
- FindTask 528
- FindToolType 529
- Flood 530
- FlushCLList 531
- Forbid 532
- FreeBoardMem 533
- FreeCLList 534
- FreeColorMap 535
- FreeConfigDev 536
- FreeCopList 537
- FreeCprList 538
- FreeDiskObject 539
- FreeEntry 540
- FreeExpansionMem 541
- FreeFreeList 542
- FreeGBuffers 543
- FreeMem 305, 544
- FreePotBits 545
- FreeRaster 546
- FreeRemember 341, 547
- FreeSignal 548
- FreeSprite 549
- FreeSysRequest 550
- FreeTrap 551
- FreeVPortCopLists 552
- FreeWBOobject 553
- GetCC 554
- GetCLBuf 555
- GetCLChar 556
- GetCLWord 557
- GetColorMap 558
- GetCurrentBinding 559
- GetDefPrefs 560
- GetDiskObject 561
- GetGBuffers 562
- GetIcon 563
- GetMsg 335, 564
- GetPacket 565
- GetPrefs 566
- GetRGB4 567
- GetScreenData 658
- GetSprite 569

- GetWBOject 570
- IEEEDPAbs 571
- IEEEDPAcos 572
- IEEEDPAdd 573
- IEEEDPAsin 574
- IEEEDPATan 575
- IEEEDPCeil 576
- IEEEDPCmp 577
- IEEEDPCos 578
- IEEEDPCosh 579
- IEEEDPDiv 580
- IEEEDPExp 581
- IEEEDPFieee 582
- IEEEDPFix 583
- IEEEDPFloor 584
- IEEEDPFlt 585
- IEEEDPLog 586
- IEEEDPLog10 587
- IEEEDPMul 588
- IEEEDPNeg 589
- IEEEDPPow 590
- IEEEDPSin 591
- IEEEDPSincos 592
- IEEEDPSinh 593
- IEEEDPSqrt 594
- IEEEDPSub 595
- IEEEDPTan 596
- IEEEDPTanh 597
- IEEEDPTieee 598
- IEEEDPTst 599
- IncrCLMark 600
- Info 601
- InitArea 602
- InitBitMap 603
- InitCLPool 604
- InitCode 605
- InitGels 606
- InitGMasks 607
- InitLayers 608
- InitMasks 609
- InitRastPort 610
- InitRequester 611
- InitResident 612
- InitSemaphore 613
- InitStruct 614
- InitTmpRas 615
- InitView 616
- InitVPort 617
- Input 618
- Insert 269, 619
- InstallClipRegion 620
- IntuiTextLength 621
- Intuition 622
- IoErr 622
- IsInteractive 623
- ItemAddress 624
- LoadRGB4 625
- LoadSeg 626
- LoadView 627
- Lock 628
- LockIBase 629
- LockLayer 630
- LockLayerInfo 631
- LockLayerRom 632
- LockLayers 633
- MakeDosNode 634
- MakeFunctions 635
- MakeLibrary 636
- MakeScreen 638
- MakeVPort 639
- MarkCList 649
- MatchToolValue 641
- MenuNumber 331
- ModifyIDCMP 642
- ModifyProp 643
- Move 645
- MoveLayer 646
- MoveLayerInFrontOf 647
- MoveScreen 648
- MoveSprite 649
- MoveWindow 650
- MrgCop 651
- NewFontContents 652
- NewLayerInfo 653

- NewList 298
- NewModifyProp 325, 654
- NewRegion 656
- ObtainConfigBinding 656
- ObtainSemaphore 657
- ObtainSemaphoreList 658
- OffGadget 659
- OffMenu 660
- OldOpenLibrary 661
- OnGadget 662
- OnMenu 663
- Open 272, 664
- OpenDevice 665
- OpenDiskFont 666
- OpenFont 667
- OpenIntuition 668
- OpenLibrary 669
- OpenResource 670
- OpenScreen 671
- OpenWindow 672
- OpenWorkBench 673
- OrRectRegion 674
- OrRegionRegion 675
- Output 676
- OwnBlitter 676
- ParentDir 677
- PeekCLMark 678
- Permit 678
- PlaneOnOff 340
- PlanePick 340
- PolyDraw 679
- PrintIText 680
- PutCLBuf 681
- PutCLChar 682
- PutCLWord 683
- PutDiskObject 684
- PutIcon 685
- PutMsg 299, 686
- PutWBOBJECT 687
- QBlit 688
- QBSBlit 689
- QueuePacket 690
- RawDoFmt 691
- RawIOInit 692
- RawKeyConvert 693
- Read 695
- ReadExpansionByte 696
- ReadExpansionROM 697
- ReadPixel 698
- RectFill 699
- RefreshGadgets 700
- RefreshGLList 701
- RefreshWindow 702
- ReleaseConfigBinding 702
- ReleaseSemaphore 703
- ReleaseSemaphoreList 704
- RemakeDisplay 705
- RemConfigDev 706
- RemDevice 707
- RemFont 708
- RemHead 270, 709
- RemIBob 710
- RemIntServer 711
- RemLibrary 712
- Remove 269, 713
- RemoveGadget 714
- RemPort 715
- RemResource 716
- RemSemaphore 717
- RemTail 718
- RemTask 719
- RemVSprite 720
- Rename 721
- ReplyMsg 336, 722
- ReportMouse 723
- Request 724
- Reschedule 725
- RethinkDisplay 725
- RomBoot 726
- Schedule 726
- ScreenToBack 727
- ScreenToFront 728
- ScrollLayer 729
- ScrollRaster 730

- ScrollVPort 731
Seek 732
SendIO 302, 734
SetAPen 735
SetBPen 736
SetCollision 737
SetComment 738
SetCurrentBinding 739
SetDMRequest 740
SetDrMd 741
SetExcept 742
SetFont 743
SetFunction 744
SetIntVector 745
SetMenuStrip 746
SetPointer 747
SetPrefs 748
SetProtection 749
SetRast 750
SetRGB4 751
SetRGB4CM 752
SetSignal 753
SetSoftStyle 754
SetSR 755
SetTaskPri 756
SetWindowTitles 757
ShowTitle 759
Signal 289, 760
SizeCList 761
SizeLayer 762
SizeWindow 763
SortGList 764
SPAbs 766
SPAcos 767
SPAdd 768
SPAsin 769
SPAtan 770
SPCeil 771
SPCmp 772
SPCos 774
SPCosh 775
SPDiv 776
SPExp 777
SPFieee 778
SPFix 779
SPFloor 780
SPFlt 781
SplitCList 765
SPLog 782
SPLog10 783
SPMul 784
SPNeg 785
SPPow 786
SPSin 787
SPSincos 788
SPSinh 789
SPSqrt 790
SPSub 791
SPTan 792
SPTanh 793
SPTieee 794
SPTst 795
SubCList 796
SubTime 797
SumKickData 798
SumLibrary 798
SuperState 799
Supervisor 799
SwapBitsRastPortClipRect 800
Switch 801
SyncSBitMap 801
Text 802
TextLength 803
ThinLayerInfo 804
Translate 805
TypeOfMem 806
UCopperListInit 807
UnGetCLChar 808
UnGetCLWord 809
UnLoadSeg 810
UnLock 811
UnlockIBase 812
UnLockLayer 813
UnLockLayerInfo 814

- UnlockLayerRom 815
- UnlockLayers 816
- UnPutCLChar 817
- UnPutCLWord 818
- UpfrontLayer 819
- UserState 820
- VBeamPos 821
- ViewAddress 822
- ViewPortAddress 823
- Wait 289, 336, 824
- WaitBlit 825
- WaitBOVP 826
- WaitForChar 827
- WaitIO 302, 828
- WaitPort 299, 335, 829
- WaitTOF 253, 830
- WBenchToBack 830
- WBenchToFront 831
- WhichLayer 832
- WindowLimits 833
- WindowToBack 834
- WindowToFront 835
- Write 836
- WriteExpansionByte 837
- WritePixel 838
- WritePotGo 839
- XorRectRegion 840
- XorRegionRegion 841
- Farbregister 61, 78
- Farbselektion 244
- FAST-RAM 12
- FastFileSystem 350
- Fat Agnus 11, 62 ff.
- Pinbelegung 69 ff.
- Fehler
 - meldungen 888 ff.
 - Vektor 888
- File
 - Header-Block 347 f.
 - List-Block 349
- FileSystem-Kommandos 352
- Flip-Flop 130
- Floppy-Controller 181
 - Programmierung 188
- Floppy-Schnittstelle 181 ff.
 - Signale der 183
- FM 187
- FORM-Chunk 938 ff.
- Frequenz 194, 196
 - modulation 215
- Funktionscodes 34
- Gadgets 318
- Game-Port 155, 161
- Game-Ports
 - Aufbau der 155
 - Schaltung der 155
- Gary, Blockschaltbild 124
- GCR 187
- Grafik 336
 - datenorganisation 77
 - format 934
- graphics.library 900, 916
- Guru-Meditation 888 ff.
- HALT* 35, 36
- Hashing 358
- Havard-Architektur 23
- Hot-Spot 257
- Hunk-Struktur 360
- icon.library 903, 919
- IDCMP 333
- Identifikationsmodus 185
- IFF
 - Datei 938 f.
 - Grafikformat 934 ff., 943
- ILBM 943
- Image 339 f.
 - Struktur 323, 339
- Include-File
 - sampleba.asm 973
 - sampleba.seh 971
 - sampleba.sei 972

- win.h 986
- INTENA 86 f.
- Interlace-Modus 240
- Interprozeßkommunikation 287
- Interrupt
 - Controller 84
 - Aufgaben 85
 - Funktion 85
 - Level 293
 - System 109
 - Datenregister 109
 - Maskenregister 109 f.
 - Programmierung 85
- Interrupts 292
 - Bearbeiten von 294
- INTREQ 86 f.
- IntuiMessage-Struktur 336
- Intuition 310
 - Texte 339
- intuition.library 904, 920
- IORequest-Struktur 301
- IPL* 33
- JOY0DAT 159 ff.
- Joystickanschluß 161
- Kanal 288
- Klangfarbe 198
- Kollisionen
 - erkennung 255
 - register 256
- Kommunikation 177 ff.
 - im System 296
- Kontrollregister 51
- Kopfpositionierung 184
- Laufwerkselektion 183
- Lautstärke 212
 - empfindung 195
- layers.library 906, 922
- LDS* 30
- Libraries 270
 - clist 894, 910
 - console 894, 910
 - diskfont 894, 910
 - dos 895, 911
 - exec 896, 912
 - expansion 899, 915
 - graphics 900, 916
 - icon 903, 919
 - intuition 904, 920
 - layers 906, 922
 - mathffp 907, 923
 - mathieeedoubbas 907, 923
 - mathieeedoubtrans 908, 924
 - mathtrans 908, 924
 - potgo 909, 925
 - romboot 909, 925
 - timer 909, 925
 - translator 909, 925
 - Eigene 274 ff.
 - Base 275
 - Einsprünge 272
 - Knoten 273
- LIST-Chunk 941
- Listen 267
 - verwaltung 267
- Longplay 216
- Makefile 966
- makefile.hnk 966
- makefile.win 967
- Maskenregister 109
- mathffp.library 907, 923
- mathieeedoubbas.library 907, 923
- mathieeedoubtrans.library 908, 924
- mathtrans.library 908, 924
- Maus 157
 - bewegung 160
 - pfeil 160
- MC68000 22
 - Ausnahmereverarbeitung 37
 - Befehlsausführungszeiten 846 ff.

- Betriebszustände 37
- Lesezyklus 136
- Pinbelegung 24
- MC68010 42
- MC68020 45
- MC68030 23, 45
- MC68881 41
- MEMF_CHIP 304
- MEMF_CLEAR 304
- MEMF_FAST 304
- MEMF_PUBLIC 304
- Memory 304
- Menu-Struktur 327
- MenuItem-Struktur 328
- Menü 327
 - punkte 331
- Messages 296
- MFM 187
- Motorsteuerung 183
- MOVE 45, 117
- Multi-Color-Sprites 246
- Multiplexer 133
- Multiprozessorbetrieb 32
- Multitasking 283 ff.
- NewScreen-Struktur 316
- NewWindow-Struktur 311, 334
- NMI 293
- Noten 223
- Packen 950 f.
- Paddles 163
- PAL-Rasterzeile 121
- Papiervorrat zuende 144
- Parallel
 - Port 97 f.
 - Schnittstelle 144
- Parameterregister 51
- Partialtöne 198
- Paula 82, 166
 - Blockdiagramm 83
 - Pinbelegung 82, 89 ff.
- Pixel 78
 - Adressierung 74
- Playfield 254
- Port 296
 - paralleler 97 f.
 - serieller 104
- Potentiometer 163
- potgo.library 909, 925
- POUT 144
- Programm
 - clibtest.c 958
 - hunk.c 959
 - mkassl.v 968
 - sl.asm 974
 - win.c 979
- Programmiermodell 48
- PROP-Chunk 942
- Proportional-Gadgets 324
- Prozessor 13
- Prüfsumme 798
- Pulse-Mode 103
- Quadraturimpuls 158
- Quantisierungsfehler 201
- Quittungssignal 135
- RAM-Bausteine 130
- RAM-Chips
 - Dynamische 130
 - Pinbelegung eines 133
- Rasterstrahl-Interrupt 239, 243
- Rauschgeräusche 199
- Raw-Keycodes 170
- READY-Liste 285
- Rechteckschwingung 222
- Refreshmechanismus 134
- Registerarchitektur 48
- Remember-Struktur 341
- Requester 332
- Reset 180
- RESET* 35, 36
- Resonanz 198

- RGB 78
- romboot.library 909, 925
- Root-Block 345
- Round-Robin-Verfahren 286
- RS232 146
- RTS 147
- RUNMODE-Bit 102
- RXD 148 f.
- Samplingfrequenz 203
- Samplingrate 212
- Schalleistung 195
- Schallpegel 196
- Schnittstelle
 - Serielle 146
 - Aufbau 148 f.
 - Schaltung 149
 - Signale der 148
 - Technische Realisierung 148 f.
- Schreibschutzfenster 185
- Screens 316 ff.
- SELECT-Leitung 184
- Serielle Schnittstelle 146
- Signal 289
- SKIP 234
- SOFT-Interrupts 88
- Software-Interrupts 295
- Speicher
 - ausstattung 128 ff.
 - belegung 122
 - organisation 137 f.
 - system 139
 - zugriff 12
 - zyklus 118
- Sprite 244
 - Datenliste 244, 249
 - Hardware, Programmierung der 249
- Statusregister 755
- String-Gadgets 326
- StringInfo-Struktur 326
- Strobe-Register 54
- Synchronisation zwischen Copper und Blitter 243
- Synchronisationstakt 105
- Synthesizer 221
- System
 - architektur 8
 - bausteine 13
 - software 262
 - steuersignale 35
 - tuning 40
- Task 283
 - stoppen 290
 - Steuerung 285
 - Wechsel 284
 - Zustände 284 f.
- Tastatur 170
 - belegung, A2000 171
 - platine 174
 - A1000 175
 - prozessor 173, 176
- Tasten-Codes 172, 928 ff.
- Tiefpaßfilter 208 f.
 - Technische Realisierung 209
- Timer 94, 98
- timer.library 909, 925
- Timerunterlauf 105
- Timeslot 118
 - Allocation-System 121
- Toggle-Mode 103
- Tonhöhe 194
- Trackdisk-Device 344
- Transistor 130
- translator.library 909, 925
- Traps 291
- TXD 148 f., 154
- UART-Schaltung 151
- UDS* 30
- Unterbrechungs
 - anforderung 85

-steuerung 33
Übertragungsfehler 101

Vertical-Blanking-Interrupt 160
Videoshifter 74

WAITING-Status 290

Warten 335

Wellenform 221

Windows 310 ff.

Wortzugriff 28

XY-Joysticks 163

Zählimpuls 160

Zellenarchitektur 131

Zugriffs

-priorität 119

-zeit 134 f.

Zustandcodes 179 f.

Zyklusdauer 134

SYBEX
Computerbücher & Software

Fordern Sie ein Gesamtverzeichnis
unserer Verlagsproduktion an:

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
D-4000 Düsseldorf 30
Tel.: (0211) 61 80 2-0
Telex: 8 588 163
Fax: 0211/6180227

Empfohlen für:

Einsteiger

✓ Fortgeschrittene

✓ Profis

Anleitung

✓ Referenz

Schulung

Amiga Profibuch

Das Amiga Profibuch gibt Ihnen die Antworten zur Hardware und Systemprogrammierung, die Sie in anderen Nachschlagewerken vergeblich gesucht haben. Hier bleibt keine Fragen offen. Auf kleinstem Raum (0,024 m³, um genau zu sein) wird eine geballte Ladung von Informationen geboten. Räumen Sie also Ihren Schreibtisch auf, schaffen Sie Platz für Ihren Amiga und die Peripherie, und nutzen Sie das geballte Wissen dieses Standardwerks, um ihr System zu optimieren oder sich über Hardware und die Systemsoftware der Amiga-Familie zu informieren.

Hier einige Highlights aus dem Inhalt:

- Grundlegende Aspekte der Systemarchitektur
- Die verschiedenen Modelle der Amiga-Familie
- Die CPU
- Die Anatomie der Customchips
- Programmiermodell und Registerarchitektur
- Die CIA-Hardware
- Vom DMA-System zur Speicherorganisation
- Die Interface-Hardware

- Das Audio-System
- Das Grafik-System
- Die Systemsoftware: Exec, Intuition
- Das Disk-Operating-System
- Die Library-Routinen

Auf der Diskette zum Buch finden Sie einige nützliche Programme, die Ihnen noch mehr Informationen über Ihr System liefern.

ISBN N 3-88745-580-0

DM 79,—
sFr 75,80
öS 616,—



9 783887 455804

Buchregal-Systematik:

Amiga