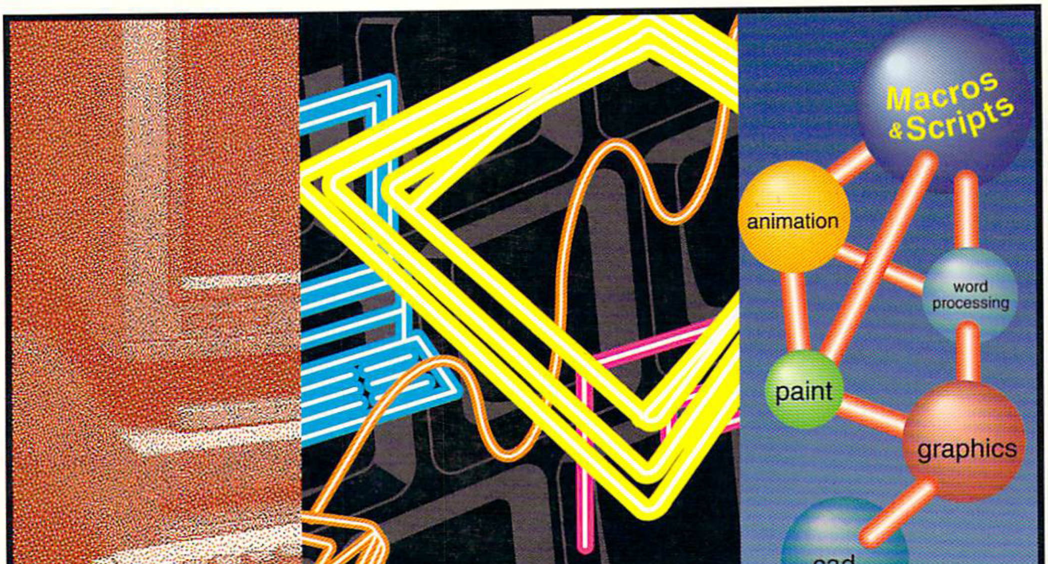
Amiga® Intern

Includes Release 2.0 Workbench 2.0 information

The definitive reference book for all Amiga computers

Christian Kuhnert, Stefan Maelger, Johannes Schemmel

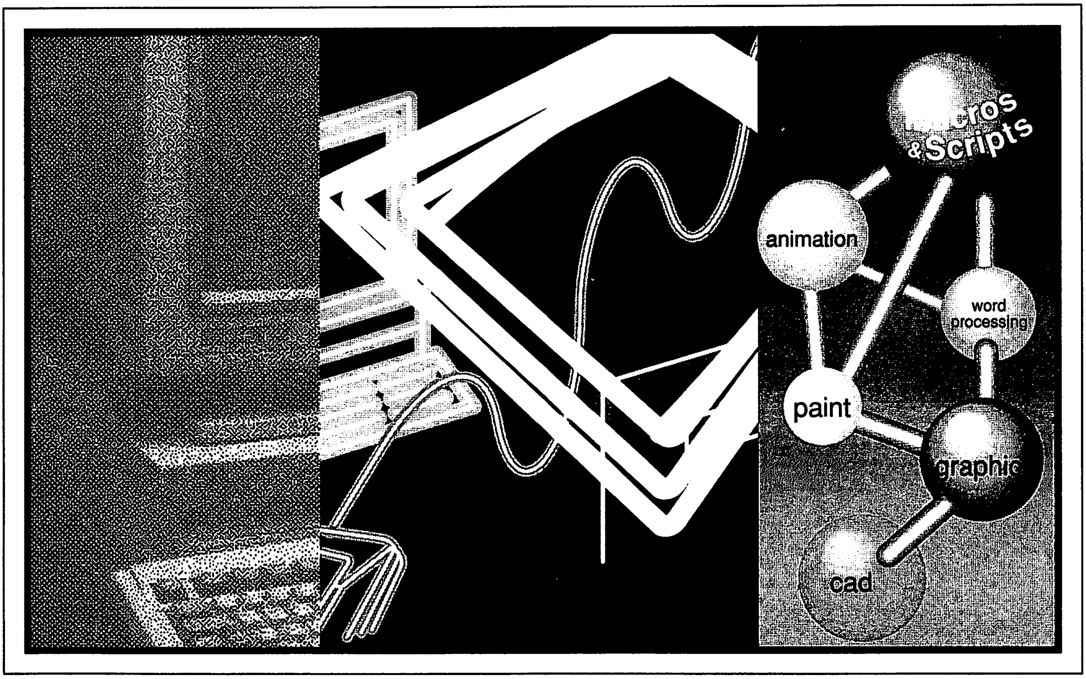


Abacus
A Data Becker Book

Amiga® Intern

The definitive reference book for all Amiga computers

Christian Kuhnert, Stefan Maelger, Johannes Schemmel



Abacus
A Data Becker Book

Library of Congress Cataloging-in-Publication Data
Maelger, Stefan, 1965-
Amiga Intern / Stefan Maelger, Christian Kuhnert, Johannes Schemmel.
p. cm.
Includes index.
ISBN 1-55755-148-0 : $39.95
1. Amiga (Computer) I. Kuhnert, Christian, 1967- . II. Schemmel, Johannes, 1958- . III. Title.
QA76.8.A177M34 1992
004.165--dc20 92-8083
CIP

Printed in the U.S.A.
10 9 8 7 6 5 4 3 2 1

Foreword

The Amiga once widely considered a little more than just a game machine, has now become a worthy and serious rival to the PC and the Mac.

Both the professional quality of software and the improvement of the Amiga's operating system have contributed to its "coming of age". With the appearance of Kickstart 2.0 (AmigaOS 2.0), the user interface has attained a professional level. It's natural that this professionalism should carry over into the quality of software. Much knowledge about hardware and software is required to master the Amiga. Assuming you're acquainted with the basics of programming, and the detailed information about how the system works, this book will provide you with the necessary professional know-how. The scope of the book alone indicates the enormous amount of knowledge and effort that have gone into its preparation. To address as many aspects of the Amiga as possible, three authors have contributed their knowledge and experience. Correspondingly, the book is divided into three parts:

Part 1:	System Programming	(Stefan Maelger)
Part 2:	ARexx	(Christian Kuhnert)
Part 3:	A3000 Intern	(Johannes Schemmel)

These sections can be read individually or consecutively; their sequence is not important. Each one constitutes in itself a useful learning tool and a guide for later reference.

We wish you many enjoyable and enlightening hours with "Amiga Intern." Maybe you will soon be publishing professional software for the Amiga.

We are grateful to Commodore and especially to Dr. Kittel for their kind support.

Table of Contents

1. Kickstart 2.04 5
1.1 Inside AmigaOS 2.x 5
1.1.1 Reset Capabilities 6
1.1.2 The Main Units of AmigaOS 2.x 7
1.1.3 Disk Libraries and Devices 9
1.2 AmigaOS 2.x Compatibility 10

2. Using the Amiga 3000 13
2.1 The Workbench 13
2.1.1 Starting AmigaOS 2.x 13
2.1.2 The Workbench Menus 14
2.1.3 The Workbench Programs 18
2.2 The Command Line Interpreter 22
2.2.1 AmigaOS 2.x Resident Commands 22
2.2.2 Using the CLI 22

3. Programming with AmigaOS 2.x 25
3.1 The Libraries and their Functions 25
3.1.1 The ASL Library 26
3.1.2 The Commodities Library 36
3.1.3 The Diskfont Library 54
3.1.4 The DOS Library 61
3.1.5 The Exec Library 161
3.1.6 The Expansion Library 225
3.1.7 The GadTools Library 233
3.1.8 The Graphics Library 251
3.1.9 The Icon Library 326
3.1.10 The IFFParse Library 331
3.1.11 The Intuition Library 350
3.1.12 The Layers Library 433
3.1.13 The MathFFP, MathIEEESingBas, and MathIEEEDoubBas Libraries 443
3.1.14 The MathTrans, MathIEEESingTrans, and MathIEEEDoubTrans Libraries448
3.1.15 The Translator Library 452
3.1.16 The Utility Library 453
3.1.17 The Workbench Library 461

4. ARexx 473

4.1 The ARexx Language 473
4.2. The Functions of ARexx 475
4.3 An Overview of ARexx 476
4.4 ARexx - Rexx on the Amiga 479
4.5 A Sample Application 480

5. ARexx Syntax 483

5.1 Using Tokens 483
5.1.1 ARexx Symbols 483
5.1.2 Character Strings in ARexx 485
5.1.3 The ARexx Operators 485
5.1.4 ARexx Special Characters 486
5.2 Expressions 487
5.2.1 Arithmetic Operators 488
5.2.2 Concatenation Operators in ARexx 490
5.2.3 Comparison Operators in ARexx 490
5.2.4 Using Logical Operators 491
5.3 ARexx Clauses 492
5.3.1 Null Clauses 492
5.3.2 ARexx Label Markers 492
5.3.3 Assignments in ARexx 492
5.3.4 ARexx Commands 493
5.3.5 Commands 493

6. Instructions 495

6.1 I/O Instructions 496
6.2 Structured Instructions 501
6.3 ARexx Control Instructions 507
6.4 Commands 513

7. ARexx Functions 515

7.1 ARexx Internal Functions 516
7.2 Built-in Functions 516
7.3 ARexx External Function Libraries 517
7.4 I/O Functions 520
7.5 ARexx String Functions 524
7.6 Bit Manipulation in ARexx 535
7.7 Numeric Functions 538
7.8 Conversion Functions in ARexx 541
7.9 ARexx System Functions 544

8. Special Features 555

8.1 Parsing Strings with Templates 555
8.1.1 Examples of Parsing 557

8.2 Error Trapping with TRACE....561
8.2.1 Trace Options....561
8.2.2 TRACE Output....562
8.2.3 Command Suppression in ARexx Programs....563
8.2.4 Interactive Tracing....564
8.2.5 SIGNAL Interrupts and Error Handling....565

9. ARexx on the Amiga....569

9.1 Commands....569
9.2 Exchanging Data with the Clip List....573
9.3 The rexxsupport.library....574
9.3.1 EXEC Functions....574
9.3.2 DOS Functions....581
9.4 Creating ARexx Function Libraries....583

10. The ARexx Interface....587

10.1 Essential Data Structures....588
10.2 Requirements for a Command Interface....590
10.2.1 Command Calls....593
10.2.2 Function Calls....594
10.2.3 ARexx Program Search Order....594
10.2.4 Expanded RexxMsg Structure Areas....595
10.2.5 Result Entries....596
10.3 The Rexx Master Procedure....598
10.3.1 Action Codes....598
10.3.2 Action Code Control Flags....600
10.3.3 Managing the Results....601
10.4 Functions in rexxsyslib.library....602
10.4.1 I/O Functions....604
10.4.2 String Manipulation....609
10.4.3 Conversion Functions in ARexx....611
10.4.4 ARexx Resource Handling....614
10.5 The RexxBase Lists....623
10.6 ARexx Error Messages....624

11. The A3000 Hardware....635

11.1 Processor Generations....637
11.2 The 68030....641
11.2.1 The PMMU....668
11.2.2 The Floating Point Coprocessor....683
11.2.3 Differences Between the MC 68881 and 68882....701

11.2.4 Cache Memory 702
11.3 The CIA 8520 707
11.4 Custom Chips and the Amiga 721
11.4.1 Basic Structure of the Amiga 722
11.4.2 The Structure of Agnus 726
11.4.3 The Structure of Denise 730
11.4.4 The Structure of Paula 734
11.5 The Amiga Interfaces 738
11.5.1 The Audio Outputs 738
11.5.2 The RGB Connector 739
11.5.3 The VGA Connector 741
11.5.4 The Video Slot 742
11.5.5 The Centronics Interface 744
11.5.6 The Serial Interface 746
11.5.7 The External Drive Connector 748
11.5.8 The Game Ports 755
11.5.9 The Zorro Bus 758
11.6 The Keyboard 764
11.6.1 Data Transfer from the Keyboard 766
11.7 Programming the Hardware 770
11.7.1 The Memory Layout 770
11.7.2 Fundamentals 780
11.7.3 Interrupts 794
11.7.4 The Copper Coprocessor 797
11.7.5 Playfields 807
11.7.6 Sprites 838
11.7.7 ECS Capabilities 860
11.7.8 The Blitter 865
11.7.9 Sound Output 905
11.7.10 Mouse, Joystick and Paddles 933
11.7.11 The Serial Interfaces 941
11.7.12 The Disk Controller 946

Bibliography 951

Index 953

Part 1

System Programming

Part 1 - Introduction

The Amiga operating system is modular. Multitasking is achieved simply and with near-optimal memory utilization through the use of libraries and virtual devices. Only what is needed is saved in memory, and several programs can share simultaneous access to system resources. The capability of Intertasking, or interprogram communication over message ports, is one of the many features of the Amiga's powerful and flexible operating system that you will read about in this book. At first, learning all these capabilities won't seem easy. The first part of the book, "System Programming", should give you the necessary background information for system programming in the AmigaOS 2.0 environment.

Author: Stefan Maelger

1. Kickstart 2.04

The new AmigaOS (Amiga Operating System) is here. It has taken some time to reach its present state of development. However, the wait has been worth it because this operating system is better than any predecessor. This contains 512K, which makes this system more powerful than any other system.

1.1 Inside AmigaOS 2.x

AmigaOS 2.x is based on the hardware environment of the Amiga 3000 (i.e., on the 68030 and 68882 processors) and the new ECS (Enhanced Chip Set) custom chips. It must be distinguished from previous beta versions based on the Amiga 1000, which are capable of calling only part of the power that the Amiga 3000 operating system provides. These beta versions, released to program developers who did not own a 3000, were intended for compatibility testing only.

Similar to the preceding versions, AmigaOS 2.x is a multitasking operating system, running several programs simultaneously. By dividing memory into two distinct types and utilizing the DMA (co-processor) concept, it is capable of actual simultaneous memory access (i.e., true hardware multitasking). The Amiga's main program is the Input task. It manages all input and transfers control to various system routines. For example, the complex display-controller called Intuition. Commands are passed from Intuition to the Input task, where, at intervals of 1/50th of a second, they are eligible for execution. Since Intuition tasks are accomplished almost exclusively by the Input task, and the work of this program is synchronized by clock pulses, all performance tests using Intuition are meaningless. But with speed, the availability of static 32 bit RAM on the non-multiplexed bus (free Fast RAM) will enhance performance considerably, by enabling the processor to be switched to burst-mode for top-speed access of the 68030's data and instruction cache.

To find out how the operating system is put together, let's try taking it apart.

1.1.1 Reset Capabilities

Begin with a "cold start" by switching on the 3000. Press both mouse buttons at once, and you will be moved to the Operating System Menu. Here you can select the operating system you want to work with and specify the source from which it should be loaded. For example, an old version of the operating system can be loaded from the (hard) drive into a RAM storage area. The 68030's integrated MMU logically shifts this area to the normal operating system address and protects it against overwrite.

Now the current operating system's normal reset routine, which can also be invoked by the sequence <Ctrl,lft-Amiga,rt-Amiga>, is initiated. Under AmigaOS 2.x, any external or internal expansions are immediately recognized and incorporated into the system (this was not the case with earlier versions). The operating system checks hardware and memory and builds the tables for routines (error handling) and interrupts. All base data structures containing variable values are then set up.

Pressing both mouse buttons again will take you to the Boot Menu. This screen allows you to select the logical or physical drive from which booting will take place. This drive will be referred to as SYS (system directory). For other Amigas, even before the start of DOS, all logical drives are recognized and drive names established. The execution of the Startup sequence can also be disabled. This can be an advantage for CLI users, since the InitialCLI itself is now a complete shell, providing a convenient and easy-to-use platform for the Command Line Interface.

Now the Device Operating System (DOS) is started and the work shell initialized. To save time and avoid problems selecting the right monitor driver, the windows aren't opened until the Workbench is activated or output in an InitialCLI window requires it.

1.1.2 The Main Units of AmigaOS 2.x

The Amiga Operating system is designed in modules. Considering the size of the entire system and that the Amiga is a multitasking device, this is a great advantage. The modular design makes the system more flexible and easier to change. The main units can be divided into four groups: Libraries, Devices, Resources and Special. Libraries are simply collections of routines of a certain type or application. Devices serve as logical device drivers and may perform one or more tasks. Resources include base routines which usually manage access to certain resources and exclude them from or reserve them for other programs.

The modules are initialized according to their priorities.

The following modules are found in AmigaOS 2.4 ROM in the order of their initialization:

Address	Pri	Typ	Name	Vers.	Date
$00f83cc0	+110	Library	expansion	37.23	(3/15/91)
$00f800b6	+105	Library	exec	37.52	(3/15/91)
$00f83cda	+105	Special	diag init		
$00fbb09a	+103	Library	utility	37.3	(2/13/91)
$00faba14	+100	Resource	potgo	37.4	(1/28/91)
$00f889e0	+80	Resource	cia	37.4	(3/15/91)
$00f98dac	+80	Resource	filesysres	37.1	(1/12/91)
$00f8f3bc	+70	Resource	disk	37.1	(1/9/91)
$00fab964	+70	Resource	misc	37.1	(1/8/91)
$00fbbb50	+65	Library	graphics	37.20	(3/14/91)
$00faebd8	+60	Device	gameport	37.8	(1/28/91)
$00fb8540	+50	Device	timer	37.57	(3/14/91)
$00f85890	+45	Resource	battclock	37.3	(3/11/91)
$00faec02	+45	Device	keyboard	37.8	(1/28/91)
$00f862d0	+44	Resource	battmem	37.3	(3/4/91)
$00fa6984	+40	Library	keymap	37.2	(1/8/91)
$00faec2c	+40	Device	input	37.8	(1/28/91)
$00fa76c4	+31	Library	layers	37.7	(3/13/91)
$00fae054	+25	Device	ramdrive	37.3	(1/9/91)
$00fb936c	+20	Device	trackdisk	37.3	(3/13/91)
$00fb0298	+10	Device	scsidisk	37.4	(2/26/1)
$00fd3f6c	+10	Library	intuition	37.220	(3/14/91)
$00f83ca4	+5	Special	alert.hook		
$00f8b358	+5	Device	console	37.85	(3/13/91)
$00fab5f4	+0	Library	mathieeesingbas	37.2	(2/7/91)
$00f86508	-35	Special	syscheck	37.2	(1/15/91)
$00fb7620	-40	Special	romboot	37.23	(3/15/91)
$00fff46c	-45	Special	Magic	36.7	(3/16/90)
$00f864c8	-50	Special	bootmenu	37.2	(1/15/91)
$00fb763a	-60	Special	strap	37.23	(3/15/91)
$00f98f3e	-81	Special	fs	37.11	(3/13/91)
$00fae70c	-100	Special	ramlib	37.13	(3/14/91)
$00f847f0	-120	Device	audio	37.7	(3/13/91)
$00f90390	-120	Library	dos	37.22	(3/15/91)
$00f9e4d0	-120	Library	gadtools	37.82	(3/14/91)
$00fa445c	-120	Library	icon	37.6	(3/2/91)
$00fab110	-120	Library	mathffp	37.1	(1/13/91)
$00fbba7a	-120	Task	Pre-2.0 LoadWB stub		
$00feccd4	-120	Library	wb	37.108	(3/14/91)
$00f88d8e	-121	Special	con-handler	37.39	(3/13/91)
$00fb2ed4	-122	Special	shell	37.37	(3/13/91)
$00fabbb8	-123	Special	ram	37.9	(3/15/91)

Some modules are only included for backward compatibility. For example, the workbench-task module and the "mathffp.library" are used. All other modules contained in ROM are used frequently or are required by other modules.

1.1.3 Disk Libraries and Devices

Modules are found in ROM, on the Workbench disk or in the system directory of the hard drive. These programs are loaded as they are used:

Name	Version	Directory
asl.library	37.25	"LIBS"
commodities.library	37.5	
diskfont.library	36.50	
iffparse.library	37.1	
mathieeedoubbas.library	37.1	
mathieeedoubtrans.library	37.1	
mathieeesingtrans.library	37.1	
mathtrans.library	37.1	
translator.library	37.1	
version.library	37.33	
rexxsyslib.library	36.19	
rexxsupport.library	34.9	
clipboard.device	37.4	"DEVS"
narrator.device	37.5	
parallel.device	37.1	
printer.device	35.603	
serial.device	37.1	
aux-handler		"L"
port-handler		
speak-handler		
queue-handler		

1.2 AmigaOS 2.x Compatibility

The addresses of routines in ROM will vary from version to version. They should not be called directly, since they are always subject to change. In short, do not rely on a specific value for anything that Commodore has not declared a constant. A disadvantage with compatibility is the memory requirements of programs. The new operating system uses more memory than earlier versions to accommodate its many new features. The same is true for program stack requirements. System routines have become much more complex, with a corresponding increase in their stack storage needs.

Many values whose contents are made up of flag bits have been expanded, and failure to handle them accordingly can lead to problems. Also, this can happen to the 68030's expanded status register. Unfortunately, some system data not defined as PUBLIC has found its way into circulation. These values are not to be trusted and changes in their definitions can most likely happen. The programmer can always rely on the address $00000004. This is the base address of the "exec.library" for all versions of the operating system. All other values are uncertain. The color and proportions of the system font can also change. Processor speed has increased dramatically. As a rule, a program will have to be synchronized with clock impulses or the monitor's electron beam.

The main and co-processors' instructions doesn't allow interval storage of values, bits in addresses or instruction codes.

Many extensions of AmigaOS 1.3 have been removed and integrated into the base module in a large expanded form. For example, the "romboot.library" was removed and the boot routine completely reprogrammed. Autobooting from devices other than the internal disks is now standard and fully supported by the system. Like the SCSI-devices, all disks come bootable from the supplier. Drives DF0 through DF3 are assigned priorities of +5, -10, -20, and -30.

Early in the reset-routine the new operating system's enhancements become apparent. Calling of the ColdCapture vector is delayed. At any time the Exception/Interrupt Table can be placed over the Vector Base Register (VBR).

There are allowances for changing the size and type of MemHeader structures, and the use of ResetWindows has been revised.

The base structure of the Expansion-library is declared as PRIVATE and may not be accessed. Any expansions are incorporated in two passes accompanied by the sorting of address slots.

The "dos.library" is greatly expanded and, like many other modules, programmed with the SAS C-compiler Version 5. Its base structure now conforms to that of the other libraries. However, for compatibility reasons, some addresses still exist as BCPL-pointers. New types of DosPackets and new locks have been implemented. The process structure has grown substantially, so that auto-creation, for example with the popular "arp.library," results in a system crash.

The Workbench, which has changed in appearance and color, can now be nearly any desired size, shade and resolution. Window frames and gadgets adjust automatically to changes in resolution and fonts. Workbench windows can be transferred to other screens. Screens, which can consist of up to 16368*16384 pixels, are capable of new display modes, overscan into the unseen border area, and several styles of horizontal and vertical scrolling. All data necessary to duplicate a screen can no longer be determined from the screen structure. Screen handling is greatly improved and, even with SimpleRefresh windows, a message is sent only when refresh is necessary. Different color borders indicate which windows are active and special effects create a 3-dimensional appearance. There are new IDCMP-flags for this, and both keyboard flags now transmit raw data for special keys.

The Layer system is improved. SimpleRefresh layers are saved and refreshed to the fullest possible extent. The routines FattenLayerInfo, ThinLayerInfo, and InitLayers should no longer be used; NewLayerInfo accomplishes all these functions.

Computing of Copper lists has been optimized. The video-hardware does not like programming errors, such as switching off the display mode in mid-display. GetColorMap() must be used to manipulate ColorMap structures, which have increased in size. Row/cols values in the GfxBase no longer relate to the Workbench.

Although the font structures have a new format, the old one continues to be supported. The system-area of "font" files has been changed. Character set sizes that are not present are now simply calculated. The topaz font is still in ROM, but now as a sans-serif variant for increased legibility at high resolutions. Size and proportions of the system font can be specified as desired.

Many CLI/Shell commands are stored in ROM, and several CLI/Shell processes can run simultaneously. Windows are now equipped with close gadgets that, when activated, cause an EOF code to be sent. The missing cursor error in SuperBitmap Console windows has been corrected.

The Audio device no longer is initialized until its first use, which can result in errors because of insufficient memory.

Several serial interfaces (expansion cards) are possible. However, this can lead to problems with the adjustment of certain parameters through the Serial device.

Trackdisk device buffers can be released, but a subsequent attempt to use a buffer may result in an error if insufficient memory is available.

Both CIAB timers are now accessible.

The current maximum for chip RAM is 2 Meg. Fast RAM is configured down from the upper boundary of memory (in full 32 bit addresses) and can be as high as 8 Gigabytes. This configuration will make it easier for a future release to break the 2 Meg chip RAM barrier, probably reaching as high as 8 Meg.

The ECS has more hardware registers, which reside in between those familiar to the previous system and can cause problems for programmers of clockcycle-optimized programs. Some old registers contain important new bits. The accubuffered truetime clock is not compatible with earlier clock chips.

2. Using the Amiga 3000

We recommend working through the following exercises step by step. While providing a quick overview of the use and capabilities of the 3000, a lot of important information is included that everyone will find useful.

2.1 The Workbench

Since the SCSI hard disk comes factory-installed, a few seconds after switching on the computer the graphical user interface, called the Workbench, appears. If you're already familiar with previous versions of the Amiga, you'll immediately notice some changes. The Workbench window is no longer just a background screen. It has acquired a border with which it can be moved around, brought into the foreground or reduced in size. There is even a close gadget (which should be used carefully). Professional color selection and the appearance of 3-dimension are impressive enhancements that dress up the Workbench window.

There is now just a single gadget for superimposing windows: the back/front gadget. Click on it once and the screen or window is brought into the foreground. A second click restores the object to the background. Next to the back/front gadget, a window has a new gadget, by which it can be toggled between two alternate sizes and positions. This is referred to as the alternate gadget.

2.1.1 Starting AmigaOS 2.x

At this point some suggestions concerning the startup of the Workbench may be helpful. Let's begin with the Startup-sequence script file in the S directory. This file contains all the commands and parameters necessary to start the system. Configuring the system to one's own wishes used to require making various changes. Remember if a command results in output to the InitialCLI window, the Workbench screen is opened and the CLI window appears on it. This is not desired, since there is the opening of windows on the Workbench screen before the screen itself is activated. Here's why.

When the LoadWB command wants to open the Workbench, the "workbench.library" attempts to use the stored display mode for the Workbench screen. If the screen is not yet present, there is no problem. If it is, an attempt is made to close it and open a new one in the desired mode. This fails when the screen to be closed contains a CLI or user window.

The result is a system requester requesting that all windows be closed. Let's assume a user is working with the A2024 monitor, which requires a special driver. Suddenly nothing can be seen on the screen, and without an understanding of the system, nothing can be done to solve this problem.

Several things could be done to prevent this situation from occurring. First of all, only those commands that must be executed before the activation of the Workbench should precede the LoadWB command. Secondly, the "Command >NIL: parameter" format should be used to null their output.

Another possibility is offered through the directory WbStartup. All programs (i.e., icons that are located here) are started after activation of the Workbench, just as if they were selected with a double-click of the left mouse button. For example, if you will be working for an extended amount of time with a particular word processing task, you can simply place the icon of the word processor, or the text itself, in this directory. Startup-sequence complications with autostarting programs can be avoided by simply modifying the placement of icons.

2.1.2 The Workbench Menus

Your acquaintance with the Amiga will require you to be familiar with the Workbench menu functions. Even CLI enthusiasts should make thorough use of them, since now the CLI can be entirely replaced by the graphical user interface. Before we proceed with the individual items, we should mention one more innovation regarding the selection of icons. If you press the left mouse button and hold it down while moving the mouse, a rectangular box appears on the Workbench or in a Workbench window. When the left mouse button is released, all the icons within the box are selected, a better procedure than multiple selections using the [Shift] key.

The "Workbench" menu contains items that are independent of file or directory selections:

Backdrop	This item is used to manipulate the Workbench window. Selecting it removes the border, enlarges the window to the full screen size and places it behind all other windows. The former condition is restored when the item is selected again.
Execute Command	Causes a CLI/Shell command to be executed. A requester appears in which a command can be entered the same as in the Shell. A new window is opened for resulting output and can again be closed with the use of a close gadget.
Redraw All	If programs have cluttered or disrupted the workbench screen, you can use this item to restore all windows and icons to their original condition.
Update All	If you are working with the Workbench and the Shell, you can use this item to show changes you have made to directories with the Shell. It updates Workbench memory and redraws the screen to reflect the current status.
Last Message	The last message to appear on the title bar is redisplayed.
About	Displays a requester showing the version numbers of the operating system and Workbench you are using. This also shows the copyright notice.
Quit	This is the same as clicking on the close gadget of the Workbench. If the Workbench is not blocked by any program windows, you can close it after confirming your decision in a requester. This frees up memory for processes such as graphics programs that may have large memory requirements.

The "Window" menu contains items that refer to directories and drives. They affect only the active window:

New Drawer	Makes a new directory and provides an icon for it. The name of the directory can be entered in a requester.
Open Parent	When one directory is located within another directory, which in turn is located within a third, it may be advisable to close the respective parent directories. Selecting this item will again open the directory in which the current window's directory is located.
Close	Closes the current window (directory).
Update	In earlier versions, directory changes that were not applied to the Workbench had to be remade with each close and subsequent reopen of the directory in order to be reflected on the screen. This item provides a simple way of keeping a window's information current.
Select Contents	If you want to work with all entries of a directory, the entire contents can be selected with this item.
Clean Up	Tidies up a window by reorganizing its icons according to the window's size.
Snapshot	Stores the size and position of the current window (submenu item "Window") and the order of all the icons it contains (submenu item "All").
Show	Determines what will be shown in the current window. The submenu item "Only Icons" is the default. This shows only those objects that have an icon file (".info" file). All other entries are also shown when you select the "All Files" submenu item. For example, this enables you to display CLI commands and double-click to start them, whereby a requester appears permitting the input of parameters.
View By	The preset submenu item "Icon" shows the directory contents by icons and, underneath them, the

corresponding filenames. All other options produce a scrollable list of entries without icons. The entries that appear in this list are determined by the "Show" criteria. Their sequence is determined by the three remaining "View By" submenu items. Entries can be sorted by "Name" of file, by "Size", or by "Date" created. Files can be selected from these lists as they can from the display of icons.

The "Icon" menu contains functions relating to icons. The upper portion consists of general activities and the lower portion consists of special icons only.

Open	Opens the selected icon, which is the same as double-clicking on the icon with the mouse.
Copy	Makes a copy of a file, directory or diskette.
Rename	Changes the name of an object.
Information	Opens a large requester in which all data about an icon can be displayed and manipulated.
Snapshot	Saves the position of the selected icon.
Unsnapshot	Deletes position information of icons saved in "Snapshot".
Leave Out	One of the most convenient features of the new Workbench. Selected icons are saved in the main Workbench window. This makes it possible to select the icon again without reopening its directory. The Leave Out configuration is saved and remains in effect even after resetting or turning off the computer.
Put Away	Removes icons placed in the Workbench window by Leave Out and displays them again with their respective directories.
Delete	Deletes all selected icons and their files or directories after confirmation using a requester.

Format Disk — Formats a diskette. The disk is initialized and given the name "Empty". The diskette icon is then displayed.

Empty Trash — Deletes the contents of the Trashcan directory.

The "Tools" menu normally contains only the "ResetWB" function, which returns the entire Workbench to its initial status. This menu was intended for user-defined items. Unfortunately, no utility for incorporating programs into menus is supplied, although the public-domain "ToolManager" (Fish 476) can be used to accomplish this.

2.1.3 The Workbench Programs

Now let's look at the programs that Workbench Version 37 Revision 64 contains. We begin with the "Prefs" directory, since you will find all the programs needed to tailor the system to your needs:

Input — This program establishes all the time constants for interrogating the keyboard and the mouse. With the "Mouse Speed" slider, you control how much the mouse must be moved to cause a corresponding movement of the mouse pointer. A low value indicates that a small movement of the mouse will change the position of the pointer. If this is not adequate, you can click on the "Acceleration" box. A check mark appears in the box when Acceleration is selected. Now the slightest movement of the mouse will cause a large displacement of the pointer. You may have to go back and adjust the Mouse Speed after selecting Acceleration.

The "Double-Click" slider sets the maximum time span that can separate two clicks before they will be recognized separately rather than as a double-click. You can try this out with the "Test" button. If a double-click is recognized, this is indicated in the "Show" box.

"Key Repeat Delay" sets the time after which a key that is struck and not released will be considered struck again.

"Key Repeat Rate" is the speed at which a letter will appear on the screen as repeated input once the Key Repeat Delay is reached and the key continues to be held down. This can be checked in the Key Repeat Test field.

IControl

IControl establishes keyboard commands that take the place of complicated mouse operations. "Verify Timeout" is the timespan that keys must be pressed to activate the corresponding action. "Command Keys" are letter keys that are pressed in combination with the left <Amiga> key to perform certain actions. For example, to move the Workbench into the foreground and the front screen into the background, or to substitute for the "OK" and "Cancel" gadgets of some system requesters. IControl allows you to specify the letters to be used for these actions.

"Mouse Screen Drag" keys are used with the mouse to drag the screen both horizontally and vertically. With IControl you can specify the keys (Shift, Ctrl, Alt and/or <Amiga>) that must be held down along with the left mouse button for this operation. When such keys are paired on the keyboard, the left one should be used.

"Avoid flicker" provides for flicker-free text in special display modes. "Preserve colors" ensures stability and fidelity of color. With "Screen menu snap", menus will always be shown in the visible area of the screen, and with "Text gadget filter", control characters are filtered out of text.

Palette

This allows the colors of the Workbench to be changed. The currently selected color appears in a box to the left of the palette. Below it the red, green and blue intensity of the selected color can be adjusted.

WBpattern

The main Workbench window and its directory windows are displayed with a background pattern. The editor WBpattern lets you choose these patterns from eight preset selections.

Font	Allows selection of the character sets to be used for the text underneath icons, for that displayed in the title bars of screens and windows, and for the default text of the system. You can also specify whether or not the background field behind text characters should be colored. Color selections for text and field are made separately.
Pointer	With this preference you can change the appearance of the mouse pointer and adjust the "hot spot" (i.e., the portion of the pointer capable of activating an object).
ScreenMode	Here the resolution and display mode of the Workbench is established. The Workbench can be made larger than the visible area of the screen. You can specify whether or not the screen display should "Autoscroll" when the selected "Width" and "Height" values exceed the visible screen dimensions. The number of possible colors can also be determined according to screen mode.
OverScan	If you have a monitor on which the border area around the Workbench is visible, then you can also enlarge the Workbench. With "Edit Text Overscan" and "Edit Standard Overscan" you can inform the system of the normal visible area of your monitor and the desired overscan display size. A program can then open a screen that extends beyond the normal text area, so that no border is visible.
Printer	This program allows you to tell the system what type of printer is connected to your computer, through which port it is connected, what kind of paper you are using, and various preferences regarding text output.
PrinterGfx	Here you can define several specifications for the output of graphics to the printer. Next to the "Color Correct" area, you can choose the "Smoothing" option to round sharp edges and offset or centering options. Other specifications include "Dithering," "Scaling," "Image," "Aspect" and "Shade".

Serial — This program sets the data transfer parameters for a modem connected to the serial port. The maximum rate supported is 31250 baud.

Time — This program establishes the date and time and sets the accubuffered truetime clock accordingly.

The "System" directory contains programs that are used primarily by the operating system. The exceptions to this rule are "SetMap," by which you can change the assigned keyboard layout, "NoFastMem," which disables the Fast RAM area, and "FixFonts," which should always be run following changes in the Fonts directory.

In the "Utilities" directory there are a few small programs that perform helpful tasks:

Clock — Displays the time in analog or digital format and has an alarm function.

More — This is a program for reading text files. You can scan through the text within a window one page or line at a time.

Display — Graphics in IFF format and even slideshows can be displayed with this program.

Say — A simple program to convert typed text into computer-synthesized speech.

Exchange and Commodities
This is the main program of an assortment of small utilities. It controls the following programs: "Autopoint" automatically activates the window over which the mouse pointer is located, "Blanker" blanks out the screen when no input has been received for a specified period of time, "FKey" assigns function keys, "IHelp" allows keyboard commands to replace many mouse operations, and "NoCapsLock" forces software disabling of the [Caps Lock] key.

2.2 The Command Line Interpreter

The Shell is a window in which you can enter command lines to control the Amiga. A command line consists of a program name and, in some cases, additional parameters.

2.2.1 AmigaOS 2.x Resident Commands

Unlike in previous versions of the Amiga operating system, under AmigaOS 2.x many programs are stored in ROM. This allows faster processing and trouble-free manipulation of system directories. Some programs stored in ROM are also kept in the current system directory, because programs written for earlier versions expect them there and require them for execution.

The following commands are implemented in ROM:

Alias	Get	Set
Ask	Getenv	Setenv
CD	If	Skip
Echo	Lab	Stack
Else	NewCLI	Unalias
EndCLI	NewShell	Unset
EndIf	Path	Unsetenv
EndShell	Prompt	Why
EndSkip	Quit	.bra
Failat	Resident	.ket
Fault	Run	.key

2.2.2 Using the CLI

The Shell or CLI provides many features to help you edit the current command line:

The left and right cursor control keys move the cursor one character position in the indicated direction. When used in conjunction with the [Shift] key, they move the cursor to the beginning or end of the line respectively.

The Backspace key erases the character to the left of the cursor. The Del key erases the character at the cursor position.

Ctrl+H — This corresponds to Backspace, Ctrl+M to Enter, and Ctrl+J to Help.

Ctrl+W — This erases the previous word.

Ctrl+L — This erases the entire window (works only in combination with Enter).

Ctrl+X — This erases the entire command line.

Ctrl+U — This erases the line from the beginning up to the cursor position.

Ctrl+K — This erases the line from the cursor position to the end.

Ctrl+C,D,E,F — This aborts an executing program, command file, etc.

Previously entered commands are stored. You can scan up or down through this list with the cursor control keys. There is a search function for quickly locating a particular command within a list. Simply type the first few characters of the command you wish to locate. Then press the up or down cursor control key together with the Shift key to begin searching in the desired direction for the first line that begins with the typed sequence.

In the Shell window, blocks of text can be marked with the mouse and with <rt-Amiga> + C as in a word processor. The block can be copied to another window by pressing <rt-Amiga>+V after activating the window that is to receive the text.

If you enter a command in a Shell window that is too small to hold the entire output, the initial lines will scroll off the top of the window and disappear from view. Enlarging the window will cause them to reappear.

3. Programming with AmigaOS 2.x

The basic concept of the operating system has been changed considerably from the old 1.x versions. In just about every area, the programmer is given opportunities to query, influence, or completely determine system processes. The operating system has become much more open, and offers good potential for multi-user systems. Many system routines were re-programmed with new capabilities. In order to maintain compatibility with old software, many of the calling conventions from the 1.x versions were implemented, and sometimes the function results were partially modified. New libraries, resources, and devices were added. The familiar system modules were expanded to the point that they can hardly be recognized anymore. All in all, AmigaOS 2.x is a completely new operating system that is compatible with the old versions.

Normal versions of AmigaOS 2.x can only work on a machine that has the same hardware configuration as the Amiga 3000 (68030, FPU, Commodore clock chip, HR chip set, etc.). This is because the reset routine starts out with 68030 commands without even querying the CPU type. Some test versions can also be installed on the 16 bit machines, but there is a lot less room in the 512K ROM, so many features are only partially functional or are missing altogether.

3.1 The Libraries and their Functions

A lot of information is required to produce a good program. All the data on AmigaOS 2.x would fill thousands of pages and extend far beyond what we could hope to effectively cover in this book. Therefore, we had to limit ourselves to a selected portion. We chose to focus on the library functions in this book. Library routines provide the building blocks and hand tools for creating more complex application programs, such as a word processor. Because there are so many functions to cover, we also chose to do without an introductory overview for beginners. For example, there are many other good books with this kind of information, such as "The Amiga System Programmers Guide".

A brief glance at the system routines will reveal the existence of two new structures = TagItem fields and Hooks. TagItem fields are variable in size and structure. They are primarily used to pass parameters. A Tag field can belong to several memory blocks. It consists of several TagItems. A TagItem consists of two 32 bit values (Longs). The first value is a code for interpreting the meaning of the second value, which is the data Long. Depending on the code, the data can be an address, a BCPL pointer, Words, Bytes, Flags, or combinations thereof. TagItems are most often used to change system routine default settings. This could be for a small change, such as setting the ECS presentation mode for a new screen, or for changes to the basic system configuration that require large numbers of parameters. TagItem fields are required in order to use certain OS-2.04 features.

Another important new structure is the Hook. Hooks give the programmer deep access into the system. In general, Hooks are structures with addresses to routines of their own. These private routines are associated with certain events or results. When a certain event or result is encountered, the system jumps to the corresponding routine. Hooks can be used to expand upon or entirely replace system functions.

And now, the description of each library in alphabetical order.

3.1.1 The ASL Library

The ASL library provides the easiest way for a programmer to create file requester boxes. Special functions can be applied to customize each requester box.

This library is found under the name "asl.library". All functions of this library, expect the base address _AslBase, is a parameter in the A6 register.

Functions of the ASL Library

1. Standard File Requester Box	*2. Complex File Requester Boxes*
AllocFileRequest	AllocAslRequest
FreeFileRequest	FreeAslRequest
RequestFile	AslRequest

Description of Functions

1. Standard File Requester Box

AllocFileRequest	**Get FileRequester structure**

Call:

```
request = AllocFileRequest()
D0        -30(a6)

STRUCT FileRequester *request;
```

Function: Obtains and initializes all data structures required for a RequestFile() function call.

Arguments: None. The initialization is automatically executed for the standard file requester. If you want to use special functions, you must obtain the data structures with AllocAslRequest().

Result: Address of a FileRequester structure which is passed to the RequestFile() function. You can read any data from the normally accessible parts of the FileRequester structure. In the case of a system error, such as no memory, the value 0 is returned.

Warning: FileRequester structures passed to RequestFile() or AslRequest must be obtained either with AllocFileRequest() or AllocAslRequest(). Reserving memory yourself or directly manipulating the entries in the structure will crash the system.

See also: RequestFile(), FreeFileRequest(), FreeAslRequest(), AslRequest()

FreeFileRequest	**Free file requester**

Call:

```
FreeFileRequest( request )
-36(a6)           A0

STRUCT FileRequester *request;
```

Function: This function is identical to Free AslRequest(). It's used to free a data structure allocated with AllocFileRequest.

Arguments: request — Address of a FileRequester structure that was obtained with AllocFileRequest().

Result: None.

See also: FreeAslRequest()

RequestFile — Display file requester and evaluate user input

Call:

```
result = RequestFile( request )
D0        -42(a6)        A0

BOOL   result;
STRUCT FileRequester *request;
```

Function: A file requester box is displayed, the user's input is processed, and the requested file is returned.

Arguments: request — FileRequester structure with address obtained via AllocFileRequest().

Result: result — 0 means Cancel was selected or a system error occurred. The exact input data can be read from the FileRequester structure.

See also: AllocFileRequest(), FreeFileRequest(), AslRequest()

2. Complex Requester Boxes

AllocAslRequest — Obtain structures for a requester box

Call:

```
request = AllocAslRequest( type, ptags )
D0          -48(a6)          D0    A0

APTR    request;
ulong   type;
STRUCT TagItem  *ptags;
```

Function: Obtains and initializes the data structures for a requester box.

Arguments: type — Type of requester box, ASL_FileRequest for a file requester or ASL"FontRequest for a font requester. The type of requester box is determined by AllocAslRequest function on the basis of the following values:

```
ASL_FileRequest = 0
ASL_FontRequest = 1
```

ptags — Address of a TagItem field used to pass special functions and parameters.

Result: Address of an initialized data structure (FileRequester or FontRequester). A value of 0 is returned in case of an error. The address of the data structure is passed to the function AslRequest() and freed with FreeAslRequest().

See also: AslRequest(), FreeAslRequest()

FreeAslRequest — **Free requester box data structures**

Call:

```
FreeAslRequest( request )
-54(a6)         A0

APTR request;
```

Function: Frees the memory occupied by a FileRequester or FontRequester structure. The address must have been previously obtained with AllocAslRequest() or AllocFileRequest().

Arguments: request — Address of a data structure obtained via AllocAslRequest() or AllocFileRequest().

Result: None.

See also: AllocAslRequest(), AslRequest(), AllocFileRequest()

AslRequest	**Display and query requester box**

Call:

```
result = AslRequest( request, ptags )
D0       -60(a6)     A0        A1

BOOL   result;
APTR   request;
STRUCT TagItem  *ptags;
```

Function: Displays a requester box and evaluates the input of the user. The type of box, special functions, and results are dependent upon the data structure and definitions passed to the TagItem field.

TagItems	
ASL_Hail (STRPTR)	Title text of the requester.
ASL_Window (struct Window *)	Window in which the requester will appear.
ASL_LeftEdge (WORD)	Left edge of the query window.
ASL_TopEdge (WORD)	Top edge of the query window.
ASL_Width (WORD)	Width of the query window.
ASL_Height (WORD)	Height of the query window.
ASL_HookFunc (APTR)	Address of an implemented function.
ASL_File (STRPTR)	Default filename of a FileRequester.
ASL_Dir (STRPTR)	Default path of a FileRequester.
ASL_FontName (STRPTR)	Default font name of a FontRequester.
ASL_FontHeight (UWORD)	Default font height.
ASL_FontStyles (UBYTE)	Default font style.
ASL_FontFlags (UBYTE)	Special flags for a FontRequester.
ASL_FrontPen (BYTE)	Foreground color of a FontRequester.
ASL_BackPen (BYTE)	Background color of a FontRequester.
ASL_MinHeight (UWORD)	Minimum height of font.
ASL_MaxHeight (UWORD)	Maximum height of font.
ASL_OKText (STRPTR)	New text for the OK button (up to 6 char.).
ASL_CancelText (STRPTR)	Same for the CANCEL button.
ASL_FuncFlags (ULONG)	Function Flags for the requester.
ASL_ExtFlags1 (ULONG)	Additional Flags.

Arguments:

request — Data structure obtained with AllocAslRequest() or AllocFileRequest().

ptags — Address of a TagItem field containing changes to the default values.

Warning: The only valid way to change the data structure entries is with TagItems.

Result: A result of 0 indicates CANCEL was pressed or a system error occurred. The exact user input can be taken from the readable parts of the data structure.

See also: AllocAslRequest(), FreeAslRequest()

Data Structures And Values:

```
Dec Hex
        STRUCTURE FileRequestr,0
  0 $00 CPTR     rf_Reserved1
  4 $04 CPTR     rf_File          ; *filename (FCHARS+1)
  8 $08 CPTR     rf_Dir           ; *directory (DSIZE+1)
 12 $0C CPTR     rf_Reserved2
 16 $10 UBYTE    rf_Reserved3
 17 $11 UBYTE    rf_Reserved4
 18 $12 APTR     rf_Reserved5
 22 $16 WORD     rf_LeftEdge
 24 $18 WORD     rf_TopEdge
 26 $1A WORD     rf_Width
 28 $1C WORD     rf_Height
 30 $1E WORD     rf_Reserved6
 32 $20 LONG     rf_NumArgs
 36 $24 APTR     rf_ArgList
 40 $28 APTR     rf_UserData
 44 $2C APTR     rf_Reserved7
 48 $30 APTR     rf_Reserved8
 52 $34 CPTR     rf_Pad
```

Interactive functions associated with a requester must look like this:

```
rf_Function( Mask, Object ,AslRequester)
             4(A7) 8(A7)     12(A7)
ULONG        Mask;
CPTR              *Object;
CPTR                          *Request;
```

The value of Mask is determined by passing a copy of ASL_FunctionFlags, which is generated for every requester. Object contains the address of data. The following bits (or Flags) are defined for a FileRequester:

```
RFB_DOWILDFUNC   =   7 ;call with AnchorPath and a name,
RFF_DOWILDFUNC   = $80 ;(FileRequester)
RFB_DOMSGFUNC    =   6 ;transmit all IDCMP events
```

```
RFF_DOMSGFUNC    = $40 ;that are not for the FileRequester
RFB_DOCOLOR      =   5 ;bit for SAVE operations
RFF_DOCOLOR      = $20 ;
RFB_NEWIDCMP     =   4 ;use own IDCMP port
RFF_NEWIDCMP     = $10 ;
RFB_MULTISELECT =   3 ;notify of multiple selection
RFF_MULTISELECT =  $8 ;
RFB_PATGAD       =   0 ;query a Pattern gadget
RFF_PATGAD       =  $1 ;

Dec Hex
        STRUCTURE FontRequester,0
  0 $00 CPTR    fo_Reserved1
  4 $04 CPTR    fo_Reserved2
  8 $08 APTR    fo_Name          ;result string
 12 $0C USHORT  fo_YSize
 14 $0E UBYTE   fo_Style
 15 $0F UBYTE   fo_Flags
 16 $10 UBYTE   fo_FrontPen
 17 $11 UBYTE   fo_BackPen
 18 $12 UBYTE   fo_DrawMode
 19 $13 UBYTE   fo_Reserved3
 20 $14 APTR    fo_UserData
 24 $18 SHORT   fo_LeftEdge
 26 $1A SHORT   fo_TopEdge
 28 $1C SHORT   fo_Width
 30 $1E SHORT   fo_Height
```

ASL_FuncFlags for FontRequester:

```
FONB_FrontColor =   0 ;query foreground color
FONF_FrontColor =  $1 ;
FONB_BackColor  =   1 ;query background color
FONF_BackColor  =  $2 ;
FONB_Styles     =   2 ;query font style
FONF_Styles     =  $4 ;
FONB_DrawMode   =   3 ;query draw mode
FONF_DrawMode   =  $8 ;
FONB_FixedWidth =   4 ;allow only fixed width fonts
FONF_FixedWidth = $10 ;
FONB_NewIDCMP   =   5 ;use own IDCM port
FONF_NewIDCMP   = $20 ;
FONB_DoMsgFunc  =   6 ;capture only events for the requester
FONF_DoMsgFunc  = $40 ;
FONB_DoWildFunc =   7 ;call with every TextAttr structure
FONF_DoWildFunc = $80 ;
```

Values for the TagItem field used with AslRequest():

```
ASL_Dummy       = TAG_USER+$80000
ASL_Hail        = ASL_Dummy+1
ASL_Window      = ASL_Dummy+2
ASL_LeftEdge    = ASL_Dummy+3
ASL_TopEdge     = ASL_Dummy+4
ASL_Width       = ASL_Dummy+5
ASL_Height      = ASL_Dummy+6
ASL_HookFunc    = ASL_Dummy+7
ASL_File        = ASL_Dummy+8
ASL_Dir         = ASL_Dummy+9
ASL_Pattern     = ASL_Dummy+10 ;FileRequester only
ASL_FontName    = ASL_Dummy+10 ;FontRequester only
ASL_FontHeight = ASL_Dummy+11
ASL_FontStyles = ASL_Dummy+12
ASL_FontFlags   = ASL_Dummy+13
ASL_FontPen     = ASL_Dummy+14
ASL_BackPen     = ASL_Dummy+15
ASL_MinHeight   = ASL_Dummy+16
ASL_MaxHeight   = ASL_Dummy+17
ASL_OKText      = ASL_Dummy+18
ASL_CancelText = ASL_Dummy+19
ASL_FuncFlags   = ASL_Dummy+20
ASL_ModeList    = ASL_Dummy+21
```

Example

Let's take a look at the creation of a simple FileRequester and how to query its result. It's rather curious that a simple routine like this does not already exist as a function:

```
**=======================================================**
**                    File selection                     **
**-------------------------------------------------------**
**          Input:   A6=_AslBase                         **
**                   A0=Buffer (FCHARS+DSIZE+1)          **
**          Output:  D0=Buffer r NULL                    **
**                   A6=_AslBase                         **
**                   A0=Buffer                           **
**=======================================================**

_File selection
 clr.b   (a0)                      ;0 bytes in buffer
 movem.l d0/a0,-(a7)               ;result+buffer
 jsr     _LVOAllocFileRequest(a6) ;get FileRequestr
 move.l  d0,(a7)                   ;save result
```

```
 beq.s    .Error                    ;on error ->
 movea.l d0,a0                      ;move FileRequestr to a0
 jsr      _LVORequestFile(a6)       ;display
 movem.l (a7),a0-a1                 ;FileRequestr+buffer
 move.l   a0,d1                     ;save FileRequestr
 move.l   d0,(a7)                   ;test Okay/Cancel
 beq.s    .Cancel                   ;on error ->
 move.l   a1,(a7)                   ;result=buffer
 movea.l rf_Dir(a0),a0              ;directory string
.CopyDir
 move.b   (a0)+,(a1)+               ;copy
 bne.s    .CopyDir
 subq.l   #1,a1                     ;return empty byte
 cmpi.b   #':',-1(a1)               ;check ending
 beq.s    .Okay                     ;if drive ->
 cmpi.b   #'/',-1(a1)               ;check ending
 beq.s    .Okay                     ;if dir ->
 move.b   #'/',(a1)+                ;insert separator byte
.Okay
 movea.l d1,a0                      ;FileRequestr
 movea.l rf_File(a0),a0             ;filename
.CopyFile
 move.b   (a0)+,(a1)+               ;append
 bne.s    .CopyFile
.Cancel
 movea.l d1,a0                      ;
 jsr      _LVOFreeFileRequest(a6)   ;free FileRequestr
.Error
 movem.l (a7)+,d0/a0                ;clear stack
 tst.l    d0                        ;set CCR
 rts
```

This routine can be easily modified to create requesters to serve your own needs.

```
**=====================================================**
**       File selection with a modified requester      **
**-----------------------------------------------------**
**         Input:  A6=_AslBase                         **
**                 A1=Buffer (FCHARS+DSIZE+1)          **
**                 A0=TagItems                         **
**         Output: D0=Buffer or NULL                   **
**                 A6=_AslBase                         **
**                 A0=Buffer                           **
**=====================================================**

_File selection
```

```
 clr.b    (a1)                    ;0 bytes in buffer
 movem.l d0/a1,-(a7)              ;result+buffer
 moveq    #ASL_FileRequest,d0     ;Tags in a0
 jsr      _LVOAllocAslRequest(a6) ;get FileRequestr
 move.l   d0,(a7)                 ;save result
 beq.s    .Error                  ;on error ->
 movea.l d0,a0                    ;move FileRequestr to a0
 jsr      _LVORequestFile(a6)     ;display
 movem.l (a7),a0-a1               ;FileRequestr+buffer
 move.l   a0,d1                   ;save FileRequestr
 move.l   d0,(a7)                 ;test Okay/Cancel
 beq.s    .Cancel                 ;on error ->
 move.l   a1,(a7)                 ;result=buffer
 movea.l rf_Dir(a0),a0            ;directory string
.CopyDir
 move.b   (a0)+,(a1)+             ;copy
 bne.s    .CopyDir
 subq.l   #1,a1                   ;return empty byte
 cmpi.b   #':',-1(a1)             ;check ending
 beq.s    .Okay                   ;if drive ->
 cmpi.b   #'/',-1(a1)             ;check ending
 beq.s    .Okay                   ;if dir ->
 move.b   #'/',(a1)+              ;else insert separator byte
.Okay
 movea.l d1,a0                    ;FileRequestr
 movea.l rf_File(a0),a0           ;filename
.CopyFile
 move.b   (a0)+,(a1)+             ;append
 bne.s    .CopyFile
.Cancel
 movea.l d1,a0                    ;
 jsr      _LVOFreeFileRequest(a6) ;free FileRequestr
.Error
 movem.l (a7)+,d0/a0              ;clear stack
 tst.l    d0                      ;set CCR
 rts
```

The address of a TagItem field is expected as an additional parameter. Here is an example of how this can look:

```
_FileReqTags
 dc.l ASL_Hail,_Titletext        ;title text for the requester.
 dc.l ASL_Dir,_DirName           ;path
 dc.l ASL_OKText,_Okay           ;OK button
 dc.l ASL_CancelText,_Cancel     ;CANCEL button.
 dc.l TAG_DONE                   ;end of field
```

```
_Titletext dc.b 'Load file',0
_Okay      dc.b 'Load',0
_Cancel    dc.b 'Return',0
_DirName   dc.b 'Work:',0
```

3.1.2 The Commodities Library

The utilities found in the Commodities directory of the Workbench are used to manipulate input queries for the A3000. These routines have been gathered into a library. This allows you to add your own expansions to the Commodities utilities.

The name "Commodities Library" is often shortened to Cx library. The base address is expected in register A6 with all function calls.

Functions of the Commodities Library

1. Object Functions

CreateCxObj
CxBroker
ActivateCxObj
DeleteCxObj
DeleteCxObjAll
CxObjType
CxObjError
ClearCxObjError
SetCxObjPri

2. Object Linking

AttachCxObj
EnqueueCxObj
InsertCxObj
RemoveCxObj

3. Special Functions

FindBroker
SetTranslate
SetFilter
SetFilterIX
ParseIX

4. General Messages

CxMsgType
CxMsgData
CxMsgID

5. Message Paths

DivertCxMsg
RouteCxMsg
DisposeCxMsg

6. InputEvent Processing

InvertKeyMap
AddIEvents

7. Control Program Functions

CopyBrokerList
FreeBrokerList)
BrokerCommand

8. Standard Macros

CxFilter
CxTypeFilter
CxSender
CxSignal
CxTranslate
CxDebug
CxCustom

Description of Functions

1. Object Functions

CreateCxObj	**Create Commodities object**

Call:

```
co = CreateCxObj(type, arg1, arg2)
D0   -30(A6)      D0     A0    A1

STRUCT CxObj *co
ULONG  type
LONG   arg1
LONG   arg2
```

Function: Creates a Commodities of type 'type'.

Parameters: type   Object type

args   Object arguments

Result: Address of a CxObj structure, a type of handle for Cx objects. A result of 0 indicates a system error, such as lack of memory.

See also: CxObjError(), CxFilter(), CxTypeFilter(), CxSender(), CxSignal(), CxTranslate(), CxDebug(), CxCustom(), CxBroker()

CxBroker	**Create CxObj of type broker**

Call:

```
broker = CxBroker(nb, error);
D0       -36(A6)  A0  D0

STRUCT CxObj *broker
STRUCT NewBroker *nb
LONG   *error
```

Function: Creates a broker according to the information passed in the NewBroker structure. As opposed to a normal CxObj, a broker is inactive when created.

Parameters:

nb	NewBroker structure used to define the broker.
error	Address of error code or 0.

```
Dec Hex STRUCTURE NewBroker,0
  0  $0 BYTE  nb_Version   ;version 5
  1  $1 BYTE  nb_Pad
  2  $2 APTR  nb_Name      ;Broker name
  6  $6 APTR  nb_Title     ;strings, description of
 10  $A APTR  nb_Descr     ;the application
 14  $E SHORT nb_Unique    ;what happens with a Broker of
                           ;the same name
 16 $10 WORD  nb_Flags
 18 $12 BYTE  nb_Pri       ;priority in the object list
 19 $13 BYTE  nb_Pad2
 20 $14 APTR  nb_Port      ;MsgPort
 24 $18 WORD  nb_ReservedChannel
```

Result: Address of a CxObj structure, or 0 in the case of an error.

If you specify an address in error, the following codes will be used at this address:

CBERR_OK No error, broker was created.

CBERR_SYSERR
System error, such as lack of memory.

CBERR_DUP
Duplicate definition with this name.

CBERR_VERSION
Unknown version number.

See also: Brokers and Application Sub-Trees (in the Reference Manual).

ActivateCxObj	Activate object functions

Call:

```
previous = ActivateCxObj(co, true);
D0         -42(A6)        A0  D0

STRUCT CxObj *co;
BOOL   true;
```

Function: Every Commodities object can be activated and deactivated. If it's active, it executes a specific operation when a Commodities message is received. This function is used to activate and deactivate objects.

Parameters:

co — CxObj structure of the object whose activation you want to control.

true — Boolean argument. A value of 0 indicates inactivation.

Result: previous — Previous status

See also: CxBroker()

DeleteCxObj	Delete Commodities object

Call:

```
DeleteCxObj(co);
-48(A6)     A0

STRUCT CxObj *co;
```

Function: Deletes a selected Commodities object. If this object is part of a list, it's also removed from the list.

If the object has some other underlying substructure(s) in the system hierarchy, then DeleteCxObjAll() must be used.

Parameters: co — CxObj

Result: None. Invalid parameter may cause system crash.

See also: exec.library/Remove(), DeleteCxObjAll()

DeleteCxObjAll
Delete Commodities object and all underlying substructures

Call:

```
DeleteCxObjAll(co);
-54(a6)        a0

STRUCT CxObj *co;
```

Function: Deletes a selected Commodities object. If the object is part of a list, it's also removed from the list.

If the object has some other underlying substructure(s) in the system hierarchy, they are also deleted.

Parameters: co   CxObj structure of any type.

Result: None. Improper use of this function will crash the system.

See also: exec.library/Remove(), DeleteCxObj()

CxObjType **Get object type**

Call:

```
type = CxObjType(co);
D0     -60(A6)    A0

ULONG  type
STRUCT CxObj *co;
```

Function: Returns the object type for a selected Commodities object. The CxObj must be known, but you will normally only have this information for your own objects. That makes this function rather meaningless.

Parameters: co   CxObj structure

Result: Object type. If you pass the value 0 as the parameter, the result is type CX_INVALID. This function only reads a data structure. If you enter the wrong parameter value, the result will be meaningless.

See also: CreateCxObj(), CxBroker()

CxObjError **Get error code**

Call:

```
error = CxObjError(co);
D0      -66(A6)    A0

LONG    error
STRUCT CxObj *co;
```

Function: When a function fails, the cause of the error is encoded in various different bits. CxObjError() gives you access to read these bits.

Parameters: co CxObj structure

Result: A longword where the set bits have the following meanings:

COERR_ISNULL
A value of 0 was passed for CxObj.

COERR_NULLATTACH
Attempt to enter a non-existent object in a Commodities list.

COERR_BADFILTER
Bad filter string.

COERR_BADTYPE
A type-specific function was attempted on an object of the wrong type.

See also: SetFilter(), SetFilterIX(), AttachCxObj(), ActivateCxObj(), ClearCxObjError()

ClearCxObjError **Delete error number of a Cx object**

Call:

```
ClearCxObjError(co);
-72(A6)          A0

STRUCT CxObj *co;
```

Function: Deletes the error code of a Commodities object.

Parameters: co CxObj structure

Result: None.

Warning: This routine may not be used with filter objects if the error bit COERR_BADFILTER is set.

See also: CxObjError()

SetCxObjPri — **Change priority of a Cx object**

Call:

```
SetCxObjPri(co, pri)
-78(A6)      A0   D0

STRUCT CxObj *co;
LONG   pri;
```

Function: This function sets the priority of an object that was entered in a list with EnqueueCxObj(). The mechanism corresponds to that of the Exec Lists System.

Parameters: co CxObj structure

pri Priority (127 through -128)

Result: None.

See also: ToolTypes and the Commodities Environment (in the Reference Manual), EnqueueCxObj()

2. Object Linking

AttachCxObj — **Attach object to a head object**

Call:

```
AttachCxObj(headobj, co);
-84(A6)      A0        A1

STRUCT CxObj *headobj
STRUCT CxObj *co
```

Function: Attaches an object to the end of the list of another object.

Parameters: headobj — CxObj structure of the head object to which this object will be attached.

co — CxObj structure of the object to be attached as a sub-object.

Result: If co is 0, then the error is noted in headobj. This can be queried with CxObjError() and cleared with ClearCxObjError().

See also: exec.library/AddTail(), Objects and Messages (in the Reference Manual), CxObjError(), ClearCxObjError()

EnqueueCxObj — Enter object as a sub-object

Call:

```
EnqueueCxObj(headobj, co);
-90(A6)      A0       A1

STRUCT CxObj *headobj
STRUCT CxObj *co
```

Function: Enters an object in the list of another object according to its priority.

Parameters: headobj — CxObj structure of the head object that possesses the sub-object list.

co — CxObj structure of the object to be entered in the sub-object list.

Result: If co has a value of 0, the error is noted in headobj. This can be queried with CxObjError() and cleared with ClearCxObjError().

See also: exec.library/Enqueue(), SetCxObjPri(), Objects and Messages (in the Reference Manual), CxObjError(), ClearCxObjError()

InsertCxObj — Insert an object in front of another object

Call:

```
InsertCxObj(headobj, co, pred);
-96(A6)     A0       A1  A2
```

```
STRUCT CxObj *headobj
STRUCT CxObj *co
STRUCT CxObj *pred
```

Function: The object co is inserted as a sub-object, in the list of object headobj, in front of sub-object pred.

Parameters:

headobj	CxObj structure that possesses the sub-object list.
co	Object to be entered in the list.
pred	Sub-object in front of which co is inserted.

Result: If co has a value of 0, the error is noted in headobj. This can be queried with CxObjError() and cleared with ClearCxObjError().

Warning: Since the Exec function Insert() needs the list header, the headobj may not be 0 in cases where pred is 0.

See also: exec.library/Insert(), Objects and Messages (in the Reference Manual), CxObjError(), ClearCxObjError()

RemoveCxObj — Remove an object from a list

Call:

```
RemoveCxObj(co);
-102(a6)    A0

STRUCT CxObj *co
```

Function: Removes a Commodities object from a selected list. This function will not crash if you pass it a value of 0 or the value of an object not found in the list.

Parameters: co — CxObj structure of the object to be removed.

Result: None.

Warning: This routine was not intended to remove a broker from the master list.

See also: Objects and Messages (in the Reference Manual)

3. Special Functions

FindBroker	**Find the broker with a given name**

Call:

```
broker=FindBroker(name)
D0      -108(A6)    A0

STRUCT CxObj *broker
APTR   name
```

Function: Returns the address of a broker when you know its name.

Parameters: name Address of the name string.

Result: broker CxObj structure of the broker or 0.

See also: exec.library/Find function

SetTranslate	**Replace the translation list**

Call:

```
Error = SetTranslate(translator, ie);
D0      -114(a6)      A0            A1

LONG Error
STRUCT CxObj *translator
STRUCT IX *ie
```

Function: Replaces the translation list of a translator object with the list at address ie. If a value of 0 is passed for ie, then all events are taken. The InputEvents are copied to Commodities messages during the translation.

Parameters: translator CxObj structure of a translator object.

ie InputEvent list

Result: 0 if the function was successfully executed.

See also: Input.Device/InputEvent, CxTranslate()

SetFilter — **Set pattern matching for a filter object**

Call:

```
SetFilter(filter, text);
-120(A6)  A0      A1

STRUCT CxObj *filter
APTR   text
```

Function: Sets the pattern matching according to the pattern string passed in text.

Parameters:

filter	CxObj structure of a filter object.
text	Pattern string

Result: None. A bad filter error can be queried with CxObjError() (COERR_BADFILTER).

See also: SetFilterIX(), CxObjError(), Commodities Input Messages and Filters, Input Expressions and Description Strings (in the Reference Manual)

SetFilterIX — **Set pattern matching of a filter object**

Call:

```
error = SetFilterIX(filter, ix);
D0      -126(A6)  A0      A1

STRUCT CxObj *filter
STRUCT IX *ix
```

Function: Sets the pattern matching according to the contents of the Input Expression structure.

Parameters:

filter	CxObj structure of a filter object.
ix	Input Expression structure

Result: error — 0 or error number

See also: SetFilter(), CxObjError(), Commodities Input Messages and Filters, Input Expressions and Description Strings (in the Reference Manual)

ParseIX — **Translate string with IX structure**

Call:

```
failurecode = ParseIX(string, ix);
D0            -132(A6) A0      A1

LONG   failurecode
APTR   string
STRUCT IX *ix
```

Function: Translates the parts of a given string to an IX structure.

Parameters: string — The string to be processed.

ix — Input Expression structure

Result: 0 if no error occurred.

See also: Input Expressions and Description Strings (in the Reference Manual)

4. General Messages

CxMsgType — **Query Commodities message type**

Call:

```
type = CxMsgType(cxm)
D0     -138(A6)  A0

ULONG  type
STRUCT CxMsg *cxm
```

Function: Returns the Commodities message type.

Parameters: cxm — Address of a Commodities message.

Result: Message type, 0 in the case of an invalid message.

See also: CxMsgData(), CxMsgID()

CxMsgData — **Obtain the data address for a CxMsg**

Call:

```
contents = CxMsgData(cxm);
D0         -144(A6)  A0
```

```
APTR    contents
STRUCT CxMsg *cxm
```

Function: Most Commodities messages contain data, for example an InputEvent structure. CxMsgData() can be used to return a pointer to this data.

Parameters: cxm Address of a CxMsg.

Result: Address of the data; 0 in the case of an invalid message.

Warning: If a message is received from a sender object, the address cannot be used after the reply is made.

See also: CxSender(), CxCustom()

CxMsgID	**Obtain the source identification of a CxMsg**

Call:

```
id = CxMsgID(cxm);
D0   -150(A6) A0

LONG   id
STRUCT CxMsg *cxm
```

Function: Returns the source identification code specified by an application for a message.

Parameters: cxm Address of a CxMsg.

Result: ID of the message; 0 if the message has no ID.

See also: CxSender(), CxCustom()

5. Message Paths

DivertCxMsg	**Send a message to a sub-object**

Call:

```
DivertCxMsg(cxm, headobj, returnobj)
-156(A6)     A0   A1       A2

STRUCT CxMsg *cxm
```

```
STRUCT CxObj *headobj
STRUCT CxObj *returnobj
```

Function: Sends a CxMsg to objects in the sub-object list of a Commodity object. The message is sent on down the list until the next object is the specified returnobj. For example, a Filter object (named 'Filter' for the sake of this example) would send a message to its sub-objects as follows: DivertCxMsg(cxm,Filter,Filter).

Parameters:

cxm	CxMsg structure to be sent.
headobj	Head object that owns the sub-objects that will receive the message.
returnobj	SUCC object that indicates the last sub-object in the chain.

Result: None.

See also: The Reference Manual

RouteCxMsg — Set the next destination for a message

Call:

```
RouteCxMsg(cxm, co)
-162(A6)    A0   A1

STRUCT CxMsg *cxm
STRUCT CxObj *co
```

Function: Determines the next object that will receive the message.

Parameters:

cxm	CxMsg to be sent.
co	CxObj that will be the next object to receive the message.

Result: None.

See also: DivertCxMsg()

DisposeCxMsg	Delete a message

Call:

```
DisposeCxMsg(cxm)
-168(A6)     A0

STRUCT CxMsg *cxm
```

Function: Deletes the specified Commodities message. This is good for disposing of InputEvents (type CXM_IEVENT).

Parameters: cxm — Address of the CxMsg.

Result: None.

6. InputEvent Processing

InvertKeyMap	Convert ANSI codes

Call:

```
retval = InvertKeyMap(ansicode, ie, km)
D0       -174(A6)     D0        A0  A1

ULONG  retval
ULONG  ansicode
STRUCT InputEvent *ie
STRUCT KeyMap *km
```

Function: The MapANSI() function determines whether an ANSI code conversion should take place when an InputEvent is received. The given KeyMap is used. Simple DeadKeys are converted.

Parameters:

ansicode	ANSI code to be checked.
ie	InputEvent structure to be filled.
km	KeyMap, default = 0

Result:

0	No conversion

See also: InvertString()

AddIEvents	**Add a list of InputEvents to the Cx list**

Call:
```
AddIEvents(ie)
-180(A6)   A0

STRUCT InputEvent *ie;
```

Function: Normally, the Commodities Library Input Handler gets its information directly from the input device. But it would not be convenient to send messages to the Commodities Library via this device. Therefore, AddIEvents was implemented. The InputEvents are copied to the Commodities messages and sent to the objects in the internal object list.

Parameters: ie Linked list of InputEvents.

Result: None.

See also: FreeIEvents()

7. Control Program Functions

CopyBrokerList	**Copy the broker list**

Call:
```
list = CopyBrokerList(blist)
D0     -186(A6)        A0
```

Warning: FOR CONTROL PROGRAMS ONLY!

FreeBrokerList	**Free broker list**

Call:
```
FreeBrokerList(list)
-192(A6)       A0
```

Warning: FOR CONTROL PROGRAMS ONLY!

BrokerCommand	**Broker command**

Call:
```
result = BrokerCommand(name,id)
D0       -198(A6)      A0   D0
```

Warning: FOR CONTROL PROGRAMS ONLY!

8. Standard Macros

Creation of CxObj:

```
CxFilter(d)          CreateCxObj(CX_FILTER,d,0)
CxTypeFilter(type)   CreateCxObj(CX_TYPEFILTER,type,0)
CxSender(port,id)    CreateCxObj(CX_SEND,port,id)
CxSignal(task,sig)   CreateCxObj(CX_SIGNAL,task,sig)
CxTranslate(ie)      CreateCxObj(CX_TRANSLATE,ie,0)
CxDebug(id)          CreateCxObj(CX_DEBUG,id,0)
CxCustom(action,id)  CreateCxObj(CX_CUSTOM,action,id)
```

Buffer size of Broker:

```
CBD_NAMELEN  = 24
CBD_TITLELEN = 40
CBD_DESCRLEN = 40
```

CxBroker() error:

```
CBERR_OK      = 0 ;no error
CBERR_SYSERR  = 1 ;system error
CBERR_DUP     = 2 ;duplicate definition
CBERR_VERSION = 3 ;unknown version

NB_VERSION    = 5 ;version of NewBroker

Dec Hex STRUCTURE NewBroker,0
  0  $0 BYTE  nb_Version  ;version 5
  1  $1 BYTE  nb_Pad
  2  $2 APTR  nb_Name     ;Broker name
  6  $6 APTR  nb_Title    ;strings, description of the
 10  $A APTR  nb_Descr    ;application
 14  $E SHORT nb_Unique   ;what happens with a Broker of the
                          ;same name
 16 $10 WORD  nb_Flags
 18 $12 BYTE  nb_Pri      ;priority in the Object list
 19 $13 BYTE  nb_Pad2
 20 $14 APTR  nb_Port     ;MsgPort
 24 $18 WORD  nb_ReservedChannel
```

Flags for nb_Unique:

```
NBU_DUPLICATE = 0 ;duplicate definition allowed
NBU_UNIQUE    = 1 ;duplicate definition not allowed
```

```
NBU_NOTIFY     = 2 ;CxMsg CXM_UNIQUE to existing Broker
```

Flag for nb_Flags:

```
COF_SHOW_HIDE = 4
```

Object types:

```
CX_INVALID    = 0 ;no object
CX_FILTER     = 1 ;for InputEvent messages only
CX_TYPEFILTER = 2 ;message Type filter
CX_SEND       = 3 ;message sender
CX_SIGNAL     = 4 ;signal sender
CX_TRANSLATE  = 5 ;InputEvent translator
CX_BROKER     = 6 ;most applications
CX_DEBUG      = 7 ;sends Debug info to serial port
CX_CUSTOM     = 8 ;custom Function
CX_ZERO       = 9 ;last entry
```

Message Types:

```
CXM_UNIQUE    = 16 ;from CxBroker()
CXM_IEVENT    = 32 ;InputEvent
CXM_COMMAND   = 64 ;from BrokerCommand()
```

ID Values:

```
CXCMD_DISABLE    = 15 ;deactivate
CXCMD_ENABLE     = 17 ;activate
CXCMD_APPEAR     = 19 ;open window
CXCMD_DISAPPEAR  = 21 ;close window
CXCMD_KILL       = 23 ;remove
CXCMD_UNIQUE     = 25 ;duplicate definition attempted
CXCMD_LIST_CHG   = 27 ;Broker list changed
```

Results of BrokerCommand():

```
CMDE_OK        =  0
CMDE_NOBROKER  = -1
CMDE_NOPORT    = -2
CMDE_NOMEM     = -3
```

Error Flags from CxObj (CxObjError()):

```
COERR_ISNULL      = 1 ;call was CxError(NULL)
COERR_NULLATTACH  = 2 ;sub-object was 0
COERR_BADFILTER   = 4 ;invalid Filter
```

```
COERR_BADTYPE     = 8 ;invalid object type
```

Version of IX Structure:

```
IX_VERSION = 2

Dec Hex
        STRUCTURE IX,0
  0  $0 UBYTE ix_Version  ;version 2
  1  $1 UBYTE ix_Class    ;Event class
  2  $2 UWORD ix_Code     ;Event data
  4  $4 UWORD ix_CodeMask ;data mask
  6  $6 UWORD ix_Qualifier;exact description
  8  $8 UWORD ix_QualMask ;QualSame mask
 10  $A UWORD ix_QualSame ;Qualifier with the same meaning
```

Flags for ix_QualSame:

```
IXSYM_SHIFT = 1 ;left and right Shift keys together
IXSYM_CAPS  = 2 ;Caps-Lock at the same time
IXSYM_ALT   = 4 ;left and right Alt keys together
```

Corresponding QualMasks (see InputEvent):

```
IXSYM_SHIFTMASK = IEQUALIFIER_LSHIFT | IEQUALIFIER_RSHIFT
IXSYM_CAPSMASK  = IXSYM_SHIFTMASK    | IEQUALIFIER_CAPSLOCK
IXSYM_ALTMASK   = IEQUALIFIER_LALT   | IEQUALIFIER_RALT

IX_NORMALQUALS  = $7FFF ;normal QualMask
```

3.1.3 The Diskfont Library

The Diskfont Library manages fonts and font styles, lists the available fonts, or loads a font in memory (if it is not already loaded).

This library is opened under the name "diskfont.library". The base address _DiskfontBase must be supplied in the A6 register with all function calls.

Functions of the Diskfont Library

OpenDiskFont
AvailFonts
NewFontContents
DisposeFontContents

NewScaledDiskFont

Description of the Functions

OpenDiskFont	**Load or scale a Diskfont**

Call:

```
font = OpenDiskFont(textAttr)
D0     -30(A6)       A0

STRUCT TextFont *font
STRUCT TextAttr *textAttr
```

Function: The font described in the TextAttr structure is loaded in memory and its address is returned. If desired, the font is scaled to the requested size.

Parameters: textAttr — TextAttr structure or TTextAttr structure that describes the font.

Result: Address of the font (TextFont structure) or 0 if the font was not found. If only the desired font size was not found and the DESIGNED flag in the TextAttr structure is not set, then the font of the desired size is created from a different size.

Warning: This routine will only work with font names up to 30 characters long.

See also: AvailFonts(), graphics.library/OpenFont()

AvailFonts	**Retrieve a list of all available fonts**

Call:

```
error = AvailFonts(buffer, bufBytes, flags);
D0      -36(A6)    A0      D0        D1

LONG   error
STRUCT AFH *buffer
LONG   bufBytes
ULONG  flags
```

Function: AvailFonts() fills a memory block of the specified size and address with font data structures. This gives the user a list of all available fonts. Certain flags can be set to indicate where to look for fonts, which fonts are stored in this memory block, and which data structures to use.

Parameters:

buffer — Address of the memory block that will contain the font list.

bufbytes — Size of the memory block.

flags — Flags for setting the AvailFonts() options.

AFF_MEMORY	Look for fonts in memory.
AFF_DISK	Look for fonts in current FONTS directory.
AFF_SCALED	Include constructed fonts in the list.
AFF_TAGGED	Fill memory with TAF (TaggedAvailFonts) structures rather than AF structures.

Result: If the buffer is too small, the number of bytes missing will be returned in error; otherwise a value of 0 is returned. If 0 is returned, the buffer is filled with an AFH structure, followed by AF or TAF structures. Memory resident fonts must be opened with OpenFont() and Diskfonts must be opened with OpenDiskFont().

Warning: If a certain font is located both in memory as well as on disk, its name will appear in the list twice.

See also: OpenDiskFont(), graphics.library/OpenFont()

NewFontContents **Recalculate xxx.font file**

Call:

```
fontContentsHeader = NewFontContents (fontsLock,fontName)
D0                   -42(A6)          A0        A1
STRUCT FontContentsHeader *fontContentsHeader
BPTR   fontsLock
APTR   fontName
```

Function: Recalculates an array with FontContents. This array begins with an FCH structure, followed by an FC structure for each size of the font with the given name. This structure corresponds to the file 'xxx.font' in the FONTS directory.

Parameters: fontsLock BPTR to a lock structure of the DOS Library. This lock must be obtained for the directory that contains the font (normally the "FONTS:" directory).

fontName Address of the font name (including the suffix ".font") which points to a FontContents file.

Result: Address of the FCH structure (FontContentsHeader) or 0, in the case of an error.

See also: DisposeFontContents(), dos.library/Lock()

DisposeFontContents **Free xxx.font buffer**

Call:

```
DisposeFontContents(fontContentsHeader)
-48(A6)             A1
STRUCT FontContentsHeader *fontContentsHeader
```

Function: Frees the buffer returned with the function NewFontContents().

Parameters: fontContentsHeader
Structure obtained with NewFontContents().

Result: None.

Warning: The system will crash if you pass an address not obtained with NewFontContents().

See also: NewFontContents()

NewScaledDiskFont — Create scaled (constructed) font

Call:

```
header = NewScaledDiskFont(srcFont, destTextAttr)
D0       -54(A6)           A0       A1

STRUCT DiskFontHeader *header
STRUCT TextFont *srcFont
STRUCT TTextAttr *destTextAttr
```

Function: Calculates a new font size based on an existing size for the given font.

Parameters: srcFont — Font for which the new size will be calculated.

destTextAttr — Attributes of the new font. This can be the address of a TextAttr structure or the address of a TTextAttr structure. The new font can be freed with StripFont() followed by UnloadSeg(). TextFont and Segment Address are components of the returned DiskFontHeader. UnloadSeg() frees all memory blocks.

Result: Address of a DiskFontHeader structure.

Warning: This function can use the blitter. Fonts with characters drawn completely outside of the normal character region cannot be processed.

See also: graphics.library/StripFont(), dos.library/UnloadSeg()

```
MAXFONTPATH = 256 ;maximum length of the font path including null byte
Dec  Hex
         STRUCTURE  FC,0
  0   $0 STRUCT   fc_FileName,MAXFONTPATH ;font name
256 $100 UWORD    fc_YSize                ;font height
258 $102 UBYTE    fc_Style                ;style
259 $103 UBYTE    fc_Flags                ;font type
260 $104 LABEL    fc_SIZEOF
```

```
        STRUCTURE  TFC,0
  0  $0 STRUCT  tfc_FileName,MAXFONTPATH-2 ;font name
        ;if the following Word contains a non-zero value,
        ;then the TagItems will be found at the end of tfc_FileName
        ;that is, at MAXFONTPATH-tfc_TagCount*TagItem_SIZEOF
254  $FE UWORD  tfc_TagCount ;number of tags including TAG_DONE
256 $100 UWORD  tfc_YSize    ;font height
258 $102 UBYTE  tfc_Style    ;style
259 $103 UBYTE  tfc_Flags    ;font type
260 $104 LABEL  tfc_SIZEOF

FCH_ID   = $f00  ;FontContentsHeader, then FontContents
TFCH_ID  = $f02  ;FontContentsHeader, then TFontContents

Dec Hex
       STRUCTURE  FCH,0          ;FontContentsHeader
  0  $0 UWORD   fch_FileID       ;FCH_ID or TFCH_ID
  2  $2 UWORD   fch_NumEntries   ;number of (T)FontContents
  4  $4 LABEL   fch_FC           ;starting here, [T]FontContents

DFH_ID      = $f80
MAXFONTNAME = 32   ;font name including ".font" and null byte

Dec Hex
       STRUCTURE  DiskFontHeader,0
;the following Longs are not part of the structure,
;but they precede it directly:
;-8 -$8 ULONG   dfh_NextSegment      ;BPTR to the next segment
;-4 -$4 ULONG   dfh_ReturnCode       ;actually MOVEQ #0,D0 : RTS
  0  $0 STRUCT  dfh_DF,LN_SIZE       ;node
 14  $E UWORD   dfh_FileID           ;DFH_ID
 16 $10 UWORD   dfh_Revision         ;revision number
 18 $12 LONG    dfh_Segment          ;segment address
 22 $16 STRUCT  dfh_Name,MAXFONTNAME ;the name
 54 $36 STRUCT  dfh_TF,tf_SIZEOF     ;TextFont
        LABEL   dfh_SIZEOF
```

If the FSB_TAGGED bit is set in dfh_TF.tf_Style:

```
dfh_TagList = dfh_Segment ;overwritten during loading
```

Bits and Flags of the AvailFonts structure:

```
AFB_MEMORY = 0 ;memory font
AFF_MEMORY = 1
AFB_DISK   = 1 ;disk font
AFF_DISK   = 2
```

```
AFB_SCALED = 2 ;constructed font (not DESIGNED!)
AFF_SCALED = 4
```

Bits and Flags of the TaggedAvailFonts structure:

```
AFB_TTATTR = 15    ;INVALID VALUE IN INCLUDES!!!
AFF_TTATTR = $8000

Dec Hex
        STRUCTURE  AF,0              ; AvailFonts
  0  $0 UWORD    af_Type             ; MEMORY, DISK, or SCALED
  2  $2 STRUCT   af_Attr,ta_SIZEOF   ; TextAttr
 10  $A LABEL    af_SIZEOF

        STRUCTURE  TAF,0               ; TAvailFonts
  0  $0 UWORD    taf_Type              ; MEMORY, DISK, or SCALED
  2  $2 STRUCT   taf_Attr,tta_SIZEOF   ; TTextAttr
 10  $A LABEL    taf_SIZEOF

        STRUCTURE  AFH,0          ; AvailFontsHeader
  0  $0 UWORD    afh_NumEntries   ; number of elements
  2  $2 LABEL    afh_AF           ; starting here, [T]AvailFonts
```

Example

You can make it difficult on yourself and create a special font for each application, or you can handle it quite easily. We will now create a font similar to the Diamond font, but with a character height of only 10 pixels.

```
 ...
 movea.l _DiskfontBase,a6
 lea     _TextAttr(pc),a0
 jsr     _LVOOpenDiskFont(a6) ;Font=OpenDiskFont(TextAttr)
 move.l  d0,_Diamond10
 beq     _Fehler
 ...
 movea.l _GfxBase,a6
 movea.l _Diamond10,a1
 jsr     _LVOCloseFont(a6)    ;CloseFont(Font)
 ...
_TextAttr dc.l _FontName ;ta_Name
          dc.w 10        ;ta_Size
                         ;ta_Style,ta_Flags
          dc.b FS_NORMAL,FPF_PROPORTIONAL!FPF_DISKFONT

_FontName dc.b 'diamond.font',0
```

Simple, isn't it? The change in size takes only a fraction of a second, so it does not add any appreciable time to the process.

3.1.4 The DOS Library

The DOS Library is completely new and expanded for Kickstart Version 2.0. The DOS Library was written in the compiler language BCPL for the old 1.x versions. This slow-executing language was replaced with faster C code, but in order to maintain compatibility, the BCPL variable management had to be kept for the most part. BCPL manages addresses in numbers of longwords (32 bits = 4 bytes), so the address 40 would be assigned the number 10 in BCPL. This is why every address must be divisible by 4.

An important change came with the transition to C. Starting with OS 2.0, DOS expects the base address of the DOS Library to be passed in register A6. This prevents the use of faster code by placing the base address in A5. Programs that utilize this will crash under Kickstart 2.0.

Functions of the DOS Library

1. DOS Structures

AllocDosObject
DupLock
DupLockFromFH
FreeDosEntry
FreeDosObject
MakeDosEntry

2. Logical Devices

AddDosEntry
AssignAdd
AssignLate
AssignLock
AssignPath
AttemptLockDosList
FindDosEntry
LockDosList
NextDosEntry
RemDosEntry
UnLockDosList

3. Handlers and Filesystems

AddBuffers
DeviceProc
DoPkt
EndNotify
Format
FreeDeviceProc
GetConsoleTask
GetDeviceProc
GetFileSysTask
Inhibit
IsFileSystem
Relabel
ReplyPkt
SendPkt
SetConsoleTask
SetFileSysTask
StartNotify

WaitPkt

4. Directories

CreateDir
CurrentDir
ExAll
Examine
ExNext
GetProgramDir
Info
MatchEnd
MatchFirst
MatchNext
ParentDir
ParentOfFH

5. Programs

AddSegment
CreateNewProc
CreateProc
Exit
FindSegment
InternalLoadSeg
InternalUnLoadSeg
LoadSeg
NewLoadSeg
RemSegment
RunCommand
UnLoadSeg

6. CLI

CheckSignal
Cli
Execute
FindCliProc
Input
MaxCli
Output
ReadArgs
ReadItem
SelectInput
SelectOutput
SetArgStr
SetCurrentDirName
SetProgramDir
SetProgramName
SetPrompt
SystemTagList
VPrintf

7. Files

ChangeMode
Close
DeleteFile
ExamineFH
FGetC
Flush
FPutC
FRead
FWrite
IsInteractive
Lock
LockRecord
LockRecords
Open
OpenFromLock
Read
Rename
SameLock
Seek
SetComment
SetFileDate
SetFileSize
SetProtection
UnGetC
UnLock
UnLockRecord
UnLockRecords
VFPrintf

VFWritef
Write

8. Strings

AddPart
DateToStr
Fault
FGets
FilePart
FindArg
FPuts
GetArgStr
GetCurrentDirName
GetProgramName
GetPrompt
MatchPattern
NameFromFH
NameFromLock
ParsePattern

PathPart
SplitName

StrToDate
StrToLong

9. Time

CompareDates
DateStamp
Delay
WaitForChar

10. Environment Variables

DeleteVar
FindVar
GetVar
SetVar

11. Errors and Requesters

ErrorReport
IoErr
PrintFault
PutStr
SetIoErr

Description of Functions

1. DOS Structures

AllocDosObject	**Create DOS data structure**

Call:

```
ptr = AllocDosObject(type, tags)
D0    -228(A6)       D1    D2

APTR   ptr
ULONG  type
STRUCT TagItem *tags
```

Function: Creates one of several possible DOS structures.

Parameters:

type Structure type

tags TagList address

Result: Data structure address or 0.

See also: FreeDosObject(), dos/dostags.h, dos/dos.h

Example: Creating a control structure for calling the new ExAll() function:

```
 movea.l _DosBase,a6
 moveq   #DOS_EXALLCONTROL,d1
 move.l  #_Dummy,d2             ;save it
 jsr     _LVOAllocDosObject(a6)
 move.l  d0,_ExAllControl
 beq     _Error

_Dummy   dc.l TAG_DONE          ;empty TagItem field
```

DupLock **Copy lock**

Call:

```
newlock = DupLock( lock )
D0           -96(A6)   D1

BPTR newlock
BPTR lock
```

Function: Copy a Filesystem SHARED_LOCK.

Parameters: lock Lock to be copied.

Result: Copy of the lock or 0.

See also: Lock(), UnLock()

DupLockFromFH **Copy a FileHandle lock**

Call:

```
lock = DupLockFromFH(fh)
D0     -372(A6)       D1

BPTR lock
BPTR fh
```

Function: Returns a copy of a FileHandle lock. The file must be open and accessible to other programs.

Parameters: fh FileHandle that owns the lock to be copied.

Result: Lock or 0, in the case of an error.

FreeDosEntry Free a structure created with MakeDosEntry()

Call:
```
FreeDosEntry(dlist)
-702(A6)     D1

STRUCT DosList *dlist
```

Function: Frees the result of a MakeDosEntry() call. This routine should not be used. Instead, use FreeDosObject() with the corresponding value.

Parameters: dlist DosList structure to be freed.

FreeDosObject Free a DOS structure

Call:
```
FreeDosObject(type, ptr)
-234(A6)       D1    D2

ULONG type
APTR  ptr
```

Function: Frees a structure created with AllocDosObject().

Parameters: type Type as specified with AllocDosObject().

ptr Result of AllocDosObject().

See also: AllocDosObject(), dos/dos.h

Example: Free an ExAll() control structure:

```
movea.l _DosBase,a6
moveq   #DOS_EXALLCONTROL,d1
move.l  _ExAllControl,d2
jsr     _LVOFreeDosObject(a6)
```

MakeDosEntry **Create a DosList structure**

Call:

```
newdlist = MakeDosEntry(name, type)
D0          -696(A6)     D1    D2

STRUCT DosList *newdlist
APTR   name
LONG   type
```

Function: Creates a DosList structure with BSTR dol_Name and dol_Type. This function should not be used. Instead, use AllocDosObject().

Parameters: name Name of the device/volume/assign node.

type Entry type

Result: DosList structure or 0.

Type for AllocDosObject():

```
DOS_FILEHANDLE    = 0  ;FileHandle
DOS_EXALLCONTROL  = 1  ;ExAllControl
DOS_FIB           = 2  ;FileInfoBlock
DOS_STDPKT        = 3  ;Standard Packet
DOS_CLI           = 4  ;CommandLineInterface
DOS_RDARGS        = 5  ;in case arguments were entered
```

Tags for AllocDosObject():

```
ADO_Dummy        = TAG_USER+2000
ADO_FH_Mode      = ADO_Dummy+1 ;for FileHandle only
ADO_DirLen       = ADO_Dummy+2 ;size of CurrentDir buffer
ADO_CommNameLen  = ADO_Dummy+3 ;size of CommandName buffer
ADO_CommFileLen  = ADO_Dummy+4 ;size of BatchFile buffer
ADO_PromptLen    = ADO_Dummy+5 ;size of Prompt buffer
```

2. Logical Devices

AddDosEntry **Add an entry to the list of logical devices**

Call:

```
success = AddDosEntry(dlist)
D0         -678(A6)     D1
```

```
BOOL   success
STRUCT DosList *dlist
```

Function: Adds a device, volume, or assign node to the DOS list of logical devices. If a logical device of the same name already exists, the function will fail. Exceptions to this are volumes nodes with different dates and DeviceNode names. This function can be called without a lock on the device list.

Parameters: dlist — Entry for the device list.

Result: 0 — Error

AssignAdd — Add a path to a directory with many paths

Call:

```
success = AssignAdd(name,lock)
D0        -630(A6)  D1   D2

BOOL success
APTR name
BPTR lock
```

Function: Sets a lock on a directory in an assign list. The assign structure must be created with AssignLock() or AssignLate(), and the lock may not be used again after this. If you need it, you can create another copy with DupLock().

Parameters: name — DeviceName without ':'

lock — Lock indicated by the name.

Result: 0 — Error, then the lock must be freed with UnLock().

See also: AssignLock(), AssignLate(), Lock(), UnLock()

Example: This allows you to define a logical device, such as 'C:' or 'DEVS:' that consists of several physical directories. Consider the following two directories:

Strings: (DIR)

```
GibsonGuitar.8SVX
RichGuitar.8SVX
WarwickBass.8SVX
WashburnGuitar.8SVX
```

Drumkit: (DIR)

```
PaisteCymbal.8SVX
PaisteGong.8SVX
PearlDrum.8SVX
PremierDrum.8SVX
```

We can assign these two directories to the logical device 'Samples:' as follows:

```
_MultiPath
 movea.l _DosBase,a6
 move.l  #_BasePath,d1
 moveq   #SHARED_LOCK,d2
 jsr     _LVOLock(a6)       ;Lock("Strings:",-2)
 move.l  d0,d2
 beq.s   .Error
 move.l  #_Samples,d1
 jsr     _LVOAssignLock(a6) ;AssignLock("Samples",Lock)
 tst.l   d0
 beq.s   .Error2
 move.l  #_AddPath,d1
 moveq   #SHARED_LOCK,d2
 jsr     _LVOLock(a6)       ;Lock("Drumkit:",-2)
 move.l  d0,d2
 beq.s   .Error3
 move.l  #_Samples,d1
 jsr     _LVOAssignAdd(a6)  ;AssignAdd("Samples",Lock)
 tst.l   d0
 beq.s   .Error4
 moveq   #0,d0
 rts
.Error4
.Error2
 move.l  d2,d1
 jsr     _LVOUnlock(a6)
.Error1
 moveq   #-1,d0
.Error3
 rts
```

```
_BasePath dc.b 'Strings:',0
_AddPath  dc.b 'Drumkit:',0
_Samples  dc.b 'Samples',0
```

If no errors occurred (result=0), you can access these files as follows:

```
"Samples:WarwickBass.8SVX"
"Samples:PaisteCymbal.8SVX"
```

If you were to store a file in the logical device 'Samples', it would go to the physical directory set with AssignLock(). In this case, this is "Strings:".

AssignLate — **Pre-define an AssignLock**

Call:

```
success = AssignLate(name,path)
D0        -618(A6)   D1   D2

BOOL success
APTR name
APTR path
```

Function: Defines an AssignLock that is only created after the first access on the given path. This is very helpful in cases where a device hasn't been activated yet.

Parameters:

	name	DeviceName without ':'
	path	Name used to address the device.

Result: 0 Error

See also: AssignLock

AssignLock — **Assign a name to a lock**

Call:

```
success = AssignLock(name,lock)
D0        -612(A6)   D1   D2

BOOL success
APTR name
BPTR lock
```

Function: Assigns a name to a lock. A value of 0 for lock will delete the entry with the given name. If an entry with the same name exists, it's replaced with the new lock. After this function, the lock may not be used again. If necessary, make a copy with DupLock().

Parameters: name — Device name (without ':') to which the lock is assigned.

lock — Lock for the name.

Result: 0 — Error, lock must then be freed with UnLock().

See also: Lock(), DupLock(), UnLock()

AssignPath **Assign a name to a path**

Call:

```
success = AssignPath(name,path)
D0         -624(A6)    D1    D2

BOOL success
APTR name
APTR path
```

Function: Assigns a name to a path. Also works with disks (volumes) that are not yet known.

Parameters: name — Device name without ':'

path — Path name replaced by 'name'.

Result: 0 — Error

See also: AssignLock(), AssignLate()

AttemptLockDosList **Lock a directory list**

Call:

```
dlist = AttemptLockDosList(flags)
D0       -666(A6)              D1

STRUCT DosList *dlist
ULONG  flags
```

Function: Prevents certain access from other programs to the list of logical devices.

Parameters: flags — Flags that indicate the nodes to be locked.

Result: dlist — Start of the list or 0 (no node address).

See also: LockDosList(), UnLockDosList()

FindDosEntry

Call:

```
newdlist = FindDosEntry(dlist,name,flags)
D0          -684(A6)       D1    D2    D3

STRUCT DosList *newdlist,*dlist
APTR   name
ULONG  flags
```

Function: Returns an entry from the list of logical devices.

Parameters:

dlist — Starting entry for the search.

name — Device name without ':'.

flags — Flags previously passed to LockDosList().

Result: Address of the entry or 0.

LockDosList — **Allow access to list of logical devices**

Call:

```
dlist = LockDosList(flags)
D0      -654(A6)    D1

STRUCT DosList *dlist
ULONG  flags
```

Function: This function allows exclusive access to the list of logical devices. If another task has the access rights, the program waits until the list is freed with UnLockDosList(). You can use nested calls of this function.

Parameters: flags — Entries to be accessed.

Result: Pointer to the list header, not an entry.

NextDosEntry — Next entry in the logical device list

Call:

```
newdlist = NextDosEntry(dlist,flags)
D0          -690(A6)     D1    D2

STRUCT DosList *newdlist,*dlist
ULONG  flags
```

Function: Finds the next entry of the desired type in the logical device list.

Parameters: dlist Current entry.

flags Type, see FindDosEntry().

Result: Next DosList structure or 0.

RemDosEntry — Remove a DosList structure from the list

Call:

```
success = RemDosEntry(dlist)
D0          -672(A6)    D1

BOOL   success
STRUCT DosList *dlist
```

Function: This function can be used to remove an entry from the logical device list. LockDosList() must be called first. The memory block used is not freed with this function.

Parameters: dlist DosList structure.

Result: 0 Error

UnLockDosList — Free logical device list

Call:

```
UnLockDosList(flags)
-660(A6)      D1

ULONG flags
```

Function: Frees a logical device list that was locked with LockDosList().

Parameters: flags Flags that were specified with LockDosList().

```
Dec Hex STRUCTURE DosList,0
  0  $0 BPTR     dol_Next       ;next entry
  4  $4 LONG     dol_Type       ;type (see below)
  8  $8 APTR     dol_Task       ;Handler task
 12  $C BPTR     dol_Lock       ;Lock
 16 $10 LABEL    dol_VolumeDate ;creation date
 16 $10 LABEL    dol_AssignName ;path name
 16 $10 BSTR     dol_Handler    ;filename (if SegList=0)
 20 $14 LABEL    dol_List       ;directory list (Assign)
 20 $14 LONG     dol_StackSize  ;stack size for the process
 24 $18 LONG     dol_Priority   ;priority of the process
 28 $1C LABEL    dol_LockList   ;available Locks
 28 $1C ULONG    dol_Startup    ;FileSysStartupMsg
 32 $20 LABEL    dol_DiskType   ;'DOS', etc.
 32 $20 BPTR     dol_SegList    ;SegList for the process
 36 $24 BPTR     dol_GlobVec    ;BCPL global vector
 40 $28 BSTR     dol_Name       ;name
 44 $2C LABEL    DosList_SIZEOF
```

Values for dl_Type:

```
DLT_DEVICE      =  0 ;logical device
DLT_DIRECTORY   =  1 ;Assign Node
DLT_VOLUME      =  2 ;diskette
DLT_LATE        =  3 ;late assignment
DLT_NONBINDING  =  4 ;free Assign (AssignPath)
DLT_PRIVATE     = -1 ;for DOS only
```

Flags for LockDosList() etc.:

```
LDB_READ    = 0, LDF_READ    =  1 ; specify either LDF_READ
LDB_WRITE   = 1, LDF_WRITE   =  2 ; or LDF_WRITE
LDB_DEVICES = 2, LDF_DEVICES =  4
LDB_VOLUMES = 3, LDF_VOLUMES =  8
LDB_ASSIGNS = 4, LDF_ASSIGNS = 16
LDB_ENTRY   = 5, LDF_ENTRY   = 32 ;for internal purposes
LDB_DELETE  = 6, LDF_DELETE  = 64
LDF_ALL = (LDF_DEVICES!LDF_VOLUMES!LDF_ASSIGNS)
```

3. Handlers and Filesystems

AddBuffers — Add to the number of buffers for a device

Call:

```
success = AddBuffers(filesystem, number)
D0        -732(A6)   D1          D2

BOOL success
APTR filesystem
LONG number
```

Function: Adds the given number of buffers to the existing number of buffers for a filesystem, then sets the number of buffers to the new number. You may also use negative values. If the call was successful, you can query the current number of buffers with IoErr().

Parameters: filesystem *String with the device name, including ':'.

number Number of buffers to add (may be positive or negative).

Result: 0 Error

See also: IoErr()

DeviceProc — Return the MsgPort for the handler of a device

Call:

```
process = DeviceProc( name )
D0        -174(A6)    D1

STRUCT MsgPort *process
APTR   name
```

Function: Returns the MessagePort that controls the given device. This is required for packet routines.

Parameters: name Device name

Result: MsgPort of the handler process or 0.

Warning: If you specify something that is only addressable as a device via ASSIGN, use the IoErr() function to get the lock associated with this name. You may only work with a copy of the lock that was created with DupLock().

See also: DoPkt(), IoErr(), DupLock()

DoPkt — **Send a DOS packet and wait for the reply**

Call:

```
result1 (/result2) = DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)
D0       (D1)         -240(A6) D1 D2     D3   D4   D5   D6   D7

LONG   result1,result2
STRUCT MsgPort *port
LONG   action,arg1,arg2,arg3,arg4,arg5
```

Function: PutMsg() sends a packet to the ProcessPort of the handler and waits for the handler to process it. Then result1 and result2 are taken from the returned packet. Since C programmers cannot use routines with two results, result2 is set up as an error code that can be queried with IoErr().

DoPkt() can also be called by an Exec task, but it will be slower and more prone to error.

Parameters:

port — pr_MsgPort of the handler.

action — Command for the handler or filesystem.

arg1, arg2, arg3, arg4,arg5 — Arguments for the command.

Result: 0 in D0 = error

See also: DeviceProc(), IoErr(), PutMsg(), WaitPort(), GetMsg()

EndNotify — **End file notification**

Call:

```
EndNotify(notifystructure)
-894(A6)  D1

STRUCT NotifyRequest *notifystructure
```

Function: Ends notification started with StartNotify().

Parameters: NotifyRequest that was passed to StartNotify().

Result: None

See also: StartNotify()

Format **Format a device**

Call:

```
success = Format(filesystem, volumename, dostype)
D0        -714(A6) D1         D2          D3

BOOL  success
APTR  filesystem,volumename
ULONG dostype
```

Function: Format a device, such as a diskette or a hard disk.

Parameters: filesystem Device name including ':'.

volumename
Name, such as the diskette name.

dostype Format type: OFS or FFS

Result: 0 Error

FreeDeviceProc **Free a structure obtained with GetDeviceProc()**

Call:

```
FreeDeviceProc(devproc)
-648(A6)       D1

STRUCT DevProc *devproc
```

Function: Frees a structure created with GetDeviceProc() and decrements the process counter.

Parameters: devproc DevProc structure from GetDeviceProc().

GetConsoleTask	Get the MsgPort of the console handler

Call:

```
port = GetConsoleTask()
D0     -510(A6)

STRUCT MsgPort *port
```

Function: Returns its own console task port (pr_ConsoleTask).

Result: pr_MsgPort of the console handler or 0.

GetDeviceProc	Get the handler for a path

Call:

```
devproc = GetDeviceProc(name, devproc)
D0        -642(A6)      D1    D2

STRUCT DevProc *devproc
APTR   name
```

Function: Returns the handler or filesystem for a path. You must supply the path name, which may be given relative to the current path, and a value of 0 as the DevProc structure. The result is a DevProc structure from which the data can be read. Kickstart 2.0 supports the division of a directory into several devices, so more than one handler/filesystem may be responsible for the path.

To get all of the data for a path, GetDeviceProc() must be called several times, and the first structure returned must be passed with each subsequent call. If you receive an ERROR_OBJECT_NOT_FOUND and if DVPF_ASSIGN is set in dvp_Flags, you must still call this function again. You will receive the DevProc structure with other values or with the value 0 and an ERROR_NO_MORE_ENTRIES from IoErr(). The function must continue to be called until 0 is returned. Then the handler/filesystem will automatically be locked. The structure returned with the first call can be freed with FreeDeviceProc. At this point, all of the data retrieved becomes invalid and must not be used anymore.

Parameters: name Path name to be accessed.

devproc DevProc structure from previous call, or 0.

Result: DevProc structure or null

GetFileSysTask — Get MsgPort of own filesystem

Call:

```
port = GetFileSysTask()
D0     -522(A6)

STRUCT MsgPort *port
```

Function: Reads the MsgPort of the filesystem from the process structure responsible for the program (pr_FileSystemTask).

Result: pr_MsgPort of the filesystem or 0.

Inhibit — Send the DOS packet ACTION_INHIBIT to a handler

Call:

```
success = Inhibit(filesystem, flag)
D0        -726(A6) D1          D2

BOOL success
APTR name
LONG flag
```

Function: Simultaneous access to a filesystem device must be locked before direct access is allowed (Workbench: DFx:BUSY). Programmers who simply jump in and access the trackdisk device or the hard disk already had many system crashes and instances of destroyed data. Normally, you would use DeviceProc() to get the handler port and then turn the filesystem off with an ACTION_INHIBIT packet. This function was implemented to give programmers a way to accomplish this.

Parameters: filesystem Device name including ':'

flag Argument for the StdPacket:

DOSTRUE	Inhibit (lock filesystem)
Null	Uninhibit (unlock filesystem)

Result: 0 Error

IsFileSystem — **Determine if a handler is a filesystem**

Call:

```
result = IsFileSystem(name)
D0       -708(A6)     D1

BOOL Result:   APTR name
```

Function: Returns a boolean argument that indicates whether a handler is a filesystem.

Parameters: name — Device name including ':'.

Result: 0 — Handler is not a filesystem.

Relabel — **Change name of a storage device**

Call:

```
success = Relabel(volumename,name)
D0        -720(A6) D1         D2

BOOL success
APTR volumename,name
```

Function: Changes the name of a storage device, such as a diskette.

Parameters: volumename — Device name including ':'.

newname — New name without ':'.

Result: 0 — Error

ReplyPkt — **Reply to a DosPacket**

Call:

```
ReplyPkt(packet, result1, result2)
-258(A6) D1      D2       D3

STRUCT DosPacket *packet
LONG result1,result2
```

Function: Places results in a packet and returns it to the sender.

Parameters: packet DosPacket structure.

result1,result2
Results

SendPkt **Send a DosPacket to a handler**

Call:

```
SendPkt(port, packet, replyport)
-246(A6) D1    D2       D3

STRUCT MsgPort *port, *replyport
STRUCT DosPacket *packet
```

Function: Sends a packet to a handler without waiting for the reply. The reply is sent to the specified ReplyPort. This is the pr_MsgPort of its own process structure.

Parameters: port pr_MsgPort of the handler (see DeviceProc()).

packet DosPacket structure to be sent.

replyport pr_MsgPort of its own process.

SetConsoleTask **Set console handler port**

Call:

```
OldPort = SetConsoleTask(port)
D0        -516(A6)       D1

STRUCT MsgPort *port, *OldPort
```

Function: Sets the port for the standard console tasks of the processor (pr_ConsoleTask).

Parameters: port pr_MsgPort of the console handler.

Result: Pointer to previous console task.

SetFileSysTask — **Set filesystem port**

Call:

```
OldPort = SetFileSysTask(port)
D0        -528(A6)        D1

STRUCT MsgPort *port, *OldPort
```

Function: Sets the port for the filesystem tasks of the process (pr_FileSystemTask).

Parameters: port — pr_MsgPort of the filesystem.

Result: Previous FileSysTask

StartNotify — **Start file notification**

Call:

```
success = StartNotify(notifystructure)
D0        -888(A6)     D1

BOOL   success
STRUCT NotifyRequest *notifystructure
```

Function: Begin notification for a file or directory. You are then notified if a change is made, as long as the filesystem supports this.

Parameters: notifystructure
Initialized NotifyRequest structure.

Result: 0 — Error

WaitPkt — **Wait for a DosPacket**

Call:

```
packet = WaitPkt()
D0       -252(A6)

STRUCT DosPacket *packet
```

Function: Waits for a DosPacket to appear in its own pr_MsgPort and picks up the StdPkt with GetMsg().

Result: packet DosPacket (LN_NAME of the message structure)

DosPacket Structure:

```
Dec Hex STRUCTURE DosPacket,0
  0  $0 APTR   dp_Link     ;Exec message
  4  $4 APTR   dp_Port     ;ReplyPort
  8  $8 LABEL  dp_Action   ;s. ACTION_...
  8  $8 LONG   dp_Type     ;'R', 'W'
 12  $C LABEL  dp_Status   ;1st result:
 12  $C LONG   dp_Res1     ;1st result
 16 $10 LABEL  dp_Status2 ;2nd result
 16 $10 LONG   dp_Res2     ;2nd result
 20 $14 LABEL  dp_BufAddr ;buffer address
 20 $14 LONG   dp_Arg1     ;1st argument
 24 $18 LONG   dp_Arg2     ;2nd argument
 28 $1C LONG   dp_Arg3     ;3rd argument
 32 $20 LONG   dp_Arg4     ;4th argument
 36 $24 LONG   dp_Arg5     ;5th argument
 40 $28 LONG   dp_Arg6     ;6th argument
 44 $2C LONG   dp_Arg7     ;7th argument
 48 $30 LABEL  dp_SIZEOF
```

Structure for sending Packets:

```
Dec Hex STRUCTURE StandardPacket,0
  0  $0 STRUCT sp_Msg,MN_SIZE    ;Exec message
 20 $14 STRUCT sp_Pkt,dp_SIZEOF ;Packet
 68 $44 LABEL  sp_SIZEOF
```

Packet Types:

```
ACTION_NIL              =    0 ;no message
ACTION_STARTUP          =    0 ;Handler startup
ACTION_GET_BLOCK        =    2 ;DO NOT USE!
ACTION_SET_MAP          =    4 ;set map
ACTION_DIE              =    5 ;end process
ACTION_EVENT            =    6 ;event
ACTION_CURRENT_VOLUME   =    7 ;current disk
ACTION_LOCATE_OBJECT    =    8 ;find object
ACTION_RENAME_DISK      =    9 ;rename disk
ACTION_WRITE            =  'W' ;write
ACTION_READ             =  'R' ;read
ACTION_FREE_LOCK        =   15 ;free Lock
ACTION_DELETE_OBJECT    =   16 ;delete object
ACTION_RENAME_OBJECT    =   17 ;rename object
```

```
ACTION_MORE_CACHE      =   18 ;add buffer
ACTION_COPY_DIR        =   19 ;copy directory
ACTION_WAIT_CHAR       =   20 ;wait for a character
ACTION_SET_PROTECT     =   21 ;set protection
ACTION_CREATE_DIR      =   22 ;create directory
ACTION_EXAMINE_OBJECT  =   23 ;examine object
ACTION_EXAMINE_NEXT    =   24 ;examine next entry
ACTION_DISK_INFO       =   25 ;info on the disk
ACTION_INFO            =   26 ;information
ACTION_FLUSH           =   27 ;invalid buffers
ACTION_SET_COMMENT     =   28 ;set comment
ACTION_PARENT          =   29 ;parent directory
ACTION_TIMER           =   30 ;Timer event
ACTION_INHIBIT         =   31 ;Handler on/off
ACTION_DISK_TYPE       =   32 ;diskette type
ACTION_DISK_CHANGE     =   33 ;diskette change
ACTION_SET_DATE        =   34 ;set date
ACTION_SAME_LOCK       =   40 ;compare Locks
ACTION_SCREEN_MODE     =  994 ;screen mode
ACTION_READ_RETURN     = 1001 ;read
ACTION_WRITE_RETURN    = 1002 ;write
ACTION_SEEK            = 1008 ;position
ACTION_FINDUPDATE      = 1004 ;open
ACTION_FINDINPUT       = 1005 ;old file
ACTION_FINDOUTPUT      = 1006 ;new file
ACTION_END             = 1007 ;end
ACTION_FORMAT          = 1020 ;format
ACTION_MAKE_LINK       = 1021 ;create a link
ACTION_SET_FILE_SIZE   = 1022 ;set file size
ACTION_WRITE_PROTECT   = 1023 ;write protect
ACTION_READ_LINK       = 1024 ;read link
ACTION_FH_FROM_LOCK    = 1026 ;get FileHandle
ACTION_IS_FILESYSTEM   = 1027 ;get Handler type
ACTION_CHANGE_MODE     = 1028 ;change access mode
ACTION_COPY_DIR_FH     = 1030 ;copy directory
ACTION_PARENT_FH       = 1031 ;get parent directory
ACTION_EXAMINE_ALL     = 1033 ;examine directory tree structure
ACTION_EXAMINE_FH      = 1034 ;examine file
ACTION_LOCK_RECORD     = 2008 ;lock record
ACTION_FREE_RECORD     = 2009 ;free record
ACTION_ADD_NOTIFY      = 4097 ;start notification
ACTION_REMOVE_NOTIFY   = 4098 ;end notification
```

Packet types from run/newcli/execute/system to the Shell:

```
RUN_EXECUTE       = -1
RUN_SYSTEM        = -2
RUN_SYSTEM_ASYNCH = -3
```

Results of GetDeviceProc():

```
Dec Hex STRUCTURE DevProc,0
  0  $0 APTR     dvp_Port     ;MsgPort
  4  $4 BPTR     dvp_Lock     ;Lock
  8  $8 ULONG    dvp_Flags    ;Flags (s.u.)
 12  $C APTR     dvp_DevNode ;DosList (DO NOT USE!)
 16 $10 LABEL    dvp_SIZEOF
```

Values for dvp_Flags

```
DVPB_UNLOCK = 0, DVPF_UNLOCK = 1
DVPB_ASSIGN = 1, DVPF_ASSIGN = 2
```

Storage device description:

```
Dec Hex STRUCTURE DosEnvec,0
  0  $0 ULONG de_TableSize      ;table size
  4  $4 ULONG de_SizeBlock      ;block size in Longs
  8  $8 ULONG de_SecOrg         ;sector organization (0)
 12  $C ULONG de_Surfaces       ;number of heads
 16 $10 ULONG de_SectorPerBlock ;sectors per block (1)
 20 $14 ULONG de_BlocksPerTrack ;blocks per track
 24 $18 ULONG de_Reserved       ;reserved blocks at the beginning
 28 $1C ULONG de_PreAlloc       ;reserved blocks at the end
 32 $20 ULONG de_Interleave     ;interleave mode (0)
 36 $24 ULONG de_LowCyl         ;first cylinder (starting with 0)
 40 $28 ULONG de_HighCyl        ;last cylinder
 44 $2C ULONG de_NumBuffers     ;normal buffer count
 48 $30 ULONG de_BufMemType     ;memory type of buffer
 52 $34 ULONG de_MaxTransfer    ;maximum speed
 56 $38 ULONG de_Mask           ;address mask
 60 $3C LONG  de_BootPri        ;boot priority
 64 $40 ULONG de_DosType        ;DOS type
 68 $44 ULONG de_Baud           ;baud rate for serial Handlers
 72 $48 ULONG de_Control        ;control Word for Handler
 76 $4C ULONG de_BootBlocks     ;number of boot blocks
 80 $50 LABEL DosEnvec_SIZEOF
```

Filesystem startup message:

```
Dec Hex STRUCTURE FileSysStartupMsg,0
  0  $0 ULONG    fssm_Unit    ;unit number for OpenDevice()
  4  $4 BSTR     fssm_Device  ;DeviceName ending in 0
  8  $8 BPTR     fssm_Environ ;structure of data storage device
 12  $C ULONG    fssm_Flags   ;Flags for OpenDevice()
 16 $10 LABEL    FileSysStartupMsg_SIZEOF
```

```
NOTIFY_CLASS = $40000000        ;this will change...
NOTIFY_CODE  = $1234            ;this too

Dec Hex STRUCTURE NotifyMessage,0
  0  $0 STRUCT nm_ExecMessage,MN_SIZE ;message
 20 $14 ULONG  nm_Class               ;s.o.
 24 $18 UWORD  nm_Code                ;s.o.
 26 $1A APTR   nm_NReq                ;Notify request (do not change)
 30 $1E ULONG  nm_DoNotTouch
 34 $22 ULONG  nm_DoNotTouch2
 38 $26 LABEL  NotifyMessage_SIZEOF

Dec Hex STRUCTURE NotifyRequest,0
  0  $0 CPTR   nr_Name          ;Name
  4  $4 CPTR   nr_FullName      ;complete DOS path
  8  $8 ULONG  nr_UserData      ;own data
 12  $C ULONG  nr_Flags         ;Flags
 16 $10 LABEL  nr_Task          ;task for SEND_SIGNAL or
 16 $10 APTR   nr_Port          ;MsgPort for SEND_MESSAGE
 20 $14 UBYTE  nr_SignalNum     ;for SEND_SIGNAL
 21 $15 STRUCT nr_pad,3
 24 $18 STRUCT nr_Reserved,4*4
 40 $28 ULONG  nr_MsgCount      ;number of Msgs sent
 44 $2C APTR   nr_Handler       ;Handler for EndNotify()
 48 $30 LABEL  NotifyRequest_SIZEOF
```

Values for nr_Flags:

```
NRB_SEND_MESSAGE     =  0, NRF_SEND_MESSAGE     =         1
NRB_SEND_SIGNAL      =  1, NRF_SEND_SIGNAL      =         2
NRB_WAIT_REPLY       =  3, NRF_WAIT_REPLY       =         8
NRB_NOTIFY_INITIAL   =  4, NRF_NOTIFY_INITIAL   =        16
NRB_MAGIC            = 31, NRF_MAGIC            = $80000000
                           NR_HANDLER_FLAGS     = $ffff0000
```

4. Directories

CreateDir	**Create a new directory**

Call:

```
lock = CreateDir( name )
D0     -120(A6)   D1

BPTR lock
APTR name
```

Function: Creates a new directory and returns a lock for it.

Parameters: name   String containing directory name.

Result: BCPL pointer to a lock or 0.

See also: UnLock()

CurrentDir **Set the current directory**

Call:

```
oldLock = CurrentDir( lock )
D0        -126(A6)    D1

BPTR oldLock
BPTR lock
```

Function: CurrentDir() sets the directory that all path specifications will use as a starting point. You are required to pass a lock for the desired directory. As a result, you receive the lock to the directory that was formerly current.

Parameters: lock   BCPL pointer to a lock.

Result: BCPL pointer to the previous current directory. A value of 0 represents the boot directory that is set by a reboot.

See also: Lock(), UnLock()

ExAll **Examine an entire directory**

Call:

```
continue = ExAll(lock, buffer, size, type, control)
D0         -432(A6) D1   D2      D3    D4    D5

BOOL continue
BPTR lock
APTR buffer
LONG size,type
STRUCT ExAllControl *control
```

Function: Examines a directory and fills a buffer with ExAllData structures.

Parameters: lock — Directory lock

buffer — Buffer address

size — Buffer size

type — The amount of data that will be stored in each file (ED_...). Higher values contains smaller values. The order is name, type, size, protection bits, date, comment.

control — ExAllControl structure, must be created with AllocDosObject(). The LastKey entry must be deleted before the call. If several calls are required, this entry may not be changed.

Entries	Number of entries in buffer.
LastKey	Delete prior to call.
MatchString	Optional pattern string.
MatchFunc	Hook address of a pattern matching routine.

Result: 0 — Cancel (delete LastKey):

IoErr() — ERROR_NO_MORE_ENTRIES: ExAll is finished; otherwise IoErr()=error code. In any other case, save the buffer contents and call ExAll() again.

See also: IoErr(), AllocDosObject(), FreeDosObject

Examine — **Examine directory or file**

Call:

```
success = Examine( lock, infoBlock )
D0        -102(A6) D1    D2

BOOL success
```

```
BPTR lock
STRUCT FileInfoBlock *infoBlock
```

Function: Examine() fills a FileInfoBlock with all available information. This data structure can only be read if it's passed as a parameter later (e.g., to ExNext()).

Parameters: lock — Lock for the file/directory to be examined.

infoBlock — Address of FileInfoBlock structure.

Result: 0 — Error

ExNext **Examine next directory entry**

Call:

```
success = ExNext( lock, infoBlock )
D0        -108(A6) D1    D2

BOOL   success
BPTR   lock
STRUCT FileInfoBlock *infoBlock)
```

Function: This function examines the next directory entry and fills the fields of the given FileInfoBlock with the values that were obtained. Prior to the first call, the FileInfoBlock must be initialized with the Examine() function.

Parameters: lock — Lock for the directory being examined. This lock must correspond with the lock from the Examine() call. File locks do not work.

infoBlock — Address of FileInfoBlock structure that was initialized by Examine().

Result: 0 — If IoErr()=ERROR_NO_MORE_ENTRIES, then no more entries are available. Otherwise, IoErr() returns the error number.

Warning: Recursive reading of the directory tree structure will only work if you use a new FileInfoBlock for each directory found.

GetProgramDir — **Get directory lock for the program**

Call:

```
lock = GetProgramDir()
D0     -600(A6)

BPTR lock
```

Function: Returns a lock for the directory from which the program is started. You can make a working copy of this lock with DupLock().

Result: Lock or 0 (for example, in the case of a resident program)

Info — **Get information about a disk**

Call:

```
success = Info( lock, parameterBlock )
D0        -114(A6) D1    D2

BOOL    success
BPTR    lock
STRUCT  InfoData *parameterBlock
```

Function: Fills the InfoData structure with information on the disk that corresponds to a given lock.

Parameters:

lock	A filesystem lock
parameterBlock	InfoData structure

Result:

0	Error

MatchEnd — **Free MatchFirst()/MatchNext() memory**

Call:

```
MatchEnd(AnchorPath)
-834(A6) D1

STRUCT AnchorPath *AnchorPath
```

Function: Free pattern matching memory.

Parameters: AnchorPath
Structure of MatchFirst()/MatchNext().

MatchFirst — **Find a file that matches the pattern**

Call:

```
error = MatchFirst(pat, AnchorPath)
D0      -822(A6)   D1   D2

BOOL error
APTR pat
STRUCT AnchorPath *AnchorPath
```

Function: Finds the first file or directory that matches the given pattern. Initializes the AnchorPath structure. Possible characters in the pattern string are:

?	Individual character
#	0 or more characters
(a|b)	Individually check components separated by |
~	Exclude the following expression
[abc]	One of the specified characters
[a-z]	Range of characters, such as "[0-9a-zA-Z]"
%	No character (useful with "(a|b|%)")
*	Can optionally be used for "#?"

Parameters: pat — Pattern string

AnchorPath
Structure for the search.

Result: 0 — Okay, otherwise error code.

MatchNext — **Find next file that matches the pattern**

Call:

```
error = MatchNext(AnchorPath)
D0      -828(A6)  D1

BOOL   error
STRUCT AnchorPath *AnchorPath
```

Function: Finds the next file or directory to match the given pattern (see MatchFirst()).

Parameters: AnchorPath
MatchFirst() structure

Result: 0 Okay, otherwise error code.

ParentDir **Get parent directory lock**

Call:

```
newlock = ParentDir( lock )
D0        -210(A6)   D1

BPTR newlock,lock
```

Function: Returns a lock for the parent directory of a file or directory.

Parameters: lock BCPL pointer to a lock structure.

Result: Lock or 0 (= boot directory, parent directory of all root directories)

ParentOfFH **Get lock for a file's parent directory**

Call:

```
lock = ParentOfFH(fh)
D0     -384(A6)   D1

BPTR lock,fh
```

Function: Returns a lock for the parent directory when given a FileHandle.

Parameters: fh FileHandle

Result: Lock or 0 (error)

Structure of Examine() and ExNext():

```
Dec  Hex STRUCTURE FileInfoBlock,0
  0   $0 LONG    fib_DiskKey            ;block number for operating system
  4   $4 LONG    fib_DirEntryType       ;type of entry (<0:file, >0:directory)
  8   $8 STRUCT fib_FileName,108        ;filename ending in 0
116  $74 LONG    fib_Protection         ;protection status
120  $78 LONG    fib_EntryType          ;for the operating system
124  $7C LONG    fib_Size               ;file size in bytes
128  $80 LONG    fib_NumBlocks          ;file size in blocks
```

```
132   $84 STRUCT fib_DateStamp,ds_SIZEOF ;revision date
144   $90 STRUCT fib_Comment,80          ;comment ending in 0
224   $E0 STRUCT fib_Reserved,36         ;reserved
260  $104 LABEL  fib_SIZEOF
```

Normal values for fib_DirEntryType:

```
ST_BOOT      =  0 ;boot directory
ST_ROOT      =  1 ;main directory
ST_USERDIR   =  2 ;directory
ST_SOFTLINK  =  3 ;soft link
ST_LINKDIR   =  4 ;HardLink to directory
ST_FILE      = -3 ;file
ST_LINKFILE  = -4 ;HardLink to file
```

Protection status bits:

```
FIBB_SCRIPT   =  6 ;batch file
FIBF_SCRIPT   = 64 ;
FIBB_PURE     =  5 ;program code is re-entrable
FIBF_PURE     = 32 ;            (=RESIDENT-capable)
FIBB_ARCHIVE  =  4 ;deleted when file is changed
FIBF_ARCHIVE  = 16 ;
FIBB_READ     =  3 ;disable read access
FIBF_READ     =  8 ;
FIBB_WRITE    =  2 ;disable write access
FIBF_WRITE    =  4 ;
FIBB_EXECUTE  =  1 ;disable program start
FIBF_EXECUTE  =  2 ;
FIBB_DELETE   =  0 ;disable delete
FIBF_DELETE   =  1 ;
```

Values for ExAll():

```
ED_NAME       = 1 ;name
ED_TYPE       = 2 ;name+type
ED_SIZE       = 3 ;name+type+length
ED_PROTECTION = 4 ;name+type+length+protection
ED_DATE       = 5 ;name+type+length+protection+date
ED_COMMENT    = 6 ;name+type+length+protection+date+comment
```

ExAll() result structure:

```
Dec Hex STRUCTURE ExAllData,0
  0  $0 APTR     ed_Next      ;next ExAllData structure
  4  $4 APTR     ed_Name      ;name
  8  $8 LONG     ed_Type      ;typ or end of structure
```

```
 12   $C ULONG   ed_Size      ;size or end of structure
 16  $10 ULONG   ed_Prot      ;protection or end of structure
 20  $14 ULONG   ed_Days      ;date stamp or end of structure
 24  $18 ULONG   ed_Mins
 28  $1C ULONG   ed_Ticks
 32  $20 APTR    ed_Comment   ;comment or end of structure
  ?    ? LABEL   ed_Strings   ;strings at end of structure
```

Control structure for ExAll():

```
Dec Hex STRUCTURE ExAllControl,0
  0   $0 ULONG   eac_Entries     ;number of buffer entries
  4   $4 ULONG   eac_LastKey     ;disk block (do not change)
  8   $8 APTR    eac_MatchString ;pattern string or 0
 12   $C APTR    eac_MatchFunc   ;pattern match Hook or 0
 16  $10 LABEL   ExAllControl_SIZEOF
```

Structure of Info():

```
Dec Hex STRUCTURE InfoData,0
  0   $0 LONG id_NumSoftErrors ;number of errors on disk
  4   $4 LONG id_UnitNumber    ;number for OpenDevice
  8   $8 LONG id_DiskState     ;diskette status (see below)
 12   $C LONG id_NumBlocks     ;number of blocks on disk
 16  $10 LONG id_NumBlocksUsed ;number of blocks used
 20  $14 LONG id_BytesPerBlock ;bytes per block
 24  $18 LONG id_DiskType      ;disk type
 28  $1C BPTR id_VolumeNode    ;BPTR to DosList structure
 32  $20 LONG id_InUse         ;Flag, 0=not active
 36  $24 LABEL id_SIZEOF
```

Diskette status:

```
ID_WRITE_PROTECTED = 80 ;write protection on
ID_VALIDATING      = 81 ;disk being checked
ID_VALIDATED       = 82 ;disk is okay
```

Diskette type:

```
ID_NO_DISK_PRESENT = -1         ;no disk in drive
ID_UNREADABLE_DISK = 'BAD'<<8   ;unreadable format or error
ID_NOT_REALLY_DOS  = 'NDOS'     ;unreadable format
ID_DOS_DISK        = 'DOS'<<8   ;OFS disk
ID_FFS_DISK        = 'DOS'<<8!1 ;FFS disk
ID_KICKSTART_DISK  = 'KICK'     ;operating system diskette
ID_MSDOS_DISK      = 'MSD'<<8   ;MS-DOS diskette
```

Pattern matching structure:

```
Dec   Hex STRUCTURE AnchorPath,0
  0    $0 LABEL    ap_First
  0    $0 CPTR     ap_Base              ;first Anchor
  4    $4 LABEL    ap_Current
  4    $4 CPTR     ap_Last              ;last Anchor
  8    $8 LONG     ap_BreakBits         ;break bits
 12    $C LONG     ap_FoundBreak        ;found bits
 16   $10 LABEL    ap_Length
 16   $10 BYTE     ap_Flags             ;Flags
 17   $11 BYTE     ap_Reserved
 18   $12 WORD     ap_Strlen            ;string length
 20   $14 STRUCT   ap_Info,fib_SIZEOF ;FileInfoBlock
280 $118 LABEL     ap_Buf               ;buffer for path
280 $118 LABEL     ap_SIZEOF

APB_DOWILD      =  0  ;OPT ALL
APF_DOWILD      =  1
APB_ITSWILD     =  1  ;Flag from MatchFirst() for MatchNext()
APF_ITSWILD     =  2
APB_DODIR       =  2  ;directory must also be checked
APF_DODIR       =  4
APB_DIDDIR      =  3  ;directory being checked
APF_DIDDIR      =  8
APB_NOMEMERR    =  4  ;not enough memory
APF_NOMEMERR    = 16
APB_DODOT       =  5  ;conversion of '.' in CurrentDir
APF_DODOT       = 32
APB_DirChanged =  6  ;directory has changed since
APF_DirChanged = 64  ;last MatchNext call
```

Anchor structure:

```
Dec   Hex STRUCTURE   AChain,0
  0    $0 CPTR     an_Child
  4    $4 CPTR     an_Parent
  8    $8 LONG     an_Lock
 12    $C STRUCT   an_Info,fib_SIZEOF  ; FileInfoBlock
272 $110 BYTE      an_Flags
273 $111 LABEL     an_String
273 $111 LABEL     an_SIZEOF

DDB_PatternBit  = 0, DDF_PatternBit  =  1
DDB_ExaminedBit = 1, DDF_ExaminedBit =  2
DDB_Completed   = 2, DDF_Completed   =  4
DDB_AllBit      = 3, DDF_AllBit      =  8
DDB_SINGLE      = 4, DDF_SINGLE      = 16
```

Tokens for Token strings:

```
P_ANY        = $80 ; Token for '*' or '#?
P_SINGLE     = $81 ; Token for '?'
P_ORSTART    = $82 ; Token for '('
P_ORNEXT     = $83 ; Token for '|'
P_OREND      = $84 ; Token for ')'
P_NOT        = $85 ; Token for '~'
P_NOTEND     = $86 ; end of expression after '~'
P_NOTCLASS   = $87 ; Token for '^'
P_CLASS      = $88 ; Token for '[]'
P_REPBEG     = $89 ; Token for '['
P_REPEND     = $8A ; Token for ']'
P_STOP       = $8B ; cancel evaluation
```

Values for an_Status:

```
COMPLEX_BIT = 1    ; pattern parsing
EXAMINE_BIT = 2    ; search in directory
```

5. Programs

AddSegment	**Insert program in resident list**

Call:

```
success = AddSegment(name, seglist, type)
D0        -774(A6)  D1    D2       D3

BOOL success
APTR name
BPTR seglist
LONG type
```

Function: Inserts a program in the resident list (to hold it in memory).

Parameters:

name	Program name
seglist	BPTR (APTR/4) to program's segment list.
type	Call counter for linking, normal value: 0.

Result: 0 Error

CreateNewProc — **Generate a new process**

Call:

```
process = CreateNewProc(tags)
D0        -498(A6)       D1

STRUCT Process *process
STRUCT TagItem *tags
```

Function: Generates a new process according to the values in the tag array. NP_Seglist or NP_Entry must be included. NP_Seglist passes a BPTR to a segment list and NP_Entry passes the address of the program. Input and output are routed to NIL: and the stack is set to 4000 bytes.

CreateNewProc can be called from a simple task, but in this case the DOS I/O will not work.

Parameters: tags — Address of a TagItems field.

Result: Process or 0

CreateProc — **Generate a new process (old)**

Call:

```
process = CreateProc( name, pri, seglist, stackSize )
D0        -138(A6)    D1    D2   D3       D4

STRUCT MsgPort *process
APTR name
LONG pri
BPTR seglist
LONG stackSize
```

Function: CreateProc starts a new process with the given name.

Parameters:

name	Address of the string with the process name.
pri	Priority of the process (-128 to 127)
seglist	BPTR to a SegList (see LoadSeg())
stackSize	Stack size (multiple of 4)

Result: Process or 0 (error)

See also: LoadSeg(), CreateNewProc()

Exit — End BCPL program

Call:

```
Exit( returnCode )
-144(A6) D1

LONG returnCode
```

Function: Exit() is used to properly end BCPL programs only. This routine must never be called by other programs.

Parameters: returnCode
Return value for CLI.

Result: None.

Warning: C programmers must be careful not to confuse the C function exit() with the DOS function Exit().

FindSegment — Retrieve a segment from the resident list

Call:

```
segment = FindSegment(name, start, system)
D0        -780(A6)     D1    D2     D3

STRUCT Segment *segment,*start
APTR   name
LONG   system
```

Function: Finds the segment of the given name in the list of resident programs. You can also specify the name of the segment from which to begin the search. If the system flag is set, then only one system segment is searched.

Parameters:

name	Segment name
start	0 or starting segment for the search
system	0 or -1 for system segment

Result: Segment address or 0

Warning: Turn off multitasking before calling.

InternalLoadSeg **Load program from FileHandle**

Call:

```
seglist = InternalLoadSeg(fh,table,functionarray,stack)
D0        -756(A6)        D0 A0    A1            A2

BPTR seglist,fh,table
APTR functionarray,stack
```

Function: Loads the program represented by a FileHandle. If no overlay is loaded, then table must be set to 0. If the stack size is integrated into the program, then it's written to the address given in stack. There may already be a value stored at this address. In this case, it's overwritten by the loaded value.

Parameters:

fh — FileHandle of the program.

table — Overlay table or 0

functionarray — Field containing addresses of three functions:

```
Actual      ReadFunc(readhandle,buffer,length),DOSBase
D0                   D1         A0       D0       A6
------->   read function,    normally Read()
Memory = AllocFunc(size,flags),Execbase
D0                 D0    D1      A6
-------> allocate memory, normally AllocMem()
         FreeFunc(memory,size),Execbase
                  A1      D0     A6
-------> free memory, normally FreeMem()
```

stack — Variable address (LONG) to which the stack size is written.

Result: SegList or -(SegList) for overlays or 0.

InternalUnLoadSeg	Free a SegList

Call:

```
success = InternalUnLoadSeg(seglist,FreeFunc)
D0        -762(A6)           D1       A1

BOOL success
BPTR seglist
FPTR FreeFunc
```

Function: Frees the segments of a SegList and closes the program file for overlays.

Parameters: seglist — SegList of a program.

FreeFunc — Free function (see InternalLoadSeg())

Result: 0 — Error

LoadSeg	Load program

Call:

```
seglist = LoadSeg( name )
D0         -150(A6) D1

BPTR seglist
APTR name
```

Function: Loads a file consisting of DOS hunks into memory. The memory blocks are linked with BPTRs in the first longword. The size of the memory block precedes the BPTR.

Parameters: name — Filename (including path)

Result: BPTR to the first segment or 0.

NewLoadSeg	Expanded LoadSeg() routine

Call:

```
seglist = NewLoadSeg(file, tags)
D0         -768(A6)   D1    D2

BPTR   seglist
APTR   file
```

```
STRUCT TagItem *tags
```

Function: Loads a file consisting of hunks, depending on the tags in a given TagItem field.

Parameters: file — Filename

tags — Address of a TagItem field.

Result: Seglist or 0

RemSegment — **Remove a program from the resident list**

Call:

```
success = RemSegment(segment)
D0          -786(A6)    D1

BOOL   success
STRUCT Segment *segment
```

Function: Removes a resident segment from the system list and frees the allocated memory.

Parameters: segment — Segment structure

Result: 0 — Error (usually because Usecount is not 0)

RunCommand — **Start a program with its own process**

Call:

```
rc = RunCommand(seglist, stacksize, argptr, argsize)
D0   -504(A6)    D1       D2          D3      D4

LONG  rc
BPTR  seglist
ULONG argsize, stacksize
APTR  argptr
```

Function: Starts a program using its own process structure.

Parameters: seglist — SegList of the program.

stacksize — Stack size

argptr	Argument string
argsize	Length of argument string

Result: Return value of the program or -1 if the stack could not be loaded.

UnLoadSeg **Free SegList**

Call:
```
success = UnLoadSeg( seglist )
D0         -156(A6)    D1

BOOL success
BPTR seglist
```

Function: Free the SegList of a file loaded with LoadSeg().

Parameters: seglist BCPL to a SegList

Result: 0 SegList was 0 or an error occurred.

CreateNewProc() Tags:

```
NP_Dummy         = TAG_USER+1000
NP_Seglist       = NP_Dummy+1  ;SegList of the program
NP_FreeSeglist   = NP_Dummy+2  ;free SegList at end?
NP_Entry         = NP_Dummy+3  ;program address
NP_Input         = NP_Dummy+4  ;input handle
NP_Output        = NP_Dummy+5  ;output handle
NP_CloseInput    = NP_Dummy+6  ;close(Inputhandle) at end?
NP_CloseOutput   = NP_Dummy+7  ;close(Outputhandle) at end?
NP_Error         = NP_Dummy+8  ;error handle
NP_CloseError    = NP_Dummy+9  ;close(Errorhandle) at end?
NP_CurrentDir    = NP_Dummy+10 ;current directory
NP_StackSize     = NP_Dummy+11 ;stack size in bytes
NP_Name          = NP_Dummy+12 ;process name
NP_Priority      = NP_Dummy+13 ;process priority
NP_ConsoleTask   = NP_Dummy+14 ;Console Handler
NP_WindowPtr     = NP_Dummy+15 ;window for Requester, etc.
NP_HomeDir       = NP_Dummy+16 ;start directory
NP_CopyVars      = NP_Dummy+17 ;copy local variables?
NP_Cli           = NP_Dummy+18 ;create CLI structure?
NP_Path          = NP_Dummy+19 ;path for CLI
NP_CommandName   = NP_Dummy+20 ;program name for CLI
NP_Arguments     = NP_Dummy+21 ;arguments for CLI
```

```
NP_NotifyOnDeath = NP_Dummy+22 ;message at end?
NP_Synchronous   = NP_Dummy+23 ;wait until process end?
NP_ExitCode      = NP_Dummy+24 ;routine to be ended
NP_ExitData      = NP_Dummy+25 ;data for NP_EndCode
```

Structure of a process (expanded Task structure):

```
Dec Hex STRUCTURE Process,0
  0  $0 STRUCT  pr_Task,TC_SIZE        ;Task structure
 92 $5C STRUCT  pr_MsgPort,MP_SIZE     ;process port
126 $7E WORD    pr_Pad
128 $80 BPTR    pr_SegList             ;SegList of the program
132 $84 LONG    pr_StackSize           ;stack size
136 $88 APTR    pr_GlobVec             ;global vector (BCPL)
140 $8C LONG    pr_TaskNum             ;CLI process number
144 $90 BPTR    pr_StackBase           ;end of stack
148 $94 LONG    pr_Result2             ;return value
152 $98 BPTR    pr_CurrentDir          ;Lock for current directory
156 $9C BPTR    pr_CIS                 ;input channel
160 $A0 BPTR    pr_COS                 ;output channel
164 $A4 APTR    pr_ConsoleTask         ;pr_MsgPort of the window Handler
168 $A8 APTR    pr_FileSystemTask      ;pr_MsgPort of the drive
172 $AC BPTR    pr_CLI                 ;CLI structure
176 $B0 APTR    pr_ReturnAddr          ;old stack
180 $B4 APTR    pr_PktWait             ;WaitPkt() function
184 $B8 APTR    pr_WindowPtr           ;Requester window
188 $BC BPTR    pr_HomeDir             ;start directory
192 $C0 LONG    pr_Flags               ;Flags
196 $C4 APTR    pr_ExitCode            ;end function
200 $C8 LONG    pr_ExitData            ;data for the function
204 $CC APTR    pr_Arguments           ;argument string
208 $D0 STRUCT  pr_LocalVars,MLH_SIZE ;local ENV variables
220 $DC APTR    pr_ShellPrivate        ;for Shell only
224 $E0 BPTR    pr_CES                 ;error channel, in case pr_COS=0
228 $E4 LABEL   pr_SIZEOF
```

pr_Flags flags:

```
PRB_FREESEGLIST = 0, PRF_FREESEGLIST =  1
PRB_FREECURRDIR = 1, PRF_FREECURRDIR =  2
PRB_FREECLI     = 2, PRF_FREECLI     =  4
PRB_CLOSEINPUT  = 3, PRF_CLOSEINPUT  =  8
PRB_CLOSEOUTPUT = 4, PRF_CLOSEOUTPUT = 16
PRB_FREEARGS    = 5, PRF_FREEARGS    = 32
```

Hunk types:

```
HUNK_UNIT       =  999 ;part of an object code file
HUNK_NAME       = 1000 ;segment name
HUNK_CODE       = 1001 ;program segment
HUNK_DATA       = 1002 ;data segment
HUNK_BSS        = 1003 ;memory block (+MEMF_CLEAR)
HUNK_RELOC32    = 1004 ;table for absolute addressing
HUNK_RELOC16    = 1005 ;offset table
HUNK_RELOC8     = 1006 ;offset table
HUNK_EXT        = 1007 ;linker data
     EXT_SYMB   =    0 ;symbol table
     EXT_DEF    =    1 ;external label
     EXT_ABS    =    2 ;absolute value
     EXT_REF32  =  129 ;32 bit symbol reference
     EXT_COMMON =  130 ;32 bit reference to global data
     EXT_REF16  =  131 ;16 bit symbol reference
     EXT_REF8   =  132 ;8 bit symbol reference
     EXT_DEXT32 =  133 ;32 bit relative data reference
     EXT_DEXT16 =  134 ;16 bit relative data reference
     EXT_DEXT8  =  135 ;8 bit relative data reference
HUNK_SYMBOL     = 1008 ;name of a Long value
HUNK_DEBUG      = 1009 ;special info for a debugger
HUNK_END        = 1010 ;end of main segment
HUNK_HEADER     = 1011 ;info on the following Hunks
HUNK_OVERLAY    = 1013 ;overlay Hunks
HUNK_BREAK      = 1014 ;end of Overlay
HUNK_DREL32     = 1015 ;relative data 32 bit
HUNK_DREL16     = 1016 ;relative data 16 bit
HUNK_DREL8      = 1017 ;relative data 8 bit
HUNK_LIB        = 1018 ;library
HUNK_INDEX      = 1019 ;table
```

6. CLI

CheckSignal	**Check for Cancel signal**

Call:

```
signals = CheckSignal(mask)
D0        -792(a6)    D1

ULONG signals
ULONG mask
```

Function: Tests the given signal bits. The signal bits are masked and passed back. All bits set in the mask are reset in the process structure.

Parameters: mask — Bit mask for signal bits.

Result: signals — Logical AND combination of the mask and the signal bits.

See also: exec.library/Signal

Cli	**Get the address of the calling CLI**

Call:

```
cli_ptr = Cli()
D0        -492(A6)

STRUCT CommandLineInterface *cli_ptr
```

Function: Returns the address of the CLI from which the program was started.

Parameters: None.

Result: Address of the CLI or 0 (Workbench).

Execute	**Execute CLI command**

Call:

```
success = Execute( commandString, input, output )
D0        -222(A6) D1             D2     D3

BOOL success
APTR commandStringExecute
BPTR input,output
```

Function: Attempts to execute a CLI command. The string that contains the command and the parameters is constructed exactly as it would be if entered from the CLI entry line. It can contain any special characters available to CLI. If an input channel is specified, then Execute() will read further instructions from this channel after the execution and change the process in the case of an interactive channel or a re-routing to NIL:. The default output is the current window, but this can be changed by specifying a different output channel.

Processes are started using the RUN command.

Parameters:	commandString	Address of a CLI command line.
	input	FileHandle
	output	FileHandle
Result:	0	Error

Warning: Programs started from the Workbench normally do not have a current output window.

FindCliProc **Find a CLI process**

Call:
```
proc = FindCliProc(num)
D0     -546(A6)    D1

STRUCT Process *proc
LONG   num
```

Function: Returns the CLI process with the given number.

Parameters: num Task number of the CLI process.

Result: Address of the Process structure or 0 if not found.

Warning: To be safe, this routine should only be called when multitasking is turned off.

Input **Get the FileHandle for the default input file**

Call:
```
file = Input()
D0     -54(A6)

BPTR file
```

Function: Returns the FileHandle that was set as the input channel when the program was started. This FileHandle may not be closed.

Result: Input FileHandle or 0

See also: Output()

MaxCli — Get the highest CLI number

Call:

```
number = MaxCli()
D0       -552(A6)

LONG number
```

Function: Returns the highest process number of all the CLI processes running.

Result: Highest CLI process number.

Warning: The highest process number does not necessarily equal the number of processes currently running, since processes with lower numbers may already have been ended.

Output — Get the FileHandle for the default output file

Call:

```
file = Output()
D0     -60(A6)

BPTR file
```

Function: Returns the FileHandle that was set as the output channel when the program was started. This FileHandle may not be closed.

Result: Output FileHandle or 0

See also: Input()

ReadArgs — Interpret CLI argument string

Call:

```
result = ReadArgs(template, array, rdargs)
D0       -798(A6) D1        D2     D3

STRUCT RDArgs *result,*rdargs
APTR   template,array
```

Function: Interprets an argument string using a pattern string, which can contain options such as "Q=Quick". Options are separated by commas in the pattern string. A result for each option is expected to be passed in the longword field. Options can be defined with '/':

/S	Switch, BOOL, 0 = not given.
/K	Keyword, this entry is only filled in if the keyword was found.
/N	Number, a number in decimal format.
/T	Switch, similar to /S.
/A	Required keyword.
/F	Remainder of the line.
/M	Multiple strings (array address with last string address=0).

The RDArgs structure is required for FreeArgs(). Such a structure is normally created with ReadArgs() (parameter = 0).

Parameters:

template	Input format
array	Longword array for results
rdargs	Optional RDArgs structure

Result: RDArgs structure or 0

ReadItem — **Read an argument from an argument string**

Call:

```
value = ReadItem(buffer, maxchars, input)
D0      -810(A6) D1      D2        D3

LONG   value,maxchars
APTR   buffer
STRUCT CHSource *input
```

Function: Reads a word or a character string enclosed in quotes from Input() or a CHSource (if given).

Parameters: buffer Result buffer

maxchars Buffer size

input CHSource structure or 0 (FGetC(Input()))

Result: See data structures.

SelectInput **Set FileHandle for default input channel**

Call:
```
old_fh = SelectInput(fh)
D0       -294(A6)     D1

BPTR old_fh,fh
```

Function: Sets the value that Input() returns for its own CLI process.

Parameters: fh New InputHandle

Result: FileHandle previously returned via Input().

SelectOutput **Set FileHandle for default output channel**

Call:
```
old_fh = SelectOutput(fh)
D0       -300(A6)      D1

BPTR old_fh,fh
```

Function: Sets the value that Output() returns for its own CLI process.

Parameters: fh New OutputHandle

Result: FileHandle previously returned by Output().

SetArgStr **Set argument string**

Call:
```
Oldptr = SetArgStr(ptr)
D0       -540(A6)  D1

APTR ptr, Oldptr
```

Function: Sets the argument string for the running process. The old string must be restored before the program is ended.

Parameters: ptr Address of new argument string.

Result: Oldptr Address of old string.

SetCurrentDirName **Sets name of the current directory in the process**

Call:

```
success = SetCurrentDirName(name)
D0         -558(A6)          D1

BOOL success
APTR name
```

Function: Manipulates the name of the current directory within the CLI structure.

Parameters: name New directory name

Result: 0 Error

SetProgramDir **Sets program directory**

Call:

```
Oldlock = SetProgramDir(lock)
D0         -594(A6)      D1

BPTR lock, Oldlock
```

Function: Sets the value returned by GetProgramDir().

Parameters: lock Directory lock

Result: Oldlock Lock on previous directory.

SetProgramName **Set program name**

Call:

```
success = SetProgramName(name)
D0         -570(A6)       D1

BOOL success
APTR name
```

Function: Changes the program name in the CLI structure.

Parameters: name — Program name

Result: 0 — Error

SetPrompt — **Set prompt for CLI/Shell**

Call:

```
success = SetPrompt(name)
D0        -582(A6)   D1

BOOL success
APTR name
```

Function: Sets prompt text in the CLI structure.

Parameters: name — Prompt string

Result: 0 — Error

SystemTagList — **Execute command from Shell**

Call:

```
error = SystemTagList(command, tags)
D0      -606(A6) D1       D2

LONG   error
APTR   command
STRUCT TagItem *tags
```

Function: Similar to Execute(), but does not read additional instructions from input FileHandle.

Parameters: command — Shell command line

tags — TagItem field for changes.

Result: Return value of command or -1 (error).

VPrintf — **Output a formatted string**

Call:

```
count = VPrintf(fmt, argv)
D0      -954(A6) D1  D2

LONG count
```

```
APTR fmt,argv[]
```

Function: Similar to VFPtrintf, but output occurs after Output().

Parameters:

fmt	Format string for exec/RawDoFmt().
argv	Field containing parameters.

Result: Number of output bytes or -1 (error).

Return values in CLI:

```
RETURN_OK     =  0 ;everything okay
RETURN_WARN   =  5 ;warning
RETURN_ERROR  = 10 ;error occurred
RETURN_FAIL   = 20 ;complete failure, nothing accomplished
```

CLI Cancel bits (CONTROL + C/D/E/F)

```
SIGBREAKB_CTRL_C = 12, SIGBREAKF_CTRL_C = $1000
SIGBREAKB_CTRL_D = 13, SIGBREAKF_CTRL_D = $2000
SIGBREAKB_CTRL_E = 14, SIGBREAKF_CTRL_E = $4000
SIGBREAKB_CTRL_F = 15, SIGBREAKF_CTRL_F = $8000
```

ReadItem() values:

```
ITEM_EQUAL    = -2 ;"=" Symbol
ITEM_ERROR    = -1 ;error
ITEM_NOTHING  =  0 ;"*N", ";", end
ITEM_UNQUOTED =  1 ;no quotes
ITEM_QUOTED   =  2 ;with quotes
```

ReadItem() structure:

```
Dec Hex STRUCTURE   CSource,0
  0  $0 APTR     CS_Buffer ;buffer
  4  $4 LONG     CS_Length ;buffer size
  8  $8 LONG     CS_CurChr ;current character
 12  $C LABEL    CS_SIZEOF
```

ReadArgs() structure:

```
Dec Hex STRUCTURE   RDArgs,0
  0  $0 STRUCT  RDA_Source,CS_SIZEOF ;source string
 12  $C APTR    RDA_DAList           ;PRIVATE
 16 $10 APTR    RDA_Buffer           ;buffer (optional)
```

```
 20 $14 LONG    RDA_BufSiz          ;buffer size
 24 $18 APTR    RDA_ExtHelp         ;optional help
 28 $1C LONG    RDA_Flags           ;Flags
 32 $20 LABEL   RDA_SIZEOF
```

RDA_Flags values:

```
RDAB_STDIN    = 0, RDAF_STDIN    = 1 ;use StdIn
RDAB_NOALLOC  = 1, RDAF_NOALLOC  = 2 ;no extra buffer
RDAB_NOPROMPT = 2, RDAF_NOPROMPT = 4 ;no input

MAX_TEMPLATE_ITEMS = 100 ;max. number of arguments (must be divisible by 4!!!)

MAX_MULTIARGS = 128      ;max. number of multiple strings
```

CLI structure:

```
Dec Hex STRUCTURE CommandLineInterface,0
  0  $0 LONG   cli_Result2         ;IoErr() value
  4  $4 BSTR   cli_SetName         ;current directory
  8  $8 BPTR   cli_CommandDir      ;command directory
 12  $C LONG   cli_ReturnCode      ;return value
 16 $10 BSTR   cli_CommandName     ;program name
 20 $14 LONG   cli_FailLevel       ;error level
 24 $18 BSTR   cli_Prompt          ;prompt string
 28 $1C BPTR   cli_StandardInput   ;default input
 32 $20 BPTR   cli_CurrentInput    ;current input
 36 $24 BSTR   cli_CommandFile     ;batch filename
 40 $28 LONG   cli_Interactive     ;BOOL if terminal
 44 $2C LONG   cli_Background      ;BOOL if RUN command
 48 $30 BPTR   cli_CurrentOutput   ;current output
 52 $34 LONG   cli_DefaultStack    ;stack size in Longs
 56 $38 BPTR   cli_StandardOutput  ;default output
 60 $3C BPTR   cli_Module          ;program's SegList
 64 $40 LABEL  cli_SIZEOF
```

System() Tags:

```
SYS_Dummy       = TAG_USER+32
SYS_Input       = SYS_Dummy+1 ;set input FileHandle
SYS_Output      = SYS_Dummy+2 ;set output FileHandle
SYS_Asynch      = SYS_Dummy+3 ;close input/output
SYS_UserShell   = SYS_Dummy+4 ;not to boot Shell
SYS_CustomShell = SYS_Dummy+5 ;specific Shell (name)
SYS_Error       = SYS_Dummy+? ;anything else = error
```

7. Files

ChangeMode — **Change access to lock or FileHandle**

Call:

```
success = ChangeMode(type, object, newmode)
D0        -450(A6)   D1    D2      D3

BOOL  success
ULONG type
BPTR  object
ULONG newmode
```

Function: Changes the access mode for a lock or FileHandle.

Parameters:

type	Data structure type: CHANGE_FH or CHANGE_LOCK
object	Lock or FileHandle (according to type)
newmode	New access mode

Result: 0 — Change not allowed

Warning: Invalid values can lead to a system crash.

See also: Lock(), Open()

Close — **Close file**

Call:

```
success = Close( file )
D0        -36(A6) D1

BOOL success
BPTR file
```

Function: Close a file opened by the program itself.

Parameters: file — BCPL address of the file's FileHandle.

Result: 0 if the file could not be closed, for example, because a buffered output is still in process.

See also: Open()

DeleteFile — **Delete a file**

Call:

```
success = DeleteFile( name )
D0        -72(A6)     D1

BOOL success
APTR name
```

Function: Attempts to delete a file or directory.

Parameters: name — String containing file or directory name.

Result: 0 — Could not be deleted.

See also: IoErr()

ExamineFH — **Retrieve information on a file**

Call:

```
success = ExamineFH(fh, fib)
D0        -390(A6)  D1  D2

BOOL success
BPTR fh
STRUCT FileInfoBlock *fib
```

Function: Examines a FileHandle and fills out a FileInfoBlock. Be careful, because fib_Size can contain invalid values.

Parameters: fh — FileHandle

fib — Address of a FileInfoBlock structure.

Result: 0 — Error

FGetC — **Read characters from a file**

Call:

```
char = FGetC(fh)
D0   -306(A6) D1

LONG char
```

```
BPTR fh
```

Function: Reads a byte from the given file (buffered).

Parameters: fh FileHandle

Result: Byte (value 0-255) or -1 if end-of-file or error.

Flush — **Clears the buffer used for a buffered I/O**

Call:

```
success = Flush(fh)
D0      -360(A6) D1

BOOL success
BPTR fh
```

Function: Deletes all buffers for a file. When reading from a file, Seek() is used to locate the old position.

Parameters: fh FileHandle

Result: 0 Error

FPutC — **Output a character**

Call:

```
char = FPutC(fh, char)
D0  -312(A6) D1  D2

LONG  char
BPTR  fh
UBYTE char
```

Function: Buffered output of an individual character.

Parameters: fh FileHandle

char Output byte

Result: The printed character or EOF in the case of an error.

FRead — **Read data blocks from a file**

Call:

```
count = FRead(fh, buf, blocklen, blocks)
D0    -324(A6) D1  D2   D3        D4

LONG  count
BPTR  fh
APTR  buf
ULONG blocklen,blocks
```

Function: Attempts a buffered read of the given number of blocks from a file.

Parameters:

fh	FileHandle to use for buffered I/O.
buf	Buffer for writing the blocks that are read.
blocklen	Block length
blocks	Number of blocks to read.

Result: Number of blocks actually read (EOF or read error aborts the read operation).

Warning: You must first use SetIoErr() to delete the error code if a query is necessary.

FWrite — **Write data blocks to a file**

Call:

```
count = FWrite(fh, buf, blocklen, blocks)
D0    -330(A6) D1  D2   D3        D4

LONG  count
BPTR  fh
APTR  buf
ULONG blocklen,blocks
```

Function: Attempts a buffered write of the given number of data blocks to a file.

Parameters:

fh	FileHandle

buf	Buffer containing the data to be written.
blocklen	Block length
blocks	Number of blocks to write.

Result: Number of blocks actually written (aborted in the case of an error).

Warning: Use SetIoErr to delete the error code before using IoErr().

IsInteractive — Is a file a virtual terminal?

Call:

```
status = IsInteractive( file )
D0       -216(A6)       D1

BOOL status
BPTR file
```

Function: Checks a file to see if it's a virtual terminal (for example, a console window).

Parameters: file — FileHandle of the file.

Result: 0 — Normal file, not a terminal.

Lock — Obtain access to a file or directory

Call:

```
lock   = Lock( name, accessMode )
D0       -84(A6) D1   D2

BPTR lock
APTR name
LONG accessMode
```

Function: Attempts to secure access to a file or directory. This can be exclusive access (ACCESS_WRITE), which prevents other programs from accessing the file, or shared access (ACCESS_READ).

Parameters: name — Filename and/or path name

accessMode
Access mode

Result: BPTR to a lock structure or 0.

LockRecord — **Obtain access to part of a file**

Call:

```
success = LockRecord(fh,offset,length,mode,timeout)
D0        -270(A6)   D1 D2     D3     D4   D5

ULONG success,offset,length,mode,timeout
BPTR  fh
```

Function: Grants access to part of a file. A specific timeout period can be set.

Parameters:

fh — FileHandle for the file.
offset — Start of record
length — End of record
mode — Access mode:

REC_EXCLUSIVE
Exclusive access

REC_EXCLUSIVE_IMMED
Exclusive access, ignore timeout

REC_SHARED — Shared access

REC_SHARED_IMMED
Shared access, ignore timeout

timeout — Timeout period in 1/50th seconds (0 allowed).

Result: 0 — Error or access not possible.

LockRecords — **Secure access to several parts of a file**

Call:

```
success = LockRecords(record_array,timeout)
D0        -276(A6)    D1           D2
```

```
BOOL   success
STRUCT RecordLock *record_array
ULONG  timeout
```

Function: This function locks several parts of the file at once. A specific timeout period can be set.

Parameters: record_array
List of RecordLock structures.

timeout Timeout period (0 allowed)

Result: 0 Error or one or more of the records not free.

Open **Open a file**

Call:

```
file = Open( name, accessMode )
D0     -30(A6) D1   D2

BPTR file
APTR name
LONG accessMode
```

Function: Attempts to open an existing file (MODE_OLDFILE) or create a new file (MODE_NEWFILE). If MODE_READWRITE is specified, a file is opened and created, if it doesn't already exist.

Parameters: name Filename

accessMode
Access mode

Result: BPTR to a FileHandle structure or 0.

OpenFromLock	Open file associated with a lock

Call:

```
fh = OpenFromLock(lock)
D0   -378(A6)      D1

BPTR fh,lock
```

Function: Opens a file associated with a given lock. The access mode is determined by the lock.

Parameters: lock — Lock structure of a file.

Result: FileHandle or 0

Read	Read data from a file

Call:

```
actualLength = Read( file, buffer, length )
D0             -42(A6) D1   D2      D3

LONG actualLength,length
BPTR file
APTR buffer
```

Function: Read data from a given file to a buffer.

Parameters:

file	FileHandle
buffer	Read buffer
length	Number of bytes to read

Result: Number of bytes actually read (0 indicates end-of-file) or -1 (error).

Rename	Rename a file or directory

Call:

```
success = Rename( oldName, newName )
D0        -78(A6) D1       D2

BOOL success
APTR oldName,newName
```

Function: Assigns a new name to a file or directory. If a new path is also given, the renamed object is moved to the new directory.

Parameters: oldName Old name

newName New name

Result: 0 Error

SameLock **Compare two locks**

Call:

```
value = SameLock(lock1, lock2)
D0      -420(A6) D1     D2

LONG value
BPTR lock1,lock2
```

Function: Compare two locks. Returns a value of LOCK_SAME if the same object is found, LOCK_SAME_HANDLER for different objects that belong to the same handler, or LOCK_DIFFERENT if the handlers are different.

Parameters: lock1,lock2
The locks to be compared.

Result: See function.

Seek **Change read/write position in a file**

Call:

```
oldPosition = Seek( file, position, mode )
D0            -66(A6) D1  D2        D3

LONG oldPosition,position,mode
BPTR file
```

Function: Seek() sets the read/write position within a file relative to the start of the file, the current position, or the end of the file. The old position is returned as the result.

Parameters: file FileHandle for the file.

position Relative value

mode Start, relative, or end

Result: Old position relative to the start of the file.

SetComment **Set file comments**

Call:

```
success = SetComment( name, comment )
D0        -180(A6)    D1    D2

BOOL success
APTR name,comment
```

Function: Sets new comments for the given file.

Parameters: name Filename

comment Comment string (max. 80 characters)

Result: D0 0 in case of error

SetFileDate **Set revision date for a file**

Call:

```
success = SetFileDate(name, date)
D0        -396(A6)    D1    D2

BOOL   success
APTR   name
STRUCT DateStamp *date
```

Function: Sets the revision date for a file or directory, as long as it's allowed by the filesystem.

Parameters: name Object name

date DateStamp structure with new date.

Result: 0 Error

SetFileSize — **Set the size of a file**

Call:

```
newsize = SetFileSize(fh, offset, mode)
D0        -456(A6)    D1  D2      D3

LONG newsize,offset,mode
BPTR fh
```

Function: Sets the file size for the given file, as long as this is allowed by the filesystem. The position is specified the same as with Seek().

Parameters:

fh	FileHandle for the file.
offset	Relative value
mode	OFFSET_BEGINNING, OFFSET_CURRENT or OFFSET_END.

Result: File length or -1 (error).

SetProtection — **Set protection status for a file**

Call:

```
success = SetProtection( name, mask )
D0        -186           D1    D2

BOOL success
APTR name
LONG mask
```

Function: Sets the protection status for a file or directory. The status consists of an OR combination of various flags:

Bit 4: A	1	file unchanged	0	file changed
Bit 3: R	1	read not allowed	0	read allowed
Bit 2: W	1	write not allowed	0	write allowed
Bit 1: E	1	not executable	0	executable
Bit 0: D	1	delete not allowed	0	delete allowed

Parameters:

name	Filename
mask	Protection status

Result: 0 Error

UnGetC **Returns a byte to the buffer**

Call:

```
value = UnGetC(fh, character)
D0     -318(A6) D1   D2

LONG value,character
BPTR fh
```

Function: Returns a byte to the input buffer. If the value -1 is passed, the last character read from the buffer is put back.

Parameters: fh FileHandle for buffered I/O.

character Character or -1

Result: Returned character or 0 (error).

UnLock **Remove lock**

Call:

```
UnLock( lock )
-90(A6) D1

BPTR lock
```

Function: Removes a lock and frees the allocated memory.

Parameters: lock BCPL pointer to a lock structure.

UnLockRecord **Free part of a file**

Call:

```
success = UnLockRecord(fh,offset,length)
D0         -282(A6)     D1 D2     D3

BOOL  success
BPTR  fh
ULONG offset,length
```

Function: Frees part of a file that was locked with LockRecord().

Parameters: fh FileHandle given with LockRecord().

	offset	Start of record
	length	Length of record
Result:	0	Error

UnLockRecords — **Free several parts of a file**

Call:

```
success = UnLockRecords(record_array)
D0          -288(A6)       D1

BOOL   success
STRUCT RecordLock *record_array
```

Function: Frees multiple records locked with LockRecords().

Parameters: record_array
List of records to free

Result: 0 Error

VFPrintf — **Write formatted string to a file**

Call:

```
count = VFPrintf(fh, fmt, argv)
D0      -354(A6) D1  D2   D3

LONG count
BPTR fh
APTR fmt,argv[]
```

Function: Formats a string and does a buffered write of the result to a file.

Parameters:	fh	FileHandle for the file.
	fmt	Format string for exec/RawDoFmt().
	argv	Address of data array.

Result: Number of bytes written or -1 (error).

VFWritef	**VFPrintf for BCPL strings**

Call:

```
count = VFWritef(fh, fmt, argv)
D0      -348(A6) D1  D2   D3

LONG count
BPTR fh
APTR fmt,argv[]
```

Functions, Parameters, and Results:

Same as VFPrintf, except the strings are BSTR or BCPL.

Write	**Write to a file**

Call:

```
returnedLength = Write( file, buffer, length )
D0               -48(A6) D1    D2      D3

LONG returnedLength,length
BPTR file
APTR buffer
```

Function: Writes a specified number of bytes to a file.

Parameters:

file	FileHandle
buffer	Address of the bytes.
length	Number of bytes to write.

Result: Number of bytes actually written.

Open() modes:

```
MODE_OLDFILE   = 1005 ;open existing file
MODE_NEWFILE   = 1006 ;create new file
MODE_READWRITE = 1004 ;open file (1005 (->1006))
```

FileHandle structure:

```
Dec Hex STRUCTURE FileHandle,0
  0  $0 APTR   fh_Link         ;Exec message
  4  $4 APTR   fh_Port         ;answer port for Packet
  8  $8 APTR   fh_Type         ;port for PutMsg()
```

```
 12  $C LONG    fh_Buf
 16 $10 LONG    fh_Pos
 20 $14 LONG    fh_End
 24 $18 LABEL   fh_Func1
 24 $18 LONG    fh_Funcs
 28 $1C LONG    fh_Func2
 32 $20 LONG    fh_Func3
 36 $24 LABEL   fh_Arg1
 36 $24 LONG    fh_Args
 40 $28 LONG    fh_Arg2
 44 $2C LABEL   fh_SIZEOF
```

Points of reference for Seek():

```
OFFSET_BEGINNING = -1 ;start of file
OFFSET_CURRENT   =  0 ;current position
OFFSET_END       =  1 ;end of file
OFFSET_BEGINING  = OFFSET_BEGINNING
```

Structure of Lock(), etc.:

```
Dec Hex STRUCTURE FileLock,0
  0  $0 BPTR     fl_Link   ;next Lock
  4  $4 LONG     fl_Key    ;block number on disk
  8  $8 LONG     fl_Access ;access mode
 12  $C APTR     fl_Task   ;Handler port
 16 $10 BPTR     fl_Volume ;Volume Node (DosList)
 20 $14 LABEL    fl_SIZEOF
```

Lock() modes:

```
SHARED_LOCK    = -2 ;shared access
EXCLUSIVE_LOCK = -1 ;exclusive access
ACCESS_READ    = SHARED_LOCK
ACCESS_WRITE   = EXCLUSIVE_LOCK
```

SameLock() values:

```
LOCK_SAME         =  0 ;objects identical
LOCK_SAME_HANDLER =  1 ;objects have same Handler
LOCK_DIFFERENT    = -1 ;completely different Locks
```

ChangeMode() types:

```
CHANGE_LOCK = 0 ;Lock structure
CHANGE_FH   = 1 ;FileHandle structure
```

MakeLink() values:

```
LINK_HARD = 0
LINK_SOFT = 1
```

LockRecord()/LockRecords() modes:

```
REC_EXCLUSIVE       = 0 ;exclusive access
REC_EXCLUSIVE_IMMED = 1 ;exclusive with no waiting
REC_SHARED          = 2 ;shared access
REC_SHARED_IMMED    = 3 ;shared with no waiting
```

LockRecords()/UnLockRecords() structure:

```
Dec Hex STRUCTURE RecordLock,0
  0  $0 BPTR     rec_FH       ;FileHandle
  4  $4 ULONG    rec_Offset   ;start (offset)
  8  $8 ULONG    rec_Length   ;record length
 12  $C ULONG    rec_Mode     ;Lock type
 16 $10 LABEL    RecordLock_SIZEOF
```

8. Strings

AddPart	**Add filename to path string**

Call:

```
success = AddPart( dirname, filename, size )
D0        -882(A6) D1       D2        D3

BOOL  success
APTR  dirname
APTR  filename
ULONG size
```

Function: Adds a filename to a path name according to DOS conventions. The filename may also contain path information. If the filename is a complete path, then the old path is replaced.

Parameters:

dirname	Path name
filename	(path +)filename, '/' or ':' allowed
size	Size of buffer that contains dirname.

Result: 0 Error (buffer too small)

See also: Filepart(), PathPart()

DateToStr **Generate string from DateStamp**

Call:

```
success = DateToStr( datetime )
D0        -744(A6)   D1

BOOL   success
STRUCT DateTime *datetime
```

Function: Generates a string for a DateStamp structure according to the given DateTime structure.

Parameters: datetime — Address of a DateTime structure, which must be initialized as follows:

dat_Stamp	Copy of the DateStamp.
dat_Format	String format (FORMAT_DOS dd-mmm-yy, FORMAT_INT yy-mmm-dd, FORMAT_USA mm-dd-yy or FORMAT_CDN dd-mm-yy).
dat_Flags	DTF_SUBST generates the day of the week (Monday, Today...).
dat_StrDay	Address of the day buffer or 0 if not used.
dat_StrDate	Address of the date buffer or 0 if not used.
dat_StrTime	Address of the time buffer or 0 if not used.

Result: 0 DateStamp error

See also: DateStamp(), StrToDate()

Fault — **Generate error message**

Call:

```
success = Fault(code, header, buffer, len)
D0        -468(A6) D1   D2      D3      D4

BOOL success
LONG code,len
APTR header,buffer
```

Function: Converts an error code into a string for the console window, printer, or a text file (with line feed). This is preceded by the given header text. Error messages should not be more than 80 characters, and headers should not be more than 60. If a certain code has no message text, the string "Error code <number>" is used.

Parameters:

code	Error code from IoErr().
header	Header text
buffer	Buffer for the complete error message.
len	Buffer length

Result:

0	Buffer too small or some other error.

FGets — **Read a line from a file**

Call:

```
buffer = FGets(fh, buf, len)
D0     -336(A6) D1  D2   D3

APTR  buffer,buf
BPTR  fh
ULONG len
```

Function: Reads a line from a file into a buffer. One character less than the length of the buffer can be read, because the last character in the buffer must always be set to 0. If the entire line fits in the buffer, the character before the null byte is an end-of-line code (LF or CR). The I/O is buffered.

Parameters:

fh	FileHandle

buf	Buffer address
len	Buffer length

Result: Address of the buffer or 0 if no characters could be read. If the end of the file is reached before the call, IoErr()=0. If an error occurs, IoErr()<>0.

FilePart — Extract the filename from a path specification

Call:

```
fileptr = FilePart( path )
D0        -870(A6)  D1

APTR fileptr,path
```

Function: Returns the start address for the file in a given path specification.

Parameters: path — Path string according to DOS conventions.

Result: Start address for the file.

See also: PathName()

FindArg — Find a keyword in an argument string

Call:

```
index = FindArg(template, keyword)
D0      -804(A6) D1       D2

LONG index
APTR keyword,template
```

Function: Returns the argument number for a given keyword.

Parameters:

keyword	Keyword to search for.
template	Argument string

Result: Argument number of the given keyword or -1 if the keyword was not found.

FPuts — **Write a string to a file**

Call:

```
error = FPuts(fh, str)
D0    -342(A6) D1  D2

LONG error
BPTR fh
APTR str
```

Function: Buffered write of a string to a file.

Parameters:

fh	FileHandle
str	String ending in 0.

Result: Negative Error

GetArgStr — **Retrieves an argument string from CLI**

Call:

```
ptr = GetArgStr()
D0     -534(A6)

APTR ptr
```

Function: Returns the argument address found in the A0 register when the program is started. This is only useful for high level languages that do not use an argument parser.

Result: Address of the argument string from CLI or 0.

GetCurrentDirName — **Retrieve the name of the current directory**

Call:

```
success = GetCurrentDirName(buf, len)
D0         -564(A6)             D1   D2

BOOL success
APTR buf
LONG len
```

Function: Gets the name of the current directory from the CLI structure of its own process.

Parameters: buf Buffer for the name.

len Buffer length

Result: 0 No CLI structure or no directory.

GetProgramName **Returns the program's own name**

Call:
```
success = GetProgramName(buf, len)
D0        -576(A6)         D1   D2

BOOL success
APTR buf
LONG len
```

Function: Copies the program name from the CLI structure to a buffer.

Parameters: buf Buffer address

len Buffer length

Result: 0 Buffer too small or CLI structure not found.

GetPrompt **Retrieve the prompt string for a process**

Call:
```
success = GetPrompt(buf, len)
D0        -588(A6)  D1   D2

BOOL success
APTR buf
LONG len
```

Function: Copies the prompt string from the CLI structure to a buffer.

Parameters: buf Buffer address

len Buffer length

Result: 0 Buffer too small or CLI structure not found.

MatchPattern	**Test a string against a pattern**

Call:

```
match = MatchPattern(pat, str)
D0      -846(A6)     D1   D2

BOOL match
APTR pat,str
```

Function: Checks to see if the given string matches a given pattern.

Parameters:

pat	Pattern string from ParsePattern().
str	String to be checked.

Result:

0	String does not match pattern.

NameFromFH	**Get the filename from the FileHandle**

Call:

```
success = NameFromFH(fh, buffer, len)
D0        -408(A6)   D1  D2      D3

BOOL success
BPTR fh
APTR buffer
LONG len
```

Function: Writes the file and path name of the given FileHandle to a buffer.

Parameters:

fh	FileHandle
buffer	Buffer for result string.
len	Buffer length

Result:

0	Error or buffer too small.

NameFromLock	**Retrieve the name and path of a lock**

Call:

```
success = NameFromLock(lock, buffer, len)
D0        -402(A6)     D1    D2      D3
```

```
BOOL success
BPTR lock
APTR buffer
LONG len
```

Function: Writes the name and path of the given lock to a buffer.

Parameters:

lock	Lock
buffer	Buffer
len	Buffer length

Result: 0 Error (IoErr()=ERROR_LINE_TOO_LONG)

ParsePattern **Generate token string for MatchPattern()**

Call:

```
IsWild = ParsePattern(Source,Dest,DestLength)
d0       -840(A6)      D1     D2   D3

LONG IsWild,DestLength
APTR Source,Dest
```

Function: Creates a token string for the MatchPattern() function.

Parameters:

source	Pattern string
dest	Buffer for token string.
DestLength	Buffer length (min. 2*Source+2).

Result:
1 string contains wildcards (#, ? etc.)
0 string contains no wildcards.
-1 buffer too small or error.

PathPart **Retrieve the end of a path specification**

Call:

```
fileptr = PathPart( path )
D0        -876(A6)  D1

APTR fileptr,path
```

Function: Returns the address of the end of a path specification.

Parameters: path — Filename (with path) according to DOS standards.

Result: Address of the part of the path that disappears when another file is selected in a file selection box.

See also: FilePart()

SplitName — Retrieve part of a path specification

Call:

```
newpos = SplitName(name, separator, buf, oldpos, size)
D0       -414(A6)  D1    D2         D3   D4      D5

WORD  newpos,oldpos
APTR  name,buf
UBYTE separator
LONG  size
```

Function: Copies the next part of a complete file/path name to a separate buffer.

Parameters:

name	Filename with path.
separator	ASCII code of the separation character.
buf	Buffer
oldpos	Old position in string.
size	Buffer size in bytes.

Result: New start position for the next call (newpos->oldpos) or -1.

StrToDate — Convert a string to a DateStamp

Call:

```
success = StrToDate( datetime )
D0        -750(A6)   D1

BOOL   success
STRUCT DateTime *datetime
```

Function: Fills in a DateStamp structure using the information from a string.

Parameters: datetime Initialized (!) DateTime structure.

Result: 0 Error

See also: DateToStr(), libraries/datetime.h

StrToLong — Convert a decimal string to a longword

Call:

```
characters = StrToLong(string,value)
D0           -816(A6)  D1     D2

LONG characters
APTR string,value
```

Function: Converts a string containing a decimal value into a longword.

Parameters: string Decimal string

value Address of the resulting longword.

Result: Number of decimal places found or -1 (longword is then set to 0).

StrToDate()/DateToStr() structure:

```
Dec Hex STRUCTURE  DateTime,0
  0  $0 STRUCT  dat_Stamp,ds_SIZEOF ;DateStamp structure
 12  $C UBYTE   dat_Format          ;dat_StrDate format
 13  $D UBYTE   dat_Flags           ;Flags (see below)
 14  $E CPTR    dat_StrDay          ;day of the week string
 18 $12 CPTR    dat_StrDate         ;date string
 22 $16 CPTR    dat_StrTime         ;time string
 26 $1A LABEL   dat_SIZEOF

LEN_DATSTRING = 16 ;length of a date string
```

Flags, Bits:

```
DTB_SUBST = 0, DTF_SUBST = 1 ;create "Today","Tomorrow"...
DTB_FUTURE= 1, DTF_FUTURE= 2 ;a future day
```

Date formats:

```
FORMAT_DOS = 0 ;dd-mmm-yy DOS format
FORMAT_INT = 1 ;yy-mm-dd  international format
FORMAT_USA = 2 ;mm-dd-yy  USA format
FORMAT_CDN = 3 ;dd-mm-yy  Canadian format
FORMAT_MAX = FORMAT_CDN
```

9. Time

CompareDates — **Compare two DateStamps**

Call:

```
result = CompareDates(date1,date2)
D0       -738(A6)      D1    D2

LONG   result
STRUCT DateStamp *date1
STRUCT DateStamp *date2
```

Function: Compares the dates given in two DateStamp structures.

Parameters: date1/date2
DateStamp structures

Result:

negative: date1 later than date2

0: date1 equals date2

positive: date2 later than date1

See also: DateStamp()

DateStamp — **Retrieves the current time**

Call:

```
DateStamp( ds )
-192(A6)   D1

STRUCT DateStamp *ds
```

Function: Fills the given DateStamp structure with the current time.

Parameters: ds Address of a DateStamp structure.

Result: The structure is filled.

Delay	**Suspend own process for a certain time period**

Call:

```
Delay( ticks )
-198(A6) D1

ULONG ticks
```

Function: Own process is suspended for the given time period.

Parameters: ticks — Time period in 1/50th second.

WaitForChar	**Wait for input**

Call:

```
status = WaitForChar( file, timeout )
D0       -204(A6)     D1    D2

BOOL status
BPTR file
LONG timeout
```

Function: Waits a specified number of microseconds (1/1000000) to see if a character can be successfully read from the given file. This is very important for working with ports and terminals.

Parameters: file — FileHandle for the file.

timeout — Time period in microseconds.

Result: 0 — No character received during the wait period.

```
Dec Hex STRUCTURE DateStamp,0
  0  $0 LONG  ds_Days   ;days since Jan. 1, 1978
  4  $4 LONG  ds_Minute ;minutes since midnight
  8  $8 LONG  ds_Tick   ;ticks since last minute
 12  $C LABEL ds_SIZEOF

TICKS_PER_SECOND = 50   ;number of ticks per second
```

10. Environment Variables

DeleteVar — **Delete local environment variable**

Call:

```
success = DeleteVar( name, flags )
D0         -912(A6)    D1     D2

BOOL  success
APTR  name
ULONG flags
```

Function: Delete a local ENV variable.

Parameters: name — String address with variable name (structured like a filename).

flags — Flags for variable type and function.

GVF_LOCAL_ONLY
Local variable (default)

GVF_GLOBAL_ONLY
Global variable

Result: 0 — Error.

See also: GetVar(), SetVar()

FindVar — **Find local variable**

Call:

```
var = FindVar( name, type )
D0    -918(A6) D1    D2

STRUCT LocalVar *var
APTR   name
ULONG  type
```

Function: Retrieves a local variable.

Parameters: name — Variable name (structured like a path name)

type — Variable type

Result: LocalVar structure or 0

See also: GetVar(), SetVar(), DeleteVar(), dos/var.h

GetVar **Retrieve the value of a variable**

Call:

```
len = GetVar( name, buffer, size, flags )
D0    -906(A6) D1   D2      D3    D4

LONG  len,size
APTR  name,buffer
ULONG flags
```

Function: Returns the value of an environment variable. If GVF_BINARY_VAR is not set, the function is interrupted when an LF character is encountered.

Parameters:

name	Variable name (AmigaDOS path)
buffer	Buffer for the variable contents.
size	Buffer size
flags	Variable type

GVF_GLOBAL_ONLY
Global ENV variable

GVF_LOCAL_ONLY
Process-specific ENV variable

GVF_BINARY_VAR
With control character

Result: Total length of the variable (may be different from the buffer contents if the buffer terminates with 0) or -1 in the case of an error (variable not found).

See also: SetVar(), DeleteVar(), dos/var.h

SetVar — **Create or set the value of a variable**

Call:

```
success = SetVar( name, buffer, size, flags )
D0        -900(A6) D1    D2      D3    D4

BOOL  success
APTR  name,buffer
LONG  size
ULONG flags
```

Function: Sets a local or environment variable. ASCII strings are only recommended.

Parameters:

name	Filename of the variable.
buffer	Contents of variable.
size	Variable size (-1 = string ending in 0)
flags	Variable type

Result: 0 Error

See also: GetVar(), DeleteVar(), dos/var.h

Structure of pr_LocalVars list:

```
Dec Hex STRUCTURE LocalVar,0
  0  $0 STRUCT    lv_Node,LN_SIZE ;node
 14  $E UWORD     lv_Flags        ;type
 16 $10 APTR      lv_Value        ;buffer
 20 $14 ULONG     lv_Len          ;buffer length
 24 $18 LABEL     LocalVar_SIZEOF
```

LN_TYPE bits in lv_Node:

```
LV_VAR      = 0      ;a variable
LV_ALIAS    = 1      ;an ALIAS definition
LVB_IGNORE  =  7, LVF_IGNORE      =  $80
```

Values for variable functions:

```
GVB_GLOBAL_ONLY =  8, GVF_GLOBAL_ONLY = $100
GVB_LOCAL_ONLY  =  9, GVF_LOCAL_ONLY  = $200
GVB_BINARY_VAR  = 10, GVF_BINARY_VAR  = $400
```

11. Errors and Requesters

ErrorReport **Display Retry/Cancel error requester**

Call:

```
status = ErrorReport(code, type, arg1, device)
D0       -480(A6)    D1    D2    D3    A0

BOOL   status
LONG   code,type
ULONG  arg1
STRUCT MsgPort *device
```

Function: Displays the appropriate error requester.

Parameters:

code — Error code (ERROR_..., ABORT_...)

type — Requester type:

REPORT_LOCK	arg1 is a lock (BPTR).
REPORT_FH	arg1 is a FileHandle (BPTR).
REPORT_VOLUME	arg1 is a volume node (CPTR).

arg1 — Parameter (according to type)

device — (optional) HandlerPort address (only needed for REPORT_LOCK with arg1=0)

Result:

DOS_TRUE — 'Cancel' or error

0 — 'Retry' or DISKINSERTED (for certain errors)

IoErr — **Retrieve additional system error information**

Call:

```
error = IoErr()
D0      -132(A6)

LONG error
```

Function: For functions that return a value of 0 when errors occur. IoErr() is used to retrieve more information on the cause of the error. Other functions use IoErr() to return a second result to accommodate programming in C.

Result: Error code or second result.

See also: Open(), DoPkt()

PrintFault — **Send error message to the output channel**

Call:

```
success = PrintFault(code, header)
D0        -474(A6)    D1    D2

BOOL success
LONG code
APTR header
```

Function: The given header string is combined with the error message associated with the given error code and sent in a buffered output to the default output channel.

Parameters:

code	Error code (see IoErr())
header	Header text to precede the error message text.

Result: 0 Error

PutStr — **Send a string to the default output channel**

Call:

```
error = PutStr(str)
D0      -948(A6) D1

LONG error
APTR str
```

Function: Buffered output of a given string to the default output channel.

Parameters: str   Output string

Result: 0 in the case of an error.

SetIoErr **Set error code**

Call:

```
oldcode = SetIoErr(code)
D0        -462(A6) D1

LONG code
```

Function: Sets a new value for the result of the IoErr() function (pr_Result2).

Parameters: code   Error code for IoErr().

Result: oldcode   Previous value of pr_Result2.

IoErr() error codes:

```
ERROR_NO_FREE_STORE            = 103 ;not enough storage space
ERROR_TASK_TABLE_FULL          = 105 ;too many Tasks
ERROR_BAD_TEMPLATE             = 114 ;command format error
ERROR_BAD_NUMBER               = 115 ;invalid value
ERROR_REQUIRED_ARG_MISSING     = 116 ;missing a required argument
ERROR_KEY_NEEDS_ARG            = 117 ;keyword with no argument
ERROR_TOO_MANY_ARGS            = 118 ;too many arguments
ERROR_UNMATCHED_QUOTES         = 119 ;quotes missing
ERROR_LINE_TOO_LONG            = 120 ;line too long
ERROR_FILE_NOT_OBJECT          = 121 ;not a normal file
ERROR_INVALID_RESIDENT_LIBRARY= 122 ;error in header Hunk
ERROR_NO_DEFAULT_DIR           = 201 ;no default directory
ERROR_OBJECT_IN_USE            = 202 ;object being used
ERROR_OBJECT_EXISTS            = 203 ;object already exists
ERROR_DIR_NOT_FOUND            = 204 ;unknown directory
ERROR_OBJECT_NOT_FOUND         = 205 ;object could not be found
ERROR_BAD_STREAM_NAME          = 206 ;invalid name
ERROR_OBJECT_TOO_LARGE         = 207 ;object is too big
ERROR_ACTION_NOT_KNOWN         = 209 ;unknown Packet
ERROR_INVALID_COMPONENT_NAME   = 210 ;invalid component name
ERROR_INVALID_LOCK             = 211 ;invalid Lock structure
ERROR_OBJECT_WRONG_TYPE        = 212 ;wrong object type
```

```
ERROR_DISK_NOT_VALIDATED       = 213 ;disk is not validated
ERROR_DISK_WRITE_PROTECTED     = 214 ;disk is write-protected
ERROR_RENAME_ACROSS_DEVICES    = 215 ;rename error
ERROR_DIRECTORY_NOT_EMPTY      = 216 ;directory is not empty
ERROR_TOO_MANY_LEVELS          = 217 ;too many levels
ERROR_DEVICE_NOT_MOUNTED       = 218 ;unknown device
ERROR_SEEK_ERROR               = 219 ;Seek() error
ERROR_COMMENT_TOO_BIG          = 220 ;comment too long
ERROR_DISK_FULL                = 221 ;disk is full
ERROR_DELETE_PROTECTED         = 222 ;delete protected
ERROR_WRITE_PROTECTED          = 223 ;write protected
ERROR_READ_PROTECTED           = 224 ;read protected
ERROR_NOT_A_DOS_DISK           = 225 ;not a DOS disk
ERROR_NO_DISK                  = 226 ;no disk found
ERROR_NO_MORE_ENTRIES          = 232 ;end was reached
ERROR_IS_SOFT_LINK             = 233 ;software link
ERROR_OBJECT_LINKED            = 234 ;object linked
ERROR_BAD_HUNK                 = 235 ;invalid Hunk type
ERROR_NOT_IMPLEMENTED          = 236 ;not implemented
ERROR_RECORD_NOT_LOCKED        = 240 ;(see LockRecord())
ERROR_LOCK_COLLISION           = 241 ;Lock collision
ERROR_LOCK_TIMEOUT             = 242 ;Lock timeout period expired
ERROR_UNLOCK_ERROR             = 243 ;Unlock error
ERROR_BUFFER_OVERFLOW          = 303 ;buffer too small
ERROR_BREAK                    = 304 ;break character
ERROR_NOT_EXECUTABLE           = 305 ;not executable

FAULT_MAX    = 82 ;max. length of an error string
```

Error message structure:

```
Dec Hex STRUCTURE ErrorString,0
  0  $0 APTR    estr_Nums
  4  $4 APTR    estr_Strings
  8  $8 LABEL   ErrorString_SIZEOF
```

ErrorReport() types:

```
REPORT_STREAM = 0
REPORT_TASK   = 1
REPORT_LOCK   = 2
REPORT_VOLUME = 3
REPORT_INSERT = 4 ;"please insert volume..."
```

ErrorReport() error codes:

```
ABORT_DISK_ERROR = 296 ;read/write error
ABORT_BUSY       = 288 ;"You MUST replace..."
```

DOS boolean values:

```
DOSTRUE  = -1 ;true
DOSFALSE =  0 ;false
```

General values:

```
BITSPERBYTE  =         8 ; 8 bits = 1 byte
BYTESPERLONG =         4 ;4 bytes = 1 long
BITSPERLONG  =        32 ;32 bits = 1 long
MAXINT       = $7FFFFFFF ;maximum LONG value
MININT       = $80000000 ;minimum LONG value
```

Basis structure:

```
Dec Hex STRUCTURE DosLibrary,0
  0  $0 STRUCT  dl_lib,LIB_SIZE ;Library node
 34 $22 APTR    dl_Root         ;RootNode
 38 $26 APTR    dl_GV           ;BCPL global vector
 42 $2A LONG    dl_A2           ;PRIVATE
 46 $2E LONG    dl_A5           ;PRIVATE
 50 $32 LONG    dl_A6           ;PRIVATE
 54 $36 APTR    dl_Errors       ;array with error messages
 58 $3A APTR    dl_TimeReq      ;PRIVATE: timer request
 62 $3E APTR    dl_UtilityBase  ;PRIVATE: utility library
 66 $42 LABEL   dl_SIZEOF

Dec Hex STRUCTURE RootNode,0
  0  $0 BPTR    rn_TaskArray          ;CLI Process Array [0]=number
  4  $4 BPTR    rn_ConsoleSegment     ;CLI SegList
  8  $8 STRUCT  rn_Time,ds_SIZEOF     ;current time
 20 $14 LONG    rn_RestartSeg         ;D\disk validator SegList
 24 $18 BPTR    rn_Info               ;Info structure
 28 $1C BPTR    rn_FileHandlerSegment ;FileHandler
 32 $20 STRUCT  rn_CliList,MLH_SIZE   ;CLI processes
 44 $2C APTR    rn_BootProc           ;PRIVATE: pr_MsgPort
 48 $30 BPTR    rn_ShellSegment       ;Shell SegList
 52 $34 LONG    rn_Flags              ;Flags
 56 $38 LABEL   rn_SIZEOF

RNB_WILDSTAR = 24, RNF_WILDSTAR = $1000000
```

```
Dec Hex STRUCTURE CliProcList,0
  0  $0 STRUCT   cpl_Node,MLN_SIZE ;for linking
  8  $8 LONG     cpl_First         ;first CLI number
 12  $C APTR     cpl_Array         ;CLI Process Array
 16 $10 LABEL    cpl_SIZEOF

Dec Hex STRUCTURE DosInfo,0
  0  $0 BPTR     di_McName            ;network name of device
  4  $4 BPTR     di_DevInfo           ;list of logical devices
  8  $8 BPTR     di_Devices           ;devices
 12  $C BPTR     di_Handlers          ;Handlers
 16 $10 APTR     di_NetHand           ;current network Handler
 20 $14 STRUCT   di_DevLock,SS_SIZE   ;PRIVATE!!!
 66 $42 STRUCT   di_EntryLock,SS_SIZE ;PRIVATE!!!
112 $70 STRUCT di_DeleteLock,SS_SIZE  ;PRIVATE!!!
158 $9E LABEL    di_SIZEOF
```

Example

The volume of these new functions is overwhelming. It's difficult to update existing programs by replacing the old functions with new ones. Assembler programmers should prepare for some big changes to their programs, because the query of arguments has been simplified and automated. This is a completely different approach to programming. As a result, programming that conforms to the operating system is easier to achieve in Assembler than in higher level languages.

Since the main routines of all CLI commands are now located in the operating system, extremely short programs are possible. As an introduction to OS 2 programming, it is recommended to try a few CLI commands first, and then gradually work up to larger programs. A disadvantage with Assembler used to be the complicated argument queries; this has been eliminated with OS 2. We will use a simple CLI command to help you through the programming procedure. For this exercise we want to emphasize the basic structure and argument queries, so we will construct a command that is executed using a new DOS function: AddBuffers.

We are not referring to the long, slow CLI command (written in C) of the same name. Instead, we are creating a completely new command that has the same function. We will also have to mention some of the dangers of using your own custom routines.

The AddBuffers functions receives a device name and a delta value, which may also be negative. This number represents the number of buffers to be added. The function result will be the current number of available buffers. This command will be able to simply query the number of available buffers or change it by passing a delta value. The first parameter is the device name, and this parameter is required with the function call. If a second parameter is given, it must be a number. This number will be taken as the delta value. We will call our new command 'Buffer'. The following is the program header:

```
************************<Part-1>************************
**====================================================**
**      CLI command structure under OS 2 (v37)        **
**      example of a new AddBuffers command           **
**----------------------------------------------------**
**      Call: Buffer DRIVE/A,BUFFERS/N                **
**      DRIVE   - drive letter                        **
**      BUFFERS - optional, number of buffers to add (+)**
**                or subtract (-)                     **
**----------------------------------------------------**
**      written (w) 1991 by Stefan Maelger            **
**====================================================**

INCLUDE_VERSION                 =   36

RETURN_OKAY                     =    0
RETURN_FAIL                     =   20

ERROR_INVALID_RESIDENT_LIBRARY =  122

ThisTask                        =  276

pr_Result2                      =  148

_LVOOpenLibrary                 = -552
_LVOCloseLibrary                = -414

_LVOIoErr                       = -132
_LVOPrintFault                  = -474
_LVOAddBuffers                  = -732
_LVOReadArgs                    = -798
_LVOFreeArgs                    = -858
_LVOVPrintf                     = -954

**********************<Part-2>**************************
```

Here we have defined the purpose of the program. All of the required system values have been set and the Include files have been linked. Our program should be re-entrable, meaning it can be kept in memory via RESIDENT after setting the PURE flag. In order to do this, we must save all registers from number 2 on up before we use them. The longword at address 4 contains the address of the operating system base structure. This can vary, depending on the operating system and the available memory. This same address is also the base address of the main library EXEC, which can then be used to get the base address of the DOS library.

```
**********************<Part-2>***************************

 SECTION Program,CODE

_Start
 movem.l d2-d6/a6,-(a7)       ;save registers
**
** Open the DOS-Library
**
 moveq.l $4.w,a6              ;load ExecBase
 lea      _DOSName(pc),a1     ;Library name
 moveq    #INCLUDE_VERSION,d0 ;OS 2, v36 and up
 jsr      _LVOOpenLibrary(a6) ;OpenLibrary(a1,d0)
 moveq    #RETURN_FAIL,d4     ;error for DOS
 move.l   d0,d5               ;save DosBase
 beq.s    _NotDOS             ;=> if DosBase=0

**********************<Part-3>***************************
```

All of the functions required for this command are available, starting with version 36 (first version of OS 2). This version number must be specified. The D4 register saves the value returned from CLI, which we immediately set to an error. This is only changed to 'no error' after successful initialization. This saves us a lot of writing. If DOS could not be opened, which should only occur with older OS versions, then we branch to the appropriate error handling routine.

Some of you will have noticed that we made no efforts to save the value returned from CLI (A0=ArgBuf, D0=ArgLen). With OS 2, this is no longer necessary. We can get the arguments with the DOS function ReadArgs, which handles all the work of passing arguments from the user.

```
**********************<Part-3>**************************
**
** Get CLI arguments
**
 exg      d5,a6               ;Exec<->Dos
;
; Store argument field in the stack
;
 clr.l    -(a7)               ;Dummy (size divisible by 16!)
 clr.l    -(a7)               ;Dummy (size divisible by 16!)
 clr.l    -(a7)               ;Arg[2]
 clr.l    -(a7)               ;Arg[1]
;
; Query arguments
;
 lea      _Template(pc),a0    ;argument description
 move.l   a0,d1               ;to d1 for call
 move.l   a7,d2               ;argument field to d2
 moveq    #0,d3               ;no RDArgs structure
 jsr      _LVOReadArgs(a6)    ;ReadArgs(d1,d2,d3)
 move.l   d0,d6               ;save RDArgs structure
 bne.s    _parseArgs          ;if RDArgs<>0 (okay)

**********************<Part-4>**************************
```

ReadArgs expects a string ending with a null byte. This string describes all of the arguments involved. In it, each argument name is given followed by the argument type. The description of each argument is separated by a comma. In our case, this string will contain 'DRIVE/A,BUFFERS/N'. Since we have described two arguments in the string, we need at least two longwords in the argument field to pass them. In order to avoid a system crash, you should always make the field size in bytes divisible by 16. There's no need to get extra memory because there is sufficient space in the stack for four longwords. A value of 0 is passed as the last parameter. An RDArgs structure obtained with ReadArgs would be passed to this location, but is not necessary in our case.

WARNING: The argument field must be filled with null bytes before the call.

The returned RDArgs structure is saved because this must be freed later. We test the result for errors or for user interrupt. If everything is okay, we continue to evaluate the arguments; otherwise an error handling routine is needed. Normally, CLI commands report the cause of an error using a readable message. This is handled by the PrintFault function, which uses the result of IoErr as a parameter.

```
**********************<Part-4>**************************
;; ReadArgs error: set return address
;
 pea     _FreeStack(pc)        ;for following routine
**
** Subroutine
** Get DOS error and output cause as message text
**
_Zerror
 jsr     _LVOIoErr(a6)         ;IoErr()
 move.l  d0,d1                 ;error code to d1
 moveq   #0,d2                 ;no header text
 jmp     _LVOPrintFault(a6)    ;->PrintFault(d1,d2)

**********************<Part-5>**************************
```

In the 'Zerror' routine, we assume that the DosBase is stored in A6 and the D2 register can be changed at any time. Therefore, we don't need to save any of the registers and can jump to the PrintFault routine with a JMP command. This corresponds to a JSR followed by an RTS. This part of the program is structured as a subroutine so that it doesn't have to be repeated for every error. In the case of a ReadArgs error, we jump directly into this routine. Therefore, we must first store a return address on the stack with PEA.

Now we come to the part of the program where the arguments are evaluated.

WARNING: Freeing RDArgs is forbidden at this point, since this could cause the entries of the argument field to point to undefined memory blocks. As long as we are working with the argument field, RDArgs must not be manipulated.

```
**********************<Part-5>**************************
**
** Evaluate arguments
**
_parseArgs
 moveq   #RETURN_OK,d4         ;save return code
;
```

```
; test if two arguments were given
;
 move.l  4(a7),d0             ;get Arg[2] (buffer)
 beq.s   _AvailBuffer         ;if Arg[2]=0
;
; execute 'Buffers xxx yyy' command
;
 movea.l d0,a0                ;Arg[2] is address of value!
 move.l  (a7),d1              ;Arg[1] to d1 (DRIVE)
 move.l  (a0),d2              ;get value from address
 jsr     _LVOAddBuffers(a6)   ;AddBuffers(d1,d2)
 tst.l   d0                   ;test result (error=0)
 bne.s   _AvailBuffer         ;if no error
;
; Error handling for RDArgs structure
;
_OutputError
 bsr.s   _Zerror              ;output message
 bra.s   _RDArgsFree          ;FreeArgs...

**********************<Part-6>**************************
```

Once the initialization is complete, we can be sure that no serious errors have occurred. Therefore, the return value (which was stored in D4) can be set to 'no error'. Next, we check to see if the number of buffers must be changed before we retrieve the number. The first argument is the address of the drive name, which can be placed directly to D1. Since this argument is required (/A), we don't have to check for its presence.

WARNING: To distinguish between a value of 0 and a missing argument, numerical values (/N) require the address of a longword in the argument field rather than the value itself. The longword then contains the actual parameter value.

This address is moved to a data register (D0). If the parameter is not present, the Z flag would have been set. Then the address is moved to an address register (A0) in order to obtain the actual value relative to the address register (D2).

If all of this is successful or if the buffer count was not asked to be changed, then the number of buffers are displayed. Otherwise, an error message is output and we jump to free RDArgs.

```
**********************<Part-6>**************************
;
; Output number of available buffers
;
_AvailBuffer
 move.l  (a7),d1              ;Arg[1] to d1 (DRIVE)
 moveq   #0,d2                ;no change
 jsr     _LVOAddBuffers(a6)   ;AddBuffers(d1,d2)
 move.l  d0,4(a7)             ;Arg[2]=Buffers
 bmi.s   _OutputError         ;if Buffers=-1
 beq.s   _RDArgsFree          ;FreeArgs...
;
; Format and output string
;
 lea     _FormatString(pc),a0 ;format string
 move.l  a0,d1                ;to d1 for call
 move.l  a7,d2                ;field with arguments
 jsr     _LVOVPrintf(a6)      ;VPrintf(d1,d2)

**********************<Part-7>**************************
```

We go to 'AvailBuffer' if no change was made to the buffer count or after the buffer count has been changed. We only need the drive name for AddBuffers, since the change in indicated by 0. The result is stored as the second argument in our argument field. In case of an error, a message is displayed or the program is ended. VPrintf is used to output a string to CLI. The control codes of this string have been replaced by the entries of the field we want to pass. This field is simply our argument field; the second entry of which we have changed to conform to our format string.

Now we still have to restore the system changes that were made when the program was started. The first thing to do is free RDArgs with FreeArgs. Then we restore the stack, which contains our longword field, and close the DOS library.

```
**********************<Part-7>**************************

**
** Free RDArgs structure
**
_RDArgsFree
 move.l  d6,d1                ;saved RDArgs
 jsr     _LVOFreeArgs(a6)     ;FreeArgs(d1)
;
; Restore stack
```

```
;
_FreeStack
 addq.l   #8,a7                 ;restore a7
 addq.l   #8,a7                 ;(all 16 bytes)
**
** Close DOS library
**
 movea.l a6,a1                  ;DosBase to a1
 movea.l d5,a6                  ;load ExecBase
 jsr     _LVOCloseLibrary(a6)   ;CloseLibrary(a1)
 bra.s   _Programend            ;->end program

**********************<Part-8>**************************
```

The error code that describes the error of a program ended with RETURN_FAIL is entered in the process structure for the program. Since every process begins with a task structure, we can access this structure via ExecBase, which always has a pointer to the currently running task. In the following section, which is used in the case of an OpenLibrary error, the error cause is sent to CLI. Then the program is ended. The return value is placed in D0 and the registers are restored. After this are the strings; you no longer have to worry about even or odd addresses since no more code follows.

```
**********************<Part-8>**************************

**
** Error opening DOS library:
** Send error cause to DOS
**
_NotDOS
 moveq   #ERROR_INVALID_RESIDENT_LIBRARY,d0 ;DOS error code
 movea.l ThisTask(a6),a0                    ;Process structure for our program
 move.l  d0,pr_Result2(a0)                  ;enter error cause
**
** End of program
**
_Programend
 move.l  d4,d0                             ;return code for CLI
 movem.l (a7)+,d2-d6/a6                    ;restore registers
 rts                                       ;return
**
** Strings
**
_DOSName      dc.b 'dos.library',0         ;library name
_Template     dc.b 'DRIVE/A,BUFFERS/N',0   ;for ReadArgs
```

```
_FormatString dc.b 'Drive %s has %ld buffers',10,0

***********************<END>***************************
```

When you combine the individual pieces of this program, you will see that things are now much simpler than they once were. Once assembled, a program such as this is less than a half a block long. Each program requires at least a FileHeader block in addition to this. So, you could store up to 439 programs of this type on a normal diskette.

In order to be able to use all mounted devices that may contain files, you first must obtain information about these 'Drives'. All such devices are included as DosEntries in the DosList. Since this list is constantly updated, it used to be necessary to turn off multitasking before searching for a certain entry. Now, you can obtain access privileges with LockDosList in order to prevent an update to the list while you are using it. Let's take a look at how OS 2 retrieves information from this list:

```
**=====================================================**
**      Retrieve info on all FileSystem devices .....   **
**-----------------------------------------------------**
**     Input:  A6 = ExecBase                            **
**             A5 = DosBase                             **
**     Output: D0 = simple linked list of the following **
**                  structures, which can be freed .....**
**                  with exec/FreeVec:                  **
**=====================================================**

 STRUCTURE FileSysDev,0
 APTR      fsd_Next               ;next structure
 STRUCT    fsd_InfoData,id_SIZEOF ;InfoData structure
 STRUCT    fsd_Name,36            ;name buffer
 LABEL     fsd_SIZEOF             ;structure size

*
* Register contents in the routine:
*
* a6,a5 ExecBase and DosBase (these are often confused)
* a4    DosList structure
* a3    InfoData structure
* a2    last FileSysDev structure
* a0,a1 continuously changed
* d3
* d6    arg4 for DosPacket: 0
* d5    arg3 for DosPacket: 0
```

```
* d4    arg2 for DosPacket: 0
* d3    arg1 for DosPacket: BPTR to InfoData structure
* d0-d2 continuously changed
*

_GetFSDevs
 moveq   #0,d0
 movem.l d0/d2-d6/a2-a4,-(a7)
 movea.l a7,a2
 moveq   #0,d4
 moveq   #0,d5
 moveq   #0,d6
;
; InfoData = AllocVec(id_SIZEOF,MEMF_PUBLIC)
;
 moveq   #id_SIZEOF,d0
 moveq   #MEMF_PUBLIC,d1
 jsr     _LVOAllocVec(a6)
 tst.l   d0
 beq.s   .Error
 movea.l d0,a3
 asr.l   #2,d0
 move.l  d0,d3
 exg     a5,a6
;
; dlist = LockDosList(LDF_DEVICES!LDF_READ)
;
 moveq   #LDF_DEVICES!LDF_READ,d1
 jsr     _LVOLockDosList(a6)
 movea.l d0,a4
.Loop
;
; dlist = NextDosEntry(dlist,LDF_DEVICES!LDF_READ)
;
 move.l  a4,d1
 moveq   #LDF_DEVICES!LDF_READ,d2
 jsr     _LVONextDosEntry(a6)
 tst.l   d0
 beq.s   .NoMoreEntries
 movea.l d0,a4
;
; res1 = DoPkt(dol_Task,ACTION_DISK_INFO,InfoData>>2,0,0,0)
;
 move.l  dol_Task(a4),d1
 beq.s   .Loop
 moveq   #ACTION_DISK_INFO,d2
 jsr     _LVODoPkt(a6)
 tst.l   d0
```

```
 beq.s    .Loop
;
; FileSysDev = AllocVec(fsd_SIZEOF,MEMF_CLEAR!MEMF_PUBLIC)
;
 moveq    #fsd_SIZEOF,d0
 move.l   #MEMF_CLEAR!MEMF_PUBLIC,d1
 exg      a5,a6
 jsr      _LVOAllocVec(a6)
 exg      a5,a6
 move.l   d0,(a2)
 beq.s    .NoMoreEntries
 movea.l d0,a2
 lea      fsd_InfoData(a2),a1
 movea.l a3,a0
 moveq    #8,d0
.CopyID
 move.l   (a0)+,(a1)+
 dbra     d0,.CopyID
 movea.l dol_Name(a4),a0
 adda.l   a0,a0
 adda.l   a0,a0
 move.b   (a0)+,d0
 moveq    #34,d1
.CopyBStr
 move.b   (a0)+,(a1)+
 subq.b   #1,d0
 dble     d1,.CopyBStr
 move.b   #':',(a1)
 bra.s    .Loop
.NoMoreEntries
;
; UnLockDosList(LDF_DEVICES!LDF_READ)
;
 moveq    #LDF_DEVICES!LDF_READ,d1
 jsr      _LVOUnLockDosList(a6)
 exg      a5,a6
;
; FreeVec(InfoData)
;
 movea.l a3,a1
 jsr      _LVOFreeVec(a6)
.Error
 move.l   (a7)+,d0
 movem.l (a7)+,d2-d6/a2-a4
 rts
```

Notice that the drive names are used here without the colon.

About the program flow:

1. Get memory for an InfoData structure.

   The memory block may not be moved, it must be allocated as PUBLIC. The length and contents do not matter. The size must correspond to that of an InfoData structure. We use the new Exec function AllocVec() here, which stores the amount of memory. If a value of 0 is returned, the memory could not be allocated and we jump to step 6.

   Note: The error cause can be output with PrintFault(IoErr(),0).

2. Obtain access to DosList (if necessary, include LDF_VOLUMES and/or LDF_ASSIGNS).

   WARNING 1: Don't forget LDF_READ.

   WARNING 2: The UnLockDosList function must be called with the same value.

   WARNING 3: With a reserved DosList, do not call functions that must change the DosList.

   WARNING 4: The returned value is not a DosList structure that can be processed.

3. Loop

3a. Get next DosList structure of the desired type. To do this, either the last DosList structure or the value returned from LockDosList is passed as the DosList structure. If a value of 0 is returned, then no more entries of the requested type are available and we jump to step 4.

3b. The dol__Task entry contains the address of the MsgPort of the FileHandler process in question (pr_MsgPort). If a value of 0 is found, then this is not a data storage device and we jump to step 3.

3c. We can get the desired information from the FileHandler. In order to do this, we must first create a StandardPacket structure, load it with the proper information, send it to the MsgPort of the FileHandler, and wait for an answer. The new DoPkt functions handle this for a simple StandardPacket. dol_Task, which is the desired action (ACTION_DISK_INFO), and a BPTR (address/4) to our InfoData structure, which is the only packet parameter, are passed to the DoPkt function. If the handler does not understand our command, then we are not dealing with a data storage device, so we jump to step 3.

3d. We use AllocVec() to reserve enough memory to hold the drive name, the complete InfoData structure, and pointers for linking the memory blocks. If this is unsuccessful, we jump to step 4.

3e. This memory block is linked to the last memory block allocated in this way. We copy the drive name and the InfoData structure. Since DOS does not use colons with drive names, we add it to complete the string. Then we jump back to the start of the loop (step 3).

4. The DosList is set free.

   WARNING: You must give the same value used with LockDosList.

5. The InfoData structure is set free. The FreeVec() function requires only the start address of its memory block.

6. End the program and return the list of linked memory blocks, that must be set free, with FreeVec():

```
.loop
   movea.l d0,a1           ;first structure
   movea.l (a1),a2
   jsr     _LVOFreeVec(a6)
   move.l  a2,d0
   bne.s   .loop
```

3.1.5 The Exec Library

Exec is the base library of the operating system. It manages all other libraries, devices, resources, interrupts, programs, and the system memory. Exec is often called 'Sys', so you may find ExecBase and SysBase used interchangeably. The routines for library management are also integrated into Exec. The base address of the Exec library is stored in the longword at $4. This address must be loaded to the A6 register for every function call.

Exec Library Functions

1. System Module

ColdReboot
FindResident
InitCode
InitResident
InitStruct
MakeFunctions
MakeLibrary
SumKickData

2. Interrupts

AddIntServer
Cause
Disable
Enable
Forbid
GetCC
Permit
RemIntServer
SetIntVector
SetSR
SuperState
Supervisor
UserState

3. Memory Management

AddMemList
AllocAbs
Allocate
AllocEntry
AllocMem
AllocVec
AvailMem
CopyMem
CopyMemQuick
Deallocate
FreeEntry
FreeMem
FreeVec
TypeOfMem

4. Structure Management

AddHead
AddTail
Enqueue
FindName
Insert
RemHead
Remove
RemTail

5. Programs

AddTask
AllocSignal
AllocTrap
CacheClearE
CacheClearU
CacheControl
FindTask
FreeSignal
FreeTrap
RemTask
SetExcept
SetSignal
SetTaskPri
Signal
Wait

6. Communications

AddPort
Alert
CreateMsgPort
Debug
DeleteMsgPort
FindPort
GetMsg
PutMsg
RawDoFmt
RemPort
ReplyMsg
WaitPort

7. Libraries

AddLibrary
CloseLibrary
OldOpenLibrary
OpenLibrary
RemLibrary
SetFunction
SumLibrary

8. Devices

AbortIO
AddDevice
CheckIO
CloseDevice
CreateIORequest
DeleteIORequest
DoIO
OpenDevice
RemDevice
SendIO
WaitIO

9. Resources

AddResource
OpenResource
RemResource

10. Semaphores

AddSemaphore
AttemptSemaphore
FindSemaphore
InitSemaphore
ObtainSemaphore
ObtainSemaphoreList
ObtainSemaphoreShared
Procure
ReleaseSemaphore
ReleaseSemaphoreList
RemSemaphore
Vacate

Description of Functions

1. System Module

ColdReboot	**Cold system reboot**

Call:

```
ColdReboot()
-726(A6)
```

Function: Resets the Amiga and all connected devices. This function corresponds to pressing the (Ctrl -Amiga-Amiga) keys simultaneously.

FindResident	**Find a system module**

Call:

```
resident = FindResident(name)
D0         -96(A6)      A1

STRUCT Resident *resident
APTR   name
```

Function: Looks for a resident tag for the given name.

Parameters: name — Address of the name (RT_NAME).

Result: Address of the resident structure or 0.

InitCode	**Initialize resident code module**

Call:

```
InitCode(startClass, version)
-72(A6)  D0          D1
```

Function: Initializes all resident modules of the given type. RTF_AFTERDOS modules should have a priority of less than -100. Modules without a startClass value should have a priority of -120.

Parameters: startClass — Flags for module type: RTF_COLDSTART, RTF_SINGLETASK or RTF_AFTERDOS.

version — Version number

InitResident — **Initialize resident module**

Call:

```
InitResident(resident, segList)
-102(A6)     A1        D1

STRUCT Resident *resident
ULONG  segList
```

Function: Initializes a ROMTag. Normally jumps to the function stored in RT_INIT (A6=ExecBase, A0=segList, D0=0). However, if RTF_AUTOINIT is set, then RT_INIT points to four consecutive longwords for calling MakeLibrary(). These longwords contain the size of the base structure, table of library functions, table for InitStruct(), and the RT_INIT function).

InitStruct — **Initialize a data structure**

Call:

```
InitStruct(initTable, memory, size)
-78(A6)    A1         A2      D0

STRUCT InitStruct *initTable
APTR   memory
ULONG  size
```

Function: Deletes a data structure and initializes it with the values in the given table. The table can be created with the MACROs from "exec/initializers.i".

Parameters:

initTable	Table containing structure data (must end in 0).
memory	Address (even) of the data structure.
size	Structure size (even count of bytes) (0=do not delete first)

MakeFunctions — **Generate library vector table**

Call:

```
tableSize = MakeFunctions(target, functionArray, funcDispBase)
D0          -90(A6)       A0      A1             A2

ULONG tableSize
APTR target,functionArray,funcDispBase
```

Function: Function for MakeLibrary(). Used to create a vector table (negative base offsets).

Parameters: target — Base address of library/device.

functionArray — Table with function addresses (ending in -1) or a table beginning with the Word -1 containing relative 16 bit offsets (ending in -1).

funcDispBase — Address to be added to the relative 16 bit values, or 0.

Result: tableSize — Vector table size (for LIB_NEGSIZE)

MakeLibrary — Create a library

Call:

```
library = MakeLibrary(vectors, structure, init, dSize, segList)
D0         -84(A6)      A0        A1         A2     D0      D1

STRUCT Library *library
APTR   vectors,init
STRUCT InitStruct *structure
ULONG  dSize
BPTR   segList
```

Function: Initializes a library structure.

Parameters: vectors — Function addresses for MakeFunctions().

structure — Data for InitStruct() or 0.

init — Library RT_INIT routine or 0.

dSize — Size of base structure.

segList — Segment list (see dos/LoadSeg())

Result: Library base address or 0.

SumKickData **Calculate check sum across resident modules**

Call:
```
checksum = SumKickData()
D0         -612(A6)

ULONG checksum
```

Function: Builds a check sum across the linked list of resident modules (KickTagPtr) and MemEntry structures (KickMemPtr). The check sum is stored in KickCheckSum, as long as "reset-proof" changes to the system will allow it.

Result: checksum Value for ExecBase->KickCheckSum

```
Dec Hex STRUCTURE RT,0      ;residentTag/ROMTag
  0  $0 UWORD RT_MATCHWORD ;ILLEGAL command
  2  $2 APTR  RT_MATCHTAG  ;start of structure (RT_MATCHWORD)
  6  $6 APTR  RT_ENDSKIP   ;RT allowed starting with this address
 10  $A UBYTE RT_FLAGS     ;Flags
 11  $B UBYTE RT_VERSION   ;version
 12  $C UBYTE RT_TYPE      ;module type (NT_...)
 13  $D BYTE  RT_PRI       ;initialization priority
 14  $E APTR  RT_NAME      ;module name
 18 $12 APTR  RT_IDSTRING  ;identification string
 22 $16 APTR  RT_INIT      ;initialization routine/data
 26 $1A LABEL RT_SIZE

RTC_MATCHWORD = $4AFC

RTB_COLDSTART   = 0, RTF_COLDSTART  =   1 ;Init from reset
RTB_SINGLETASK = 1, RTF_SINGLETASK =   2 ;task
RTB_AFTERDOS    = 2, RTF_AFTERDOS   =   4 ;Init after DOS
RTB_AUTOINIT    = 7, RTF_AUTOINIT   = $80 ;RT_INIT = data

RTW_NEVER       = 0 ;do not initialize
```

2. Interrupts

AddIntServer **Insert an interrupt in a server list**

Call:
```
AddIntServer(intNum, interrupt)
-168(A6)     D0      A1

ULONG  intNum
STRUCT IS *interrupt
```

Function: Links an IS structure in a server list of an interrupt server. The given interrupt number is the number of an Amiga interrupt source, not that of a processor interrupt. The interrupt routines must end with RTS and must set the processor's Z flag if other interrupt routines are to be processed. The function is called with IS_DATA in A1.

Parameters:

intNum	Interrupt source with a server (PORTS, COPER, VERTB, EXTER or NMI).
interrupt	IS structure

Warning: Not suitable for high-level languages. For VERTB, the value $DFF000 must remain in the A0 register if the interrupt has a priority of 10 or greater.

See also: RemIntServer(), SetIntVector(), hardware/intbits.i

Example: Linking an interrupt that is executed with every vertical blank of the monitor.

```
 ...
_Interrupt_link
 movea.l $4.w,a6
 lea     _VertBIS(pc),a1
 moveq   #INTB_VERTB,d0
 jsr     _LVOAddIntServer(a6)
 ...

 ...
_Interrupt_remove
 movea.l $4.w,a6
 lea     _VertBIS(pc),a1
 moveq   #INTB_VERTB,d0
 jsr     _LVORemIntServer(a6)
 ...

_VertBIS
 dc.l    0,0                ;LN_SUCC,LN_PRED
 dc.b    NT_INTERRUPT,127   ;LN_TYPE,LN_PRI
 dc.l    _VertBName         ;LN_NAME
 dc.l    _Data,_Interrupt   ;IS_DATA,IS_CODE
```

```
**====================================================**
**        Interrupt Routine                           **
**----------------------------------------------------**
**        Input: a0 = _Custom ($dff000)               **
**               a1 = _Data   (IS_DATA)               **
**        Output:d0,cc = 0,Z                          **
**====================================================**

_Interrupt
 movem.l d2-d6/a0-a6,-(a7)
 ...
 movem.l (a7)+,d2-d6/a0-a6
 moveq   #0,d0
 rts

_Data       ;data block for the interrupt routine
```

Cause — **Calls a software interrupt**

Call:

```
Cause(interrupt)
-180(A6) A1

STRUCT IS *interrupt
```

Function: Executes a software interrupt.

Parameters: interrupt IS structure of the interrupt.

Disable — **Turn off interrupts**

Call:

```
Disable()
-120(A6)
```

Function: Turns off all interrupts along with multitasking. This can be a nested call.

Warning: Essential operating system functions can be destroyed by turning the interrupts off for more than 0.00025 seconds. It's best to let these calls go through other operating system functions.

Wait() calls within a Disable()/Enable() call turn multitasking back on until signaled.

Enable — **Allow interrupts**

Call:
```
Enable()
-126(A6)
```

Function: Reverses the effect of Disable(). Interrupt processing is restored as long as the number of Enable() calls correspond to the number of preceding Disable() calls.

Forbid — **Turn off multitasking**

Call:
```
Forbid()
-132(A6)
```

Function: Turns off multitasking capabilities. Forbid() calls can be nested.

Warning: Wait() calls within a Forbid()/Permit() call turn multitasking back on until signaled.

See also: Permit()

GetCC — **Retrieve CCR in CPU-compatible format**

Call:
```
conditions = GetCC()
D0           -528(A6)

UWORD conditions
```

Function: "MOVE SR,<ea>" is only allowed on the 68000 in user mode. This function replaces that command so that all processors can read the status register.

Result: The 680x0 ConditionCodes

Permit — **Turn multitasking back on**

Call:
```
Permit()
-138(A6)
```

Function: Allows multitasking again. Multitasking is restored as long as the number of Permit() calls correspond to the number of preceding Forbid() calls.

See also: Forbid()

RemIntServer — **Remove IS from a server list**

Call:

```
RemIntServer(intNum, interrupt)
-174(A6)     D0      A1

ULONG  intNum
STRUCT IS *interrupt
```

Function: Opposite of AddIntServer().

Parameters:

intNum — Interrupt source, as with AddIntServer().

interrupt — IS structure, as with AddIntServer().

SetIntVector — **Set interrupt handler**

Call:

```
oldInterrupt = SetIntVector(intNumber, interrupt)
D0             -162(A6)     D0         A1

STRUCT IS *oldInterrupt,*interrupt
ULONG  intNumber
```

Function: Assigns a handler to an interrupt source. The previous handler for this source is removed and returned to its IS structure. The routine, which must end with RTS, contains an AND combination of intenar and intreqr in D1, the address of the custom chip in A0, and IS_DATA in A1.

Parameters:

intNum — Interrupt source with no server.

interrupt — IS structure of the handler.

Result: IS structure of the previous handler.

See also: AddIntServer(),exec/interrupts.i,exec/hardware.i

SetSR	**Read and change status register**

Call:

```
oldSR = SetSR(newSR, mask)
D0      -144(A6) D0   D1

ULONG oldSR,newSR,mask
```

Function: Reads the SR according to the installed processor and sets the bits in a given bit mask according to the passed values.

Parameters:

newSR — Condition to which the bits will be changed.

mask — Bit mask containing the bits to be changed.

Result: The complete status register prior to the change.

```
**=====================================================**
**      Read status register                            **
**=====================================================**

 movea.l $4.w,a6
 moveq   #0,d0
 moveq   #0,d1
 jsr     _LVOSetSR(a6)
 move.w  d0,...

**=====================================================**
**      Set interrupt level 4                           **
**=====================================================**

 movea.l $4.w,a6
 move.w  #$400,d0
 move.w  #$700,d1
 jsr     _LVOSetSR(a6)
 move.w  d0,...
```

SuperState	**Change processor to supervisor mode**

Call:

```
oldSysStack = SuperState()
D0            -150(A6)

APTR oldSysStack
```

Function: Switches the processor to supervisor mode. Keeps the user stack, which contains all interrupt data.

Result: Address of the system stack or 0 (called from supervisor mode).

See also: UserState(), Supervisor()

Supervisor — Execute routine in supervisor mode

Call:
```
result = Supervisor(userFunc)
Rx       -30(A6)    A5
```

Function: Executes an Assembler routine ending with RTE in supervisor mode. The registers are not changed.

Parameters: userFunc Address of the Assembler routine (RTE).

Result: All register changes during the execution of the routine (up to 15 changes).

See also: SuperState(), UserState()

Example: Get the location of the exception vector table for higher processors:

```
 movea.l $4.w,a6
 moveq   #AFF_68010!AFF_68020!AFF_68030!AFF_68040,d7
 and.w   AttnFlags(a6),d7
 beq.s   _TableFound
 lea     _Exception(pc),a5
 jsr     _LVOSupervisor(a6)
_TableFound
 ...

_Exception
 movec.l vbr,d7 ;VBR nach d7
 rte
```

UserState — Return processor to user mode

Call:
```
UserState(sysStack)
-156(A6)  D0
```

```
APTR sysStack
```

Function: Switches the processor back to user mode.

Parameters: sysStack Supervisor stack from SuperState().

See also: SuperState(), Supervisor()

```
Dec  Hex STRUCTURE  ExecBase,LIB_SIZE ;Exec base structure
 34  $22 UWORD   SoftVer             ;Kickstart version
 36  $24 WORD    LowMemChkSum        ;trap vector check sum
 38  $26 ULONG   ChkBase             ;inverted base address
 42  $2A APTR    ColdCapture         ;cold boot vector
 46  $2E APTR    CoolCapture         ;reset vector
 50  $32 APTR    WarmCapture         ;warm boot vector
 54  $36 APTR    SysStkUpper         ;system stack upper limit
 58  $3A APTR    SysStkLower         ;system stack lower limit
 62  $3E ULONG   MaxLocMem           ;size of chip memory
 66  $42 APTR    DebugEntry          ;global debugger
 70  $46 APTR    DebugData           ;debugger data
 74  $4A APTR    AlertData           ;alarm data
 78  $4E APTR    MaxExtMem           ;FastRAM
 82  $52 WORD    ChkSum              ;check sum up to this point
 84  $54 LABEL   IntVects            ;interrupt vectors
 84  $54 STRUCT IVTBE,IV_SIZE        ;serial output
 96  $60 STRUCT IVDSKBLK,IV_SIZE     ;DiskDMA finished
108  $6C STRUCT IVSOFTINT,IV_SIZE    ;software interrupt
120  $78 STRUCT IVPORTS,IV_SIZE      ;CIA interrupts
132  $84 STRUCT IVCOPER,IV_SIZE      ;copper interrupt
144  $90 STRUCT IVVERTB,IV_SIZE      ;vertical blank
156  $9C STRUCT IVBLIT,IV_SIZE       ;blitter finished
168  $A8 STRUCT IVAUD0,IV_SIZE       ;start of sound channel 0
180  $B4 STRUCT IVAUD1,IV_SIZE       ;start of sound channel 1
192  $C0 STRUCT IVAUD2,IV_SIZE       ;start of sound channel 2
204  $CC STRUCT IVAUD3,IV_SIZE       ;start of sound channel 3
216  $D8 STRUCT IVRBF,IV_SIZE        ;serial input
228  $E4 STRUCT IVDSKSYNC,IV_SIZE    ;DiskDMA synchronized
240  $F0 STRUCT IVEXTER,IV_SIZE      ;external interrupt
252  $FC STRUCT IVINTEN,IV_SIZE      ;level 6 interrupt
264 $108 STRUCT IVNMI,IV_SIZE        ;level 7 interrupt
276 $114 APTR    ThisTask            ;currently running program
280 $118 ULONG   IdleCount           ;wait counter
284 $11C ULONG   DispCount           ;dispatch counter
288 $120 UWORD   Quantum             ;time period
290 $122 UWORD   Elapsed             ;elapsed time
```

```
292 $124 UWORD  SysFlags                    ;internal Flags
294 $126 BYTE   IDNestCnt                   ;interrupt forbid counter
295 $127 BYTE   TDNestCnt                   ;multitask forbid counter
296 $128 UWORD  AttnFlags                   ;special system Flags
298 $12A UWORD  AttnResched                 ;execution Flags
300 $12C APTR   ResModules                  ;ROMTags
304 $130 APTR   TaskTrapCode                ;standard trap handler
308 $134 APTR   TaskExceptCode              ;standard exception handler
312 $138 APTR   TaskExitCode                ;standard return address
316 $13C ULONG  TaskSigAlloc                ;system signal mask
320 $140 UWORD  TaskTrapAlloc               ;system trap task
322 $142 STRUCT MemList,LH_SIZE             ;free memory PRIVATE!
336 $150 STRUCT ResourceList,LH_SIZE        ;Resources  PRIVATE!
350 $15E STRUCT DeviceList,LH_SIZE          ;Devices    PRIVATE!
364 $16C STRUCT IntrList,LH_SIZE            ;Interrupts PRIVATE!
378 $17A STRUCT LibList,LH_SIZE             ;Libraries  PRIVATE!
392 $188 STRUCT PortList,LH_SIZE            ;MsgPorts   PRIVATE!
406 $196 STRUCT TaskReady,LH_SIZE           ;programs  PRIVATE!
420 $1A4 STRUCT TaskWait,LH_SIZE            ;waiting tasks PRIVATE!
434 $1B2 STRUCT SoftInts,SH_SIZE*5          ;SoftwareInts PRIVATE!
514 $202 STRUCT LastAlert,4*4               ;last system error
530 $212 UBYTE  VBlankFrequency             ;vertical blank frequency
531 $213 UBYTE  PowerSupplyFrequency        ;power supply frequency
532 $214 STRUCT SemaphoreList,LH_SIZE       ;signal Semaphores
546 $222 APTR   KickMemPtr                  ;reset-protected memory blocks
550 $226 APTR   KickTagPtr                  ;reset-protected user module
554 $22A APTR   KickCheckSum                ;check sum across Mem and Tags
558 $22E UWORD  ex_Pad0
560 $230 ULONG  ex_Reserved0
564 $234 APTR   ex_RamLibPrivate            ;RAM library PRIVATE!
568 $238 ULONG  ex_EClockFrequency          ;CPU E pin frequency
572 $23C ULONG  ex_CacheControl             ;CACR
576 $240 ULONG  ex_TaskID                   ;next possible Task
580 $244 ULONG  ex_PuddleSize
584 $248 ULONG  ex_PoolThreshold
588 $24C STRUCT ex_PublicPool,MLN_SIZE
596 $254 APTR   ex_MMULock
600 $258 STRUCT ex_Reserved,12
612 $264 LABEL  SYSBASESIZE

AFB_68010 = 0, AFF_68010 =   1 ;also with 68020
AFB_68020 = 1, AFF_68020 =   2 ;also with 68030
AFB_68030 = 2, AFF_68030 =   4 ;also with 68040
AFB_68040 = 3, AFF_68040 =   8 ;CPU 68040
AFB_68881 = 4, AFF_68881 =  16 ;also with 68882
AFB_68882 = 5, AFF_68882 =  32 ;FPU 68882
```

```
CACRB_EnableI =  0, CACRF_EnableI =   1          ;command cache
CACRB_FreezeI =  1, CACRF_FreezeI =   2          ;freeze command cache
CACRB_ClearI  =  3, CACRF_ClearI  =   8          ;clear command cache
CACRB_IBE     =  4, CACRF_IBE     =  16          ;burst mode commands
CACRB_EnableD =  8, CACRF_EnableD = 256          ;data cache
CACRB_FreezeD =  9, CACRF_FreezeD = 512          ;freeze data cache
CACRB_ClearD  = 11, CACRF_ClearD  =2048          ;clear data cache
CACRB_DBE     = 12, CACRF_DBE     =4096          ;data burst
CACRB_WriteAllocate = 13, CACRF_WriteAllocate = 8192 ;always

Dec Hex STRUCTURE  IS,LN_SIZE ;Interrupt Structure
 14  $E APTR     IS_DATA      ;data for IS_CODE
 18 $12 APTR     IS_CODE      ;interrupt routine
 22 $16 LABEL    IS_SIZE

Dec Hex STRUCTURE  IV,0 ;Execs Interrupt Vectors
  0  $0 APTR     IV_DATA ;data for IS_CODE
  4  $4 APTR     IV_CODE ;interrupt Handler/Server
  8  $8 APTR     IV_NODE ;IS structure/0
 12  $C LABEL    IV_SIZE

SB_SAR  = 15, SF_SAR  = $8000 ;execution plan
SB_TQE  = 14, SF_TQE  = $4000 ;time exceeded
SB_SINT = 13, SF_SINT = $2000 ;SoftInt

Dec Hex STRUCTURE  SH,LH_SIZE ;SoftInt Header
 14  $E UWORD    SH_PAD
 16 $10 LABEL    SH_SIZE

SIH_PRIMASK = $F0 ;priority mask
SIH_QUEUES  = 5   ;5 SoftInt queues
```

3. Memory Management

AddMemList	**Add memory to the free memory list**

Call:

```
AddMemList( size, attributes, pri, base, name )
-618(A6)    D0    D1          D2   A0    A1

ULONG size,attributes
LONG  pri
APTR  base,name
```

Function: Adds a memory block to the list of free memory. A MemHeader structure is created at the beginning of the block.

Parameters: size — Size of memory block.

attributes — Memory type

pri — Allocation priority

base — Address

name — Name for memory or 0

AllocAbs — **Attempt to allocate a certain memory block**

Call:

```
memoryBlock = AllocAbs(byteSize, location)
D0            -204(A6) D0        A1

APTR  memoryBlock,location
ULONG byteSize
```

Function: Allocates a memory block at a set address. Normally, this routine is only used by reset-protected programs to protect themselves from being overwritten.

Parameters: byteSize — Size

location — Address

Result: Address of the memory block (divisible by 8) or 0.

Allocate — **Allocate private memory block**

Call:

```
memoryBlock=Allocate(memHeader, byteSize)
D0          -186(a6) A0         D0

APTR   memoryBlock
STRUCT MemHeader *memHeader
ULONG  byteSize
```

Function: Assign a private MemHeader to a memory block.

Parameters: memHeader

Private MemHeader

byteSize   Size of the desired block.

Result: Address of the reserved memory block or 0.

See also: Deallocate, exec/memory.h

AllocEntry — **Allocate several memory blocks**

Call:

```
memList = AllocEntry(memList)
D0        -222(A6)  A0

STRUCT MemList *memList
```

Function: Allocates all of the blocks stored in a MemList structure.

Parameters: memList   Structure containing MemEntry structures.

Result: New MemList structure with the results (not identical to the structure passed as a parameter). If a block could not be allocated, then the memory type with a matching bit 31 is passed back (negative value).

AllocMem — **Allocate memory**

Call:

```
memoryBlock = AllocMem(byteSize, attributes)
D0            -198(A6) D0        D1

APTR  memoryBlock
ULONG byteSize,attributes
```

Function: Allocates the requested type and amount of memory.

Parameters: byteSize   Size of block

attributes   Memory type (MEMF_...)

Result: Address of the memory block or 0.

Warning: Memory that cannot be freed must be MEMF_PUBLIC.

See also: FreeMem()

AllocVec — **Allocate memory and store the size**

Call:

```
memoryBlock = AllocVec(byteSize, attributes)
D0            -684(A6) D0        D1
```

Functions, Parameters, Results:

Same as AllocMem(), except that Exec stores the block size for FreeVec().

See also: FreeVec(), AllocMem()

AvailMem — **Query free memory**

Call:

```
size = AvailMem(attributes)
D0     -216(A6) D1

ULONG size,attributes
```

Function: Query the amount of free system memory.

Parameters: requirements
Memory type (MEMF_...)

Result: Number of free bytes of the desired type. This number may not be correct because of multitasking.

CopyMem — **Copy a memory block**

Call:

```
CopyMem( source, dest, size )
-624(A6) A0      A1    D0

APTR  source,dest
ULONG size
```

Function: Super-fast copying of a memory block.

Parameters:

source	Source address
dest	Destination address
size	Block size (0 allowed)

See also: CopyMemQuick()

CopyMemQuick — Optimized memory copy

Call:

```
CopyMemQuick( source, dest, size )
-630(A6)      A0      A1    D0
```

Function: Highly optimized memory copying function.

Parameters:

source — Source address (divisible by 4)

dest — Destination address (divisible by 4)

size — Block size (divisible by 4, 0 allowed)

See also: CopyMem()

Deallocate — Free memory block allocated with Allocate()

Call:

```
Deallocate(memHeader, memoryBlock, byteSize)
-192(A6)   A0         A1           D0

STRUCT MemHeader *memHeader
APTR   memoryBlock
ULONG  byteSize
```

Function: Frees a memory block that was allocated with the Allocate() command.

Parameters:

memHeader
Own MemHeader

memoryBlock
Address of memory block

byteSize — Block size, 0 allowed

See also: Allocate(), exec/memory.h

FreeEntry — **Free several memory blocks**

Call:

```
FreeEntry(memList)
-228(A6)  A0

STRUCT MemList *memList
```

Function: Frees all memory blocks in a MemList structure (result of AllocEntry).

Parameters: memList MemList structure

See also: AllocEntry()

FreeMem — **Free memory block**

Call:

```
FreeMem(memoryBlock, byteSize)
-210(A6) A1          D0

APTR  memoryBlock
ULONG byteSize
```

Function: Frees a memory block.

Parameters: memoryBlock
Block address

byteSize Block size

See also: AllocMem(),AllocAbs()

FreeVec — **Free memory allocated with AllocVec()**

Call:

```
FreeVec(memoryBlock)
-690(a6) A1

APTR memoryBlock
```

Function: Frees a memory block allocated with AllocVec().

Parameters: memoryBlock
Result of AllocVec() or 0

See also: AllocVec()

TypeOfMem **Get memory type**

Call:

```
attributes = TypeOfMem(address)
D0           -534(A6)   A1

ULONG attributes
APTR  address
```

Function: Queries the memory type of the memory block at the given address (MEMF_...).

Parameters: address Memory address

Result: Memory type or 0 (ROM, not linked, or does not exist).

```
Dec Hex STRUCTURE ML,LN_SIZE  ;MemList
 14  $E UWORD   ML_NUMENTRIES ;number of ME structures to follow
 16 $10 LABEL   ML_ME         ;start of the ME structures
 16 $10 LABEL   ML_SIZE =     ;size excluding ME structures

Dec Hex STRUCTURE ME,0      ;MemEntry
  0  $0 LABEL   ME_REQS =   ;memory type for AllocMem()
  0  $0 APTR    ME_ADDR     ;memory address follows
  4  $4 ULONG   ME_LENGTH   ;block size
  8  $8 LABEL   ME_SIZE

MEMF_ANY = 0 ;any memory type (do not use!)
MEMB_PUBLIC    =  0, MEMF_PUBLIC   =      1 ;usable memory
MEMB_CHIP      =  1, MEMF_CHIP     =      2 ;ChipRAM
MEMB_FAST      =  2, MEMF_FAST     =      4 ;FastRAM
MEMB_LOCAL     =  8, MEMF_LOCAL    =   $100 ;UserRAM
MEMB_24BITDMA  =  9, MEMF_24BITDMA =   $200 ;DMA capable, 24 bit
MEMB_CLEAR     = 16, MEMF_CLEAR    = $10000 ;delete beforehand
MEMB_LARGEST   = 17, MEMF_LARGEST  = $20000 ;largest block
MEMB_REVERSE   = 18, MEMF_REVERSE  = $40000 ;inverted
MEMB_TOTAL     = 19, MEMF_TOTAL    = $80000 ;total size

MEM_BLOCKSIZE = 8 ;smallest available memory block

Dec Hex STRUCTURE  MH,LN_SIZE ;start of memory
 14  $E UWORD   MH_ATTRIBUTES ;memory type
 16 $10 APTR    MH_FIRST      ;first free block
 20 $14 APTR    MH_LOWER      ;start of block
```

```
 24 $18 APTR     MH_UPPER       ;end of block
 28 $1C ULONG    MH_FREE        ;free bytes
 32 $20 LABEL    MH_SIZE

Dec Hex STRUCTURE  MC,0 ;start of a free block
  0  $0 APTR     MC_NEXT ;next free block
  4  $4 ULONG    MC_BYTES ;block size
  8  $8 LABEL    MC_SIZE
```

4. Structure Management

AddHead	**Insert a node at the start of a list**

Call:

```
AddHead(list, node)
-240(A6) A0    A1

STRUCT LH *list
STRUCT LN *node
```

Function: Inserts a node at the start of a double linked list.

Parameters:

list	LH structure of the double linked list.
node	LN structure of the list entry.

AddTail	**Insert node at the end of a list**

Call:

```
AddTail(list, node)
-246(A6) A0    A1

STRUCT LH *list
STRUCT LN *node
```

Function: Like AddHead(), but the node is added to the end of the double linked list.

Parameters:

list	LH structure of the double linked list.
node	LN structure of the list entry.

Enqueue	**Adds a node to a list**

Call:

```
Enqueue(list, node)
-270(A6) A0   A1

STRUCT LH *list
STRUCT LN *node
```

Function: Adds an LN structure to a double linked list using the given priority (LN_PRI).

Parameters: list — LH structure of the double linked list.

node — LN structure of the list entry.

FindName	**Find a node in a list**

Call:

```
node = FindName(start, name)
D0,CC  -276(A6) A0     A1

STRUCT LN *node
STRUCT LH *start
APTR   name
```

Function: Finds a node with the given name (LN_NAME) in a double linked list. In order to find multiple nodes with the same name, the next call must use the node structure returned from the previous call instead of the ListHeader structure.

Parameters: start — ListHeader or ListNode

name — String ending in 0, containing node name.

Result: Node address or 0.

Insert	**Insert a node into a list after another node**

Call:

```
Insert(list, node, listNode)
-234(A6) A0   A1    A2

STRUCT LH *list
STRUCT LN *node,*listNode
```

Function: Inserts a node after another node in a double linked list.

Parameters: list — ListHeader (if listNode=0)

node — ListNode to be inserted.

listNode — Node after which the new node will be inserted or 0.

RemHead	Remove the first node in a list

Call:

```
node = RemHead(list)
D0     -258(A6) A0

STRUCT LN *node
STRUCT LH *list
```

Function: Gets the address of the first node in a double linked list and removes the node from the list.

Parameters: list — ListHeader structure

Result: Address of the ListNode or 0 (list was empty).

Remove	Remove a node from a list

Call:

```
Remove(node)
-252(A6) A1

STRUCT LN *node
```

Function: Takes the given ListNode out of the list.

Parameters: node — ListNode to be removed.

RemTail	Remove last node from a list

Call:

```
node = RemTail(list)
D0     -264(A6) A0

STRUCT LN *node
STRUCT LH *list
```

Function: Gets the address of the last node in a double linked list and removes the node from the list.

Parameters: list ListHeader structure

Result: Address of the ListNode or 0 (list was empty).

```
Dec Hex STRUCTURE LH,0    ;list, ListHeader
  0  $0 APTR  LH_HEAD     ;first node
  4  $4 APTR  LH_TAIL     ;0 (end marker)
  8  $8 APTR  LH_TAILPRED ;last node
 12  $C UBYTE LH_TYPE     ;list type
 13  $D UBYTE LH_pad
 14  $E LABEL LH_SIZE

Dec Hex STRUCTURE MLH,0    ;same structure, minimal configuration
  0  $0 APTR  MLH_HEAD     ;first node
  4  $4 APTR  MLH_TAIL     ;0
  8  $8 APTR  MLH_TAILPRED ;last node
 12  $C LABEL MLH_SIZE

Dec Hex STRUCTURE LN,0 ;ListNode
  0  $0 APTR  LN_SUCC  ;next node
  4  $4 APTR  LN_PRED  ;previous node
  8  $8 UBYTE LN_TYPE  ;node type
  9  $9 BYTE  LN_PRI   ;node priority
 10  $A APTR  LN_NAME  ;node name
 14  $E LABEL LN_SIZE  ;data begins here

Dec Hex STRUCTURE MLN,0 ;same structure, minimal configuration
  0  $0 APTR  MLN_SUCC  ;next node
  4  $4 APTR  MLN_PRED  ;previous node
  8  $8 LABEL MLN_SIZE  ;data starts here

NT_UNKNOWN      =  0 ;not defined
NT_TASK         =  1 ;Exec task
NT_INTERRUPT    =  2 ;interrupt
NT_DEVICE       =  3 ;device
NT_MSGPORT      =  4 ;MP structure
NT_MESSAGE      =  5 ;message sent
NT_FREEMSG      =  6 ;message without ReplyPort
NT_REPLYMSG     =  7 ;reply message
NT_RESOURCE     =  8 ;resource
NT_LIBRARY      =  9 ;library
NT_MEMORY       = 10 ;memory
NT_SOFTINT      = 11 ;software interrupt
NT_FONT         = 12 ;font
```

```
NT_PROCESS      = 13 ;AmigaDOS process
NT_SEMAPHORE    = 14 ;message semaphore
NT_SIGNALSEM    = 15 ;SignalSemaphore
NT_BOOTNODE     = 16 ;boot node
NT_KICKMEM      = 17 ;operating system memory
NT_GRAPHICS     = 18 ;graphics data
NT_DEATHMESSAGE = 19 ;end message
NT_USER         =254 ;maximum user definition
NT_EXTENDED     =255 ;extended node
```

5. Programs

AddTask — **Start a program**

Call:

```
AddTask(task, initialPC, finalPC)
-282(A6) A1   A2         A3

STRUCT TC *task
APTR   initialPC,finalPC
```

Function: Adds a task to the system, redistributes the processor time, and starts the task with the highest priority. Most of the parameters are taken from the initialized Task structure that is passed to this routine. A stack larger than 256 bytes is needed for calling Exec functions. The minimum for other operating system functions is 4096 bytes. The TC_FLAGS are cleared.

Parameters:

task	Initialized TC structure
initialPC	Program start address
finalPC	Return address or 0 (normal)

Warning: Exec tasks cannot use DOS routines, since these require a greatly expanded Task structure (process).

AllocSignal — **Allocate a signal bit**

Call:

```
signalNum = AllocSignal(signalNum)
D0          -330(A6)    D0

BYTE signalNum
```

Function: Allocates a free signal bit from its own task. You can specify a certain bit or the value -1 if any bit will do (this is the normal procedure). Up to 16 different bits can be reserved per task. The other bits are used by the operating system, for example, bit 8 signals an incoming DOS packet.

Parameters: signalNum Bit number (0-31) or -1 (any bit)

Result: Bit number or -1 (bit not free or not bit free)

AllocTrap **Allocate a CPU trap vector**

Call:

```
trapNum = AllocTrap(trapNum)
D0        -342(A6)  D0

LONG trapNum
```

Function: Gets the number of a free CPU trap vector (TRAP #). A certain trap vector can be specified, or -1 can be passed to get the next free vector. Traps are sent to the trap handler in the following format, which is entered in tc_TrapCode: the number of the exception vector is on the stack (32-47 correspond to TRAP #1-#15) followed by the 680x0 exception frame.

Parameters: trapNum Trap number (0-15) or -1

Result: trapNum Number of the allocated trap vector (0-15) or -1 (no free vector).

CacheClearE **Clear cache memory**

Call:

```
CacheClearE(address, length, caches)
-642(A6)     A0       D0      D1

ULONG length, caches
APTR address
```

Function: Clears the internal command and data cache memory of the CPU.

Parameters: address Start address

length — Size of block to be cleared, or -1 to clear all addresses.

caches — The following bits are supported at this time:

CACRF_ClearI	Clear instruction cache
CACRF_ClearD	Clear data cache

CacheClearU — **Clear cache memory**

Call:
```
CacheClearU
-636(A6)
```

Function: Clears all internal command and data cache memory of the CPU.

CacheControl — **Cache control in user mode**

Call:
```
oldBits = CacheControl(cacheBits,cacheMask)
D0        -648(A6)     D0        D1

ULONG oldBits,cachBits,cacheMask
```

Function: Global control via the CACR register of the 68030. All changes to the cache pertain to the entire system. This allows the programmer to turn off the caches of programs not normally executable (self-modifying code, construction of private vector tables, etc.) and run them with extremely reduced processor expenditures.

Parameters: cacheBits — New bit values for the bits to be changed.

cacheMask — Bit mask for the bits to be changed.

Result: The complete CACR register prior to the manipulation.

FindTask — **Find the address of a Task structure**

Call:
```
task = FindTask(name)
D0     -294(A6) A1
```

```
STRUCT TC *task
APTR   name
```

Function: Gets the Task structure of the program with the given name. If no name is given, the routine reads ThisTask from the ExecBase. Since tasks can also remove themselves, it is usually necessary to turn off multitasking.

Parameters: name — String ending in 0 containing the program name.

Result: Task control block, process, or 0

FreeSignal — **Free a signal bit**

Call:

```
FreeSignal(signalNum)
-336(A6)   D0

BYTE signalNum
```

Function: Free a signal bit that was allocated with AllocSignal().

Parameters: signalNum Bit number (0-31) from AllocSignal()

FreeTrap — **Free a CPU trap vector**

Call:

```
FreeTrap(trapNum)
-348(A6) D0

ULONG trapNum
```

Function: Frees a vector allocated with AllocTrap().

Parameters: trapNum — Vector number from AllocTrap().

RemTask — **Remove a program**

Call:

```
RemTask(task)
-288(A6) A1

STRUCT TC *task
```

Function: Remove a task from the system. All linked MemList structures in TC)MEMENTRY are freed (see AllocEntry(), FreeEntry()).

Parameters: task — Address of a task control block or 0 (task removes itself).

SetExcept — **Define exception signal bits**

Call:

```
oldSignals = SetExcept(newSignals, signalMask)
D0           -312(A6)  D0          D1

ULONG oldSignals,newSignals,signalMask
```

Function: Sets the signal bits produced by an exception processed by the task exception handler in tc_ExceptionCode. The handler is passed the ExecBase in A6, the contents of tc_ExceptCode in A1, and the signal bits in d0. It returns a bit mask in which all of the signal bits to be reset are set.

Parameters: newSignals
New bit values for the bits to be changed.

signalMask
Mask with the bits to be changed.

Result: Status of the signal bits prior to the reset.

SetSignal — **Define task signal status**

Call:

```
oldSignals = SetSignal(newSignals, signalMask)
D0           -306(A6)  D0          D1

ULONG oldSignals,newSignals,signalMask
```

Function: Queries and resets received signals.

Parameters: newSignals
New bit values for the bits to be changed.

signalMask
Mask with the bits to be changed.

Result: Signal bits prior to the change.

Example:

```
**=======================================================**
**       Read signal bits                                **
**=======================================================**

 movea.l $4.w,a6
 moveq   #0,d0
 moveq   #0,d1
 jsr     _LVOSetSignal(a6)
 move.l  d0,...

**=======================================================**
**       Clear signal bits                               **
**=======================================================**

 movea.l $4.w,a6
 moveq   #0,d0
 moveq   #-1,d1
 jsr     _LVOSetSignal(a6)
 move.l  d0,...

**=======================================================**
**       Clear signal bit for CONTROL-C                  **
**=======================================================**

 movea.l $4.w,a6
 moveq   #0,d0
 move.l  #SIGBREAKF_CTRL_C,d1
 jsr     _LVOSetSignal(a6)
 move.l  d0,...
```

SetTaskPri	**Change priority of a task**

Call:

```
oldPriority = SetTaskPri(task, priority)
D0            -300(A6)   A1    D0

BYTE   oldPriority
LONG   priority
STRUCT TC *task
```

Function: Changes the priority of a program and updates the distribution of processor time throughout the system.

Parameters: task — Task control block

	priority New priority (127 to -128)
Result:	Previous priority

Signal — **Sends a signal to a program**

Call:

```
Signal(task, signals)
-324(A6) A1   D0

STRUCT TC *task
ULONG  signals
```

Function: Sends the signal bits in the given signal mask to a task. If the task was waiting for one of the signals, it is re-activated and the processor time distribution is recalculated.

Parameters: task — Task control block

signals — Signal mask

Wait — **Wait for a signal**

Call:

```
signals = Wait(signalSet)
D0          -318(A6) D0

ULONG signals,signalSet
```

Function: Turns off own task and waits for one of the given signal bits.

Parameters: signalSet — Signal bit mask

Result: The received signal.

```
Dec Hex STRUCTURE  TC_Struct,LN_SIZE ;previously TC
 14  $E UBYTE    TC_FLAGS            ;Flags
 15  $F UBYTE    TC_STATE            ;Status
 16 $10 BYTE     TC_IDNESTCNT        ;saved IDNestCnt
 17 $11 BYTE     TC_TDNESTCNT        ;saved TDNestCnt
 18 $12 ULONG    TC_SIGALLOC         ;allocated Signalbits
 22 $16 ULONG    TC_SIGWAIT          ;expected Signalbits
 26 $1A ULONG    TC_SIGRECVD         ;received Signalbits
 30 $1E ULONG    TC_SIGEXCEPT        ;signal for Exception Handler
```

```
 34 $22 APTR    tc_ETask             ;extension structure
 38 $26 APTR    TC_EXCEPTDATA        ;data for Exception Handler
 42 $2A APTR    TC_EXCEPTCODE        ;Exception Handler
 46 $2E APTR    TC_TRAPDATA          ;data for Trap Handler
 50 $32 APTR    TC_TRAPCODE          ;Trap Handler
 54 $36 APTR    TC_SPREG             ;StackPointer
 58 $3A APTR    TC_SPLOWER           ;lower limit of stack
 62 $3E APTR    TC_SPUPPER           ;upper limit of stack
 66 $42 FPTR    TC_SWITCH            ;routine task switch
 70 $46 FPTR    TC_LAUNCH            ;routine task start
 74 $4A STRUCT  TC_MEMENTRY,LH_SIZE  ;memory for task
 88 $58 APTR    TC_Userdata          ;data for task
 92 $5C LABEL   TC_SIZE

Dec Hex STRUCTURE ETask,MN_SIZE       ;task extension
 20 $14 APTR   et_Parent              ;TC_Struct
 24 $18 ULONG  et_UniqueID            ;task ID
 28 $1C STRUCT et_Children,MLH_SIZE   ;sub-tasks
 40 $28 UWORD  et_TRAPALLOC           ;allocated Traps
 42 $2A UWORD  et_TRAPABLE            ;possible Traps
 44 $2C ULONG  et_Result1             ;1. result
 48 $30 APTR   et_Result2             ;result address (AllocVec)
 52 $34 STRUCT et_TaskMsgPort,MP_SIZE ;TaskPort
 86 $56 LABEL  ETask_SIZEOF           ;not the true size!!!

CHILD_NOTNEW   = 1 ;call to old task (TC)
CHILD_NOTFOUND = 2 ;sub-task not found
CHILD_EXITED   = 3 ;sub-task ended
CHILD_ACTIVE   = 4 ;sub-task active

TB_PROCTIME = 0, TF_PROCTIME =   1
TB_ETASK    = 3, TF_ETASK    =   8
TB_STACKCHK = 4, TF_STACKCHK = $10
TB_EXCEPT   = 5, TF_EXCEPT   = $20
TB_SWITCH   = 6, TF_SWITCH   = $40
TB_LAUNCH   = 7, TF_LAUNCH   = $80

TS_INVALID  =     0
TS_ADDED    =     TS_INVALID+1
TS_RUN      =     TS_ADDED+1
TS_READY    =     TS_RUN+1
TS_WAIT     =     TS_READY+1
TS_EXCEPT   =     TS_WAIT+1
TS_REMOVED  =     TS_EXCEPT+1

SIGB_ABORT     = 0, SIGF_ABORT     =    1
SIGB_CHILD     = 1, SIGF_CHILD     =    2
SIGB_BLIT      = 4, SIGF_BLIT      =  $10
```

```
SIGB_SINGLE     = 4, SIGF_SINGLE     =  $10
SIGB_INTUITION = 5, SIGF_INTUITION =  $20
SIGB_DOS        = 8, SIGF_DOS        = $100

SYS_SIGALLOC  = $FFFF ;system signal bits
SYS_TRAPALLOC = $8000 ;system traps (TRAP #15)
```

6. Communications

AddPort	**Make MsgPort available to other tasks**

Call:

```
AddPort(port)
-354(A6) A1

STRUCT MsgPort *port
```

Function: Adds the given MsgPort to the system list so that other programs can access it with FindPort() and address it.

Parameters: port — MessagePort structure (LN_NAME <> 0 if the port must be found with FindPort.).

Alert	**Indicates an error**

Call:

```
Alert(alertNum)
-108(A6) D7

ULONG alertNum
```

Function: Indicates a catastrophic error (Guru Meditation). Debugging with a second computer attached via the serial port is usually possible (9600 baud, 8 bits, n parity).

Parameters: alertNum — Error code

See also: exec/alerts.h

CreateMsgPort	**Create MP structure**

Call:

```
port = CreateMsgPort()
d0     -666(A6)

STRUCT MsgPort *port
```

Function: Allocates the memory required for a MsgPort and initializes it. The message queue list is created, a signal bit is allocated, the task is entered, and the port is set to PA_SIGNAL (for WaitPort()). The port can only be freed with DeleteMsgPort().

Result: MsgPort or 0

Debug — **Starts system debugger**

Call:

```
Debug(flags)
-114(A6) D0

ULONG flags
```

Function: Calls the system debugger. Normally, this is the "ROM-WACK", but you can also patch the Debug() function with SetFunction().

Parameters: flags 0 at this time

DeleteMsgPort — **Free MP created with CreateMsgPort()**

Call:

```
DeleteMsgPort(msgPort)
-672(A6)      a0

STRUCT MsgPort *msgPort
```

Function: Frees a MessagePort created with CreateMsgPort().

Parameters: msgPort MP structure from CreateMsgPort() or0.

FindPort — **Find MsgPort**

Call:

```
port = FindPort(name)
D0     -390(A6) A1

STRUCT MP *port
APTR   name
```

Function: Finds port in the system list with the given name (LN_NAME).

Parameters: name — Port name string ending in 0.

Result: MsgPort address or 0

GetMsg — Get next MessageNode from the port

Call:

```
message = GetMsg(port)
D0        -372(A6) A0

STRUCT MN *message
STRUCT MP *port
```

Function: Gets the next message from the port's queue. WaitPort() or Wait() are used to wait for messages. Messages must be answered with ReplyMsg(). A signal does not always indicate a message has arrived, it may also indicate several messages have arrived (security prompt).

Parameters: port — MessagePort

Result: MessageNode or 0 if no message has arrived at the port.

PutMsg — Send a MessageNode to a port

Call:

```
PutMsg(port, message)
-366(A6) A0   A1

STRUCT MP *port
STRUCT MN *message
```

Function: Sends a message to a port. Depending on MP_FLAGS, the port program is also notified.

Parameters: port — MP structure of the destination port.

message — MessageNode to be sent.

RawDoFmt — Format a string

Call:

```
RawDoFmt(FormatString, DataStream, PutChProc, PutChData)
   -522(A6) a0              a1          a2          a3
```

```
APTR FormatString,DataStream,PutChData
FPTR PutChProc
```

Function: A format string is loaded with the given arguments (this is the basis of C routines such as PrintF(), etc.). The arguments are in word or longword widths. The prefix code for an argument is the % character. To get a % character in the result string, the format string must contain %%. The output is sent to the result buffer one character at a time using the given Assembler routine.

Parameters: FormatString

String with arguments in the following format:

```
%[flag][width.limit][length]type
```

flag '-'	Left justify
width	Width of argument. If the first character is '0', the given width to the left is filled with zeros.
limit	Maximum width, if the argument is a string.
length 'l'	Longword, otherwise word (only with numbers).
type	Argument type (in DataStream):

b	BSTR (BPTR to a BCPL string)
d	Decimal number
x	Hexadecimal number (characters 0-F only)
s	String address
c	Individual character

DataStream
: Memory block containing the values and/or addresses of the arguments one after another.

PutChProc
: Address of an Assembler routine that writes a character to PutChData. This routine receives the character in d0 and PutChData in a3. This routine normally looks like this: 'MOVE.B D0,(A3)+ :RTS'. The last character is a 0 byte.

PutChData
: Buffer for storing the result string.

Example: Format text and output to a RastPort:

```
**
** Example (Result: "reading cyl 1, 78 to go")
**
 ...
 movea.l _RastPort,a2
 lea     _Format,a0
 lea     _Parameter,a1
 bsr     _Print
 ...

_Format
 dc.b    '%s cyl %d, %d to go',0
 cnop    0,2

_Parameter
dc.l    _Action
 dc.w    1
 dc.w    78

_Action
 dc.b    'reading',0
 dc.b    'writing',0
 dc.b    "ver'ing",0

_Print
 movem.l a2-a3/a6,-(a7)
 lea     .PutChar(pc),a2
 move.l  a7,-4(a2)
 lea     -100(a7),a7
```

```
 movea.l a7,a3
 movea.l $4.w,a6
 jsr     _LVORawDoFmt(a6)
 movea.l 100(a7),a1
 movea.l a7,a0
.Loop
 tst.b   (a3)+
 bne.s   .Loop
 subq.l  #2,a3
 move.l  a3,d0
 sub.l   a7,d0
 movea.l _GfxBase,a6
 jsr     _LVOText(a6)
 lea     100(a7),a7
 movem.l (a7)+,a2-a3/a6
 rts
.BufferEnd
 dc.l    0
.PutChar
 cmpa.l  .BufferEnd(pc),a3
 beq.s   .Overflow
 move.b  d0,(a3)+
 rts
.Overflow
 clr.b   -1(a3)
 rts
```

RemPort — **Remove a MessagePort from the system list**

Call:

```
RemPort(port)
-360(A6) A1

STRUCT MP *port
```

Function: Removes a port added with AddPort() from the list.

Parameters: port MessagePort

ReplyMsg — **Reply to a message**

Call:

```
ReplyMsg(message)
-378(a6) A1

STRUCT MN *message
```

Function: After processing a message, this routine sends a MessageNode back to the sender or its port (MN_REPLYPORT).

Parameters: message Address of the MessageNode.

WaitPort **Wait for a message**

Call:

```
message = WaitPort(port)
D0        -384(A6) A0

STRUCT MN *message
STRUCT MP *port
```

Function: Turns off own task and waits for the receipt of one or more messages at the given port. MP_SIGTASK and MP_SIGBIT must be initialized and MP_FLAGS must be set to PA_SIGNAL.

Parameters: port MsgPort

Result: Address of the first MessageNode (not removed from the port. Use GetMsg()).

Alarm Types:

```
AT_DeadEnd      = $80000000 ;reset after display
AT_Recovery     = $00000000 ;recovery possible
```

Alarm Groups:

```
AG_NoMemory     = $00010000 ;no memory
AG_MakeLib      = $00020000 ;create library
AG_OpenLib      = $00030000 ;open library
AG_OpenDev      = $00040000 ;open device
AG_OpenRes      = $00050000 ;open resource
AG_IOError      = $00060000 ;I/O error
AG_NoSignal     = $00070000 ;no signal
AG_BadParm      = $00080000 ;bad parameter
AG_CloseLib     = $00090000 ;closed too many times
AG_CloseDev     = $000A0000 ;closed too many times
AG_ProcCreate   = $000B0000 ;create process
```

Alarm Objects:

```
AO_ExecLib       = $00008001 ;Exec Library
AO_GraphicsLib   = $00008002 ;Gfx Library
AO_LayersLib     = $00008003 ;Layers Library
AO_Intuition     = $00008004 ;Intuition Library
AO_MathLib       = $00008005 ;Math Library
AO_DOSLib        = $00008007 ;DOS Library
AO_RAMLib        = $00008008 ;RAM Library
AO_IconLib       = $00008009 ;Icon Library
AO_ExpansionLib  = $0000800A ;Expansion Library
AO_DiskfontLib   = $0000800B ;Diskfont Library
AO_UtilityLib    = $0000800C ;Utility Library
AO_AudioDev      = $00008010 ;Audio Device
AO_ConsoleDev    = $00008011 ;Console Device
AO_GamePortDev   = $00008012 ;Gameport Device
AO_KeyboardDev   = $00008013 ;Keyboard Device
AO_TrackDiskDev  = $00008014 ;Trackdisk Device
AO_TimerDev      = $00008015 ;Timer Device
AO_CIARsrc       = $00008020 ;CIAx Resource
AO_DiskRsrc      = $00008021 ;Disk Resource
AO_MiscRsrc      = $00008022 ;Misc. Resource
AO_BootStrap     = $00008030 ;Strap
AO_Workbench     = $00008031 ;Workbench Library
AO_DiskCopy      = $00008032 ;Diskcopy
AO_GadTools      = $00008033 ;GadTools Library
AO_Unknown       = $00008035 ;unknown object

Dec Hex STRUCTURE MP,LN_SIZE        ;MsgPort
 14  $E UBYTE    MP_FLAGS           ;signal type
 15  $F UBYTE    MP_SIGBIT          ;signal bit number
 16 $10 APTR     MP_SIGTASK         ;task or interrupt
 20 $14 STRUCT   MP_MSGLIST,LH_SIZE ;message queue
 34 $22 LABEL    MP_SIZE

MP_SOFTINT   =   MP_SIGTASK   ;for PA_SOFTINT

PF_ACTION    =   3 ;mask
PA_SIGNAL    =   0 ;signal to task MP_SIGTASK
PA_SOFTINT   =   1 ;execute software interrupt MP_SOFTINT
PA_IGNORE    =   2 ;ignore

Dec Hex STRUCTURE  MN,LN_SIZE ;message
 14  $E APTR     MN_REPLYPORT ;MsgPort for reply
 18 $12 UWORD    MN_LENGTH    ;total structure size
 20 $14 LABEL    MN_SIZE      ;data begins here
```

Example:

```
RawKeyMapping:

 movea.l _SysBase,a6
 movea.l _Window,a3
 movea.l wd_UserPort(a3),d1
 beq.s   _ErrorNoUserPort
 movea.l d1,a3
 bra.s   _GetMessage

_WaitMsg
 moveq   #-1,d0
 jsr     _LVOAllocSignal(a6)
 tst.b   d0
 bmi.s   _GetMessage

 move.b  d1,MP_SIGBIT(a3)
 move.l  ThisTask(a6),MP_SIGTASK(a3)
 clr.b   MP_FLAGS(a3)

 movea.l a3,a0
 jsr     _LVOWaitPort(a6)

 addq.b  #PA_IGNORE,MP_FLAGS(a3)

 move.b  MP_SIGBIT(a3),d0
 jsr     _LVOFreeSignal(a6)

_GetMessage
 movea.l a3,a0
 jsr     _LVOGetMsg(a6)
 tst.l   d0
 beq.s   _WaitMsg

 movea.l d0,a4
 move.l  im_Class(a4),d0
 cmpi.l  #RAWKEY,d0
 beq     _RawKey

 ...

_ErrorNoUserPort

 ...

_RawKey
 movea.l _KeymapBase,a6
```

```
 lea      -ie_SIZEOF(a7),a7
 movea.l a7,a0
 clr.l    (a0)
 move.b   #IECLASS_RAWKEY,ie_Class(a0)
 clr.b    ie_SubClass(a0)
 move.w   im_Code(a4),ie_Code(a0)
 move.w   im_Qualifier(a4),ie_Qualifier(a0)
 move.l   im_IAddress(a4),ie_EventAddress(a0)
 lea      _Buffer(pc),a1
 moveq    #79,d1
 lea      $0.w,a2
 jsr      _LVOMapRawKey(a6)
 move.l   d0,_CharsInBuffer
 lea      ie_SIZEOF(a7),a7

 movea.l a4,a1
 movea.l _SysBase,a6
 jsr      _LVOReplyMsg(a6)

 ...

_CharsInBuffer
 dc.l     0

 ...

_Buffer
 ds.b     80
```

7. Libraries

AddLibrary	**Adds a library to the system list**

Call: `AddLibrary(library)`
`-396(A6)    A1`

`STRUCT Library *library`

Function: Makes a complete, initialized library available to other programs. Also, calculates the check sum for the library.

Parameters: library   Base address of the library.

CloseLibrary — **Close a library**

Call:

```
CloseLibrary(library)
-414(A6)     A1

STRUCT Library *library
```

Function: Closes a library. This is necessary in order to free the memory occupied by unused libraries.

Parameters: library — Base address of the library or 0.

OldOpenLibrary — **For Kickstart 1.0 compatibility**

Call:

```
library = OldOpenLibrary(libName)
D0         -408(A6)        A1

STRUCT Library *library
APTR   libname
```

Function: This function exists only to maintain compatibility with operating system Version 1.0. It corresponds to OpenLibrary(libName,0) and should no longer be used.

OpenLibrary — **Open a library**

Call:

```
library = OpenLibrary(libName, version)
D0         -552(A6)     A1       D0

STRUCT Library *library
APTR   libName
ULONG  version
```

Function: Opens a library, gets the base address, and prevents the library from being removed from memory. This function also checks to make sure that the library has the given minimum version number. A value of 0 will accept any version, but this should never be used. Since there is no documentation on which operating system version contains which library versions, here is a list:

```
Kick ?.? = LibVersion  0 (no longer supported!!!)
Kick 1.0 = LibVersion 30 (no longer supported!!!)
Kick 1.1 (NTSC) =  V. 31 (no longer supported!!!)
Kick 1.1 (+PAL) =  V. 32 (no longer supported!!!)
Kick 1.2 = LibVersion 33
Kick 1.3 = LibVersion 34
Kick 1.3 (+A2024) =   34/35
Kick 2.0 = LibVersion 36 (described in this book)
```

If the library is not in the list, DOS loads it from disk (the default directory is LIBS:). Because of this, only DOS processes can call this function for non-resident libraries. A complete path can also be given instead of a name.

Parameters: libName — Library name (+path if desired). Upper and lowercase letters are also distinguished in paths.

version — Minimum version number

Result: Base address of the library or 0.

RemLibrary — Attempt to delete a library

Call:

```
RemLibrary(library)
-402(A6)   A1

STRUCT Library *library
```

Function: Calls the LIB_EXPUNGE routine of the given library. This sets the automatic removal feature for extra libraries. The library will automatically be removed when it is no longer needed.

Parameters: library — Base address of the library.

Example: Attempt to remove a library from memory:

```
**
** Input: a1=LibName
**
 movea.l $4.w,a6
 addq.b  #1,TDNestCnt(a6)
```

```
 lea      LibList(a6),a0
 jsr      _LVOFindName(a6)
 tst.l    d0
 beq.s    .notfound

 movea.l d0,a1
 jsr      _LVORemLibrary(a6)

.notfound
 subq.b   #1,TDNestCnt(a6)

 ...
```

SetFunction — **Divert a library function**

Call:
```
oldFunc = SetFunction(library, funcOffset, funcEntry)
D0        -420(A6)    A1       A0.W        D0

APTR   oldFunc,funcEntry
STRUCT Library *library
LONG   funcOffset
```

Function: Routine for patching operating system functions.

Parameters:

library — Base address of the library.

funcOffset — Offset of the routine (LVO).

funcEntry — Address of the new function.

Result: Address of the old function.

SumLibrary — **Calculate check sum for a library**

Call:
```
SumLibrary(library)
-426(A6)    A1

STRUCT Library *library
```

Function: Recalculates the check sum of a library. If the results does not agree with the given check sum and the CHANGED flag is not set, then the Alert() function is called.

Parameters: library Base address of the library.

```
LIB_OPEN    =  -6 ;LVO open library
LIB_CLOSE   = -12 ;LVO close library
LIB_EXPUNGE = -18 ;LVO remove library
LIB_EXTFUNC = -24 ;LVO future extension

Dec Hex STRUCTURE LIB,LN_SIZE ;library base structure
 14  $E UBYTE   LIB_FLAGS     ;Flags
 15  $F UBYTE   LIB_pad
 16 $10 UWORD   LIB_NEGSIZE   ;vector table size
 18 $12 UWORD   LIB_POSSIZE   ;size of base structure
 20 $14 UWORD   LIB_VERSION   ;version number
 22 $16 UWORD   LIB_REVISION  ;revision number
 24 $18 APTR    LIB_IDSTRING  ;identification string
 28 $1C ULONG   LIB_SUM       ;check sum
 32 $20 UWORD   LIB_OPENCNT   ;number of opens
 34 $22 LABEL   LIB_SIZE
```

LIB_Flags values:

```
LIBB_SUMMING = 0, LIBF_SUMMING =  1 ;check sum calculation
LIBB_CHANGED = 1, LIBF_CHANGED =  2 ;library changed
LIBB_SUMUSED = 2, LIBF_SUMUSED =  4 ;calculate check sum
LIBB_DELEXP  = 3, LIBF_DELEXP  =  8 ;self-removal
LIBB_EXP0CNT = 4, LIBF_EXP0CNT = 16 ;same for system
```

8. Devices

AbortIO — **Abort I/O process**

Call:
```
AbortIO(iORequest)
-480(A6) A1

STRUCT IORequest *iORequest
```

Function: Attempts to abort a currently running I/O process. Regardless of whether or not this is successful, it must use WaitIO() to wait for the official end of the process.

Parameters: iORequest IO structure of any size (active or complete).

AddDevice	Make a device available to other programs

Call:

```
AddDevice(device)
-432(A6)   A1

STRUCT Device *device
```

Function: Enters a fully initialized Device structure into the system list.

Parameters: device   Base address of the device.

CheckIO	Check to see if an I/O process is completed

Call:

```
result = CheckIO(iORequest)
D0       -468(A6) A1

BOOL   result
STRUCT IORequest *iORequest
```

Function: This function checks to see if an I/O process started with SendIO() is still running or is finished. Even if the process has finished, WaitIO() must be used to wait for the official process end.

Parameters: iORequest IO structure of any size (active or complete).

Result: 0 if the process is still running; otherwise the address of the IO structure is returned.

CloseDevice	Close a device

Call:

```
CloseDevice(iORequest)
-450(A6)    A1

STRUCT IORequest *iORequest
```

Function: Closes access to a device and the sub-objects of the device.

Parameters: iORequest IO structure from OpenDevice().

CreateIORequest **Create IO structure**

Call:

```
ioReq = CreateIORequest( ioReplyPort, size )
D0      -654(A6)         A0           D0

STRUCT IORequest *ioReq
STRUCT MsgPort *ioReplyPort
ULONG  size
```

Function: Creates and initializes an IO structure of any size.

Parameters: ioReplyPort
Address of a fully initialized MsgPort (see CreateMsgPort()).

size — Size of the IO structure.

Result: IO structure or 0 (error).

DeleteIORequest
Free an IO structure created with CreateIORequest()

Call:

```
DeleteIORequest( ioReq )
-660(A6)         a0

STRUCT IORequest *ioReq
```

Function: Frees a structure created with CreateIORequest().

Parameters: ioReq — Result form CreateIORequest() or 0.

DoIO **Execute I/O process**

Call:

```
error = DoIO(iORequest)
D0      -456(A6) A1

BYTE   error
STRUCT IORequest *iORequest
```

Function: Transfers an IO structure containing the required data to a device which extracts the command and executes it. This function returns at the end of the process.

Parameters: iORequest Initialized IO structure from OpenDevice() which was manually loaded with device-specific data.

Result: 0 or a device-specific error code.

OpenDevice **Register access to a device**

Call:

```
error = OpenDevice(devName, unitNumber, iORequest, flags)
D0      -444(A6)   A0       D0          A1         D1

BYTE   error
APTR   devName
ULONG  unitNumber,flags
STRUCT IORequest *iORequest
```

Function: Attempts to obtain access to a device. The passed IO structure is supplied the necessary data if it's successful. If the device is not in memory, it attempts to load it from (hard) disk. Possible to specify a complete path.

Parameters: devName Name of the device (distinguishes uppercase and lowercase notation).

unitNumber
Number of a subunit (e.g., 1-DF1:) or null.

iORequest I/O structure

flags Special information

Result: Null or error code.

Example: Attempt to remove a device from memory:

```
**
** Input: a1=DevName
**
 movea.l $4.w,a6
 addq.b  #1,TDNestCnt(a6)

 lea     DeviceList(a6),a0
 jsr     _LVOFindName(a6)
```

```
tst.l    d0
beq.s    .notfound

movea.l d0,a1
jsr      _LVORemDevice(a6)

.notfound
subq.b  #1,TDNestCnt(a6)

...
```

RemDevice	**Remove device**

Call:
```
RemDevice(device)
-438(A6)  A1

STRUCT Device *device
```

Function: Attempts to initiate a device removing itself from memory.

Parameters: device Base address of the device.

SendIO	**Start I/O process**

Call:
```
SendIO(iORequest)
-462(A6) A1

STRUCT IORequest *iORequest
```

Function: Starts an I/O process without waiting for the end.

Parameters: iORequest I/O structure

WaitIO	**Wait for the end of an I/O process**

Call:
```
error = WaitIO(iORequest)
D0      -474(A6) A1

BYTE   error
STRUCT IORequest *iORequest
```

Function: Waits for the end of an I/O process started with SendIO().

Parameters: iORequest I/O structure (active or completed)

Result: Null or error code.

```
Dec Hex STRUCTURE  DD,LIB_SIZE      ;Device structure
 34 $22 LABEL      DD_SIZE

Dec Hex STRUCTURE  UNIT,MP_SIZE     ;Unit structure
 34 $22 UBYTE   UNIT_FLAGS          ;Flags
 35 $23 UBYTE   UNIT_pad
 36 $24 UWORD   UNIT_OPENCNT        ;Number of openings
 38 $26 LABEL   UNIT_SIZE

UNITB_ACTIVE = 0, UNITF_ACTIVE = 1 ;working now
UNITB_INTASK = 1, UNITF_INTASK = 2 ;in the device task

IOERR_OPENFAIL    = -1 ;Error opening
IOERR_ABORTED     = -2 ;Process aborted
IOERR_NOCMD       = -3 ;Unknown command
IOERR_BADLENGTH   = -4 ;Length not okay
IOERR_BADADDRESS  = -5 ;Address not okay
IOERR_UNITBUSY    = -6 ;Unit still working
IOERR_SELFTEST    = -7 ;Hardware error
ERR_OPENDEVICE    =  IOERR_OPENFAIL

Dec Hex STRUCTURE  IO,MN_SIZE ;I/O structure
 20 $14 APTR    IO_DEVICE     ;Device base address
 24 $18 APTR    IO_UNIT       ;Unit structure
 28 $1C UWORD   IO_COMMAND    ;Command
 30 $1E UBYTE   IO_FLAGS      ;Flags
 31 $1F BYTE    IO_ERROR      ;Error code
 32 $20 LABEL   IO_SIZE
 32 $20 ULONG   IO_ACTUAL     ;Moved bytes etc.
 36 $24 ULONG   IO_LENGTH     ;Length
 40 $28 APTR    IO_DATA       ;Data address
 44 $2C ULONG   IO_OFFSET     ;Offset for positioning
 48 $30 LABEL   IOSTD_SIZE

IOB_QUICK = 0, IOF_QUICK = 1  ;execute immediately

CMD_INVALID = 0 ;No command
CMD_RESET   = 1 ;reset device
CMD_READ    = 2 ;Read
CMD_WRITE   = 3 ;Write
CMD_UPDATE  = 4 ;Write buffer
CMD_CLEAR   = 5 ;Clear buffer
CMD_STOP    = 6 ;Stop
```

```
CMD_START   = 7 ;Continue
CMD_FLUSH   = 8 ;Delete commands
CMD_NONSTD  = 9 ;1. Device specific command
```

9. Resources

AddResource — Make a resource accessible to other programs

Call:
```
AddResource(resource)
-486(A6)    A1

APTR resource
```

Function: Adds a completely initialized resource to the system list.

Parameters: resource — Library node of the resource.

OpenResource — Get the base address of a resource

Call:
```
resource = OpenResource(resName)
D0         -498(A6)     A1

APTR resource,resName
```

Function: Retrieves the base address of a resource.

Parameters: resName — Resource name

Result: Base address or 0 (error).

RemResource — Attempt to remove a resource

Call:
```
RemResource(resource)
-492(A6)    A1

APTR resource
```

Function: Attempts to initiate self-removal of the given resource.

Parameters: resource — Base address of the resource.

10. Semaphores

AddSemaphore	**Initialize and link semaphore**

Call:

```
AddSemaphore(signalSemaphore)
-600(A6)     A1

STRUCT SS *signalSemaphore
```

Function: Initializes an SS structure containing a name and priority and adds it to the system list.

Parameters: signalSemaphore
SS structure

AttemptSemaphore	**Attempt to allocate a semaphore**

Call:

```
success = AttemptSemaphore(signalSemaphore)
D0        -576(A6)         A0

LONG success
STRUCT SS *signalSemaphore
```

Function: Attempts to allocate a semaphore and returns to the caller if this is not possible.

Parameters: signalSemaphore
SS structure

Result: 0 SS was not free.

FindSemaphore	**Find a semaphore**

Call:

```
signalSemaphore = FindSemaphore(name)
D0                -594(A6)      A1

STRUCT SS *signalSemaphore
APTR   name
```

Function: Attempts to find a semaphore with the given name.

Parameters: name Semaphore name

Result: SS structure or 0

InitSemaphore — **Initialize signal semaphore**

Call:

```
InitSemaphore(signalSemaphore)
-558(A6)       A0

STRUCT SS *signalSemaphore
```

Function: Initializes an SS structure.

Parameters: signalSemaphore
Deleted SS structure

ObtainSemaphore — **Obtain exclusive access to a semaphore**

Call:

```
ObtainSemaphore(signalSemaphore)
-564(A6)        A0

STRUCT SS *signalSemaphore
```

Function: Allocates an SS structure. If this is not possible, the task is turned off until the semaphore is freed.

Parameters: signalSemaphore
SS structure

ObtainSemaphoreList — **Allocate semaphores in a list**

Call:

```
ObtainSemaphoreList(list)
-582(A6)            A0

STRUCT LH *list
```

Function: Allocates all semaphores in the list or waits for them to be freed.

Parameters: list — Semaphore list

ObtainSemaphoreShared — **Shared semaphore access**

Call:

```
ObtainSemaphoreShared(signalSemaphore)
-678(A6)                a0

STRUCT SS *signalSemaphore
```

Function: Obtains shared access to a semaphore or waits for it to be freed.

Parameters: signalSemaphore
SS structure

Procure **Allocate message semaphore**

Call:

```
result = Procure(semaphore, bidMessage)
D0       -540(A6) A0         A1

BYTE result
STRUCT Semaphore *semaphore
STRUCT MN *bidMessage
```

Function: Attempts to allocate a semaphore.

Parameters: semaphore A semaphore MsgPort

Result: 0 Semaphore was not free.

ReleaseSemaphore **Free semaphore**

Call:

```
ReleaseSemaphore(signalSemaphore)
-570(A6)         A0

STRUCT SS *signalSemaphore
```

Function: Frees a given semaphore.

Parameters: signalSemaphore
SS structure

ReleaseSemaphoreList **Free a semaphore list**

Call:

```
ReleaseSemaphoreList(list)
-588(A6)             A0

STRUCT LH *list
```

Function: Frees a semaphore list.

Parameters: list Semaphore list

RemSemaphore **Remove a semaphore**

Call:
```
RemSemaphore(signalSemaphore)
-606(A6)     A1

STRUCT SS *signalSemaphore
```

Function: Removes a semaphore from its list.

Parameters: signalSemaphore
SS structure

Vacate **Free a message semaphore**

Call:
```
Vacate(semaphore)
-546(A6) A0

STRUCT Semaphore *semaphore
```

Function: Frees a semaphore.

Parameters: semaphore Semaphore MsgPort

```
Dec Hex STRUCTURE  SSR,MLN_SIZE           ;PRIVATE!
  8  $8 APTR     SSR_WAITER
 12  $C LABEL    SSR_SIZE

Dec Hex STRUCTURE  SS,LN_SIZE             ;SignalSemaphore
 14  $E WORD     SS_NESTCOUNT             ;number of tasks
 16 $10 STRUCT   SS_WAITQUEUE,MLH_SIZE    ;wait queue
 28 $1C STRUCT   SS_MULTIPLELINK,SSR_SIZE ;link
 40 $28 APTR     SS_OWNER                 ;Task
 44 $2C WORD     SS_QUEUECOUNT            ;queued Tasks
 46 $2E LABEL    SS_SIZE

Dec Hex STRUCTURE  SM,MP_SIZE             ;Message semaphore
 34 $22 WORD     SM_BIDS                  ;number of bids
 36 $24 LABEL    SM_SIZE

SM_LOCKMSG    =  MP_SIGTASK
```

Example for Exec Library

Exec has several new functions that make access to devices considerably easier. As an example, let's take a look at how direct access to a disk drive can be programmed:

```
**====================================================**
**      Direct access to a floppy disk drive        **
**----------------------------------------------------**
**      Input:   A6 = ExecBase                        **
**               A5 = DosBase                         **
**               D0 = Drive (0...3)                   **
**      Output:  D0 = IOEXTTD                         **
**====================================================**

_GetAccess movem.l d2-d5,-(a7)
           move.l  d0,d5                    ;drive number

           jsr     _LVOCreateMsgPort(a6)    ;get port
           move.l  d0,d3                    ;save address
           beq.s   .Error

           movea.l d0,a0                    ;port to a0
           moveq   #IOTD_SIZE,d0            ;size to d0
           jsr     _LVOCreateIORequest(a6)  ;get IORequest
           move.l  d0,d4                    ;save address
           beq.s   .DelPort

           lea     _TDName(pc),a0           ;name to a0
           move.l  d5,d0                    ;number to d0
           movea.l d0,a1                    ;IORequest
           moveq   #0,d1                    ;3.5" disks
           jsr     _LVOOpenDevice(a6)       ;open
           tst.l   d0                       ;error test
           bne.s   .DelIOReq

           exg     a5,a6                    ;DosBase to a6
           lsl.l   #8,d5                    ;number << 1 byte
           addi.l  #'DF0:',d5               ;add string
           clr.w   -(a7)                    ;end of string
           move.l  d5,-(a7)                 ;move string
           move.l  a7,d1                    ;string to d1
           jsr     _LVODeviceProc(a6)       ;Handler port
           addq.l  #6,a7                    ;clear stack
           move.l  d0,d1                    ;port to d1
           beq.s   .NoDevProc
```

```
            moveq    #DOSTRUE,d2              ;set Flag
            jsr      _LVOInhibit(a6)          ;inhibit access
            exg      a5,a6                    ;Exec to a6
            tst.l    d0                       ;error test
            beq.s    .CloseDev
            move.l   d4,d0                    ;IORequest -> d0

.Exit       movem.l  (a7)+,d2-d5              ;clean up
            rts                               ;end

.NoDevProc  exg      a5,a6                    ;Exec to a6
.CloseDev   movea.l  d4,a1                    ;IOReq to a1
            jsr      _LVOCloseDevice(a6)      ;close Dev

.DelIOReq   movea.l  d4,a0                    ;IOReq to a0
            jsr      _LVODeleteIORequest(a6)  ;delete IOReq

.DelPort    movea.l  d3,a0                    ;port to a0
            jsr      _LVODeleteMsgPort(a6)    ;delete port
            moveq    #0,d0                    ;no result
            bra.s    .Exit                    ;end

**=======================================================**
**                 Free drive                            **
**-------------------------------------------------------**
**          Input:  A6 = ExecBase                        **
**                  A5 = DosBase                         **
**                  A1 = IORequest                       **
**                  D0 = Drive (0...3)                   **
**          Output: D0 = Success (0=Error)               **
**=======================================================**

_FreeDrive  movem.l  d2-d3,-(a7)
            move.l   d0,d3                    ;save drive

            move.l   a1,d2                    ;save IOReq
            jsr      _LVOCloseDevice(a6)      ;close Dev

            movea.l  d2,a0                    ;IOReq to a0
            move.l   MN_REPLYPORT(a0),d2      ;save port
            jsr      _LVODeleteIORequest(a6)  ;delete IOReq

            movea.l  d2,a0                    ;port to a0
            jsr      _LVODeleteMsgPort(a6)    ;delete port

            exg      a5,a6                    ;DOS to a6
            lsl.l    #8,d3                    ;number << 1 byte
            addi.l   #'DF0:',d3               ;add string
```

```
            clr.w    -(a7)                  ;end of string
            move.l   d3,-(a7)               ;move string
            move.l   a7,d1                  ;string to d1
            jsr      _LVODeviceProc(a6)     ;Handler port
            addq.l   #6,a7                  ;clear stack
            move.l   d0,d1                  ;port to d1
            beq.s    .NoDevProc

            moveq    #DOSFALSE,d2           ;code to free
            jsr      _LVOInhibit(a6)        ;free

.NoDevProc  exg      a5,a6                  ;Exec to a6
            tst.l    d0                     ;set CC
            movem.l  (a7)+,d2-d3            ;clean up
            rts                             ;end

_TDName     dc.b     'trackdisk.device',0   ;DeviceName
```

While we are working with the trackdisk device, here is a program that turns off the annoying clicking sound made by an empty disk drive. This program can be started from the CLI/Shell or the Workbench. It is made possible by a new flag in the Unit structures. We will also see an example of minimum message handling for Workbench starts, especially at the end of the program, which is responsible for freeing memory when the program is segmented:

```
 OPT O+
 INCLUDE IncAll.i
**
**
** NoClick
**
**
_Startup
 movea.l $4.w,a6                        ;load ExecBase

 movea.l ThisTask(a6),a5                ;get process
 moveq   #0,d7                          ;WbStartup to 0
 tst.l   pr_CLI(a5)                     ;test CLI
 bne.s   _CLIstart                      ;->if available

 lea     pr_MsgPort(a5),a0              ;ProcessPort
 jsr     _LVOWaitPort(a6)               ;wait for message
 lea     pr_MsgPort(a5),a0              ;ProcessPort
 jsr     _LVOGetMsg(a6)                 ;get message
 move.l  d0,d7                          ;save WbStartup
```

```
_CLIstart
 cmpi.w  #36,LIB_VERSION(a6)             ;test OS 2
 blt.s   _ReplyStartup                   ;->if not OS 2

 jsr     _LVOCreateMsgPort(a6)           ;create MsgPort
 move.l  d0,d6                           ;and save
 beq.s   _ReplyStartup                   ;->if error

 movea.l d0,a0                           ;MsgPort to a0
 moveq   #IOSTD_SIZE,d0                  ;structure size
 jsr     _LVOCreateIORequest(a6)         ;get IORequest
 move.l  d0,d5                           ;and save
 beq.s   _delport                        ;->if error

 moveq   #3,d4                           ;4 drives
_NoClickLoop

 lea     _tdname(pc),a0                  ;DeviceName
 move.l  d4,d0                           ;drive number
 movea.l d5,a1                           ;IORequest
 moveq   #0,d1                           ;3.5" only
 jsr     _LVOOpenDevice(a6)              ;open
 tst.l   d0                              ;error test
 bne.s   _next                           ;->if error

 movea.l d5,a1                           ;IORequest
 movea.l IO_UNIT(a1),a0                  ;get UnitPort
 ori.b   #TDPF_NOCLICK,TDU_PUBFLAGS(a0)  ;save Flag
 jsr     _LVOCloseDevice(a6)             ;close device

_next
 dbra    d4,_NoClickLoop                 ;all drives

_delio
 movea.l d5,a0                           ;IORequest to a0
 jsr     _LVODeleteIORequest(a6)         ;delete IORequest

_delport
 movea.l d6,a0                           ;port to a0
 jsr     _LVODeleteMsgPort(a6)           ;delete port

_ReplyStartup
 move.l  d7,d0                           ;WbStartup to d0
 beq.s   _fromCLI                        ;->if not there

 movea.l d0,a1                           ;WbStartup to a1
 jmp     _LVOReplyMsg(a6)                ;reply
```

```
;Return to program would lead to a crash. If necessary, turn
;multitasking off first (it will activate itself again after the
;program ends).

_fromCLI
 rts                                    ;end of program

_tdname
 dc.b     'trackdisk.device',0          ;DeviceName
```

Cache Control

The 68030 uses internal memory to store the last command and the last memory access during the execution of the command. This internal memory, called a cache, can greatly speed up processing. If the values that the processor needs are found in a cache, then no more RAM access is necessary, which with a non-multiplexed bus in a 32 bit architecture is rather time-consuming. Normally, the processor does not access the memory block containing the program code when executing a command. The separation of command and data caches can therefore speed things up greatly. Self-modifying code must be excluded from this, however, because the changes would be made in the data cache and not in the command cache. The Amiga's coprocessors, the DMA chips, are another problem. If one of these manipulates the memory, the contents of the caches do not change and the processor will be working with the wrong values. This could make it necessary to turn off the caches or delete them. Assembler programmers can use the CACR (CAche Control Register) and CAAR (CAche Address Register) to delete individual cache entries, but this is not in conformance with the operating system.

Another way of managing the caches is needed for developing high speed programs. The 68030 offers the ability to "freeze" the contents of its caches. The contents of a frozen cache cannot be changed, but they can be read. This allows you to freeze the cache of a frequently used subroutine after you have run it. General program processing is a little slower because of this, but the subroutine will be extremely fast the next time it is called.

The Exec takes care of managing and storing the contents of the CACR in our example:

```
**=====================================================**
**                 Turn off caches                     **
**=====================================================**

 ...
 movea.l $4.w,a6                           ;load ExecBase
 moveq   #0,d0                             ;new cache bits (value=0)
 move.l  #CACRF_EnableI!CACRF_EnableD,d1 ;mask
 jsr     _LVOCacheControl(a6)              ;save caches
 ...


**=====================================================**
**                 Activate caches                     **
**=====================================================**

 ...
 movea.l $4.w,a6                           ;load ExecBase
 move.l  #CACRF_EnableI!CACRF_EnableD,d0 ;new cache bits
 move.l  d0,d1                             ;mask
 jsr     _LVOCacheControl(a6)              ;activate caches
 ...


**=====================================================**
**                 Turn off caches                     **
**=====================================================**

 ...
 movea.l $4.w,a6                           ;load ExecBase
 moveq   #0,d0                             ;new cache bits (value=0)
 move.l  #CACRF_EnableI!CACRF_EnableD,d1 ;mask
 jsr     _LVOCacheControl(a6)              ;lock caches
 ...


**=====================================================**
**             Delete caches (User mode)               **
**=====================================================**

 ...
 movea.l $4.w,a6               ;load ExecBase
 moveq   #-1,d0                ;both caches
 jsr     _LVOCacheClearU(a6)   ;delete caches
 ...
```

```
**=======================================================**
**              Store subroutine in cache                **
**=======================================================**

 ...
 movea.l $4.w,a6                          ;load ExecBase
 jsr     _VeryWichtigHighTech             ;subroutine
 move.l  #CACRF_FreezeI!CACRF_FreezeD,d0 ;new cache bits
 move.l  d0,d1                            ;mask
 jsr     _LVOCacheControl(a6)             ;freeze caches
 ...

**=======================================================**
**                Free caches                            **
**=======================================================**

 ...
 movea.l $4.w,a6                          ;load ExecBase
 moveq   #0,d0                            ;new cache bits
 move.l  #CACRF_FreezeI!CACRF_FreezeD,d1 ;mask
 jsr     _LVOCacheControl(a6)             ;free caches
 ...
```

Another problem can arise using Burst mode. If the hardware is properly designed, the 68030 can move 16 bytes from cache to RAM (or RAM to cache) in only 5 clock cycles (= 2-1-1-1 burst). The data transfer is done in 16 byte steps and is based on modulo 16 addresses. This is a good reason for keeping your data well-organized, as the C structures of the operating system are. The speed in Burst mode is determined to a large extent by which memory chips are used. Dynamic Nibble mode RAM, as used in the ChipMem region, will only allow a 4-1-1-1 burst (7 clock cycles). Also, if the memory chips have added WaitStates during the last three longword accesses, this can slow down the processor even more, since each WaitState costs two clock cycles. But regardless of the speed, problems can still occur because of DMA accesses when the data is disorganized. The solution here involves CACRF_IBE and CACRF_DBE, which can be used to turn the Instruction burst and the Data burst on and off via CacheControl.

3.1.6 The Expansion Library

The Expansion library, called "expansion.library" with the OpenLibrary() function, manages hardware and software expansions and the configuration of the strap routines (for booting). As always, the base address must be passed in A6.

Functions of the Expansion Library

AddBootNode
AddConfigDev
AddDosNode
AllocConfigDev
AllocExpansionMem
FindConfigDev
FreeConfigDev
FreeExpansionMem
GetCurrentBinding
MakeDosNode
ObtainConfigBinding
ReleaseConfigBinding
RemConfigDev
SetCurrentBinding

Description of the Routines

AddBootNode	**Add a bootable device**

Call:

```
ok = AddBootNode( bootPri, flags, deviceNode, configDev )
D0   -36(A6)      D0       D1     A0          A1

BOOL   ok
BYTE   bootPri
ULONG  flags
STRUCT DeviceNode *deviceNode
STRUCT ConfigDev *configDev
```

Function: A logical AutoBoot device is added to the DOS list. If DOS does not exist yet, the data is stored in a buffer.

Parameters, Results:
See AddDosNode(), the only difference is that an AutoBoot requires a ConfigDev structure.

AddConfigDev — **Add a ConfigDev structure**

Call:

```
AddConfigDev( configDev )
-30(A6)       A0

STRUCT ConfigDev *configDev
```

Function: Adds the given ConfigDev structure to the system list.

Parameters: configDev Initialized ConfigDev structure

See also: RemConfigDev()

AddDosNode — **Mounts a data storage device**

Call:

```
ok = AddDosNode( bootPri, flags, deviceNode )
D0   -150(A6)    D0       D1     A0

BOOL   ok
BYTE   bootPri
ULONG  flags
STRUCT DeviceNode *deviceNode
```

Function: Adds a filesystem device to the system list. If DOS is not active yet, the information is stored in a buffer. If no handler is given, the new filesystem automatically takes over the management.

Parameters:

bootPri — AutoBoot priority (127 to -128). Only works if the corresponding ConfigDev structure is in the system list.

flags — ADNF_STARTPROC (bit 0) start handler immediately.

deviceNode — Initialized DOS device node.

Result: 0 Error

See also: MakeDosNode(), AddBootNode()

Example: Add a bootable drive to the system and activate a FileHandler:

```
 movea.l _ExpansionBase,a6
 lea     _Parms(pc),a0
 jsr     _LVOMakeDosNode(a6)
 tst.l   d0
 beq     _Error

 movea.l d0,a0
 moveq   #0,d0
 moveq   #ADNF_STARTPROC,d1
 jsr     _LVOAddDosNode(a6)
 ...

_DosNode
 dc.l    0

_Parms
 dc.l    _DOSname,_ExecName
 dc.l    1,0          ;Unit, Flags
 dc.l    16           ;Tablesize
 dc.l    128          ;Longwords per block
 dc.l    0,2          ;sector location, heads
 dc.l    1,11         ;sectors per block, blocks per track
 dc.l    2,0,0        ;boot blocks, unused, interleave
 dc.l    0,79         ;first and last cylinders
 dc.l    5,MEMF_CHIP  ;number of buffers, memory type
 dc.l    $7fffffff    ;maximum transfer rate
 dc.l    $fffffffe    ;mask
 dc.l    0            ;boot priority
 dc.b    'DOS',0      ;FileSystem type

_DOSname
 dc.b    'df1',0

_ExecName
 dc.b    'trackdisk.device',0
```

AllocConfigDev — **Allocate a ConfigDev structure**

Call:

```
configDev = AllocConfigDev()
D0          -48(A6)
```

Function: Allocates a deleted ConfigDev structure.

Result: ConfigDev structure or 0.

See also: FreeConfigDev()

AllocExpansionMem — **Allocate expansion memory**

Call:

```
startSlot = AllocExpansionMem( numSlots, slotOffset )
D0          -54(A6)            D0        D1
```

Function: Allocates the given number of slots.

Parameters:

numSlots   Number of slots required.

slotOffset   Memory location

Result: First slot number or -1.

See also: FreeExpansionMem()

FindConfigDev — **Find appropriate ConfigDev**

Call:

```
configDev = FindConfigDev( oldConfigDev, manufacturer, product )
D0          -72(A6)        A0            D0            D1

STRUCT ConfigDev *configDev,*oldConfigDev
LONG   manufacturer,product
```

Function: Finds a ConfigDev structure that fits the given description. In order to be able to test several ConfigDev structures, the previously retrieved ConfigDev can be specified. Values of -1 will accept every manufacturer code and every product ID.

Parameters: oldConfigDev

Last result or 0 (start of list)

manufacturer
Manufacturer's code or -1

product Product ID or -1

Result: The next appropriate ConfigDev structure or 0.

FreeConfigDev — Free a ConfigDev structure

Call:
```
FreeConfigDev( configDev )
-84(A6)        A0

STRUCT ConfigDev *configDev
```

Function: Frees a structure allocated with AllocConfigDev().

Parameters: configDev ConfigDev structure

See also: AllocConfigDev()

FreeExpansionMem — Free memory

Call:
```
FreeExpansionMem( startSlot, numSlots )
-90(A6)           D0         D1
```

Function: Frees memory allocated with AllocExpansionMem().

Parameters: Same as with AllocExpansionMem().

See also: AllocExpansionMem()

GetCurrentBinding — Gets a copy of CurrentBinding

Call:
```
actual = GetCurrentBinding( currentBinding, size )
D0       -138(A6)           A0              D0:16
```

Function: Copies the contents of the CurrentBinding structure to the given buffer.

Parameters: currentBinding
CurrentBinding structure

size Structure size

Result: The true size of the CurrentBinding structure.

See also: SetCurrentBinding()

MakeDosNode **Create a DosList entry**

Call:

```
deviceNode = MakeDosNode( parameterPkt )
D0           -144(A6)     A0

STRUCT DeviceNode *deviceNode
APTR   parameterPkt
```

Function: Creates all of the data structures required to add a device with AddDosNode().

Parameters: parameterPkt

Longword field with all the required information:

Device name (DOS, for example "df1"), device name (Exec, for example "trackdisk.device"), unit number, flags for OpenDevice(), number of following longwords, environment table for the FileHandler.

Result: Initialized structure or 0.

See also: AddDosNode()

Example: Create a DosNode for a 3.5" drive as "DF1:":

```
 movea.l _ExpansionBase,a6
 lea     _Parms(pc),a0
 jsr     _LVOMakeDosNode(a6)
 move.l  d0,_DosNode
 ...

_DosNode
 dc.l    0

_Parms
 dc.l    _DOSname,_ExecName
 dc.l    1,0          ;Unit, Flags
```

```
dc.l    16          ;Table size
dc.l    128         ;Longwords per block
dc.l    0,2         ;sector location, heads
dc.l    1,11        ;sectors per block, blocks per track
dc.l    2,0,0       ;boot blocks, unused, interleave
dc.l    0,79        ;first and last cylinders
dc.l    5,MEMF_CHIP ;number of buffers, memory type
dc.l    $7fffffff   ;maximum transfer rate
dc.l    $fffffffe   ;mask
dc.l    0           ;boot priority
dc.b    'DOS',0     ;FileSystem type

_DOSname
 dc.b   'df1',0

_ExecName
 dc.b   'trackdisk.device',0
```

ObtainConfigBinding — **Enable configuration binding**

Call: `ObtainConfigBinding()`
`-120(A6)`

Function: Obtains the approval to add drivers to ConfigDev structures.

See also: ReleaseConfigBinding()

ReleaseConfigBinding — **Release configuration binding**

Call: `ReleaseConfigBinding()`
`-126(A6)`

Function: Allows access by other programs.

See also: ObtainConfigBinding()

RemConfigDev — **Remove a ConfigDev from the system list**

Call: `RemConfigDev( configDev )`
`-108(A6)        A0`

Function: Removes the given ConfigDev structure from the system list.

Parameters: configDev ConfigDev structure

See also: AddConfigDev()

SetCurrentBinding **Set CurrentBinding**

Call:
```
SetCurrentBinding( currentBinding, size )
-132(A6)           A0              D0:16
```

Function: Copies the contents of the given buffer to the system's CurrentBinding structure.

Parameters: currentBinding
Buffer with the new contents for the CurrentBinding structure.

size Buffer size

See also: GetCurrentBinding()

```
ADNB_STARTPROC=0, ADNF_STARTPROC=1 ;start Handler immediately

Dec Hex STRUCTURE BootNode,LN_SIZE ;boot node
 14  $E UWORD bn_Flags             ;Flags
 16 $10 APTR  bn_DeviceNode        ;DosList
 20 $14 LABEL BootNode_SIZEOF

Dec Hex STRUCTURE ExpansionBase,LIB_SIZE ;library
 34 $22 UBYTE   eb_Flags                               ;readable
 35 $23 UBYTE   eb_Private01                           ;private
 36 $24 ULONG   eb_Private02                           ;private
 40 $28 ULONG   eb_Private03                           ;private
 44 $2C STRUCT  eb_Private04,CurrentBinding_SIZEOF ;private
 60 $3C STRUCT  eb_Private05,LH_SIZE                   ;private
 74 $3A STRUCT  eb_MountList,LH_SIZE ;BootNodes         private
 ... ;more private data...

EE_OK          =  0 ;no errors
EE_LASTBOARD   = 40 ;cannot be closed
EE_NOEXPANSION = 41 ;not enough memory
EE_NOMEMORY    = 42 ;no normal memory free
EE_NOBOARD     = 43 ;no board available
EE_BADMEM      = 44 ;defective memory
```

```
EBB_CLOGGED     = 0, EBF_CLOGGED     =  1 ;close error
EBB_SHORTMEM    = 1, EBF_SHORTMEM    =  2 ;less memory
EBB_BADMEM      = 2, EBF_BADMEM      =  4 ;defective memory
EBB_DOSFLAG     = 3, EBF_DOSFLAG     =  8 ;for AmigaDOS
EBB_KICKBACK33 = 4, EBF_KICKBACK33 = 16 ;OS change (DOS)
EBB_KICKBACK36 = 5, EBF_KICKBACK36 = 32 ;OS change (DOS)
```

3.1.7 The GadTools Library

The GadTools library, which uses the name "gadtools.library" for OpenLibrary(), is used to simplify the programming of gadgets, menus, and Intuition events. Previous operating system versions required many data structures to be created by hand. Now, an application can be made more user-friendly with just a few calls to the functions of the GadTools library.

GadTools Library Functions

CreateContext
CreateGadgetA
CreateMenusA
DrawBevelBoxA
FreeGadgets
FreeMenus
FreeVisualInfo
GetVisualInfoA
GT_BeginRefresh

GT_EndRefresh
GT_FilterIMsg
GT_GetIMsg
GT_PostFilterIMsg
GT_RefreshWindow
GT_ReplyIMsg
GT_SetGadgetAttrsA
LayoutMenuItemsA
LayoutMenusA

Description of the Functions

CreateContext — **Reserve a data block**

Call:

```
gad = CreateContext(glistpointer)
D0    -114(A6)      A0

STRUCT Gadget *gad,**glistpointer
```

Function: Reserves room for the context data. This function must be called before creating gadgets with the GadTools library.

Parameters: glistptr — Address of a longword ending in 0 where GadTools will store the address of the gadget being generated. The gadget address can then be given to Intuition later (AddGList() etc.).

Result: Address of a context gadget or 0.

CreateGadgetA — **Create a GadTools gadget**

Call:

```
gad = CreateGadgetA(kind, previous, newgad, taglist)
D0    -30(A6)       D0    A0        A1      A2

STRUCT Gadget *gad,*previous
ULONG  kind
STRUCT NewGadget *newgad
STRUCT TagItem *taglist
```

Function: Gets a gadget of the given type, initializes it as indicated by the tags and the NewGadget structure, and adds it to an existing gadget.

Parameters:

kind — Gadget type

previous — Gadget to which the new GG will be added.

newgad — NewGadget structure that describes the gadget.

taglist — TagItem field with special instructions.

Tags:

GT_Underscore (Char (starting with version 37)) defines the character for which the following character will be underlined in the gadget text (for example, to indicate the "hotkey" that will activate the gadget). If the "_" character is selected and the gadget text reads "_Color", then the gadget text will appear on screen with the "C" underlined.

GA_Disabled (BOOL) is used to turn off the gadget (TRUE). By default, the gadget is active.

GTCB_Checked (BOOL) is used to display a check mark (TRUE) in a Checkbox gadget. The default is no check mark.

GTCY_Labels (STRPTR *) sets the 0-terminated string address field for Cycle gadgets.

GTCY_Active (UWORD) sets the number (0...) of the active text for a Cycle gadget. The default string is 0.

GTIN_Number (ULONG) sets the contents (value) of an Integer gadget. The default value is 0.

GTIN_MaxChars (UWORD) sets the maximum number of decimal places for an Integer gadget. The default is 10.

STRINGA_ExitHelp (BOOL) (V37 and up) If TRUE, an Integer gadget can be ended by pressing the Help key. You will then get a GADGETUP with the RawKey code of the Help key ($5F).

GA_TabCycle (BOOL) (V37 and up) If TRUE, pressing [Tab] or [Shift]-[Tab] will activate the next or the previous gadget. The default is TRUE.

GTLV_Top (UWORD) sets the number of the first visible entry in a ListView gadget (scrollable list). The default is Entry 0.

GTLV_Labels (STRUCT List *) passes a list whose LN_NAME entries will appear in the ListView gadget (box with scrollable list).

GTLV_ReadOnly (BOOL) sets the read-only attribute for a ListView gadget (TRUE).

GTLV_ScrollWidth (UWORD) sets the width of the scroll bar. The default is 16 pixels.

GTLV_ShowSelected (STRUCT Gadget *) passes a String gadget, in which the selected entry can be edited, to a ListView gadget. If the value 0 is passed, the selected item is displayed below the ListView gadget.

GTLV_Selected (UWORD) sets the number of the pre-selected item in a ListView gadget. The default is -1, which means no item is pre-selected.

LAYOUTA_Spacing sets the number of lines between two items in a ListView gadget. The default is 0.

GTMX_Labels (STRPTR *) is a 0-terminated string address field containing the texts that will be displayed next to the selection buttons in a mutually exclusive selection table (MutualXclusive gadget).

GTMX_Active (UWORD) sets the active button number for an MX gadget. The default button is 0.

GTMX_Spacing (UWORD) sets the distance between two items in an MX gadget. The default is one line (1).

GTNM_Number (LONG) sets the value to be displayed as a decimal string in a non-revisable gadget (default: 0).

GTNM_Border (BOOL) displays a border (TRUE).

GTPA_Depth (UWORD) sets the number of bit-planes for a Palette gadget. The default is one bit-plane (2^1 colors).

GTPA_Color (UBYTE) sets the default for the selected color of a Palette gadget (otherwise 1 is used).

GTPA_ColorOffset (UBYTE) determines the number of the first color to be queried in a Palette gadget. The default is color 0.

GTPA_IndicatorWidth (UWORD) sets the width of the palette's color indicator if it is used.

GTPA_IndicatorHeight (UWORD) is the same for the height of the color indicator.

GTSC_Top (WORD) sets the start of a ScrollGadget (similar to the old PropGadget). The default is 0.

GTSC_Total (WORD) sets the number of available units (ScrollGadget, default: 0 units).

GTSC_Visible (WORD) sets how many units will be visible at once (ScrollGadget, default: 2 units from GTSC_Total).

GTSC_Arrows (UWORD) equips the ScrollGadget with arrow symbols. The value defines the height of the arrow and ScrollGadget for a horizontal gadget and the width of the arrow and Scroll Gadget for a vertical gadget.

PGA_Freedom is used to define a vertical ScrollGadget (LORIENT_VERT). The default is a horizontal ScrollGadget (LORIENT_HORIZ).

GA_Immediate (BOOL) causes every IDCMP_GADGETDOWN event to be passed (TRUE).

GA_RelVerify (BOOL) same for IDCMP_GADGETUP events.

GTSL_Min (WORD) sets the minimum value for a SliderGadget (default: 0).

GTSL_Max (WORD) is the same for the maximum value (default: 15).

GTSL_Level (WORD) sets a SliderGadget to a specified location (default 0).

GTSL_MaxLevelLen (UWORD) sets the maximum length of the string containing the location for the SliderGadget.

GTSL_LevelFormat (STRPTR) determines a format string for the 32 bit value indicating the location for the SliderGadget. The format string is formatted with the Exec routine RawDoFmt().

GTSL_LevelPlace determines where the position value will be output (PLACETEXT_LEFT (default), PLACETEXT_RIGHT, PLACETEXT_ABOVE, or PLACETEXT_BELOW).

GTSL_DispFunc (FPTR) associates a function with a SliderGadget. The function is passed the gadget address and position value on the stack. The slider position is calculated from this information and passed back as a longword in D0.

GTST_String (STRPTR) sets the string used to initialize the contents of a StringGadget (default: empty = 0).

GTST_MaxChars (UWORD) sets the maximum number of characters in a StringGadget buffer.

GTTX_Text (STRPTR) sets the contents of a TextGadget (default: empty=0).

GTTX_CopyText (BOOL) causes the TextGadget to make a copy of GTTX_Text (TRUE).

GTTX_Border (BOOL) makes a border for the TextGadget (TRUE).

Result: Address of a new gadget or 0.

CreateMenusA	**Create a GadTools menu**

Call:

```
menu = CreateMenusA(newmenu, taglist)
D0     -48(A6)        A0       A1

STRUCT Menu *menu
STRUCT NewMenu *newmenu
STRUCT TagItem *taglist
```

Function: Creates a complete MenuStrip according to the information in the NewMenu structure and the tags.

Parameters: newmenu List with initialized NewMenu structure.

taglist TagItem field

Tags: GTMN_FrontPen (UBYTE) text color (or else 0).

GTMN_FullMenu (BOOL (Version 37 and up)) indicates that the menu description of the NewMenu structure pertains to a complete MenuStrip (TRUE).

GTMN_SecondaryError (ULONG * (Version 37 and up)) passes the address of a long initialized to 0, to which an error code can be written:

GTMENU_INVALID
Invalid NewMenu structure (result 0).

GTMENU_TRIMMED
Too many items (some are trimmed).

GTMENU_NOMEM
Not enough memory.

Result: MenuStrip address, might not be complete (GTMENU_TRIMMED) or 0. MenuStrips are created without locations. Therefore, LayoutMenusA() or LayoutMenuItemsA() must be called before they are added.

DrawBevelBoxA — **Draw a box**

Call:
```
DrawBevelBoxA(rport, left, top, width, height, taglist)
-120(A6)      A0     D0    D1   D2     D3      A3

STRUCT RastPort *rport
WORD   left,top,width,height
STRUCT TagItem *taglist
```

Function: Draws a box in a RastPort.

Parameters:

rport	RastPort
left	Left edge of box
top	Top edge of box
width	Width of box
height	Height of box
taglist	TagItem field

Tags: GTBB_Recessed (BOOL) is used to draw a new box; otherwise the box is removed.

GT_VisualInfo (APTR) must be given with the result of a GetVisualInfoA() call.

FreeGadgets — **Free gadgets**

Call:
```
FreeGadgets(glist)
-36(A6)     A0

STRUCT Gadget *glist
```

Function: Frees all memory for a gadget list whose components were allocated with CreateGadgetA().

Parameters: glist — One or more linked gadget structures.

FreeMenus — **Free menus**

Call:
```
FreeMenus(menu)
-54(A6)   A0

STRUCT Menu *menu
```

Function: Free all memory for menus created with CreateMenusA().

Parameters: menu — Menu or MenuItem from CreateMenusA().

FreeVisualInfo — **Free VisualInfo**

Call:

```
FreeVisualInfo(vi)
-132(A6)       A0

APTR vi
```

Function: Frees memory and resources allocated with GetVisualInfoA(). This function may only be called after gadgets are used, for example, after a window is closed. It must be called before closing or unlocking a screen.

Parameters: vi — Result from GetVisualInfoA()

GetVisualInfoA — **Get information on the screen display**

Call:

```
vi = GetVisualInfoA(screen, taglist)
D0   -126(A6)        A0      A1

APTR vi
STRUCT Screen *screen
STRUCT TagItem *taglist
```

Function: Gets the information that the GadTools library needs to create the best possible gadgets or menus. After a window is closed for the last time, the result must be freed with FreeVisualInfo().

Parameters: screen — Screen where the window is to be opened.

taglist — TagItems field

Result: Address of a private data field.

GT_BeginRefresh — **BeginRefresh for GadTools windows**

Call:

```
GT_BeginRefresh(win)
-90(A6)         A0
```

```
STRUCT Window *win
```

Function: Executes the BeginRefresh() (known from Intuition) for windows with GadTools structures (GadTools works with NOCAREREFRESH windows).

Parameters: win — Window that receives an IDCMP_REFRESHWINDOW message.

GT_EndRefresh — **End refresh**

Call:

```
GT_EndRefresh(win, complete)
-96(A6)        A0   D0

STRUCT Window *win
BOOL   complete
```

Function: Ends a window refresh that was started with GT_BeginRefresh().

Parameters:

win	Window structure
complete	Flag: TRUE=refresh completed

GT_FilterIMsg — **Pass Intuition message to GadTools**

Call:

```
modimsg = GT_FilterIMsg(imsg)
D0         -102(A6)      A1

STRUCT IntuiMessage *modimsg,*imsg
```

Function: Passes an Intuition message to the GadTools library to assure proper control of GadTools gadgets.

Parameters: imsg — Normal IntuiMessage from a window UserPort.

Result: 0 if GadTools was not interested in the message; otherwise an IntuiMessage modified with information from GadTools.

GT_GetIMsg	**Get and process an IntuiMessage**

Call:

```
imsg = GT_GetIMsg(intuiport)
D0     -72(A6)    A0

STRUCT IntuiMessage *imsg
STRUCT MsgPort *intuiport
```

Functions, Results:
Similar to GT_FilterIMsg(), except that the message is first retrieved from the given port with GetMsg().

Parameters: intuiport UserPort for a window.

GT_PostFilterIMsg	**Restore an IntuiMessage**

Call:

```
imsg = GT_PostFilterIMsg(modimsg)
D0     -108(A6)          A1

STRUCT IntuiMessage *imsg,*modimsg
```

Function: Messages modified by GadTools must not be answered with ReplyMsg(). This function returns the original message and frees the memory used by the modified message.

Parameters: modimsg A GadTools message from GT_GetIMsg() or GT_FilterIMsg().

Result: The original IntuiMessage.

GT_RefreshWindow	**Refresh all GadTools gadgets**

Call:

```
GT_RefreshWindow(win, req)
-84(A6)          A0   A1

STRUCT Window *win
STRUCT Requester *req
```

Function: Calling intuition/RefreshGList() after intuition/AddGList() is not enough to properly display GadTools gadgets. This function must also be called. Afterwards, it is no longer needed.

Parameters: win — Window with GadTools gadgets.

req — Requester address (not yet supported: 0)

GT_ReplyIMsg — Reply to a GadTools message

Call:

```
GT_ReplyIMsg(imsg)
-78(A6)      A1

STRUCT IntuiMessage *imsg
```

Function: Replies to a message obtained with GT_GetIMsg().

Parameters: imsg — A modified IntuiMessage from GT_GetIMsg().

GT_SetGadgetAttrsA — Change attributes of a GadTools gadget

Call:

```
GT_SetGadgetAttrsA(gad, win, req, taglist)
-42(A6)             A0   A1   A2   A3

STRUCT Gadget *gad
STRUCT Window *win
STRUCT Requester *req
STRUCT TagItem *taglist
```

Function: Changes the attributes of a GadTools gadget according to the information in a TagItem field.

Parameters: gad — GadTools gadget

win — Window containing the gadget.

req — Requester for the gadget (not yet supported: 0).

taglist — TagItem field

Tags:

BUTTON-Gadget: GA_Disabled.
CHECKBOX-Gadget: GTCB_Checked, GA_Disabled.

CYCLE-Gadget:	GTCY_Active, GTCY_Labels, GA_Disabled.
INTEGER-Gadget:	GTIN_Number, GA_Disabled.
LISTVIEW-Gadget:	GTLV_Top, GTLV_Labels, GTLV_Selected.
MX-Gadget:	GTMX_Active.
NUMBER-Gadget:	GTNM_Number.
PALETTE-Gadget:	GTPA_Color, GA_Disabled.
SCROLLER-Gadget:	GTSC_Top, GTSC_Total, GTSC_Visible, GA_Disabled.
SLIDER-Gadget:	GTSL_Min, GTSL_Max, GTSL_Level, GA_Disabled.
STRING-Gadget:	GTST_String, GA_Disabled.
TEXT-Gadget:	GTTX_Text.

LayoutMenuItemsA **Enter positions for MenuItems**

Call:

```
success = LayoutMenuItemsA(menuitem, vi, taglist)
D0        -60(A6)          A0        A1  A2

BOOL   success
STRUCT MenuItem *menuitem
APTR   vi
STRUCT TagItem *taglist
```

Function: Enters the positions of MenuItems and SubItems.

Parameters: menuitem First MenuItem

vi Result of GetVisualInfoA().

taglist Description of the items.

Tags: GTMN_TextAttr (STRUCT TextAttr *) defines the font for MenuItems and SubItems. The font must be accessible with graphics/OpenFont().

GTMN_Menu (STRUCT Menu *) gives GadTools the address of the Menu structure of the given items (needed for calculations).

Result: 0 Error

LayoutMenusA **Enter position in MenuStrip**

Call:

```
success = LayoutMenusA(menu, vi, taglist)
D0          -66(A6)       A0    A1  A2

BOOL   success
STRUCT Menu *menu
APTR   vi
STRUCT TagItem *taglist
```

Function: Enters positions for an entire MenuStrip.

Parameters:

menu MenuStrip address from CreateMenusA().

vi Result from GetVisualInfoA().

taglist TagItem field

Tags: GTMN_TextAttr (STRUCT TextAttr *) see LayoutMenuItemsA().

Result: 0 Error

```
GENERIC_KIND    =  0 ;standard gadget
BUTTON_KIND     =  1
CHECKBOX_KIND   =  2
INTEGER_KIND    =  3
LISTVIEW_KIND   =  4
MX_KIND         =  5
NUMBER_KIND     =  6
CYCLE_KIND      =  7
PALETTE_KIND    =  8
SCROLLER_KIND   =  9
SLIDER_KIND     = 11
STRING_KIND     = 12
TEXT_KIND       = 13
NUM_KINDS       = 14 ;number of new gadgets

GADTOOLBIT      = $8000 ;GadTools gadget
GADTOOLMASK     = $7FFF ;user bits
```

Required IDCMP Flags:

```
ARROWIDCMP     = GADGETUP!GADGETDOWN!INTUITICKS!MOUSEBUTTONS
BUTTONIDCMP    = GADGETUP
CHECKBOXIDCMP  = GADGETUP
INTEGERIDCMP   = GADGETUP
LISTVIEWIDCMP  = GADGETUP!GADGETDOWN!MOUSEMOVE!ARROWIDCMP
MXIDCMP        = GADGETDOWN
NUMBERIDCMP    = 0
CYCLEIDCMP     = GADGETUP
PALETTEIDCMP   = GADGETUP
SCROLLERIDCMP  = GADGETUP!GADGETDOWN!MOUSEMOVE ;without arrows!
SLIDERIDCMP    = GADGETUP!GADGETDOWN!MOUSEMOVE
STRINGIDCMP    = GADGETUP
TEXTIDCMP      = 0
```

Spacing:

```
INTERWIDTH     = 8
INTERHEIGHT    = 4

Dec Hex STRUCTURE NewGadget,0
  0  $0 WORD   gng_LeftEdge     ;gadget position
  2  $2 WORD   gng_TopEdge
  4  $4 WORD   gng_Width        ;gadget size
  6  $6 WORD   gng_Height
  8  $8 APTR   gng_GadgetText   ;text
 12  $C APTR   gng_TextAttr     ;font for the text
 16 $10 UWORD  gng_GadgetID     ;ID
 18 $12 ULONG  gng_Flags        ;Flags
 22 $16 APTR   gng_VisualInfo   ;see GetVisualInfo()
 26 $1A APTR   gng_UserData     ;user data
 30 $1E LABEL  gng_SIZEOF

PLACETEXT_LEFT  = $0001 ;next to slider, left
PLACETEXT_RIGHT = $0002 ;right next to slider, right
PLACETEXT_ABOVE = $0004 ;above slider
PLACETEXT_BELOW = $0008 ;below slider
PLACETEXT_IN    = $0010 ;in the gadget
NG_HIGHLABEL    = $0020 ;highlight

Dec Hex STRUCTURE NewMenu,0
  0  $0 UBYTE  gnm_Type           ;type
  1  $1 UBYTE  gnm_Pad
  2  $2 APTR   gnm_Label          ;text
  6  $6 APTR   gnm_CommKey        ;character
 10  $A UWORD  gnm_Flags          ;structure
 12  $C LONG   gnm_MutualExclude  ;exclude
```

```
 16 $10 APTR   gnm_UserData       ;user data
 20 $14 LABEL  gnm_SIZEOF
```

UserData always comes after the normal structure, for example, as with mu_SIZEOF(Menu).

```
NM_TITLE = 1 ;Menu
NM_ITEM  = 2 ;MenuItem
NM_SUB   = 3 ;SubItem
NM_END   = 0 ;end of field

MENU_IMAGE = 128               ;Image flag
IM_ITEM    = NM_ITEM!MENU_IMAGE ;item with image
IM_SUB     = NM_SUB!MENU_IMAGE  ;SubItem with image

NM_BARLABEL = -1 ;dividing line

NM_MENUDISABLED = MENUENABLED
NM_ITEMDISABLED = ITEMENABLED

NM_FLAGMASK = ~(COMMSEQ!ITEMTEXT!HIGHFLAGS)

GT_TagBase       = TAG_USER+$80000   ;first Tag
GTVI_NewWindow   = GT_TagBase+$01    ;NewWindow for VisualInfo
GTVI_NWTags      = GT_TagBase+$02    ;NewWindow Tags
GT_Private0      = GT_TagBase+$03    ;private
GTCB_Checked     = GT_TagBase+$04    ;checkbox status
GTLV_Top         = GT_TagBase+$05    ;top of ListView
GTLV_Labels      = GT_TagBase+$06    ;ListView contents
GTLV_ReadOnly    = GT_TagBase+$07    ;ListView type
GTLV_ScrollWidth= GT_TagBase+$08     ;ListView scroller width
GTMX_Labels      = GT_TagBase+$09    ;MX contents
GTMX_Active      = GT_TagBase+$0A    ;MX prefix
GTTX_Text        = GT_TagBase+$0B    ;text
GTTX_CopyText    = GT_TagBase+$0C    ;copy text
GTNM_Number      = GT_TagBase+$0D    ;number value
GTCY_Labels      = GT_TagBase+$0E    ;cycle contents
GTCY_Active      = GT_TagBase+$0F    ;cycle prefix
GTPA_Depth       = GT_TagBase+$10    ;palette bit planes
GTPA_Color       = GT_TagBase+$11    ;palette prefix
GTPA_ColorOffset= GT_TagBase+$12     ;palette start
GTPA_IndicatorWidth =GT_TagBase+$13  ;palette indicator width
GTPA_IndicatorHeight=GT_TagBase+$14  ;palette indicator height
GTSC_Top         = GT_TagBase+$15    ;top of scroller
GTSC_Total       = GT_TagBase+$16    ;total contents of scroller
GTSC_Visible     = GT_TagBase+$17    ;scroller contents
GTSC_Overlap     = GT_TagBase+$18    ;not used
```

```
GTSL_Min          = GT_TagBase+$26    ;slider minimum
GTSL_Max          = GT_TagBase+$27    ;slider maximum
GTSL_Level        = GT_TagBase+$28    ;slider position
GTSL_MaxLevelLen= GT_TagBase+$29      ;slider text length
GTSL_LevelFormat= GT_TagBase+$2A      ;slider format string
GTSL_LevelPlace = GT_TagBase+$2B      ;slider text position
GTSL_DispFunc     = GT_TagBase+$2C    ;slider function
GTST_String       = GT_TagBase+$2D    ;string contents
GTST_MaxChars     = GT_TagBase+$2E    ;string text length
GTIN_Number       = GT_TagBase+$2F    ;integer value
GTIN_MaxChars     = GT_TagBase+$30    ;integer text length
GTMN_TextAttr     = GT_TagBase+$31    ;MenuItem font
GTMN_FrontPen     = GT_TagBase+$32    ;MenuItem text color
GTBB_Recessed     = GT_TagBase+$33    ;BevelBox recessed
GT_VisualInfo     = GT_TagBase+$34    ;VisualInfo
GTLV_ShowSelected=GT_TagBase+$35      ;ListView display
GTLV_Selected     = GT_TagBase+$36    ;ListView prefix
GT_Reserved0      = GT_TagBase+$37    ;reserved
GT_Reserved1      = GT_TagBase+$38    ;reserved
GTTX_Border       = GT_TagBase+$39    ;text border
GTNM_Border       = GT_TagBase+$3A    ;number border
GTSC_Arrows       = GT_TagBase+$3B    ;scroller arrows
GTMN_Menu         = GT_TagBase+$3C    ;menu address
GTMX_Spacing      = GT_TagBase+$3D    ;MX spacing
```

Example

Create gadgets. In some cases, using GadTools can be more difficult than creating the gadgets yourself. But your efforts will be rewarded with gadgets that have the new, professional-looking standard appearance. Also, you won't have to program the query routines for the new gadget types yourself.

```
 ...
 bsr      _CreateGadgets
 beq      _NoGadgets

 movea.l _Window,a0
 movea.l _GadgetListe(pc),a1
 moveq    #-1,d0
 moveq    #-1,d1
 lea      $0.w,a2
 movea.l _IntuiBase,a6
 jsr      _LVOAddGList(a6)

 movea.l _GadgetListe(pc),a0
 movea.l _Window,a1
```

```
 moveq    #-1,d0
 jsr      _LVORefreshGList(a6)

 movea.l _Window,a0
 movea.l a2,a1
 movea.l _GadToolsBase,a6
 jsr      _LVOGT_RefreshWindow(a6)

 ...


_CreateGadgets
 movea.l _GadToolsBase,a6
 movea.l _MyScreen,a0
 lea      _DummyTags,a1
 jsr      _LVOGetVisualInfoA(a6)
 move.l   d0,_VisualInfo
 beq      _Zerror1

 lea      _Gadgetliste(pc),a0
 jsr      _LVOCreateContext(a6)
 tst.l    d0
 beq      _Zerror2

 movea.l d0,a0
 moveq    #CYCLE_KIND,d0
 lea      _NewGadget(pc),a1
 lea      _TagList(pc),a2
 jsr      _LVOCreateGadgetA(a6)
 tst.l    d0
 beq      _Zerror3

 ...

 rts

_Zerror3
 movea.l _GadgetListe(pc),a0
 jsr      _LVOFreeGadgets(a6)
_Zerror2
 movea.l _VisualInfo,a0
 jsr      _FreeVisualInfo(a6)
_Zerror1
 moveq    #0,d0
 rts

_DummyTags
 dc.l TAG_DONE
```

```
_Gadgetliste
 dc.l 0

_NewGadget
 dc.w 10,10,80,12
 dc.l 0,_Topaz8
 dc.w 1
 dc.l 0
_VisualInfo
 dc.l 0,0

_Topaz8
 dc.l _TopazName
 dc.w 8
 dc.b 0,0

_TagList
 dc.l GTCY_Labels,_Strings
 dc.l TAG_DONE

_Strings
 dc.l _Text0,_Text1,_Text2,_Text3,0

_TopazName
 dc.b 'topaz.font',0

_Text0
 dc.b 'DF0:',0

_Text1
 dc.b 'DF1:',0

_Text2
 dc.b 'DF2:',0

_Text3
 dc.b 'DF3:',0
```

3.1.8 The Graphics Library

Programmers often refer to "graphics.library" (its proper name for the OpenDevice() function) as the Gfx library. Gfx is responsible for all display and graphics operations. This library is used to program the blitter and the copper which control the video hardware. These routines are used for such operations as drawing, text output, and displaying movable objects. The base address must always be passed in A6.

Functions of the Gfx Library

1. The Video Hardware

CBump
CloseMonitor
CMove
CWait
FindDisplayInfo
FreeCopList
FreeCprList
FreeVPortCopLists
GetDisplayInfoData
GetVPModeID
LoadRGB4
LoadView
MakeVPort
ModeNotAvailable
MrgCop
NextDisplayInfo
OpenMonitor
ScrollVPort
SetRGB4
VBeamPos
VideoControl
WaitBOVP
WaitTOF

2. General Blitter Control

BitMapScale
BltBitMap
BltBitMapRastPort
BltClear
BltMaskBitMapRast
Port
BltPattern
BltTemplate
ClipBlit
CopySBitMap
DisOwnBlitter
OwnBlitter
QBlit
QBSBlit
ScalerDiv
ScrollRaster
SyncSBitMap
WaitBlit

3. Refresh Functions

AndRectRegion
AndRegionRegion
ClearRectRegion
ClearRegion
DisposeRegion
NewRegion
OrRectRegion
OrRegionRegion
XorRectRegion
XorRegionRegion

4. Data Structures

AllocRaster
AttemptLockLayerRom
FreeColorMap
FreeRaster
GetColorMap
GetRGB4
InitBitMap
InitRastPort
InitTmpRas
InitView
InitVPort
LockLayerRom

SetRGB4CM
UnlockLayerRom

5. Draw Functions

AreaDraw
AreaEllipse
AreaEnd
AreaMove
Draw
DrawEllipse
EraseRect
Flood
InitArea
Move
PolyDraw
ReadPixel
ReadPixelArray8
ReadPixelLine8
RectFill
SetAPen
SetBPen
SetDrMd
SetRast
WritePixel
WritePixelArray8
WritePixelLine8

6. Text Output

AddFont
AskFont
AskSoftStyle
ClearEOL
ClearScreen
CloseFont
ExtendFont
FontExtent
OpenFont
RemFont
SetFont
SetSoftStyle
StripFont
Text
TextExtent
TextFit
TextLength
WeighTAMatch

7. Movable Objects

AddAnimOb
AddBob
AddVSprite
Animate
ChangeSprite
DoCollision
DrawGList
FreeGBuffers
FreeSprite
GetGBuffers
GetSprite
InitGels
InitGMasks
InitMasks
MoveSprite
RemIBob
RemVSprite
SetCollision
SortGList

Description of Functions

1. The Video Hardware

CBump	**UCopList pointer to the next instruction**

Call:

```
CBump( c )
-366(A6) a1

STRUCT UCopList *c
```

Function: Sets the command pointer of a user Copper list to the next command.

Parameters: c — Address of a UCopList structure.

CloseMonitor	**Close MonitorSpec**

Call:

```
error = CloseMonitor( monitor_spec )
d0      -720(A6)      a0

LONG error
STRUCT MonitorSpec *monitor_spec
```

Function: Closes the given MonitorSpec.

Parameters: monitor_spec
MonitorSpec address from OpenMonitor().

Result: 0 — MonitorSpec closed

CMove	**Write a Copper move instruction to the UCopList**

Call:

```
CMove( c , a , v )
-372(A6)a1 d0 d1

STRUCT UCopList *c
SHORT  a,v
```

Function: Writes a Copper move command to a user Copper list without changing the edit pointer.

Parameters:	c	UCopList structure
	a	Hardware register offset from $DFF000.
	v	Word to which the register will be written.

CWait — Enter Copper wait in UCopList

Call:

```
CWait( c , v , h )
-378(A6) a1 d0 d1

STRUCT UCopList *c
SHORT  v,h
```

Function: Writes a Copper wait command to the user Copper list without changing the edit position.

Parameters:	c	UCopList structure
	v	Vertical wait position (end = 10000).
	h	Horizontal wait position (end = 255).

FindDisplayInfo — Get info on the display mode

Call:

```
handle = FindDisplayInfo( ID )
d0       -726(A6)         d0

ULONG  ID
LONG   handle
```

Function: Finds the information structure for a given display mode.

Parameters:	ID	32 bit display mode (monitor specific ViewMode).

Result: Handle to DisplayInfoRecord or 0.

FreeCopList — Free Copper list buffer

Call:

```
FreeCopList(coplist)
-546(A6)    a0
```

```
STRUCT CopList *coplist
```

Function: Frees the memory used by a Copper list.

Parameters: coplist   CopList structure

FreeCprList — **Free a hardware Copper list**

Call:

```
FreeCprList(cprlist)
-564(A6)    a0

STRUCT cprlist *cprlist
```

Function: Frees the memory of a hardware Copper list.

Parameters: cprlist   cprlist structure

FreeVPortCopLists — **Free ViewPort Copper lists**

Call:

```
FreeVPortCopLists(vp)
-540(A6)          a0

STRUCT ViewPort *vp
```

Function: Frees the memory for all Copper lists of a ViewPort.

Parameters: vp   ViewPort

GetDisplayInfoData — **Get data associated with a display mode**

Call:

```
result = GetDisplayInfoData(handle, buf, size, tagID, ID)
d0       -756(A6)           a0      a1   d0    d1     d2

LONG  handle
APTR  buf
ULONG size,tagID,ID,result
```

Function: Fills a buffer with data associated with a display mode.

Parameters: handle   DisplayInfo handle of the display mode.

buf	Buffer to be filled	
size	Buffer size	
tagID	Desired data type:	
	DTAG_DISP	DisplayInfo structure
	DTAG_DIMS	DimensionInfo structure
	DTAG_MNTR	MonitorInfo structure
	DTAG_NAME	Display mode name
ID	32 bit display mode (if handle=0).	

Result: Number of bytes in buffer, 0 (unknown mode or error).

GetVPModeID — **Get a monitor specific display mode**

Call:

```
modeID = GetVPModeID( vp )
d0       -792(A6)     a0

STRUCT ViewPort *vp
ULONG  modeID
```

Function: Retrieves the monitor specific 32 bit display mode of a ViewPort.

Parameters: vp — ViewPort structure

Result: ModeID or INVALID_ID

LoadRGB4 — **Set color table**

Call:

```
LoadRGB4( vp, colors , count )
-192(A6)  a0  a1       d0:16

STRUCT ViewPort *vp
APTR   colors
SHORT  count
```

Function: Loads the 3x4 bit RGB color values from a table to the ColorMap of the ViewPort, recalculates the Copper lists, and controls the video hardware.

Parameters: vp — ViewPort whose colors are to be changed.

colors — Word array with color values ($0RGB).

count — Number of colors (including 0).

LoadView — Activate a Copper list

Call:

```
LoadView( View )
-222(A6)  A1

STRUCT View *View
```

Function: Activates the Copper list of a view (available after MakeView(), MrgCop()) until the next call. Many programs, handlers (Intuition, Workbench...) and operating system routines call this function, so a good knowledge of the system is required for error-free programming.

Parameters: View — View structure with Copper lists or 0 (screen off but DMA still running, as, for example, with the sprite DMA).

MakeVPort — Assemble the Copper lists of a ViewPort

Call:

```
MakeVPort( view, viewport )
-216(A6)   a0    a1

STRUCT View *view
STRUCT ViewPort *viewport
```

Function: Derives the Copper lists of a ViewPort.

Parameters: view — View structure of the ViewPorts.

viewport — ViewPort structure with RasInfo.

ModeNotAvailable — Checks on availability of a display mode

Call:

```
error = ModeNotAvailable( modeID )
d0      -798(A6)          d0
```

```
ULONG modeID,error
```

Function: Checks for the availability of a monitor specific 32 bit display mode.

Parameters: modeID 32 bit display mode

Result: Error code that describes why the mode is not available, or 0 if the system does not have a reason why this mode can't be used.

MrgCop **Merge Copper lists**

Call:

```
MrgCop( View )
-210(A6) A1

STRUCT View *View
```

Function: Merges all partial Copper lists into a proper Copper list.

Parameters: View View structure with partial Copper lists.

NextDisplayInfo **Read through list of display modes**

Call:

```
next_ID = NextDisplayInfo( last_ID )
d0        -732(A6)         d0

ULONG last_ID,next_ID
```

Function: Gets the next available monitor specific display mode.

Parameters: last_ID Result of the last call or INVALID_ID for the start of the list.

Result: 32 bit display mode or INVALID_ID (no more modes).

OpenMonitor **Open MonitorSpec**

Call:

```
mspc = OpenMonitor( monitor_name , display_id)
d0     -714(A6)     a1             d0

APTR   monitor_name
```

```
ULONG  display_id
STRUCT MonitorSpec *mspc
```

Function: Opens a MonitorSpec which is given the monitor name or the 32 bit ID. If both parameters are 0, then the default monitor is returned.

Parameters: monitor_name
Monitor name or 0.

display_id 32 bit display mode or 0.

Result: MonitorSpec structure or 0.

ScrollVPort **Scroll ViewPort contents**

Call:

```
ScrollVPort( vp )
-588(A6)     a0

STRUCT ViewPort *vp
```

Function: Called after changing the RasInfo offsets and BitMap pointer to recalculate the Copper lists. Warning: high level languages are too slow.

Parameters: vp Visible ViewPort

SetRGB4 **Change colors**

Call:

```
SetRGB4( vp, n, r,   g,   b)
-288(A6) a0  d0 d1:4 d2:4 d3:4

STRUCT ViewPort *vp
SHORT  n
UBYTE  r,g,b
```

Function: Sets the color intensity of a color register, recalculates the Copper list, which controls the hardware.

Parameters: vp ViewPort

n Color number (0...31)

r	Red intensity (0...15)
g	Green intensity (0...15)
b	Blue intensity (0...15)

VBeamPos **Get the vertical beam position**

Call:

```
pos = VBeamPos()
d0    -384(A6)

LONG pos
```

Function: Gets the position of the monitor's vertical beam.

Result: Vertical beam position (0...511). The uncertainty is extremely high and is only acceptable for a task with the highest priority.

VideoControl **Change the color operations of a ViewPort**

Call:

```
error = VideoControl( cm , tags )
d0      -708(A6)      a0   a1

LONG error
STRUCT ColorMap *cm
STRUCT TagItem  *tags
```

Function: Change the operation of a ViewPort's ColorMap according to the commands in a TagItem field.

Parameters:

cm	ColorMap, result of GetColorMap().
tags	TagItem field

Tags: VTAG_ATTACH_CM_.. get the ViewPort of the ColorMap (..GET), set (..SET).

VTAG_VIEWPORTEXTRA_.. get vp_extra (..GET), set (..SET).

VTAG_NORMAL_DISP_.. get DisplayInfoHandle in normal mode (..GET), set (..SET).

VTAG_COERCE_DISP_.. same for coerced mode (..GET, ..SET).

VTAG_BORDERBLANK_.. Genlock: set border blanking (..SET), clear (..CLR), get (..GET).

VTAG_BORDERNOTRANS_.. set no-transparency in the border region (..SET), clear (..CLR), get (..GET).

VTAG_CHROMAKEY_.. set Chroma mode (..SET), clear (..CLR), get (..GET).

VTAG_BITPLANEKEY_.. set BitPlane mode (..SET), clear (..CLR), get (..GET).

VTAG_CHROMA_PEN_.. set Chroma color number (..SET), clear (..CLR), get (..GET).

VTAG_CHROMA_PLANE_.. set BitPlane number (..SET), get (..GET).

VTAG_NEXTBUF_CM next command list.

VTAG_END_CM last command.

Result: 0 Okay, followed by adding the next MakeVPort().

WaitBOVP	**Wait until a ViewPort is scanned**

Call:

```
WaitBOVP( vp )
-402(A6)  a0

STRUCT ViewPort *vp
```

Function: Waits until the monitor beam has displayed the last visible line of the given ViewPort.

Parameters: vp ViewPort

WaitTOF **Wait for vertical scan interrupts**

Call: `WaitTOF()`
`-270(A6)`

Function: Waits for the monitor's next vertical scan and for all VertB interrupt routines to be processed (turning off tasks, signal through VertB handler).

Pseudo Opcodes for lists:

```
COPPER_MOVE =     0 ;Pseudo Opcode for MOVE #...,...
COPPER_WAIT =     1 ;Pseudo Opcode for WAIT ...,...
CPRNXTBUF   =     2 ;end of buffer
CPR_NT_LOF  = $8000 ;command for Shortframes
CPR_NT_SHT  = $4000 ;command for Longframes (2. Interlace)
CPR_NT_SYS  = $2000 ;User command

Dec Hex STRUCTURE CopIns,0   ;Copper Pseudo Opcode
  0  $0 WORD   ci_OpCode     ;0 = move, 1 = wait
  2  $2 STRUCT ci_nxtlist,0  ;address of the next buffer
  2  $2 STRUCT ci_VWaitPos,0 ;or wait position
  2  $2 STRUCT ci_DestAddr,2 ;or destination address
  4  $4 STRUCT ci_HWaitPos,0 ;2. partial wait position
  4  $4 STRUCT ci_DestData,2 ;or value
  6  $6 LABEL  ci_SIZEOF

Dec Hex STRUCTURE cprlist,0 ;management of true Copper lists
  0  $0 APTR  crl_Next      ;address
  4  $4 APTR  crl_start     ;start
  8  $8 WORD  crl_MaxCount  ;length
 10  $A LABEL crl_SIZEOF

Dec Hex STRUCTURE CopList,0 ;management of temporary lists
  0  $0 APTR  cl_Next       ;next structure
  4  $4 APTR  cl__CopList   ;private
  8  $8 APTR  cl__ViewPort  ;private
 12  $C APTR  cl_CopIns     ;start of block
 16 $10 APTR  cl_CopPtr     ;command address
 20 $14 APTR  cl_CopLStart  ;LongFrame address from MrgCop()
 24 $18 APTR  cl_CopSStart  ;ShortFrame address from MrgCop()
 28 $1C WORD  cl_Count      ;counter
 30 $1E WORD  cl_MaxCount   ;block length in Pseudo Opcodes
 32 $20 WORD  cl_DyOffset   ;vertical start position
```

```
 34 $22 LABEL cl_SIZEOF

Dec Hex STRUCTURE UCopList,0    ;User Copper list
  0  $0 APTR   ucl_Next         ;next list
  4  $4 APTR   ucl_FirstCopList ;first node
  8  $8 APTR   ucl_CopList      ;current node
 12  $C LABEL  ucl_SIZEOF

MODE_640    = $8000 ;HiRes
PLNCNTMSK   =     7 ;bplcon0 bit plane mask
PLNCNTSHFT  =    12 ;bplcon0 bit plane bit
PF2PRI      =   $40 ;Playfield 2 priority
COLORON     =  $200 ;suppress Color Burst
DBLPF       =  $400 ;DualPlayfield mode
HOLDNMODIFY=  $800 ;Hold-And-Modify mode
INTERLACE   =     4 ;Interlace mode

PFA_FINE_SCROLL        =  15 ;Softscrolling planes 0,2,4
PFB_FINE_SCROLL_SHIFT =   4 ;bit position for planes 1,3,5
PF_FINE_SCROLL_MASK    =  15 ;Softscrolling planes 1,3,5

DIW_HORIZ_POS        =  $7F ;horizontal mask
DIW_VRTCL_POS        = $1FF ;vertical mask
DIW_VRTCL_POS_SHIFT =    7 ;bit position
DFTCH_MASK           =  $FF ;data fetching mask
VPOSRLOF             =$8000 ;LongFrame Flag vpos

DTAG_DISP = $80000000 ;Display Tags
DTAG_DIMS = $80001000
DTAG_MNTR = $80002000
DTAG_NAME = $80003000

Dec Hex STRUCTURE QueryHeader,0
  0  $0 ULONG  qh_StructID  ;Display Tag ID
  4  $4 ULONG  qh_DisplayID ;32 bit mode
  8  $8 ULONG  qh_SkipID    ;TAG_SKIP
 12  $C ULONG  qh_Length    ;length in 8 byte segments
 16 $10 LABEL  qh_SIZEOF

Dec Hex STRUCTURE DisplayInfo,qh_SIZEOF
 16 $10 UWORD  dis_NotAvailable                   ;Flag: 0=available
 18 $12 ULONG  dis_PropertyFlags                  ;characteristics
 22 $16 STRUCT dis_Resolution,tpt_SIZEOF          ;pixel resolution X/Y
 26 $1A UWORD  dis_PixelSpeed                     ;nanoseconds per pixel
 28 $1C UWORD  dis_NumStdSprites                  ;number of sprites
 30 $1E UWORD  dis_PaletteRange                   ;available colors
 32 $20 STRUCT dis_SpriteResolution,tpt_SIZEOF ;sprite resolution
 36 $24 STRUCT dis_pad,4
```

```
 40 $28 STRUCT dis_reserved,8
 48 $30 LABEL  dis_SIZEOF

DI_AVAIL_NOCHIPS         = 1
DI_AVAIL_NOMONITOR       = 2
DI_AVAIL_NOTWITHGENLOCK  = 4
DIPF_IS_LACE             = $00000001
DIPF_IS_DUALPF           = $00000002
DIPF_IS_PF2PRI           = $00000004
DIPF_IS_HAM              = $00000008
DIPF_IS_ECS              = $00000010
DIPF_IS_PAL              = $00000020
DIPF_IS_SPRITES          = $00000040
DIPF_IS_GENLOCK          = $00000080
DIPF_IS_WB               = $00000100
DIPF_IS_DRAGGABLE        = $00000200
DIPF_IS_PANELLED         = $00000400
DIPF_IS_BEAMSYNC         = $00000800
DIPF_IS_EXTRAHALFBRITE   = $00001000

Dec Hex STRUCTURE DimensionInfo,qh_SIZEOF
 16 $10 UWORD  dim_MaxDepth               ;number of bit planes
 18 $12 UWORD  dim_MinRasterWidth         ;minimum width
 20 $14 UWORD  dim_MinRasterHeight        ;minimum height
 22 $16 UWORD  dim_MaxRasterWidth         ;maximum width
 24 $18 UWORD  dim_MaxRasterHeight        ;maximum height
 26 $1A STRUCT dim_Nominal,ra_SIZEOF      ;standard dimensions
 34 $22 STRUCT dim_MaxOScan,ra_SIZEOF     ;maximum OverScan
 42 $2A STRUCT dim_VideoOScan,ra_SIZEOF   ;video OverScan
 50 $32 STRUCT dim_TxtOScan,ra_SIZEOF     ;text OverScan Prefs
 58 $3A STRUCT dim_StdOScan,ra_SIZEOF     ;standard OverScan Prefs
 66 $42 STRUCT dim_pad,14
 80 $50 STRUCT dim_reserved,8
 88 $58 LABEL  dim_SIZEOF

Dec Hex STRUCTURE MonitorInfo,qh_SIZEOF
 16 $10 APTR   mtr_Mspc                           ;MonitorSpec
 20 $14 STRUCT mtr_ViewPosition,tpt_SIZEOF        ;Prefs
 24 $18 STRUCT mtr_ViewResolution,tpt_SIZEOF      ;resolution
 28 $1C STRUCT mtr_ViewPositionRange,ra_SIZEOF    ;range
 36 $24 UWORD  mtr_TotalRows                      ;number of rows
 38 $26 UWORD  mtr_TotalColorClocks               ;width in 1/280ns
 40 $28 UWORD  mtr_MinRow                         ;minimum height
 42 $2A WORD   mtr_Compatibility                  ;compatibility
 44 $2C STRUCT mtr_pad,36
 80 $50 STRUCT mtr_reserved,8
 88 $58 LABEL  mtr_SIZEOF
```

```
MCOMPAT_MIXED  =  0 ;mixed display allowed
MCOMPAT_SELF   =  1 ;with this monitor type only
MCOMPAT_NOBODY = -1 ;only on ViewPort allowed
DISPLAYNAMELEN = 32 ;length of display name

Dec Hex STRUCTURE NameInfo,qh_SIZEOF
 16 $10 STRUCT nif_Name,DISPLAYNAMELEN ;name
 48 $30 STRUCT nif_reserved,8
 56 $38 LABEL  nif_SIZEOF

INVALID_ID                 =        -1
MONITOR_ID_MASK            = $FFFF1000
DEFAULT_MONITOR_ID         = $00000000
NTSC_MONITOR_ID            = $00011000
PAL_MONITOR_ID             = $00021000

LORES_KEY                  = $00000000 ;LoRes
HIRES_KEY                  = $00008000 ;HiRes
SUPER_KEY                  = $00008020 ;SuperHiRes
HAM_KEY                    = $00000800 ;HoldAndModify
LORESLACE_KEY              = $00000004 ;Interlace
HIRESLACE_KEY              = $00008004 ;HiRes-Interlace
SUPERLACE_KEY              = $00008024 ;SuperHiRes-Interlace
HAMLACE_KEY                = $00000804 ;HAM-Interlace
LORESDPF_KEY               = $00000400 ;DualPlayfield
HIRESDPF_KEY               = $00008400 ;HiRes-DblPf
SUPERDPF_KEY               = $00008420 ;SuperHiRes-DblPf
LORESLACEDPF_KEY           = $00000404 ;Interlace-DblPf
HIRESLACEDPF_KEY           = $00008404 ;HiRes-Interlace-DblPf
SUPERLACEDPF_KEY           = $00008424 ;SuperHiRes-ILace-DblPf
LORESDPF2_KEY              = $00000440 ;DualPlayfield2
HIRESDPF2_KEY              = $00008440 ;HiRes-DblPf2
SUPERDPF2_KEY              = $00008460 ;SuperHiRes-DblPf2
LORESLACEDPF2_KEY          = $00000444 ;Interlace-DblPf2
HIRESLACEDPF2_KEY          = $00008444 ;HiRes-Interlace-DblPf2
SUPERLACEDPF2_KEY          = $00008464 ;SuperHRes-ILace-DblPf2
EXTRAHALFBRITE_KEY         = $00000080 ;ExtraHalfbrite
EXTRAHALFBRITELACE_KEY     = $00000084 ;ExtraHalfbrite-ILace

VGA_MONITOR_ID             = $00031000 ;VGA monitor
VGAEXTRALORES_KEY          = $00031004 ;ExtraLoRes
VGALORES_KEY               = $00039004 ;LoRes
VGAPRODUCT_KEY             = $00039024 ;Productivity
VGAHAM_KEY                 = $00031804 ;HAM
VGAEXTRALORESLACE_KEY      = $00031005 ;ExtraLoRes-ILace
VGALORESLACE_KEY           = $00039005 ;Interlace
VGAPRODUCTLACE_KEY         = $00039025 ;Productivity-ILace
VGAHAMLACE_KEY             = $00031805 ;HAM-Interlace
```

```
VGAEXTRALORESDPF_KEY     = $00031404 ;ExtraLoRes-DblPf
VGALORESDPF_KEY          = $00039404 ;DualPlayfield
VGAPRODUCTDPF_KEY        = $00039424 ;Productivity-DblPf
VGAEXTRALORESLACEDPF_KEY= $00031405 ;XLoRes-ILace-DblPf
VGALORESLACEDPF_KEY      = $00039405 ;Interlace-DblPf
VGAPRODUCTLACEDPF_KEY    = $00039425 ;Prod-ILace-DblPf
VGAEXTRALORESDPF2_KEY    = $00031444 ;XLoRes-DblPf2
VGALORESDPF2_KEY         = $00039444 ;DualPlayfield2
VGAPRODUCTDPF2_KEY       = $00039464 ;Productivity-DblPf2
VGAEXTRALORESLACEDPF2_KEY=$00031445 ;XLoRes-ILace-DblPf2
VGALORESLACEDPF2_KEY     = $00039445 ;Interlace-DblPf2
VGAPRODUCTLACEDPF2_KEY   = $00039465 ;Prod-ILace-DblPf2
VGAEXTRAHALFBRITE_KEY    = $00031084 ;ExtraHalfbrite
VGAEXTRAHALFBRITELACE_KEY=$00031085 ;EHB-Interlace

A2024_MONITOR_ID         = $00041000 ;monochrome monitor
A2024TENHERTZ_KEY        = $00041000 ;10 Hz mode
A2024FIFTEENHERTZ_KEY    = $00049000 ;15 Hz mode

PROTO_MONITOR_ID         = $00051000 ;prototype

Dec Hex STRUCTURE tPoint,0 ;resolution per point
  0  $0 WORD   tpt_x
  2  $2 WORD   tpt_y
  4  $4 LABEL  tpt_SIZEOF

Dec Hex STRUCTURE AnalogSignalInterval,0
  0  $0 UWORD asi_Start
  2  $2 UWORD asi_Stop
  4  $4 LABEL asi_SIZEOF

Dec Hex STRUCTURE SpecialMonitor,XLN_SIZE
 24 $18 UWORD  spm_Flags
 26 $1A APTR   spm_do_monitor
 30 $1E APTR   spm_reserved1
 34 $22 APTR   spm_reserved2
 38 $26 APTR   spm_reserved3
 42 $2A STRUCT spm_hblank,asi_SIZEOF
 46 $2E STRUCT spm_vblank,asi_SIZEOF
 50 $32 STRUCT spm_hsync,asi_SIZEOF
 54 $36 STRUCT spm_vsync,asi_SIZEOF
 58 $3A LABEL  spm_SIZEOF

Dec Hex STRUCTURE MonitorSpec,XLN_SIZE
 24 $18 UWORD  ms_Flags
 26 $1A LONG   ms_ratioh
 30 $1E LONG   ms_ratiov
 34 $22 UWORD  ms_total_rows
```

```
 36 $24 UWORD  ms_total_colorclocks
 38 $26 UWORD  ms_DeniseMaxDisplayColumn
 40 $28 UWORD  ms_BeamCon0
 42 $2A UWORD  ms_min_row
 44 $2C APTR   ms_Special
 48 $30 UWORD  ms_OpenCount
 50 $32 APTR   ms_transform
 54 $36 APTR   ms_translate
 58 $3A APTR   ms_scale
 62 $3E UWORD  ms_xoffset
 64 $40 UWORD  ms_yoffset
 66 $42 STRUCT ms_LegalView,ra_SIZEOF
 74 $4A APTR   ms_maxoscan
 78 $4E APTR   ms_videoscan
 82 $52 UWORD  ms_DeniseMinDisplayColumn
 84 $54 UWORD  ms_DisplayCompatible
 86 $56 STRUCT ms_DisplayInfoDataBase,LH_SIZE
100 $64 STRUCT ms_DIDBSemaphore,SS_SIZE
146 $92 ULONG  ms_reserved00
150 $96 ULONG  ms_reserved01
154 $9A LABEL  ms_SIZEOF

MSB_REQUEST_NTSC    = 0, MSF_REQUEST_NTSC    = 1
MSB_REQUEST_PAL     = 1, MSF_REQUEST_PAL     = 2
MSB_REQUEST_SPECIAL = 2, MSF_REQUEST_SPECIAL = 4
MSB_REQUEST_A2024   = 3, MSF_REQUEST_A2024   = 8

STANDARD_VIEW_X  = $81
STANDARD_VIEW_Y        = $2C

Dec  Hex STRUCTURE GfxBase,LIB_SIZE ;base structure
 34  $22 APTR    gb_ActiView        ;active View
 38  $26 APTR    gb_copinit         ;Copper start list
 42  $2A APTR    gb_cia             ;CIA
 46  $2E APTR    gb_blitter         ;Blitter
 50  $32 APTR    gb_LOFlist         ;current Copper list
 54  $36 APTR    gb_SHFlist         ;current Copper list
 58  $3A APTR    gb_blthd           ;bltnode
 62  $3E APTR    gb_blttl
 66  $42 APTR    gb_bsblthd
 70  $46 APTR    gb_bsblttl
 74  $4A STRUCT  gb_vbsrv,IS_SIZE
 96  $60 STRUCT  gb_timsrv,IS_SIZE
118  $76 STRUCT  gb_bltsrv,IS_SIZE
140  $8C STRUCT  gb_TextFonts,LH_SIZE
154  $9A APTR    gb_DefaultFont
158  $9E UWORD   gb_Modes      ;bltcon0
160  $A0 BYTE    gb_VBlank
```

```
161  $A1 BYTE    gb_Debug
162  $A2 UWORD   gb_BeamSync
164  $A4 WORD    gb_system_bplcon0
166  $A6 BYTE    gb_SpriteReserved
167  $A7 BYTE    gb_bytereserved
168  $A8 WORD    gb_Flags
170  $AA WORD    gb_BlitLock
172  $AC WORD    gb_BlitNest
174  $AE STRUCT  gb_BlitWaitQ,LH_SIZE
188  $BC APTR    gb_BlitOwner
192  $C0 STRUCT  gb_TOF_WaitQ,LH_SIZE
206  $CE WORD    gb_DisplayFlags
208  $D0 APTR    gb_SimpleSprites
212  $D4 WORD    gb_MaxDisplayRow
214  $D6 WORD    gb_MaxDisplayColumn
216  $D8 WORD    gb_NormalDisplayRows
218  $DA WORD    gb_NormalDisplayColumns
220  $DC WORD    gb_NormalDPMX
222  $DE WORD    gb_NormalDPMY
224  $E0 APTR    gb_LastChanceMemory
228  $E4 APTR    gb_LCMptr
232  $E8 WORD    gb_MicrosPerLine ;microseconds times 256
234  $EA WORD    gb_MinDisplayColumn
236  $EC UBYTE   gb_ChipRevBits0  ;new Agnus/Denise
237  $ED STRUCT  gb_crb_reserved,5
242  $F2 STRUCT  gb_monitor_id,2
244  $F4 STRUCT  gb_hedley,4*8
276 $114 STRUCT  gb_hedley_sprites,4*8
308 $134 STRUCT  gb_hedley_sprites1,4*8
340 $154 WORD    gb_hedley_count
342 $156 WORD    gb_hedley_flags
344 $158 WORD    gb_hedley_tmp
346 $15A APTR    gb_hash_table
350 $15E UWORD   gb_current_tot_rows
352 $160 UWORD   gb_current_tot_cclks
354 $162 UBYTE   gb_hedley_hint
355 $163 UBYTE   gb_hedley_hint2
356 $164 STRUCT  gb_nreserved,4*4
372 $174 APTR    gb_a2024_sync_raster
376 $178 WORD    gb_control_delta_pal
378 $17A WORD    gb_control_delta_ntsc
380 $17C APTR    gb_current_monitor
384 $180 STRUCT  gb_MonitorList,LH_SIZE
398 $18E APTR    gb_default_monitor
402 $192 APTR    gb_MonitorListSemaphore
406 $196 APTR    gb_DisplayInfoDataBase
410 $19A APTR    gb_ActiViewCprSemaphore
414 $19E APTR    gb_UtilityBase
```

```
418 $1A2 APTR    gb_ExecBase
422 $1A6 LABEL   gb_SIZE

OWNBLITTERn = 0 ;Blitter occupied
QBOWNERn    = 1 ;Blitter occupied by queue

GFXB_BIG_BLITS = 0 ;ChipRevBits0
GFXB_HR_AGNUS  = 0 ;HiRes Agnus
GFXB_HR_DENISE = 1 ;HiRes Denise

NTSCn            = 0 ;display bits
GENLOCn          = 1
PALn             = 2
TODA_SAFEn       = 3

BLITMSG_FAULTn   = 2

Dec Hex STRUCTURE XLN,0 ;graphics node
  0  $0 APTR   XLN_SUCC
  4  $4 APTR   XLN_PRED
  8  $8 UBYTE  XLN_TYPE
  9  $9 BYTE   XLN_PRI
 10  $A APTR   XLN_NAME
 14  $E UBYTE  XLN_SUBSYSTEM
 15  $F UBYTE  XLN_SUBTYPE
 16 $10 LONG   XLN_LIBRARY
 20 $14 LONG   XLN_INIT
 24 $18 LABEL  XLN_SIZE

SS_GRAPHICS = $02 ;GfxSemaphore

VIEW_EXTRA_TYPE      = 1
VIEWPORT_EXTRA_TYPE  = 2
SPECIAL_MONITOR_TYPE = 3
MONITOR_SPEC_TYPE    = 4

VTAG_END_CM              = $00000000
VTAG_CHROMAKEY_CLR       = $80000000
VTAG_CHROMAKEY_SET       = $80000001
VTAG_BITPLANEKEY_CLR     = $80000002
VTAG_BITPLANEKEY_SET     = $80000003
VTAG_BORDERBLANK_CLR     = $80000004
VTAG_BORDERBLANK_SET     = $80000005
VTAG_BORDERNOTRANS_CLR   = $80000006
VTAG_BORDERNOTRANS_SET   = $80000007
VTAG_CHROMA_PEN_CLR      = $80000008
VTAG_CHROMA_PEN_SET      = $80000009
VTAG_CHROMA_PLANE_SET    = $8000000A
```

```
VTAG_ATTACH_CM_SET      = $8000000B
VTAG_NEXTBUF_CM         = $8000000C
VTAG_BATCH_CM_CLR       = $8000000D
VTAG_BATCH_CM_SET       = $8000000E
VTAG_NORMAL_DISP_GET    = $8000000F
VTAG_NORMAL_DISP_SET    = $80000010
VTAG_COERCE_DISP_GET    = $80000011
VTAG_COERCE_DISP_SET    = $80000012
VTAG_VIEWPORTEXTRA_GET  = $80000013
VTAG_VIEWPORTEXTRA_SET  = $80000014
VTAG_CHROMAKEY_GET      = $80000015
VTAG_BITPLANEKEY_GET    = $80000016
VTAG_BORDERBLANK_GET    = $80000017
VTAG_BORDERNOTRANS_GET  = $80000018
VTAG_CHROMA_PEN_GET     = $80000019
VTAG_CHROMA_PLANE_GET   = $8000001A
VTAG_ATTACH_CM_GET      = $8000001B
VTAG_BATCH_CM_GET       = $8000001C
VTAG_BATCH_ITEMS_GET    = $8000001D
VTAG_BATCH_ITEMS_SET    = $8000001E
VTAG_BATCH_ITEMS_ADD    = $8000001F
VTAG_VPMODEID_GET       = $80000020
VTAG_VPMODEID_SET       = $80000021
VTAG_VPMODEID_CLR       = $80000022
VTAG_USERCLIP_GET       = $80000023
VTAG_USERCLIP_SET       = $80000024
VTAG_USERCLIP_CLR       = $80000025

GENLOCK_VIDEO     =    $2 ;composite video signal
V_LACE            =    $4 ;Interlace
V_SUPERHIRES      =   $20 ;SuperHiRes
V_PFBA            =   $40 ;switch Playfields
V_EXTRA_HALFBRITE =   $80 ;Halfbrite
GENLOCK_AUDIO     =  $100 ;audio signal
V_DUALPF          =  $400 ;DualPlayfield
V_HAM             =  $800 ;Hold And Modify
V_EXTENDED_MODE   = $1000 ;extended structure
V_VP_HIDE         = $2000 ;hide ViewPort
V_SPRITES         = $4000 ;Sprite DMA activated
V_HIRES           = $8000 ;HiRes

EXTEND_VSTRUCT    = $1000

VPF_DENISE = $80
VPF_A2024  = $40
VPF_AGNUS  = $20
VPF_TENHZ  = $20
VPF_ILACE  = $10
```

```
Dec Hex STRUCTURE ColorMap,0
  0  $0 BYTE   cm_Flags
  1  $1 BYTE   cm_Type
  2  $2 WORD   cm_Count
  4  $4 APTR   cm_ColorTable
  8  $8 APTR   cm_vpe
 12  $C APTR   cm_TransparencyBits
 16 $10 BYTE   cm_TransparencyPlane
 17 $11 BYTE   cm_reserved1
 18 $12 WORD   cm_reserved2
 20 $14 APTR   cm_vp
 24 $18 APTR   cm_NormalDisplayInfo
 28 $1C APTR   cm_CoerceDisplayInfo
 32 $20 APTR   cm_batch_items
 36 $24 LONG   cm_VPModeID
 40 $28 LABEL  cm_SIZEOF

COLORMAP_TYPE_V1_2 = 0 ;old ColorMap
COLORMAP_TYPE_V36  = 1 ;new ColorMap

COLORMAP_TRANSPARENCY    = $01
COLORPLANE_TRANSPARENCY = $02
BORDER_BLANKING          = $04
BORDER_NOTRANSPARENCY    = $08
VIDEOCONTROL_BATCH       = $10
USER_COPPER_CLIP         = $20

Dec Hex STRUCTURE ViewPort,0
  0  $0 LONG    vp_Next
  4  $4 LONG    vp_ColorMap
  8  $8 LONG    vp_DspIns
 12  $C LONG    vp_SprIns
 16 $10 LONG    vp_ClrIns
 20 $14 LONG    vp_UCopIns
 24 $18 WORD    vp_DWidth
 26 $1A WORD    vp_DHeight
 28 $1C WORD    vp_DxOffset
 30 $1E WORD    vp_DyOffset
 32 $20 WORD    vp_Modes
 34 $22 BYTE    vp_SpritePriorities
 35 $23 BYTE    vp_ExtendedModes
 36 $24 APTR    vp_RasInfo
 40 $28 LABEL   vp_SIZEOF

Dec Hex STRUCTURE View,0
  0  $0 LONG    v_ViewPort
  4  $4 LONG    v_LOFCprList
```

```
  8   $8 LONG     v_SHFCprList
 12   $C WORD     v_DyOffset
 14   $E WORD     v_DxOffset
 16 $10 WORD     v_Modes
 18 $12 LABEL    v_SIZEOF

Dec Hex STRUCTURE ViewExtra,XLN_SIZE
 24 $18 APTR     ve_View
 28 $1C APTR     ve_Monitor
 32 $20 LABEL    ve_SIZEOF

Dec Hex STRUCTURE ViewPortExtra,XLN_SIZE
 24 $18 APTR     vpe_ViewPort
 28 $1C STRUCT   vpe_DisplayClip,ra_SIZEOF
 36 $24 LABEL    vpe_SIZEOF

Dec Hex STRUCTURE collTable,0
  0   $0 LONG     cp_collPtrs,16
 64 $40 LABEL    cp_SIZEOF

Dec Hex STRUCTURE RasInfo,0
  0   $0 APTR     ri_Next
  4   $4 LONG     ri_BitMap
  8   $8 WORD     ri_RxOffset
 10   $A WORD     ri_RyOffset
 12   $C LABEL    ri_SIZEOF
```

2. General Blitter Control

BitMapScale	**Change the size of bit-map contents**

Call:

```
BitMapScale( bitScaleArgs )
-678(A6)     a0

STRUCT BitScaleArgs *bitScaleArgs
```

Function: Copies a portion of a bit-map to another bit-map, changing the size to correspond to the size of the destination bit-map.

Parameters: bitScaleArgs

Structure with the following parameters:

bsa_srcX, bsa_srcY Upper left corner of the source bit-map.

bsa_srcWidth, bsa_srcHeight
Size of source bit-map.

bsa_destX, bsa_destY
Position in the destination bit-map.

bsa_destWidth, bsa_destHeight
New size (result)

bsa_xSrcFactor:bsa_xDestFactor
Scaling factor, corresponds with bsa_srcWidth:bsa_destWidth; Range: 1..16383.

bsa_ySrcFactor:bsa_yDestFactor
Same for bsa_srcHeight:bsa_destHeight.

bsa_srcBitMap
Source bit-map

bsa_destBitMap
Destination bit-map (may not overlap with srcBitMap).

bsa_flags 0 (not yet supported)

Result: destWidth and destHeight are filled with the new size.

Example: Double the size of an image. A LoRes bit-map in 320*256 pixel format is copied to fill a bit-map in 640*512 HiRes Interlace format. The size change is accomplished as follows:

```
bsa_DestWidth=bsa_SrcWidth*bsa_XDestFactor/bsa_XSrcFactor
bsa_DestHeight=bsa_SrcHeight*bsa_YDestFactor/bsa_YSrcFactor
```

In our example:

```
bsa_DestWidth  = 320 * 2 / 1 = 640
bsa_DestHeight = 256 * 2 / 1 = 512
```

Here is the simple demo routine:

```
movea.l _GfxBase,a6
lea     _BitScaleArgs(pc),a0
jsr     _BitMapScale(a6)

...

_BitScaleArgs
 dc.w    0,0                ;bsa_SrcX, bsa_SrcY
 dc.w    320,256            ;bsa_SrcWidth, bsa_SrcHeight
 dc.w    1,1                ;bsa_XSrcFactor, bsa_YSrcFactor
 dc.w    0,0                ;bsa_DestX, bsa_DestY
 dc.w    0,0                ;bsa_DestWidth, bsa_DestHeight
 dc.w    2,2                ;bsa_XDestFactor,bsa_YDestFactor
 dc.l    _LoResBitMap       ;bsa_SrcBitMap
 dc.l    _HiResILaceBitMap  ;bsa_DestBitMap
 dc.l    0                  ;bsa_Flags
 dc.w    0,0                ;bsa_XDDA, bsa_YDDA
 dc.l    0,0                ;bsa_Reserved1, bsa_Reserved2
```

BltBitMap — Copy a portion of a bit-map

Call:

```
planecnt = BltBitMap(SrcBitMap, SrcX, SrcY, DstBitMap,
D0         -30(A6)    A0         D0:16 D1:16 A1
           DstX, DstY, SizeX, SizeY, Minterm, Mask, TempA)
           D2:16 D3:16 D4:16  D5:16  D6:8     D7:8  A2

ULONG  planecnt
STRUCT BitMap *SrcBitMap,DstBitMap
WORD   SrcX,SrcY,DstX,DstY,SizeX,SizeY
UBYTE  Minterm,Mask
APTR   TempA
```

Function: Copies part of a bit-map to the given position in another bit-map. Both bit-maps can be the same and the ranges may overlap. If a bit-plane address is set to 0, it is handled like an empty bit-plane. If the bit-plane address is -1, it is handled like a filled bit-plane.

Parameters: SrcBitMap Source bit-map

DstBitMap
Destination bit-map

SrcX, SrcY Coordinates in the source bit-map.

DstX, DstY
Coordinates in the destination bit-map.

SizeX,SizeY
Size of the region to be copied.

Minterm Logical combination of source and destination:

Blitter source A is filled within the region.
Blitter source B is the source.
Blitter sources C and D are the destination.
$C0 copies, $30 copies the inverted source, $50 inverts only the destination, etc.

Mask Bit mask for destination bit-plane.

TempA Buffer for one line (source A) that must be scrolled horizontally if the regions overlap.

Result: Number of affected bit-planes.

BltBitMapRastPort — Copy a bit-map range to a RastPort

Call:

```
BltBitMapRastPort
(srcbm,srcx,srcy,destrp,destX,destY,sizeX,sizeY,minterm)
-606(A6)         a0     d0    d1    a1    d2    d3    d4  d5  d6

STRUCT BitMap *srcbm
WORD   srcx,srcy,destX,destY,sizeX,sizeY
STRUCT RastPort *destrp
UBYTE  minterm
```

Function: Similar to BltBitMap(), except that the destination is the given RastPort and a mask cannot be used.

Parameters: srcbm Source bit-map

srcx,srcy Position in the source bit-map.

destrp	Destination RastPort
destX,destY	Position in RastPort
sizeX,sizeY	Size of range
minterm	Logical combination

BltClear — Clear memory block (ChipRAM)

Call:
```
BltClear( memBlock, bytecount, flags )
-300(A6)  a1        d0         d1

APTR  memBlock
ULONG bytecount,flags
```

Function: Clears a memory block in ChipRAM.

Parameters:

memBloc	Address of block	
flags	Bit 0: 1	Call WaitBlit()
	Bit 1: 0	bytecount = size of range
	1	bytecount = lines <<16+BytesPerLine
	Bit 2: 1	bytecount = full value<<16+size of range

BltMaskBitMapRastPort — Copy bit-map to a RastPort with a mask

Call:
```
BltMaskBitMapRastPort
(srcbm,srcx,srcy,destrp,destX,destY,sizeX,sizeY,minterm,bltmask)
-636(A6)            a0   d0   d1   a1    d2    d3    d4    d5    d6     a2
```

Functions, Parameters:

Same as BltBitMapRastPort(), with the addition of the address of a single bit-plane (bltmask) in which the affected bits are set.

BltPattern — Blit using a mask

Call:

```
BltPattern(rp, mask, xl, yl, maxx, maxy, bytecnt)
-312(A6)   a1   a0    d0  d1  d2    d3    d4

STRUCT RastPort *rp
APTR   mask
SHORT  xl,yl,maxx,maxy,bytecnt
```

Function: Blits a rectangular region at a given position via a mask, using the Drawmode and Areafill pattern entries from the RastPort.

Parameters:

rp — RastPort

mask — MaskBitPlane or 0 (rectangle)

xl,yl — Position in RastPort

maxx,maxy — Size of range

bytecnt — Bytes per line in the mask

BltTemplate — Copy a rectangular region to the RastPort

Call:

```
BltTemplate(SrcTemplate, SrcX, SrcMod, rp, DstX, DstY, SizeX, SizeY)
-36(A6)      A0           D0    D1      A1  D2    D3    D4     D5

APTR   SrcTemplate
WORD   SrcX,SrcMod,DstX,DstY,SizeX,SizeY
STRUCT RastPort *rp
```

Function: Copies a rectangular portion of a bit-plane with the selected color and Drawmode to a given position in a RastPort.

Parameters:

SrcTemplate — Address of the first word in the BitImage.

SrcX — X bit offset from SrcTemplate (0..15).

SrcMod — Bytes per line in the BitImage.

rp Destination RastPort.

DstX, DstY
Coordinates in RastPort.

SizeX, SizeY
Size of range.

ClipBlit **BltBitMap(), with layers**

Call:

```
ClipBlit(Src, SrcX, SrcY, Dest, DestX, DestY, XSize, YSize, Minterm)
-552(A6) a0   d0    d1    a1    d2     d3     d4     d5     d6

STRUCT RastPort *Src,*Dest
WORD   SrcX,SrcY,DestX,DestY,XSize,YSize
UBYTE  Minterm
```

Function: Same as BltBitMap(), except that the ClipRects are considered here. With windows, this function must be called instead of BltBitMap().

Parameters: Src Source RastPort

SrcX,SrcY Position in source RastPort.

Dest Destination RastPort

DestX,DestY
Position in destination RastPort.

XSize,YSize
Size of range

Minterm Logical combination (B=source, C=destination)

CopySBitMap **Copy SuperBitMap range to a layer**

Call:

```
CopySBitMap( layer )
-450(A6)     a0

STRUCT Layer *layer
```

Function: Opposite of SyncSBitMap() - copies the current excerpt of a SuperBitMap to the given SuperBitMap layer.

Parameters: layer — SuperBitMap layer (must be allocated with LockLayerROM())

DisownBlitter — **Free Blitter**

Call:
```
DisownBlitter()
-462(A6)
```

Function: Frees the Blitter for use by other programs.

OwnBlitter — **Obtain use of Blitter**

Call:
```
OwnBlitter()
-456(A6)
```

Function: Prevents other programs from using the Blitter. The Blitter becomes available only after it finishes its current operation (see WaitBlit()).

QBlit — **Enter BltNode in the Blitter list**

Call:
```
QBlit( bp )
-276(A6) a1

STRUCT bltnode *bp
```

Function: Enters a BltNode in the wait queue of the Blitter. If the indicated routine is called, the Blitter stops work and becomes available, meaning it can be directly programmed. The routine must be executable in both supervisor and user modes.

Parameters: bp — Initialized BltNode

QBSBlit — **QBlit with raster synchronization**

Call:
```
QBSBlit( bsp )
-294(A6) a1
```

```
STRUCT bltnode *bsp
```

Function: Same as QBlit(), except that the routine is only called when the monitor beam reaches a certain position. BltNodes entered with QBSBlit() take priority over QBlit() BltNodes. Access by several tasks can lead to synchronization errors or true timing problems.

Parameters: bsp Initialized BltNode

ScalerDiv **Calculate scaling**

Call:

```
result = ScalerDiv(factor, numerator, denominator)
d0       -684(A6)  d0      d1         d2

UWORD result,factor,numerator,denominator
```

Function: Calculates factor*numerator/denominator just like BitMapScale(). For example, the new width can be calculated as width*XDestFactor/XSrcFactor.

Parameters: factor Width or height from BitMapScale().

numerator ?DestFactor

denominator
?SrcFactor

Result: factor*numerator/denominator

ScrollRaster **Scroll a rectangular range**

Call:

```
ScrollRaster( rp, dx, dy, xmin, ymin, xmax, ymax)
-396(A6)      a1  d0  d1  d2    d3    d4    d5

STRUCT RastPort *rp
WORD  dx,dy,xmin,ymin,xmax,ymax
```

Function: Moves the contents of a rectangular range by the given delta value in the direction of coordinates (0,0). The bug that occurred in Kick 1.x, if the TmpRas structure was missing, has been fixed.

Parameters: rp RastPort

dx,dy Delta value (right and down NEGATIVE)

xmin,ymin Upper left corner

xmax,ymax
Lower right corner

SyncSBitMap **Copy layer contents to a SuperBitMap**

Call:
```
SyncSBitMap( layer )
-444(A6)     a0

STRUCT Layer *layer
```

Function: Copies the contents of a SuperBitMap layer to the current position of the SuperBitMap.

Parameters: layer SuperBitMap layer (locked)

WaitBlit **Wait for the Blitter**

Call:
```
WaitBlit()
-228(A6)
```

Function: Waits until the Blitter finished its current work. This function is normally used after OwnBlitter() and/or before DisownBlitter().

```
Dec Hex STRUCTURE BitMap,0      ;BitMap
  0  $0 WORD     bm_BytesPerRow ;bytes per row
  2  $2 WORD     bm_Rows        ;rows
  4  $4 BYTE     bm_Flags       ;Flags
  5  $5 BYTE     bm_Depth       ;number of BitPlanes
  6  $6 WORD     bm_Pad
  8  $8 STRUCT   bm_Planes,8*4  ;PlanePointer
 40 $28 LABEL    bm_SIZEOF

Dec Hex STRUCTURE BitScaleArgs,0 ;BitMapScale() argument
  2  $2 UWORD    bsa_SrcX        ;source position
  4  $4 UWORD    bsa_SrcY
  6  $6 UWORD    bsa_SrcWidth    ;source size
  8  $8 UWORD    bsa_SrcHeight
```

```
10  $A UWORD   bsa_XSrcFactor   ;denominators
12  $C UWORD   bsa_YSrcFactor
14  $E UWORD   bsa_DestX        ;destination position
16 $10 UWORD   bsa_DestY
18 $12 UWORD   bsa_DestWidth    ;result
20 $14 UWORD   bsa_DestHeight
22 $16 UWORD   bsa_XDestFactor  ;numerators
24 $18 UWORD   bsa_YDestFactor
26 $1A APTR    bsa_SrcBitMap    ;source BitMap
30 $1E APTR    bsa_DestBitMap   ;destination BitMap
34 $22 ULONG   bsa_Flags        ;0!
38 $26 UWORD   bsa_XDDA
40 $28 UWORD   bsa_YDDA
42 $2A LONG    bsa_Reserved1
46 $2E LONG    bsa_Reserved2
50 $32 LABEL   bsa_SIZEOF
```

3. Refresh Functions

AndRectRegion	**Preserve contents of a rectangle**

Call:

```
AndRectRegion(region,rectangle)
-504(A6)      a0     a1

STRUCT Region *region
STRUCT Rectangle *rectangle
```

Function: Deletes everything in the region outside of the given rectangle.

Parameters:

region — Region structure

rectangle — Rectangle structure

AndRegionRegion	**Trim a region**

Call:

```
status   AndRegionRegion(region1,region2)
d0       -624(A6)        a0      a1

BOOL status
STRUCT Region *region1,*region2
```

Function: Cut off surfaces from region2 that are not part of region1.

Parameters: region1 Mask region

region2 Destination region

Result: 0 Error (no memory)

ClearRectRegion **Clear a rectangle within a region**

Call:

```
status = ClearRectRegion(region,rectangle)
d0       -522(A6)         a0     a1

BOOL error
STRUCT Region *region
STRUCT Rectangle *rectangle
```

Function: Cuts a rectangle out of a region.

Parameters: region Region containing the rectangle.

rectangle Rectangle to be deleted.

Result: 0 Error (no memory)

ClearRegion **Clear all rectangles within a region**

Call:

```
ClearRegion(region)
-528(A6)    a0

STRUCT Region *region
```

Function: Clears an entire region.

Parameters: region Region to be cleared

DisposeRegion **Free region**

Call:

```
DisposeRegion(region)
-534(A6)       a0

STRUCT Region *region
```

Function: Frees the memory of a region.

Parameters: region Region structure

NewRegion **Get a Region structure**

Call:

```
region   NewRegion()
d0       -516(A6)

STRUCT Region *region
```

Function: Allocates memory for a Regions structure and initializes it.

Result: Region structure or 0.

OrRectRegion **Insert rectangle into a region**

Call:

```
status = OrRectRegion(region,rectangle)
d0       -510(A6)     a0     a1

BOOL   status
STRUCT Region *region
STRUCT Rectangle *rectangle
```

Function: Inserts the given rectangle (not contained in the region) into the given region.

Parameters: region Region structure

rectangle Rectangle structure

Result: 0 Error

OrRegionRegion **Join Region structures**

Call:

```
status   OrRegionRegion(region1,region2)
d0       -612(A6)       a0      a1

BOOL   status
STRUCT Region *region1,*region2
```

Function: Adds region1 to region2.

Parameters: region1,region2
Region structures

Result: 0 Error

XorRectRegion	**Exclusive OR combination of two areas**

Call:

```
status = XorRectRegion(region,rectangle)
d0       -558(A6)      a0     a1

BOOL   status
STRUCT Region *region
STRUCT Rectangle *rectangle
```

Function: Adds the given rectangle (not contained in the region) to the given region, and deletes the part of the region common to both.

Parameters: region Region structure

rectangle Rectangle structure

Result: 0 Error

XorRegionRegion	**Exclusive OR combination of two regions**

Call:

```
status = XorRegionRegion(region1,region2)
d0       -618(A6)         a0      a1

BOOL   status
STRUCT Region *region1,*region2
```

Function: Adds one region to the other and deletes the overlapping area.

Parameters: region1,region2
Regions to be combined

Result: 0 Error

```
Dec Hex STRUCTURE Rectangle,0 ;rectangle
  0  $0 WORD  ra_MinX         ;dimensions
  2  $2 WORD  ra_MinY
  4  $4 WORD  ra_MaxX
  6  $6 WORD  ra_MaxY
```

```
  8  $8 LABEL ra_SIZEOF

Dec Hex STRUCTURE Rect32,0 ;32 bit rectangle
  0  $0 LONG   r32_MinX     ;dimensions
  4  $4 LONG   r32_MinY
  8  $8 LONG   r32_MaxX
 12  $C LONG   r32_MaxY
 16 $10 LABEL  r32_SIZEOF

Dec Hex STRUCTURE Region,0
  0  $0 STRUCT rg_bounds,ra_SIZEOF
  8  $8 APTR   rg_RegionRectangle
 12  $C LABEL  rg_SIZEOF

Dec Hex STRUCTURE RegionRectangle,0
  0  $0 APTR   rr_Next
  4  $4 APTR   rr_Prev
  8  $8 STRUCT rr_bounds,ra_SIZEOF
 16 $10 LABEL  rr_SIZEOF
```

4. Data Structures

AllocRaster	**Allocate memory for a bit-plane**

Call:

```
planeptr = AllocRaster( width, height )
d0         -492(A6)     d0:16  d1:16

APTR   planeptr
USHORT width,height
```

Function: Allocates the ChipRAM required for a bit-plane of the given size.

Parameters: width — Bit-plane width in pixels.

height — Bit-plane height in pixels.

Result: Address of memory block or 0.

AttemptLockLayerRom	**Attempt to lock a layer**

Call:

```
gotit = AttemptLockLayerRom( layer )
d0      -654(A6)             a5

BOOLEAN gotit
```

```
STRUCT Layer *layer
```

Function: Attempts to lock a layer with exclusive access rights.

Parameters: layer — Layer structure

Result: 0 — No access to layer.

FreeColorMap — **Free a ColorMap**

Call:

```
FreeColorMap( colormap )
-576(A6)       a0

STRUCT ColorMap *colormap
```

Function: Frees the memory used by a structure allocated with GetColorMap().

Parameters: colormap — Address of the ColorMap.

FreeRaster — **Free a bit-plane**

Call:

```
FreeRaster( p, width, height)
-498(A6)     a0 d0:16  d1:16

APTR   p
USHORT width,height
```

Function: Frees the memory used for a bit-plane.

Parameters:

p	PlaneAddress
width	Width in bits
height	Height of bit-plane

GetColorMap — **Allocate a ColorMap**

Call:

```
cm = GetColorMap( entries )
d0   -570(A6)     d0

STRUCT ColorMap *cm
```

```
LONG entries
```

Function: Allocates memory for a ColorMap and initializes the structure.

Parameters: entries Number of colors

Result: ColorMap or 0.

GetRGB4 — Allocate a 3x4 bit color value

Call:

```
value = GetRGB4( colormap, entry )
d0      -582(A6) a0        d0

ULONG  value
STRUCT ColorMap *colormap
LONG   entry
```

Function: Reads the color value of a color number from the given ColorMap.

Parameters: colormap ColorMap structure

entry Color number (0...)

Result: red value<<8+green value<<4+blue value (4 bits each: 0...15) or -1 (entry not available, error)

Example: Get the color values for the background color of a ViewPort and set them in a second ViewPort (Warning: doing this by hand could cause problems with the new 24 bit ColorMaps):

```
...
movea.l _GfxBase,a6
movea.l _ViewPort1,a0
movea.l vp_ColorMap(a0),a0
moveq   #0,d0
jsr     _LVOGetRGB4(a6)
tst.w   d0
bmi     _Zerror
moveq   #15,d3              ;mask for blue value
and.w   d0,d3               ;blue value
```

```
lsr.w   #4,d0
moveq   #15,d2             ;green value mask
and.w   d0,d2              ;green value
lsr.w   #4,d0
moveq   #15,d1             ;red value mask
and.w   d0,d1              ;red value
moveq   #0,d0              ;color number
movea.l _ViewPort2,a0      ;2nd ViewPort
jsr     _LVOSetRGB4(a6)
...
```

InitBitMap — **Initialize a BitMap structure**

Call:

```
InitBitMap( bm, depth, width, height )
-390(A6)    a0  d0     d1     d2

STRUCT BitMap *bm
BYTE   depth
UWORD  width,height
```

Function: Initializes a BitMap structure. The bit-plane addresses are excluded in order to keep the size of the structure variable.

Parameters:

bm	BitMap structure to be initialized.
depth	Number of bit-planes
width	Width in bits
height	Height of bit-plane

InitRastPort — **Initialize a RastPort**

Call:

```
InitRastPort( rp )
-198(A6)      a1

STRUCT RastPort *rp
```

Function: Initializes a RastPort structure with the standard values (Mask=-1, FgPen=-1, AOLPen=-1, LinePtrn=-1, DrawMode=JAM2, Font=Systemfont).

Parameters:

rp	RastPort structure

InitTmpRas — **Initialize TmpRas**

Call:

```
InitTmpRas(tmpras, buffer, size)
-468(A6)   a0      a1      d0

STRUCT TmpRas *tmpras
APTR   buffer
LONG   size
```

Function: Initializes a TmpRas structure with a buffer for intensive graphics operations (AreaEnd(), Flood(), Text()).

Parameters:

tmpras	TmpRas structure
buffer	ChipRAM buffer
size	Buffer size

InitView — **Initialize View structure**

Call:

```
InitView( view )
-360(A6)  a1

STRUCT View *view
```

Function: Initializes a View structure with the standard values.

Parameters:

view	View structure

InitVPort — **Initialize ViewPort structure**

Call:

```
InitVPort( vp )
-204(A6)   a0

STRUCT ViewPort *vp
```

Function: Initializes a ViewPort structure with the standard values.

Parameters:

vp	ViewPort structure

LockLayerRom **Obtain access to a layer**

Call:

```
LockLayerRom( layer )
-432(A6)      a5

STRUCT Layer *layer
```

Function: Obtains exclusive access to a layer. No Intuition functions may be called during this time, since most of the work is done by the input Handler Intuition(), which also must use locking to obtain exclusive access. There is no problem calling LockLayerRom() with libraries that are not based on a single task, since this function first checks to see if the active task already has access to the layer.

Parameters: layer Layer structure

SetRGB4CM **Enter a color**

Call:

```
SetRGB4CM(cm, n,  r,   g,   b)
-630(A6)  a0  d0  d1:4 d2:4 d3:4

STRUCT ColorMap *cm
SHORT  n
UBYTE  r,g,b
```

Function: Enters the intensity values for a color in a ColorMap. This function is used to create color tables before entering in a ViewPort.

Parameters:

cm	ColorMap
n	Color number (0...31)
r,g,b	4 bit intensity value (0...15)

UnlockLayerRom **Free a layer**

Call:

```
UnlockLayerRom( layer )
-438(A6)        a5
```

Function: Frees exclusive access rights to a layer.

Parameters: layer Locked layer

```
NEWLOCKS = 1 ;new Layer lock

Dec Hex STRUCTURE Layer,0             ;range for clipping
  0  $0 LONG    lr_front              ;foreground Layer
  4  $4 LONG    lr_back               ;background Layer
  8  $8 LONG    lr_ClipRect           ;ClipRect
 12  $C LONG    lr_rp                 ;RastPort
 16 $10 WORD    lr_MinX               ;range
 18 $12 WORD    lr_MinY               ;
 20 $14 WORD    lr_MaxX               ;
 22 $16 WORD    lr_MaxY               ;
 24 $18 STRUCT  lr_reserved,4         ;reserved
 28 $1C WORD    lr_priority           ;priority
 30 $1E WORD    lr_Flags              ;Flags
 32 $20 LONG    lr_SuperBitMap        ;BitMap
 36 $24 LONG    lr_SuperClipRect      ;ClipRect
 40 $28 APTR    lr_Window             ;window
 44 $2C WORD    lr_Scroll_X           ;BitMap offsets
 46 $2E WORD    lr_Scroll_Y           ;
 48 $30 APTR    lr_cr                 ;ClipRect
 52 $34 APTR    lr_cr2                ;ClipRect
 56 $38 APTR    lr_crnew              ;ClipRect
 60 $3C APTR    lr_SuperSaverClipRects ;ClipRects
 64 $40 APTR    lr__cliprects         ;ClipRects
 68 $44 APTR    lr_LayerInfo          ;LayerInfo
 72 $48 STRUCT  lr_Lock,SS_SIZE       ;SignalSemaphore
118 $76 APTR    lr_BackFill           ;backfill Hook
122 $7A ULONG   lr_reserved1          ;reserved
126 $7E APTR    lr_ClipRegion         ;region
130 $82 APTR    lr_saveClipRects      ;ClipRects
134 $86 STRUCT  lr_reserved2,22       ;reserved (SS_SIZE)
156 $9C APTR    lr_DamageList         ;damage list
160 $A0 LABEL   lr_SIZEOF

Dec Hex STRUCTURE  ClipRect,0
  0  $0 LONG     cr_Next    ;next ClipRect
  4  $4 LONG     cr_prev    ;previous ClipRect
  8  $8 LONG     cr_lobs
 12  $C LONG     cr_BitMap  ;BitMap
 16 $10 WORD     cr_MinX    ;range
 18 $12 WORD     cr_MinY    ;
 20 $14 WORD     cr_MaxX    ;
 22 $16 WORD     cr_MaxY    ;
 24 $18 APTR     cr__p1
 28 $1C APTR     cr__p2
```

```
 32 $20 LONG    cr_reserved
 36 $24 LONG    cr_Flags  ;Flags
 40 $28 LABEL   cr_SIZEOF

CR_NEEDS_NO_CONCEALED_RASTERS = 1 ;internal Flag
CR_NEEDS_NO_LAYERBLIT_DAMAGE = 2

ISLESSX = 1 ;Flags for clipping
ISLESSY = 2
ISGRTRX = 4
ISGRTRY = 8

LAYERSIMPLE          =    1
LAYERSMART           =    2
LAYERSUPER           =    4
LAYERUPDATING        =  $10
LAYERBACKDROP        =  $40
LAYERREFRESH         =  $80
LAYER_CLIPRECTS_LOST = $100
LMN_REGION           =   -1

Dec Hex STRUCTURE Layer_Info,0
  0  $0 APTR   li_top_layer ;top Layer
  4  $4 APTR   li_check_lp
  8  $8 APTR   li_obs
 12  $C STRUCT li_FreeClipRects,MLH_SIZE
 24 $18 STRUCT li_Lock,SS_SIZE
 70 $46 STRUCT li_gs_Head,LH_SIZE
 84 $54 LONG   li_long_reserved
 88 $58 WORD   li_Flags
 90 $5A BYTE   li_fatten_count
 91 $5B BYTE   li_LockLayersCount
 92 $5C WORD   li_LayerInfo_extra_size
 94 $5E APTR   li_blitbuff
 98 $62 APTR   li_LayerInfo_extra
102 $66 LABEL  li_SIZEOF

NEWLAYERINFO_CALLED = 1
```

5. Draw Functions

AreaDraw	**Define corner point for AreaFill**

Call:

```
error = AreaDraw( rp,  x, y)
d0      -258(A6)  A1  D0,D1

LONG  error
```

```
STRUCT RastPort *rp
SHORT  x,y
```

Function: Inserts a point in the vector list for AreaFill.

Parameters: rp — RastPort with AreaInfo

x,y — Point coordinates

Result: 0 — No error

AreaEllipse — Insert an ellipse for AreaFill in AreaInfo

Call:

```
error = AreaEllipse( rp, cx, cy, a, b)
d0      -186(A6)     a1  d0  d1 d2 d3

LONG error
STRUCT RastPort *rp
SHORT  cx,cy,a,b
```

Function: Stores an ellipse in the vector buffer.

Parameters: rp — RastPort with AreaInfo

cx,cy — Center of ellipse

a — Horizontal radius (a>0)

b — Vertical radius (b>0)

Result: 0 — No error

AreaEnd — Execute AreaFill according to vector table contents

Call:

```
error = AreaEnd(rp)
d0      -264(A6) A1

LONG error
STRUCT RastPort *rp
```

Function: Processes the vector buffer of the Area routines and fills the calculated area. Re-initializes for new AreaMove() calls.

Parameters: rp — RastPort

Result: 0 — No error

AreaMove — **Define starting point for AreaFill**

Call:

```
error =   AreaMove( rp,    x, y)
d0        -252(A6)  a1    d0 d1

LONG   error
STRUCT RastPort *rp
SHORT  x,y
```

Function: Closes the last polygon and begins a new one.

Parameters: rp — RastPort with AreaInfo

x,y — Position of the starting point

Result: 0 — No error

Draw — **Draw a line**

Call:

```
Draw( rp, x, y)
-246(A6) a1 d0 d1

STRUCT RastPort *rp
SHORT  x,y
```

Function: Draws a line from the current position to the given coordinates.

Parameters: rp — RastPort

x,y — Destination coordinates

DrawEllipse — **Draw an ellipse**

Call:

```
DrawEllipse( rp, cx, cy, a, b )
-180(A6)     a1  d0  d1  d2 d3

STRUCT RastPort *rp
```

```
SHORT  cx,cy,a,b
```

Function: Draws an ellipse in RastPort.

Parameters:

rp	RastPort
cx,cy	Center of ellipse
a	Horizontal radius (a>0)
b	Vertical radius (b>0)

EraseRect **Fill a rectangle using the BackFill hook**

Call:

```
EraseRect( rp, xmin, ymin, xmax, ymax)
-810(A6)   a1  d0:16 d1:16 d2:16 d3:16

STRUCT RastPort *rp
SHORT xmin,ymin,xmax,ymax
```

Function: Fills a rectangular area in a RastPort. If the RastPort layer is showing, then the BackFill hook is used. Otherwise, the rectangle is deleted.

Parameters:

rp	RastPort
xmin,ymin	Upper left corner of rectangle
xmax,ymax	Lower right corner of rectangle

Flood **Fill an area**

Call:

```
error = Flood( rp, mode, x, y)
d0      -330(A6) a1 d2   d0 d1

BOOL   error
STRUCT RastPort rp
ULONG  mode
SHORT  x,y
```

Function: Fills an area of any complexity with the color or pattern set in the current draw mode.

Parameters: rp — RastPort with TmpRas

x,y — Starting point for fill

mode — Fill mode (0: through AOLPen, 1: only points with the color at x-y)

Result: 0 — Okay

InitArea — **Initialize AreaInfo vector matrix**

Call:

```
 InitArea( areainfo, buffer, maxvectors )
-282(A6)   a0         a1       d0

STRUCT AreaInfo *areainfo
APTR buffer
SHORT maxvectors
```

Function: Initializes the vector table for Area commands. The given buffer must have at least five bytes per vector. Remember that AreaEllipse() needs two vectors, and AreaEnd() needs one.

Parameters: areainfo — AreaInfo structure

buffer — Vector buffer (5*maxvectors+5)

maxvectors — Maximum vectors

Move — **Set coordinates for graphics output**

Call:

```
Move(     rp, x, y)
-240(A6) a1  d0 d1

STRUCT RastPort *rp
SHORT  x,y
```

Function: Sets the coordinates for graphics output in the RastPort.

Parameters: rp RastPort

x,y Coordinates

PolyDraw — Draw a line according to coordinates in a table

Call:

```
PolyDraw( rp, count , array )
-336(A6)  a1  d0      a0

STRUCT RastPort *rp
WORD   count
APTR   array
```

Function: Draws from point to point according to the values in a coordinate table. This function is the same as a Move() call to the first coordinates followed by subsequent Draw() calls.

Parameters: rp RastPort

count Number of coordinate points.

array Array with two words per entry (x and y).

ReadPixel — Read the color number of a pixel

Call:

```
penno = ReadPixel( rp, x, y )
d0      -318(A6)   a1  d0 d1

LONG   penno
STRUCT RastPort *rp
SHORT  x,y
```

Function: Gets the color number of the pixel at the given coordinates in a RastPort.

Parameters: rp RastPort

x,y Coordinates

Result: Color number (0..255) or -1 (coordinates outside of RastPort)

ReadPixelArray8	Read the color numbers of a rectangle

Call:

```
count = ReadPixelArray8(rp,xstart,ystart,xstop,ystop,array,temprp)
d0      -780(A6)        a0 d0:16  d1:16  d2:16 d3:16 a2    a1

LONG   count
STRUCT RastPort *rp,*temprp
UWORD  xstart,ystart,xstop,ystop
APTR   array
```

Function: ReadPixel() for each point within a rectangular area of a RastPort. The results are written as bytes to the given buffer.

Parameters:

rp — RastPort structure

xstart,ystart — Starting point in RastPort.

xstop,ystop — End point in RastPort.

array — Results buffer, at least ((((width+15)>>4)<<4)*(ystop-ystart+1)) bytes.

temprp — Temporary RastPort (copy with layer=0 and a bit-map that can store one row of the rectangular area).

Result: Number of pixels read.

Example: Read a 16*16 pixel area:

```
movea.l _GfxBase,a6
movea.l _RastPort,a0
moveq   #8,d0
moveq   #16,d1
moveq   #24,d2
moveq   #32,d3
lea     _Array,a2
lea     _TmpRp,a1
jsr     _LVOReadPixelArray8(a6)
```

```
 ...

_Array
 ds.b    16*16

 ...
_TmpRp   ;previously initialized
 ...
```

ReadPixelLine8 **Read color numbers of a horizontal line**

Call:

```
count = ReadPixelLine8(rp,xstart,ystart,width,array,temprp)
d0      -768(A6)        a0 d0:16  d1:16  d2    a2    a1

LONG   count
STRUCT RastPort *rp,*temprp
UWORD  xstart,ystart,width
APTR   array
```

Function: Like ReadPixelArray8(), but for only one line.

Parameters:

rp	RastPort
x,y	Starting point
width	Line width (in pixels)
array	Results buffer, at least (((width+15)>>4)<<4) bytes.
temprp	Same as with ReadPixelArray8().

RectFill **Fill a rectangle**

Call:

```
RectFill( rp, xmin, ymin, xmax, ymax)
-306(A6)  a1  d0:16 d1:16 d2:16 d3:16

STRUCT RastPort *rp
SHORT  xmin,ymin,xmax,ymax
```

Function: Fills a rectangle in a RastPort with the set color or pattern.

Parameters: rp RastPort

xmin,ymin Upper left corner of rectangle

xmax,ymax
Lower right corner of rectangle

SetAPen — **Set color for drawing**

Call:

```
SetAPen( rp, pen )
-342(A6) a1  d0

STRUCT RastPort *rp
UBYTE  pen
```

Function: Sets the foreground color for graphics operations.

Parameters: rp RastPort

pen Color number (0...255)

SetBPen — **Set the background color**

Call:

```
SetBPen( rp, pen )
-348(A6) a1  d0

STRUCT RastPort *rp
UBYTE  pen
```

Function: Sets the second color for graphics operations.

Parameters: rp RastPort

pen Color number (0...255)

SetDrMd — **Set draw mode**

Call:

```
SetDrMd( rp, mode )
-354(A6) a1  d0:8

STRUCT RastPort *rp
UBYTE  mode
```

Function: Sets the draw mode for drawing, text output, and filling areas.

Parameters: rp RastPort

mode JAM1, JAM2, etc.

Example: Output shaded text:

```
**===========================================================**
**       Shadow print                                        **
**-----------------------------------------------------------**
**       Input:  a1 = RastPort                               **
**               a0 = Text                                   **
**               d0 = Text color                             **
**               d1 = Shadow color                           **
**               d2 = xPos                                   **
**               d3 = yPos                                   **
**===========================================================**

_ShadowPrint
 movem.l d0-d4/a0-a1/a6,-(a7)
 movea.l _GfxBase,a6
 moveq   #RP_JAM1,d0
 jsr     _LVOSetDrMd(a6)    ;foreground color only

.StrLen
 moveq   #-1,d0
 sub.l   a0,d0
.StrLenLoop
 tst.b   (a0)+
 bne.s   .StrLenLoop
 add.l   a0,d0
 move.l  d0,d4

 move.l  4(a7),d0
 addq.w  #1,d2
 addq.w  #1,d3
 bsr.s   .GiveOut
 move.l  (a7),d0
 subq.w  #1,d2
 subq.w  #1,d3
 bsr.s   .GiveOut

 movem.l (a7)+,d0-d4/a0-a1/a6
 rts
```

```
.GiveOut
 movea.l 28(a7),a1
 jsr     _LVOSetAPen(a6)
 movea.l 28(a7),a1
 move.l  d2,d0
 move.l  d3,d1
 jsr     _LVOMove(a6)
 movem.l 24(a7),a0-a1
 move.l  d4,d0
 jmp     _LVOText(a6)
```

SetRast — **Fill an area with a color**

Call:

```
SetRast( rp, pen )
-234(A6) a1  d0

STRUCT RastPort *rp
UBYTE  pen
```

Function: Fills the RastPort with the given color.

Parameters:

rp	RastPort
pen	Color number (0...255)

WritePixel — **Draw a pixel**

Call:

```
error = WritePixel( rp, x, y)
d0      -324(A6)    a1  D0 D1

LONG   error
STRUCT RastPort *rp
SHORT  x,y
```

Function: Places a pixel at the given coordinates in the RastPort using the foreground color.

Parameters:

rp	RastPort
x,y	Pixel coordinates

Result: 0 Okay, -1=coordinates outside of RastPort.

WritePixelArray8 **Draw a multi-colored rectangle**

Call:

```
count = WritePixelArray8(rp,xstart,ystart,xstop,ystop,array,temprp)
d0      -786(A6)          a0 d0:16  d1:16  d2:16 d3:16 a2    a1

LONG   count
STRUCT RastPort *rp,*temprp
UWORD  xstart,ystart,xstop,ystop
APTR   *array
```

Function: Fills a rectangle with pixels. The color numbers are given in a byte field.

Parameters: See ReadPixelArray8().

Result: Number of pixels drawn.

Example: Write a 16*16 pixel area:

```
movea.l _GfxBase,a6
movea.l _RastPort,a0
moveq   #8,d0
moveq   #16,d1
moveq   #24,d2
moveq   #32,d3
lea     _Array,a2
lea     _TmpRp,a1
jsr     _LVOWritePixelArray8(a6)
...

_Array
ds.b    16*16 ;previously read, manipulated, etc.

...
_TmpRp  ;already initialized
...
```

WritePixelLine8 **Draw a multi-colored horizontal line**

Call:

```
count = WritePixelLine8(rp,xstart,ystart,width,array,temprp)
d0      -774(A6)          a0 d0:16  d1:16  d2    a2    a1

LONG   count
STRUCT RastPort *rp,*temprp
```

```
UWORD  xstart,ystart,width
APTR   array
```

Function: Draws a horizontal line with the color numbers given in a byte field.

Parameters: See ReadPixelLine8().

Result: Number of pixels drawn.

```
Dec Hex STRUCTURE TmpRas,0 ;temporary raster
  0  $0 APTR    tr_RasPtr   ;buffer
  4  $4 LONG    tr_Size     ;buffer size
  8  $8 LABEL   tr_SIZEOF

RPB_FRST_DOT     = 0, RPF_FRST_DOT     =  1 ;first pixel also
RPB_ONE_DOT      = 1, RPF_ONE_DOT      =  2 ;pixel line
RPB_DBUFFER      = 2, RPF_DBUFFER      =  4 ;double buffering
RPB_AREAOUTLINE = 3, RPF_AREAOUTLINE =  8 ;outline mode
RPB_NOCROSSFILL = 5, RPF_NOCROSSFILL = 16 ;AreaFill mode

RP_JAM1         = 0 ;without background
RP_JAM2         = 1 ;with background
RP_COMPLEMENT   = 2 ;complement
RP_INVERSVID    = 4 ;invert

RPB_TXSCALE = 0, RPF_TXSCALE = 1

Dec Hex STRUCTURE RastPort,0
  0  $0 LONG    rp_Layer
  4  $4 LONG    rp_BitMap
  8  $8 LONG    rp_AreaPtrn
 12  $C LONG    rp_TmpRas
 16 $10 LONG    rp_AreaInfo
 20 $14 LONG    rp_GelsInfo
 24 $18 BYTE    rp_Mask
 25 $19 BYTE    rp_FgPen
 26 $1A BYTE    rp_BgPen
 27 $1B BYTE    rp_AOLPen
 28 $1C BYTE    rp_DrawMode
 29 $1D BYTE    rp_AreaPtSz
 30 $1E BYTE    rp_linpatcnt
 31 $1F BYTE    rp_Dummy
 32 $20 WORD    rp_Flags
 34 $22 WORD    rp_LinePtrn
 36 $24 WORD    rp_cp_x
```

```
 38 $26 WORD   rp_cp_y
 40 $28 STRUCT rp_minterms,8
 48 $30 WORD   rp_PenWidth
 50 $32 WORD   rp_PenHeight
 52 $34 LONG   rp_Font
 56 $38 BYTE   rp_AlgoStyle
 57 $39 BYTE   rp_TxFlags
 58 $3A WORD   rp_TxHeight
 60 $3C WORD   rp_TxWidth
 62 $3E WORD   rp_TxBaseline
 64 $40 WORD   rp_TxSpacing
 66 $42 APTR   rp_RP_User
 70 $46 STRUCT rp_longreserved,8
 78 $4E STRUCT rp_wordreserved,14
 92 $5C STRUCT rp_reserved,8
100 $64 LABEL  rp_SIZEOF

Dec Hex STRUCTURE AreaInfo,0
  0  $0 LONG     ai_VctrTbl
  4  $4 LONG     ai_VctrPtr
  8  $8 LONG     ai_FlagTbl
 12  $C LONG     ai_FlagPtr
 16 $10 WORD     ai_Count
 18 $12 WORD     ai_MaxCount
 20 $14 WORD     ai_FirstX
 22 $16 WORD     ai_FirstY
 24 $18 LABEL    ai_SIZEOF
```

Example: A routine could calculate the color values for an apple man (fractal, Mandelbrot set) and store them in byte arrays, which can then be output at the end of a line with WritePixelLine8().

6. Text Output

AddFont — **Add a font to the system list**

Call:

```
AddFont(textFont)
-480(A6) a1

STRUCT TextFont *textFont
```

Function: Adds the given font to the system list.

Parameters: textFont TextFont structure

AskFont	**Get the attributes of the current font**

Call:

```
AskFont(rp, textAttr)
-474(A6) a1 a0

STRUCT RastPort *rp
STRUCT TextAttr *textAttr
```

Function: Fills the given TextAttr structure with information on the RastPort font.

Parameters: rp RastPort

textAttr TextAttr structure

AskSoftStyle	**Get the current style**

Call:

```
enable = AskSoftStyle(rp)
d0       -84(A6)      a1

ULONG enable
STRUCT RastPort *rp
```

Function: For the given RastPort, it returns the style currently being generated by the software.

Parameters: rp RastPort

Result: Style (bit mask, undefined bits are set)

ClearEOL	**Delete the rest of the line**

Call:

```
ClearEOL( rp )
-42(A6)   a1

STRUCT RastPort *rp
```

Function: Deletes the rest of a text line starting from the current position.

Parameters: rp RastPort

ClearScreen **Deletes the RastPort from the current position**

Call:

```
ClearScreen( rp )
-48(A6)      a1

STRUCT RastPort *rp
```

Function: Deletes the rest of the RastPort starting from the current text position (ClearEOL(), then continue down to the bottom edge).

Parameters: rp RastPort

CloseFont **Free a font**

Call:

```
CloseFont(font)
-78(A6)   a1

STRUCT TextFont *font
```

Function: Notifies the system that the given font is no longer being used.

Parameters: font Result from OpenFont()/OpenDiskFont()

ExtendFont **Create tf_Extension**

Call:

```
success = ExtendFont(font, fontTags)
D0        -816(A6)   A0    A1

ULONG success
STRUCT TextFont *font
STRUCT TagItem *fontTags
```

Function: Assure that tf_Extension is available.

Parameters: font TextFont structure

fontTags TagItem field

Result: 0 Error

FontExtent — **Get font attributes**

Call:

```
FontExtent(font, fontExtent)
-762(A6)    a0    a1

STRUCT TextFont *font
STRUCT TextExtent *fontExtent
```

Function: Fills the given FontExtent structure with information on the given font.

Parameters: font TextFont

fontExtent FontExtent structure to be filled.

OpenFont — **Open a font**

Call:

```
font = OpenFont(textAttr)
d0     -72(A6)  a0

STRUCT TextFont *font
STRUCT TextAttr *textAttr
```

Function: Searches the GfxLibrary list to find the font that most closely matches the data given in the TextAttr structure, and opens it.

Parameters: textAttr TextAttr or XTextAttr structure

Result: TextFont address if a font with the given name was found, or 0. Warning: the attributes of the returned font may not exactly match the requested attributes (different height, etc.).

RemFont — **Remove a font from the system list**

Call:

```
RemFont(textFont)
-486(A6) a1

STRUCT TextFont *textFont
```

Function: Removes a font from the system list (as long as it is no longer required).

Parameters: textFont TextFont structure for the font.

SetFont **Set font**

Call:

```
SetFont(rp, font)
-66(A6) a1  a0

STRUCT RastPort *rp
STRUCT TextFont *font
```

Function: Sets the font to be used by a RastPort. If the font does not conform with the standards, then an attempt is made to convert it to a usable format.

Parameters: rp RastPort

font Result from OpenFont() or OpenDiskFont().

SetSoftStyle **Set software style**

Call:

```
newStyle = SetSoftStyle(rp, style, enable)
d0         -90(A6)      a1  d0     d1

ULONG newStyle,style,enable
STRUCT RastPort *rp
```

Function: Changes the software style of the current font.

Parameters: rp RastPort

style New values for the bits.

enable Bit mask with bits to be changed.

Result: Value with the new style.

StripFont **Remove tf_Extension**

Call:

```
StripFont(font)
-822(A6)   A0

STRUCT TextFont *font
```

Function: Converts a 2.x font into a 1.x font.

Parameters: font TextFont structure for the font.

Text **Output a string**

Call:

```
Text(   rp, string, length)
-60(A6) a1  a0      d0

STRUCT RastPort *rp
APTR   string
WORD   length
```

Function: Outputs text to the current position in the RastPort.

Parameters: rp RastPort

string Address of first character of the output string.

length Number of characters.

TextExtent **Calculate dimensions of a text output**

Call:

```
TextExtent(rp, string, count, textExtent)
-690(A6)   a1  a0      d0:16  a2

STRUCT RastPort *rp
APTR   string
WORD   count
STRUCT TextExtent *textExtent
```

Function: Fills a TextExtent structure with the calculated dimensions of an output string.

Parameters: rp — RastPort

string — Address of first character

count — Number of characters

textExtent — Data structure for result

Result: Filled TextExtent structure

TextFit — Calculate proper text length

Call:

```
chars = TextFit(rp, s, sL, tE, cE, sD, cBW, cBH)
d0      -696(A6) a1 a0 d0  a2  a3  d1  d2   d3

ULONG  chars
STRUCT RastPort *rp
APTR   s
UWORD  sL,sD,cBW,cBH
STRUCT TextExtent *tE,cE
```

Function: Checks how many characters and returns a TextExtent structure of the proper length.

Parameters: rp — RastPort

s — String

sL — String length

tE — TextExtent structure for the result.

cE — Given TextExtent structure of 0.

sD — Offset from one character of the string to the next.

cBW — Bit width (alternative to cE)

cBH — Bit height (alternative to cE)

Result: Number of characters that will fit in the area (0 is possible).

TextLength — **Length of a text output in a RastPort**

Call:

```
length = TextLength(rp, string, count)
d0       -54(A6)    a1  a0      d0:16

WORD   length,count
STRUCT RastPort *rp
APTR   string
```

Function: Returns the length of a text output in pixels.

Parameters:

rp — RastPort

string — String address

count — String length

Result: Text length in pixels.

WeighTAMatch — **Compare fonts**

Call:

```
weight = WeighTAMatch(reqTextAttr, targetTextAttr, targetTags)
d0       -804(A6)     a0           a1              a2

WORD weight
STRUCT TTextAttr *reqTextAttr
STRUCT TextAttr *targetTextAttr
STRUCT TagItem *targetTags
```

Function: Compares two TextAttr structures and returns a value that describes how well they match. The best result is MAXFONTMATCHWEIGHT (perfect match), the worst case is 0. The names are not compared.

Parameters:

reqTextAttr
Desired TextAttribute

targetTextAttr
Potential TextAttribute

targetTags Extended attributes for targetTextAttr or 0.

Result: Match value (0...MAXFONTMATCHWEIGHT)

```
FS_NORMAL = 0 ;normal style
FSB_UNDERLINED = 0, FSF_UNDERLINED =   1 ;underline
FSB_BOLD       = 1, FSF_BOLD       =   2 ;bold
FSB_ITALIC     = 2, FSF_ITALIC     =   4 ;italics
FSB_EXTENDED   = 3, FSF_EXTENDED   =   8 ;extended
FSB_COLORFONT  = 6, FSF_COLORFONT  = $40 ;colored
FSB_TAGGED     = 7, FSF_TAGGED     = $80 ;TTextAttr

FPB_ROMFONT      = 0, FPF_ROMFONT      =   1 ;font from ROM
FPB_DISKFONT     = 1, FPF_DISKFONT     =   2 ;font from disk
FPB_REVPATH      = 2, FPF_REVPATH      =   4 ;change output direction
FPB_TALLDOT      = 3, FPF_TALLDOT      =   8 ;HiRes font
FPB_WIDEDOT      = 4, FPF_WIDEDOT      = $10 ;LoRes Interlace font
FPB_PROPORTIONAL = 5, FPF_PROPORTIONAL = $20 ;proportional font
FPB_DESIGNED     = 6, FPF_DESIGNED     = $40 ;designed (not derived)
FPB_REMOVED      = 7, FPF_REMOVED      = $80 ;not available

Dec Hex STRUCTURE TextAttr,0
  0  $0 APTR    ta_Name  ;font name
  4  $4 UWORD   ta_YSize ;height
  6  $6 UBYTE   ta_Style ;style
  7  $7 UBYTE   ta_Flags ;preference Flags
  8  $8 LABEL   ta_SIZEOF

Dec Hex STRUCTURE TTextAttr,0
  0  $0 APTR    tta_Name  ;font name
  4  $4 UWORD   tta_YSize ;height
  6  $6 UBYTE   tta_Style ;style
  7  $7 UBYTE   tta_Flags ;preference Flags
  8  $8 APTR    tta_Tags  ;TagItem field
 12  $C LABEL   tta_SIZEOF

TA_DeviceDPI = TAG_USER!1 ;XDPI<<16!YDPI

MAXFONTMATCHWEIGHT = 32767 ;perfect match

Dec Hex STRUCTURE TextFont,MN_SIZE ;font
 20 $14 UWORD tf_YSize             ;height
 22 $16 UBYTE tf_Style             ;style
 23 $17 UBYTE tf_Flags             ;preference Flags
 24 $18 UWORD tf_XSize             ;normal width
 26 $1A UWORD tf_Baseline          ;base line
 28 $1C UWORD tf_BoldSmear         ;bold value
```

```
 30 $1E UWORD tf_Accessors         ;number of users
 32 $20 UBYTE tf_LoChar            ;first character
 33 $21 UBYTE tf_HiChar            ;last character
 34 $22 APTR  tf_CharData          ;packed BitImages
 38 $26 UWORD tf_Modulo            ;bytes per line of CharData
 40 $28 APTR  tf_CharLoc           ;offsets and character widths
 44 $2C APTR  tf_CharSpace         ;proportional spaces
 48 $30 APTR  tf_CharKern          ;image offsets
 52 $34 LABEL tf_SIZEOF

tf_Extension = MN_REPLYPORT

TE0B_NOREMFONT = 0, TE0F_NOREMFONT = 1 ;not removable

Dec Hex STRUCTURE TextFontExtension,0 ;read only!
  0  $0 UWORD tfe_MatchWord           ;ID for compatibility
  2  $2 UBYTE tfe_Flags0              ;system Flags (TE0..)
  3  $3 UBYTE tfe_Flags1              ;
  4  $4 APTR  tfe_BackPtr             ;check address
  8  $8 APTR  tfe_OrigReplyPort       ;old contents of tf_Extension
 12  $C APTR  tfe_Tags                ;TagItem field
 16 $10 APTR  tfe_OFontPatchS         ;private
 20 $14 APTR  tfe_OFontPatchK         ;private
 24 $18 LABEL tfe_SIZEOF

CT_COLORFONT = 1 ;color values are set
CT_GREYFONT  = 2 ;grey scale only (dark to light)
CT_ANTIALIAS = 4 ;AntiAliasing

CTB_MAPCOLOR = 0, CTF_MAPCOLOR = 1 ;set rp_FgPen first

Dec Hex STRUCTURE ColorFontColors,0
  0  $0 UWORD   cfc_Reserved   ;0!!
  2  $2 UWORD   cfc_Count      ;number of color values
  4  $4 APTR    cfc_ColorTable ;color table $xRGB
  8  $8 LABEL   cfc_SIZEOF

Dec Hex STRUCTURE ColorTextFont,tf_SIZEOF
 52 $34 UWORD   ctf_Flags           ;additional Flags
 54 $36 UBYTE   ctf_Depth           ;number of BitPlanes
 55 $37 UBYTE   ctf_FgColor         ;rp_FgPen
 56 $38 UBYTE   ctf_Low             ;lowest color
 57 $39 UBYTE   ctf_High            ;highest color
 58 $3A UBYTE   ctf_PlanePick       ;ImagePlanes
 59 $3B UBYTE   ctf_PlaneOnOff      ;BitMap mask
 60 $3C APTR    ctf_ColorFontColors ;colors
 64 $40 STRUCT  ctf_CharData,8*4    ;BitPlanePointer
 96 $60 LABEL   ctf_SIZEOF
```

```
Dec Hex STRUCTURE TextExtent,0
  0  $0 UWORD   te_Width     ;TextLength
  2  $2 UWORD   te_Height    ;tf_YSize
  4  $4 STRUCT  te_Extent,8 ;MinX,MinY,MaxX,MaxY (relative)
 12  $C LABEL   te_SIZEOF
```

7. Movable Objects

AddAnimOb	**Add AnimOb to RastPort list**

Call:

```
AddAnimOb(anOb, anKey, rp)
-156(A6)  a0     a1    a2

STRUCT AnimOb *anOb,**anKey
STRUCT RastPort *rp
```

Function: Adds an AnimOb structure to the given list and initializes the Timer values of the structure. GelsInfo for the RastPort must be initialized.

Parameters:

anOb — AnimOb structure

anKey — Address of the address of the first AnimOb.

rp — RastPort of the AnimOb.

AddBob	**Add a bob structure to the GEL list**

Call:

```
AddBob(Bob, rp)
-96(A6) a0  a1

STRUCT Bob *Bob
STRUCT RastPort *rp
```

Function: Adds the given Blitter object to the RastPort's list.

Parameters:

Bob — Blitter object

rp — RastPort of the bob

AddVSprite — **Add a virtual sprite to the GEL list**

Call:

```
AddVSprite(vs, rp)
-102(A6)    a0  a1

STRUCT VSprite *vs
STRUCT RastPort *rp
```

Function: Adds a VSprite structure to the RastPort's list.

Parameters:

vs	VSprite
rp	RastPort

Animate — **Move AnimObs**

Call:

```
Animate(anKey, rp)
-162(A6) a0    a1

STRUCT AnimOb **anKey
STRUCT RastPort *rp
```

Function: Animates all AminObs and their components.

Parameters:

anKey	Address of the pointer to the first AnimOb.
rp	RastPort of the AnimOb.

ChangeSprite — **Change a sprite**

Call:

```
ChangeSprite( vp, s, newdata)
-420(A6)      a0  a1 a2

STRUCT ViewPort *vp
STRUCT SimpleSprite *s
APTR   newdata
```

Function: Changes the appearance of a sprite.

Parameters:

vp	ViewPort of the sprite or 0 (=relative to start of display).

s	Address of the SimpleSprite structure.
newdata	Address (ChipRAM) of the new hardware sprite data list.

DoCollision — Check elements of the GEL list for collisions

Call:

```
DoCollision(rp)
-108(A6)    a1

STRUCT RastPort *rp
```

Function: Checks every movable object for border and object collisions and calls the GEL collision routine if one is found.

Parameters: rp — RastPort with sorted GEL list (see SortGList()).

DrawGList — Display movable objects

Call:

```
DrawGList(rp, vp)
-114(A6)  a1  a0

STRUCT RastPort *rp
STRUCT ViewPort *vp
```

Function: Calculates a new Copper list for sprites and draws bobs.

Parameters:

rp	RastPort of the bob.
vp	ViewPort of the VSprite.

FreeGBuffers — Free the AminOb component buffers

Call:

```
FreeGBuffers(anOb, rp, db)
-600(A6)      a0    a1  d0

STRUCT AnimOb *anOb
STRUCT RastPort *rp
BOOL   db
```

Function: Frees all buffers of all AnimOb components (SaveBuffer, BorderLine, CollMask=ImageShadow). If desired, double buffering memory (DBufPacket, BufBuffer) is also set free.

Parameters:

anOb	AnimOb
rp	RastPort
db	Flag for double buffering (TRUE).

FreeSprite — **Free hardware sprite**

Call:

```
FreeSprite( pick )
-414(A6)    d0

WORD pick
```

Function: Frees a hardware sprite for use by other programs.

Parameters:

pick	Sprite number (0...7)

GetGBuffers — **Allocate all buffers for an AnimOb**

Call:

```
status = GetGBuffers(anOb, rp, db)
d0       -168(A6)    a0    a1  d0

BOOL status,db
STRUCT AnimOb *anOb
STRUCT RastPort *rp
```

Function: Attempts to allocate all memory for the components of an AnimOb (SaveBuffer, BorderLine, CollMask=ImageShadow). Memory for double buffering (DBufPacket, BufBuffer) is also allocated if indicated. If an error occurs, memory already allocated is not set free.

Parameters:

anOb	AnimOb structure
rp	RastPort of the AnimOb.
db	Flag for double buffering (TRUE).

Result:	0	Error

GetSprite — **Allocate a hardware sprite**

Call:

```
Sprite_Number = GetSprite( sprite, pick )
d0              -408(A6)   a0      d0

SHORT  Sprite_Number,pick
STRUCT SimpleSprite *sprite
```

Function: Attempts to allocate one of the 8 hardware sprites.

Parameters:

sprite	SimpleSprite structure for the sprite.
pick	Sprite number (0...7) or -1 (any Sprite).

Result: Sprite number of the allocated sprite or -1 (already in use/none free).

InitGels — **Initialize GELs**

Call:

```
InitGels(head, tail, GInfo)
-120(A6) a0    a1    a2

STRUCT VSprite *head,*tail
STRUCT GelsInfo *GInfo
```

Function: Links the VSprite structures to the GfxBase.

Parameters:

head	Start of list
tail	End of list
GInfo	GelsInfo structure to be initialized.

InitGMasks — **Initialize AnimOb mask**

Call:

```
InitGMasks(anOb)
-174(A6)   a0

STRUCT AnimOb *anOb
```

Function: Calculate and enter the mask values for an AnimOb.

Parameters: anOb AnimOb

InitMasks **Initialize VSprite mask**

Call:

```
InitMasks(vs)
-126(A6)  a0

STRUCT VSprite *vs
```

Function: Calculates BorderLine and CollMask for a VSprite/bob.

Parameters: vs VSprite structure of the object.

MoveSprite **Move a hardware sprite**

Call:

```
MoveSprite( vp, sprite, x, y )
-426(A6)    a0  a1      d0 d1

STRUCT ViewPort *vp
STRUCT SimpleSprite *sprite
SHORT  x,y
```

Function: Positions a hardware sprite relative to the ViewPort.

Parameters:

vp	ViewPort of the sprite or 0 (relative to View).
sprite	SimpleSprite structure
x,y	Position (x-coordinate +1)

RemIBob **Remove a bob from the RastPort list**

Call:

```
RemIBob(bob, rp, vp)
-132(A6) a0  a1  a2

STRUCT Bob *bob
STRUCT RastPort *rp
STRUCT ViewPort *vp
```

Function: Removes a bob from the RastPort's GEL list.

Parameters: bob — Blitter object to remove.

rp — RastPort

vp — ViewPort for raster synchronization.

RemVSprite — **Remove a VSprite from the RastPort list**

Call:
```
RemVSprite(vs)
-138(A6)   a0

STRUCT VSprite *vs
```

Function: Removes a VSprite from the GEL list of the RastPort.

Parameters: vs — VSprite

SetCollision — **Set the collision routine**

Call:
```
SetCollision(num, routine, GInfo)
-144(A6)     d0   a0        a1

ULONG  num
APTR   routine
STRUCT GelsInfo *GInfo
```

Function: Sets the collision routine for an entry.

Parameters: num — Number of entries

routine — Collision routine

GInfo — GelsInfo

SortGList — **Sort list of movable objects**

Call:
```
SortGList(rp)
-150(A6)  a1
```

Function: Sorts the objects list by y-x coordinates.

Parameters: rp — RastPort with GelsInfo

```
SUSERFLAGS   = $00FF ;mask for User VSprite Flags
VSB_VSPRITE     =  0, VSF_VSPRITE     =    1 ;VSprite, ~BOb
VSB_SAVEBACK    =  1, VSF_SAVEBACK    =    2 ;save background
VSB_OVERLAY     =  2, VSF_OVERLAY     =    4 ;mask
VSB_MUSTDRAW    =  3, VSF_MUSTDRAW    =    8 ;draw
VSB_BACKSAVED   =  8, VSF_BACKSAVED   = $100 ;background
VSB_BOBUPDATE   =  9, VSF_BOBUPDATE   = $200 ;update BOb
VSB_GELGONE     = 10, VSF_GELGONE     = $400 ;outside of View
VSB_VSOVERFLOW  = 11, VSF_VSOVERFLOW  = $800 ;overflow

BUSERFLAGS   = $00FF ;mask for User BOb Flags
BB_SAVEBOB      =  0, BF_SAVEBOB      =    1 ;do not delete
BB_BOBISCOMP    =  1, BF_BOBISCOMP    =    2 ;AnimOb component
BB_BWAITING     =  8, BF_BWAITING     = $100 ;BOb waiting
BB_BDRAWN       =  9, BF_BDRAWN       = $200 ;BOb drawn
BB_BOBSAWAY     = 10, BF_BOBSAWAY     = $400 ;remove BOb
BB_BOBNIX       = 11, BF_BOBNIX       = $800 ;BOb gone
BB_SAVEPRESERVE= 12, BF_SAVEPRESERVE=$1000 ;background from Dbuf
BB_OUTSTEP      = 13, BF_OUTSTEP      =$2000 ;clear Dbuf

ANFRACSIZE  =     6 ;animation Flags
ANIMHALF    = $0020
RINGTRIGGER = $0001

Dec Hex STRUCTURE VS,0 ;vSprite
  0  $0 APTR  vs_NextVSprite   ;next structure
  4  $4 APTR  vs_PrevVSprite   ;previous structure
  8  $8 APTR  vs_DrawPath      ;overlay vSprite
 12  $C APTR  vs_ClearPath     ;delete vSprite
 16 $10 WORD  vs_Oldy          ;old position
 18 $12 WORD  vs_Oldx          ;
 20 $14 WORD  vs_VSFlags       ;vSprite Flags
 22 $16 WORD  vs_Y             ;position
 24 $18 WORD  vs_X             ;
 26 $1A WORD  vs_Height        ;height
 28 $1C WORD  vs_Width         ;width in Words
 30 $1E WORD  vs_Depth         ;number of BitPlanes
 32 $20 WORD  vs_MeMask        ;collision mask
 34 $22 WORD  vs_HitMask       ;collision mask
 36 $24 APTR  vs_ImageData     ;image
 40 $28 APTR  vs_BorderLine    ;mask of all bits
 44 $2C APTR  vs_CollMask      ;collision image
 48 $30 APTR  vs_SprColors     ;Sprite colors
 52 $34 APTR  vs_VSBob         ;BOb
 56 $38 BYTE  vs_PlanePick     ;BitPlane mask image
 57 $39 BYTE  vs_PlaneOnOff    ;same for other planes
 58 $3A LABEL vs_SUserExt      ;start of user extension
 58 $3A LABEL vs_SIZEOF
```

```
Dec Hex STRUCTURE BOB,0          ;Blitter object
  0  $0 WORD    bob_BobFlags     ;Flags
  2  $2 APTR    bob_SaveBuffer   ;background buffer
  6  $6 APTR    bob_ImageShadow  ;Image mask
 10  $A APTR    bob_Before       ;previous Bob
 14  $E APTR    bob_After        ;next Bob
 18 $12 APTR    bob_BobVSprite   ;vSprite structure
 22 $16 APTR    bob_BobComp      ;AnimComp
 26 $1A APTR    bob_DBuffer      ;dBufPacket
 30 $1E LABEL   bob_BUserExt
 30 $1E LABEL   bob_SIZEOF

Dec Hex STRUCTURE AC,0           ;AnimComp
  0  $0 WORD    ac_CompFlags     ;Flags
  2  $2 WORD    ac_Timer         ;activation time
  4  $4 WORD    ac_TimeSet       ;start time
  6  $6 APTR    ac_NextComp      ;next component
 10  $A APTR    ac_PrevComp      ;previous component
 14  $E APTR    ac_NextSeq       ;next sequence
 18 $12 APTR    ac_PrevSeq       ;previous sequence
 22 $16 APTR    ac_AnimCRoutine  ;animation routine
 26 $1A WORD    ac_YTrans        ;y translation
 28 $1C WORD    ac_XTrans        ;x translation
 30 $1E APTR    ac_HeadOb        ;AnimOb
 34 $22 APTR    ac_AnimBob       ;BOB
 38 $26 LABEL   ac_SIZE

Dec Hex STRUCTURE  AO,0           ;AnimOb
  0  $0 APTR     ao_NextOb        ;next AnimOb
  4  $4 APTR     ao_PrevOb        ;previous AnimOb
  8  $8 LONG     ao_Clock         ;number of Animate()
 12  $C WORD     ao_AnOldY        ;old position
 14  $E WORD     ao_AnOldX        ;
 16 $10 WORD     ao_AnY           ;position
 18 $12 WORD     ao_AnX           ;
 20 $14 WORD     ao_YVel          ;velocity
 22 $16 WORD     ao_XVel          ;
 24 $18 WORD     ao_XAccel        ;acceleration
 26 $1A WORD     ao_YAccel        ;
 28 $1C WORD     ao_RingYTrans    ;ring translation
 30 $1E WORD     ao_RingXTrans    ;
 32 $20 APTR     ao_AnimORoutine  ;animation routine
 36 $24 APTR     ao_HeadComp      ;AnimComp
 40 $28 LABEL    ao_AUserExt
 40 $28 LABEL    ao_SIZEOF
```

```
Dec Hex STRUCTURE DBP,0         ;dBufPacket
  0  $0 WORD     dbp_BufY       ;screen position
  2  $2 WORD     dbp_BufX       ;
  4  $4 APTR     dbp_BufPath    ;vSprite
  8  $8 APTR     dbp_BufBuffer  ;buffer
 12  $C APTR     dbp_BufPlanes  ;background PlanePointer
 16 $10 LABEL    dbp_SIZEOF

Dec Hex STRUCTURE GelsInfo,0
  0  $0 BYTE    gi_sprRsrvd     ;Sprite numbers
  1  $1 BYTE    gi_Flags        ;Flags
  2  $2 APTR    gi_gelHead      ;start of list
  6  $6 APTR    gi_gelTail      ;end of list
 10  $A APTR    gi_nextLine     ;Sprite lines
 14  $E APTR    gi_lastColor    ;color field
 18 $12 APTR    gi_collHandler  ;collision routine
 22 $16 WORD    gi_leftmost
 24 $18 WORD    gi_rightmost
 26 $1A WORD    gi_topmost
 28 $1C WORD    gi_bottommost
 30 $1E APTR    gi_firstBlissObj
 34 $22 APTR    gi_lastBlissObj
 38 $26 LABEL   gi_SIZEOF

Dec Hex STRUCTURE SimpleSprite,0
  0  $0 APTR         ss_posctldata
  4  $4 WORD         ss_height
  6  $6 WORD         ss_x
  8  $8 WORD         ss_y
 10  $A WORD         ss_num
 12  $C LABEL        ss_SIZEOF
```

3.1.9 The Icon Library

The "icon.library" is used to process '.info' files. So, the base address must be given in A6 with the function calls.

Functions of the Icon Library

AddFreeList
BumpRevision
FindToolType
FreeDiskObject
FreeFreeList
GetDefDiskObject

GetDiskObject
GetDiskObjectNew
MatchToolValue
PutDefDiskObject
PutDiskObject

Description of Functions

AddFreeList	**Add memory to FreeList**

Call:

```
status = AddFreeList(free, mem, len)
D0        -72(A6)      A0    A1   A2

BOOL   status
STRUCT FreeList *free
APTR   mem
ULONG  len
```

Function: Adds the given memory block to a FreeList.

Parameters:

free	FreeList
mem	Memory address
len	Size of block

Result: 0 Error

BumpRevision	**Create a filename for a copy**

Call:

```
result = BumpRevision(newbuf, oldname)
D0        -108(A6)      A0      A1

APTR result,newbuf,oldname
```

Function: Creates a filename with "copy_of_" etc.

Parameters:

newbuf	New name (copy_of_...), min. 31 bytes
oldname	Original filename

Result: Address of the new name or 0.

FindToolType — **Find the value of a ToolType variable**

Call:

```
value = FindToolType(toolTypeArray, typeName)
D0      -96(A6)      A0             A1

APTR value,toolTypeArray,typeName
```

Function: Finds a ToolType field according to the contents of the given variable.

Parameters: toolTypeArray
String field (APTRs)

typeName Variable name

Result: Address in ToolType string after the typeName equals sign or 0.

FreeDiskObject — **Free an icon's memory**

Call:

```
FreeDiskObject(diskobj)
-90(A6)        A0

STRUCT DiskObject *diskobj
```

Function: Frees the memory used by an icon allocated with GetDiskObject().

Parameters: diskobj DiskObject structure

FreeFreeList — **Free the FreeList**

Call:

```
FreeFreeList(free)
-54(A6)      A0

STRUCT FreeList *free
```

Function: Frees the memory entered in an icon's FreeList, including the memory for the FreeList structure.

Parameters: free FreeList

GetDefDiskObject — **Get default DiskObject**

Call:

```
diskobj = GetDefDiskObject(def_type)
D0        -120(A6)         D0

STRUCT DiskObject *diskobj
LONG   def_type
```

Function: Reads the default Workbench icon for an object of the given type.

Parameters: def_type Icon type

Result: DiskObject or 0

GetDiskObject — **Get an icon file**

Call:

```
diskobj = GetDiskObject(name)
D0        -78(A6)       A0

STRUCT DiskObject *diskobj
APTR   name
```

Function: Reads the icon of the given object.

Parameters: name Object name or 0 (empty structure).

Result: DiskObject or 0

GetDiskObjectNew — **Get a new icon file**

Call:

```
diskobj = GetDiskObjectNew(name)
D0        -132(A6)         A0

STRUCT DiskObject *diskobj
APTR   name
```

Functions, Parameters, Results:
Same as GetDiskObject(), except that if no ".info" file is available, an attempt is made to get the default settings with GetDefDiskObject().

MatchToolValue — **Compare ToolType variable**

Call:
```
result = MatchToolValue(typeString, value)
D0       -102(A6)       A0          A1

BOOL  result
APTR  typeString,value
```

Function: Compares a string with a ToolType variable value (may be several values separated with '|').

Parameters: typeString ToolType values as from FindToolType().

value Comparison string

Result: 0 Value was not in typeString.

PutDefDiskObject — **Set a Workbench icon**

Call:
```
status = PutDefDiskObject(diskobj)
D0       -126(A6)         A0

BOOL   status
STRUCT DiskObject *diskobj
```

Function: Changes the standard Workbench icon for the given DiskObject type.

Parameters: diskobj DiskObject

Result: 0 Error

PutDiskObject

Call:
```
status = PutDiskObject(name, diskobj)
D0       -84(A6)       A0    A1

BOOL   status
APTR   name
STRUCT DiskObject *diskobj
```

Function: Writes an icon file to disk.

Parameters:	name	Filename
	diskobj	DiskObject
Result:	0	Error

Structures: See Workbench Library.

3.1.10 The IFFParse Library

The IFF file format became an Amiga standard very quickly. Today, all sound and graphics programs use it. A standard file format makes it simple to transfer data from one program to another. The "iffparse.library" offers you the easiest way to introduce this standard to your own programs. All functions are called with the base address in A6.

Functions of the IFFParse Library

1. Base Functions

AllocIFF
CloseClipboard
CloseIFF
FreeIFF
GoodID
GoodType
IDtoStr
InitIFF
InitIFFasDOS
InitIFFasClip
OpenClipboard
OpenIFF
ParseIFF
ReadChunkBytes
ReadChunkRecords
WriteChunkBytes
WriteChunkRecords

2. Context

CurrentChunk
FindCollection
FindProp
FindPropContext
ParentChunk
PopChunk
PushChunk

3. Handlers

CollectionChunk
CollectionChunks
EntryHandler
ExitHandler
PropChunk
PropChunks
StopChunk
StopChunks
StopOnExit

4. Local ContextItems

AllocLocalItem
FindLocalItem
FreeLocalItem

LocalItemData | StoreItemInContext
SetLocalItemPurge | StoreLocalItem

Description of the Functions

1. Base Functions

AllocIFF	**Allocate an IFFHandle**

Call:

```
iff = AllocIFF ()
d0    -30(A6)

STRUCT IFFHandle *iff
```

Function: Allocates and initializes an IFFHandle structure.

Result: IFFHandle or 0

CloseClipboard	**Close ClipboardHandle**

Call:

```
CloseClipboard (clip)
-252(A6)        a0

STRUCT ClipboardHandle *clip
```

Function: Closes the "clipboard.device" and frees the ClipboardHandle structure.

Parameters: clip — ClipboardHandle structure from OpenClipboard().

CloseIFF	**Close IFF**

Call:

```
CloseIFF (iff)
-48(A6)   a0

STRUCT IFFHandle *iff
```

Function: Closes an IFF file, leaving the IFFHandle structure intact for a new OpenIFF() call.

Parameters: iff — IFFHandle structure from OpenIFF().

FreeIFF	**Free IFFHandle**

Call:

```
FreeIFF (iff)
-54(A6)  a0

STRUCT IFFHandle *iff
```

Function: Frees an IFFHandle that was closed with CloseIFF().

Parameters: iff   IFFHandle structure

GoodID	**Check IFF ID**

Call:

```
isok = GoodID (id)
d0     -258(A6) d0

LONG isok, id
```

Function: Checks to see if a chunk ID conforms with the Electronic Arts IFF 85 standard.

Parameters: id   32 bit ChunkID

Result: 0   ID not valid

GoodType	**Check FORM type**

Call:

```
isok = GoodType (type)
d0      -264(A6)  d0

LONG isok, type
```

Function: Checks to see if the ID is a type of FORM chunk (EA IFF 85).

Parameters: type   32 bit FORMat chunk ID

Result: 0   Not a FORM type

IDtoStr — **Store ID as a string**

Call:

```
str = IDtoStr (id, buf)
d0    -270(A6) d0  a0

APTR  str,buf
LONG  id
```

Function: Writes the ID (longword) to the given buffer and deletes the following byte.

Parameters:

id	Longword
buf	5 byte buffer

Result: Buffer

InitIFF — **Initialize IFFHandle as UserStream**

Call:

```
InitIFF (iff, flags, streamhook)
-228(A6) a0   d0     a1

STRUCT IFFHandle *iff
LONG   flags
STRUCT Hook  *streamhook
```

Function: Initializes IFFHandle with the user routines for positioning, reading, and writing. The hook routines are passed to the Hook, IFFStreamCmd, and IFFHandle structures in registers A0-A2.

Parameters:

iff	IFFHandle structure
flags	I/O flags
hook	Hook with stream routine.

InitIFFasClip — **Initialize IFFHandle as ClipboardStream**

Call:

```
InitIFFasClip (iff)
-240(A6)       a0
```

```
STRUCT IFFHandle *iff
```

Function: Initializes an IFFHandle for the "clipboard.device". Another ClipboardHandle from OpenClipboard() must be entered in iff_Stream.

Parameters: iff IFFHandle

InitIFFasDOS **Initialize IFFHandle as DOSStream**

Call:

```
InitIFFasDOS (iff)
-234(A6)      a0

STRUCT IFFHandle *iff
```

Function: Initializes an IFFHandle for DOS. Another FileHandle from Open() must be entered in iff_Stream (BPTR).

Parameters: iff IFFHandle structure

OpenClipboard **Get ClipboardHandle**

Call:

```
ch = OpenClipboard (unit)
d0   -246(A6)      d0

STRUCT ClipboardHandle  *ch
LONG   unit
```

Function: Opens the given unit of the "clipboard.device" (normally PRIMARY_CLIP) and returns a structure for InitIFFasClip().

Parameters: unit "clipboard.device" unit

Result: ClipboardHandle or 0

OpenIFF **Prepare IFFHandle for I/O**

Call:

```
error = OpenIFF (iff, rwmode)
d0      -36(A6)  a0   d0

LONG   error,rwmode
```

```
STRUCT IFFHandle *iff
```

Function: Initializes an IFFHandle structure for reading or writing (IFFF_READ or IFFF_WRITE).

Parameters:

iff	IFFHandle
rwmode	IFFF_READ or IFFF_WRITE

Result: Error code or 0

ParseIFF — **Analyze IFF**

Call:

```
error = ParseIFF (iff, control)
d0      -42(A6)   a0   d0

LONG   error,control
STRUCT IFFHandle *iff
```

Function: Reads an IFF file, puts the chunks on the context stack, and retrieves them in the correct order. The corresponding chunk type handler is called.

Parameters:

iff	IFFHandle structure
control	IFFPARSE_SCAN, IFFPARSE_STEP, or IFFPARSE_RAWSTEP

Result: Error code or 0

ReadChunkBytes — **Read bytes of the current chunk**

Call:

```
actual = ReadChunkBytes (iff, buf, size)
d0       -60(A6)         a0   a1   d0

LONG   actual,size
STRUCT IFFHandle *iff
APTR   buf
```

Function: Reads the given number of bytes from IFFHandle to the buffer.

Parameters: iff — IFFHandle

buf — Read buffer

size — Number of bytes

Result: Number of bytes read or negative (-error code).

ReadChunkRecords — **Read structures of the current chunk**

Call:

```
actual = ReadChunkRecords (iff, buf, recsize, numrec)
d0       -72(A6)           a0   a1   d0       d1

LONG   actual,recsize,numrec
STRUCT IFFHandle *iff
APTR   buf
```

Function: Reads numrec structures of length recsize to the buffer.

Parameters: iff — IFFHandle

buf — Read buffer

recsize — Size of structure

numrec — Number of structures

Result: Number or negative (-error code)

WriteChunkBytes — **Write to the current chunk**

Call:

```
error = WriteChunkBytes (iff, buf, size)
d0      -66(A6)          a0   a1   d0
```

Function: Writes size bytes to the current chunk.

Parameters: iff — IFFHandle

buf — Write buffer

size — Buffer size

Result: Number of written bytes or negative (-error code).

WriteChunkRecords — Write data structures to chunk

Call:

```
error = WriteChunkRecords (iff, buf, recsize, numrec)
d0      -78(A6)           a0   a1   d0       d1

LONG   error,recsize,numrec
STRUCT IFFHandle *iff
APTR   buf
```

Function: Writes numrec structures of size recsize to the current chunk.

Parameters:

iff	IFFHandle
buf	Buffer
recsize	Structure size
numrec	Number of structures

Result: Number or negative (-error code)

```
Dec Hex STRUCTURE IFFHandle,0
  0  $0 ULONG iff_Stream
  4  $4 ULONG iff_Flags
  8  $8 LONG  iff_Depth  ;stack depth
 12  $C LABEL iff_SIZEOF ;not end of structure!!!

iff_Flags:
IFFF_READ     = 0         ;read
IFFF_WRITE    = 1         ;write
IFFF_FSEEK    = 2         ;forward only
IFFF_RSEEK    = 4         ;any direction
IFFF_RESERVED = $FFFF0000 ;important system bits

Dec Hex STRUCTURE ClipboardHandle,iocr_SIZEOF ; cbh_Reg
 52 $34 STRUCT cbh_CBport,MP_SIZE
 86 $56 STRUCT cbh_SatisfyPort,MP_SIZE
120 $78 LABEL  cbh_SIZEOF

IFFERR_EOF        =  -1 ;end of file
IFFERR_EOC        =  -2 ;end of context
```

```
IFFERR_NOSCOPE    =  -3 ;invalid values for PROPs
IFFERR_NOMEM      =  -4 ;no free memory
IFFERR_READ       =  -5 ;read error
IFFERR_WRITE      =  -6 ;write error
IFFERR_SEEK       =  -7 ;seek error
IFFERR_MANGLED    =  -8 ;defective data in file
IFFERR_SYNTAX     =  -9 ;IFF syntax error
IFFERR_NOTIFF     = -10 ;not an IFF file
IFFERR_NOHOOK     = -11 ;no Hook
IFF_RETURN2CLIENT = -12 ;return to program

ID_FORM = 'FORM'
ID_LIST = 'LIST'
ID_CAT  = 'CAT '
ID_PROP = 'PROP'
ID_NULL = '    '

IFFPARSE_SCAN    = 0
IFFPARSE_STEP    = 1
IFFPARSE_RAWSTEP = 2
```

2. Context

CurrentChunk	**ContextNode of the current chunk**

Call:

```
top = CurrentChunk (iff)
d0    -174(A6)      a0

STRUCT ContextNode  *top
STRUCT IFFHandle *iff
```

Function: Returns the current ContextNode of the IFFHandle.

Parameters: iff IFFHandle

Result: ContextNode or 0

FindCollection	**Get CollectionItem list**

Call:

```
ci = FindCollection (iff, type, id)
d0   -162(A6)        a0   d0    d1

STRUCT CollectionItem   *ci
STRUCT IFFHandle *iff
LONG   type, id
```

Function: Gets the address of a list of CollectionItem structures of the given chunk type.

Parameters: iff IFFHandle

type Type

id ID

Result: Address of the last collection chunk.

FindProp **Find StoredProperty for a context**

Call:

```
sp = FindProp (iff, type, id)
d0   -156(A6)  a0    d0    d1

STRUCT StoredProperty *sp
STRUCT IFFHandle *iff
LONG   type, id
```

Function: Finds the StoredProperty structure set with PropChunk() or PropChunks(), which was automatically created by ParseIFF().

Parameters: iff IFFHandle

type FORM type (for example "ILBM")

id ChunkID (for example "CMAP")

Result: StoredProperty or 0

FindPropContext **Find ContextNode of the StoredProperty**

Call:

```
cn = FindPropContext (iff)
d0   -168(A6)          a0

STRUCT ContextNode  *cn
STRUCT IFFHandle *iff
```

Function: Retrieves the ContextNode, which is contained as the highest level of the current position, for example "FORM".

Parameters: iff IFFHandle

Result: ContextNode

ParentChunk **Get the ContextNode of the next higher level**

Call:

```
parent = ParentChunk (cn)
d0       -180(A6)     a0

STRUCT ContextNode *parent, *cn
```

Function: Gets the ContextNode of the next highest level, for example "FORM", from the ContextNode of a chunk.

Parameters: cn ContextNode for which parent node is sought.

Result: ContextNode or 0

PopChunk **Get ContextNode from context stack**

Call:

```
error = PopChunk (iff)
d0      -90(A6)   a0

LONG   error
STRUCT IFFHandle *iff
```

Function: Gets the next context chunk from the stack and frees all LocalItems.

Parameters: iff IFFHandle

Result: 0 or error code

PushChunk **Move ContextNode to the context stack**

Call:

```
error = PushChunk (iff, type, id, size)
d0      -84(A6)    a0   d0    d1  d2

LONG   error
STRUCT IFFHandle *iff
LONG   type, id, size
```

Function:	Places a new ContextNode from IFFStream on the context stack.	
Parameters:	iff	IFFHandle
	type	Type (e.g. "ILBM")
	id	ID (e.g. "CMAP")
	size	Chunk size or IFFSIZE_UNKNOWN
Result:	0 or error code	

```
Dec Hex STRUCTURE ContextNode,MLN_SIZE ;cn_Node
  8  $8 LONG  cn_ID                    ;ChunkID
 12  $C LONG  cn_Type                  ;FORM type
 16 $10 LONG  cn_Size                  ;Chunk size
 20 $14 LONG  cn_Scan                  ;byte offset
 24 $18 LABEL cn_SIZEOF                ;not end of structure!!!
```

3. Handlers

CollectionChunk — **Declare a CollectionChunk**

Call:

```
error = CollectionChunk (iff, type, id)
d0       -138(A6)         a0   d0   d1

LONG   error,type,id
STRUCT IFFHandle *iff
```

Function:	Declares a chunk to be a collection chunk and installs a handler that is activated when the chunk is accessed.	
Parameters:	iff	IFFHandle (must not be open)
	type	Type (such as "ILBM")
	id	ID (such as "CRNG")
Result:	0 or error code	

CollectionChunks	Declare CollectionChunks

Call:

```
error = CollectionChunks (iff, list, n)
d0      -144(A6)          a0   a1   d0

LONG   error,n
STRUCT IFFHandle *iff
APTR   list
```

Function: CollectionChunk() for several chunk types. The list is a field of two longwords: type, ID.

Parameters:

iff — IFFHandle

list — Field with types and IDs

n — Number of list entries

Result: 0 or error code

EntryHandler	Link handler to context

Call:

```
error = EntryHandler (iff, type, id, position, hook, object)
d0      -102(A6)      a0   d0    d1  d2        a1    a2

LONG   error,type,id,position
STRUCT IFFHandle *iff
STRUCT Hook  *hook
APTR   object
```

Function: Installs hook for a chunk type. The hook routine is called for every new chunk of the given type.

Parameters:

iff — IFFHandle

type — Typ (such as "ILBM")

id — ID (such as "CMAP")

position — IFFSLI_...

hook — Hook structure with handler routine object User data (hook routine: A2)

Result: 0 or error code

ExitHandler — **Install chunk exit handler**

Call:

```
error = ExitHandler (iff, type, id, position, hook, object)
d0      -108(A6)     a0   d0    d1  d2        a1    a2
```

Functions, Parameters, Results:

Same as EntryHandler(), except that this routine is called prior to removing a chunk.

PropChunk — **Declare a StoredProperty chunk**

Call:

```
error = PropChunk (iff, type, id)
d0      -114(A6)   a0   d0    d1

LONG   error,type,id
STRUCT IFFHandle *iff
```

Function: Installs a handler for chunks of the given type.

Parameters:

iff	IFFHandle (does not have to be open)
type	Type (such as "ILBM")
id	ID (such as "CMAP")

Result: 0 or error code

PropChunks — **Declare chunks as PropChunks**

Call:

```
error = PropChunks (iff, list, n)
d0      -120(A6)    a0   a1    d0

LONG   error,n
STRUCT IFFHandle *iff
APTR   list
```

Function: PropChunk() for several chunks. The list parameter is a field with two longwords: type and ID.

Parameters: iff IFFHandle

list List with type, ID

n Number of chunks

Result: 0 or error code

StopChunk **Declare StopChunk**

Call:
```
error = StopChunk (iff, type, id)
d0      -126(A6)   a0   d0    d1

LONG   error,type,id
STRUCT IFFHandle *iff
```

Function: Declares a chunk to be a StopChunk, which stops ParseIFF() when encountered (IFFPARSE_SCAN-Modus).

Parameters: iff IFFHandle

type Type (such as "ILBM")

id ID (such as "BODY")

Result: 0 or error code

StopChunks **Declare StopChunks**

Call:
```
error = StopChunks (iff, list, n)
d0      -132(A6)    a0    a1   d0

LONG   error,n
STRUCT IFFHandle *iff
APTR   list
```

Function: Several StopChunk() calls (like PropChunks() etc.).

Parameters: iff IFFHandle

list Field with type, ID

n Number of chunks

Result: 0 or error code

StopOnExit **Stop after chunk**

Call:

```
error = StopOnExit (iff, type, id)
d0      -150(A6)    a0    d0    d1

LONG   error,type,id
STRUCT IFFHandle *iff
```

Function: Stops ParseIFF() in scan mode after the given chunk is processed.

Parameters:

iff IFFHandle

type Type (such as "ILBM")

id ID (such as "BODY")

Result: 0 or error code

```
Dec Hex STRUCTURE IFFStreamCmd,0
  0  $0 LONG  isc_Command ;IFFCMD_...
  4  $4 APTR  isc_Buf     ;data buffer
  8  $8 LONG  isc_NBytes  ;number of bytes
 12  $C LABEL isc_SIZEOF

Dec Hex STRUCTURE StoredProperty,0
  0  $0 LONG spr_Size
  4  $4 APTR spr_Data
  8  $8 LABEL spr_SIZEOF

Dec Hex STRUCTURE CollectionItem,0
  0  $0 APTR cit_Next
  4  $4 LONG cit_Size
  8  $8 APTR cit_Data
 12  $C LABEL cit_SIZEOF

IFFCMD_INIT     = 0 ;initialization
IFFCMD_CLEANUP  = 1 ;end
IFFCMD_READ     = 2 ;read
IFFCMD_WRITE    = 3 ;write
```

```
IFFCMD_SEEK     = 4 ;seek
IFFCMD_ENTRY    = 5 ;new Context
IFFCMD_EXIT     = 6 ;exit Context
IFFCMD_PURGELCI = 7 ;free LocalContextItem
```

4. Local ContextItems

AllocLocalItem — Allocate a LocalContextItem

Call:

```
item = AllocLocalItem (type, id, ident, usize)
d0     -186(A6)        d0    d1  d2     d3

STRUCT LocalContextItem *item
LONG   type, id, ident, usize
```

Function: Allocates and initializes a LocalContextItem structure for the given amount of user data.

Parameters:

type,id	Type, ID
ident	ContextItem type
usize	Size of user data buffer

Result: LocalContextItem or 0

FindLocalItem — Get LocalContextItem from the context stack

Call:

```
lci = FindLocalItem (iff, type, id, ident)
d0    -210(A6)       a0   d0    d1  d2

STRUCT LocalContextItem *lci
STRUCT IFFHandle *iff
LONG   type, id, ident
```

Function: Searches the context stack for the given LocalContextItem.

Parameters:

iff	IFFHandle
type	Type
id	ID

ident Item type (such as "exhd" = ExitHandler)

Result: LocalContextItem or 0

FreeLocalItem — Free a LocalContextItem

Call:

```
FreeLocalItem (lci)
-204(A6)       a0

STRUCT LocalContextItem *lci
```

Function: Frees the memory of a LocalContextItem allocated with AllocLocalItem().

Parameters: lci LocalContextItem from AllocLocalItem().

LocalItemData — Get address of the user data

Call:

```
data = LocalItemData (lci)
d0     -192(A6)       a0

APTR   data
STRUCT LocalContextItem *lci
```

Function: Returns the address of the user buffer of an LCI.

Parameters: lci LocalContextItem or 0

Result: Data address or 0

SetLocalItemPurge — Install purge handler

Call:

```
SetLocalItemPurge (item, purgehook)
-198(A6)          a0    a1

STRUCT LocalContextItem *item
STRUCT Hook  *purgehook
```

Function: Installs a purge handler in an LCI (A0=Hook, A1=*IFFCMD_PURGELCI, A2=LCI).

Parameters: item LocalContextItem

purgehook
Hook with purge routine.

StoreItemInContext	**Store LCI in ContextNode**

Call:

```
StoreItemInContext (iff, item, cn)
-222(A6)            a0   a1    a2

STRUCT IFFHandle *iff
STRUCT LocalContextItem *item
STRUCT ContextNode  *cn
```

Function: Adds a LocalContextItem to a ContextNode's list.

Parameters:
iff IFFHandle

item LocalContextItem

cn ContextNode

StoreLocalItem	**Store LCI on the context stack**

Call:

```
error = StoreLocalItem (iff, item, position)
d0      -216(A6)        a0   a1    d0

LONG   error,position
STRUCT IFFHandle *iff
STRUCT LocalContextItem *item
```

Function: Add LCI to a ContextNode's list.

Parameters:
iff IFFHandle

item LocalContextItem

position IFFSLI_ROOT, IFFSLI_TOP, or IFFSLI_PROP

Result: 0 or error code

```
Dec Hex STRUCTURE LocalContextItem,MLN_SIZE ; lci_Node
  8  $8 ULONG lci_ID
 12  $C ULONG lci_Type
```

```
 16 $10 ULONG lci_Ident
 20 $14 LABEL lci_SIZEOF ;not end of structure!!!

IFFLCI_PROP         = 'prop'
IFFLCI_COLLECTION   = 'coll'
IFFLCI_ENTRYHANDLER = 'enhd'
IFFLCI_EXITHANDLER  = 'exhd'

IFFSLI_ROOT = 1 ;LCI in previously set Context
IFFSLI_TOP  = 2 ;LCI in current Context
IFFSLI_PROP = 3 ;LCI in FORM or LIST

IFFSIZE_UNKNOWN = -1
```

3.1.11 The Intuition Library

The "intuition.library" handles global management of the display and input from the keyboard and mouse. The base address must be given in A6.

Functions of the Intuition Library

1. Screens

CloseScreen
CloseWorkBench
FreeScreenDrawInfo
GetDefaultPubScreen
GetScreenData
GetScreenDrawInfo
LockPubScreen
LockPubScreenList
MakeScreen
MoveScreen
NextPubScreen
OpenScreen
OpenScreenTagList
OpenWorkBench
PubScreenStatus
QueryOverscan
RemakeDisplay
RethinkDisplay
ScreenToBack
ScreenToFront
SetDefaultPubScreen
SetPubScreenModes
ShowTitle
UnlockPubScreen
UnlockPubScreenList
ViewAddress
WBenchToBack
WBenchToFront

2. Windows

ActivateWindow
BeginRefresh
ChangeWindowBox
ClearMenuStrip
ClearPointer
CloseWindow
EndRefresh
ItemAddress
ModifyIDCMP

MoveWindow
MoveWindowInFrontOf
OffMenu
OnMenu
OpenWindow
OpenWindowTagList
RefreshWindowFrame
ResetMenuStrip
SetMenuStrip
SetMouseQueue
SetPointer
SetWindowTitles
SizeWindow
ViewPortAddress
WindowLimits
WindowToBack
WindowToFront
ZipWindow

3. Requesters

AutoRequest
BuildEasyRequestArgs
BuildSysRequest
ClearDMRequest
DisplayAlert
EasyRequestArgs
EndRequest
FreeSysRequest
InitRequester
Request
SetDMRequest
SysReqHandler

4. Gadgets

ActivateGadget
AddGadget
AddGList
GadgetMouse
ModifyProp
NewModifyProp
ObtainGIRPort
OffGadget
OnGadget
RefreshGadgets
RefreshGList
ReleaseGIRPort
RemoveGadget
RemoveGList
ReportMouse
SetEditHook
SetGadgetAttrsA

5. Output Functions

DisplayBeep
DrawBorder
DrawImage
DrawImageState
EraseImage
IntuiTextLength
PrintIText

6. Other Functions

AddClass
AllocRemember
CurrentTime
DisposeObject
DoubleClick
FreeClass
FreeRemember
GetAttr
GetDefPrefs
GetPrefs
LockIBase
MakeClass
NewObjectA
NextObject
PointInImage
RemoveClass
SetAttrsA
SetPrefs
UnlockIBase

Description of the Functions

1. Screens

CloseScreen — Attempt to close a screen

Call:

```
Success = CloseScreen(Screen)
D0         -66(A6)      A0

BOOL    Success
STRUCT Screen *Screen
```

Function: Attempts to close a screen. If successful and the screen was the last screen, then an attempt is made to open the Workbench.

Parameters: Screen — Screen to be closed.

Result: 0 — Screen could not be closed because it still contains windows.

CloseWorkBench — Attempt to close Workbench

Call:

```
Success = CloseWorkBench()
D0         -78(A6)

BOOL Success
```

Function: Attempts to close the Workbench.

Result: 0 — Workbench still open because there are still windows from other programs on screen.

FreeScreenDrawInfo — Free DrawInfo

Call:

```
FreeScreenDrawInfo( Screen, DrInfo )
-696(A6)            A0      A1

STRUCT Screen *Screen
STRUCT DrawInfo *DrInfo
```

Function: Frees the screen's DrawInfo structure (important for future operating system versions).

Parameters: Screen — Screen with the DrawInfo structure.

DrInfo — DrawInfo from GetScreenDrawInfo().

GetDefaultPubScreen	**Get default PublicScreen**

Call:

```
GetDefaultPubScreen( Namebuff )
-582(A6)             A0

APTR Namebuff
```

Function: Gets the name of the default PublicScreen.

Parameters: Namebuff — Buffer of size MAXPUBSCREENNAME bytes, or 0.

Result: None. Will provide the string "Workbench" in Namebuff if there is no current default public screen.

GetScreenData	**Copy Screen structure**

Call:

```
Success = GetScreenData(Buffer, Size, Type, Screen )
D0        -426(A6)      A0      D0    D1    A1

BOOL Success
STRUCT Screen *Screen
UWORD Size,Type
APTR Buffer
```

Function: With CUSTOMSCREEN, copies the given Screen structure to a buffer. If you specify a different screen type, the data structure of the given screen type is copied. This will open the Workbench if it was closed at the time the call was made.

Parameters: Buffer — Buffer for the Screen structure.

Size — Buffer size

	Type	Screen type (such as WBENCHSCREEN)
	Screen	Screen address of a CUSTOMSCREEN.
Result:	0	Error ('Type' Screen could not be opened)

GetScreenDrawInfo — **Get DrawInfo for a screen**

Call:

```
DrInfo = GetScreenDrawInfo( Screen )
D0       -690(A6)           A0

STRUCT DrawInfo *DrInfo
STRUCT Screen *Screen
```

Function: Gets a DrawInfo structure for the given screen.

Parameters: Screen — Address of a Screen structure.

Result: DrawInfo structure

LockPubScreen — **Lock a PublicScreen**

Call:

```
screen = LockPubScreen( Name )
D0       -510(A6)       A0

STRUCT Screen *screen
APTR   Name
```

Function: Prevents a PublicScreen from being closed while data is being read. This is needed to open a window according to the screen dimensions. If a value of 0 is given, the default public screen is addressed - usually the Workbench screen. If this is closed at the time, it is opened again (OpenWorkBench()).

Parameters: Name — Screen name, *"Workbench" or 0

Result: Screen address or 0

LockPubScreenList **Lock PublicScreen list**

Call:

```
List = LockPubScreenList()
D0     -522(A6)

STRUCT List *List
```

Function: Prevents the PublicScreen list from being changed and gets a user copy of this system list (PubScreenNodes).

Result: Address of the PublicScreen list.

MakeScreen **MakeVPort() for Intuition screens**

Call:

```
MakeScreen(Screen)
-378(A6)   A0

STRUCT Screen *Screen
```

Function: Allows changes to the screen display in a compatible way. RethinkDisplay() should be called afterwards.

Parameters: Screen Address of the changed screen.

MoveScreen **Move screen**

Call:

```
MoveScreen(Screen, DeltaX, DeltaY)
-162(A6)   A0      D0      D1

STRUCT Screen *Screen
WORD   DeltaX, DeltaY
```

Function: Moves the given screen according to the given delta value. Starting with AmigaOS2, the screen may also be moved horizontally and scrolled up to the left and out of the display (negative positions).

Parameters:

Screen	Screen to be moved.
DeltaX	Horizontal interval in (screen) pixels.
DeltaY	Vertical interval in (screen) pixels.

NextPubScreen — **Get name of the next PublicScreen**

Call:

```
Name = NextPubScreen( Screen, NameBuff)
D0     -534(A6)       A0      A1

STRUCT Screen *Screen
APTR   NameBuff, Name
```

Function: Gets the name of the next PublicScreen.

Parameters: Screen — Screen or 0

NameBuff — Buffer consisting of MAXPUBSCREENNAME bytes.

Result: Address of buffer or 0 (not a PublicScreen).

OpenScreen — **Open screen**

Call:

```
Screen = OpenScreen(NewScreen)
D0       -198(A6)   A0

STRUCT Screen *Screen
STRUCT (Ext)NewScreen *NewScreen
```

Function: Opens a screen that's given the definition of a NewScreen or ExtNewScreen structure.

Parameters: NewScreen

Initialized NewScreen or ExtNewScreen structure.

Tags (ExtNS):

Old function (see NS): SA_Left, SA_Top, SA_Width, SA_Height, SA_Depth, SA_DetailPen, SA_BlockPen, SA_Title, SA_Font, SA_Type, SA_BitMap, SA_ShowTitle, SA_Behind, SA_Quiet.

SA_DisplayID: 32 bit display mode.

SA_Overscan: OSCAN_TEXT, OSCAN_STANDARD, OSCAN_MAX or OSCAN_VIDEO.

SA_DClip: DisplayClip region, see QueryOverscan().

SA_AutoScroll: Bool
For oversized screens.

SA_PubName: Screen becomes a PublicScreen.

SA_Pens: Field for DrawInfo.dri_Pens which allows all OS 2.0 options.

SA_PubTask: Task that is informed of the last "Visitor" window to leave the PubScreen.

SA_PubSig: Signal bit for SA_PubTask.

SA_Colors: Colors, ending with -1.

SA_FullPalette: Take over all 32 Preferences colors (BOOL).

SA_ErrorCode: Address for error codes.

SA_SysFont: Use Preferences font (0 = old font, 1=Workbench font)

Result: Address of the Screen structure or 0 (see SA_ErrorCode).

OpenScreenTagList **Open screen**

Call:

```
Screen = OpenScreenTagList( NewScreen, TagItems )
D0       -612(A6)           A0         A1

STRUCT Screen *Screen
STRUCT NewScreen *NewScreen
STRUCT TagItem *TagItems
```

Function: Same as OpenScreen() ExtNewScreen data structure, but the TagItem field is passed as a parameter in place of the old ExtNewScreen method (test versions).

Functions, Tags:
See OpenScreen()

Parameters: NewScreen
Optional NewScreen structure

TagItems Optional TagItem field, ending with TAG_END.

Result: Address of the Screen structure or 0.

OpenWorkBench — Get Workbench address

Call:

```
WBScreen = OpenWorkBench()
D0         -210(A6)

STRUCT Screen *WBScreen
```

Function: Searches for the Workbench screen and tries to open it if it is not already open.

Result: Screen structure or 0

PubScreenStatus — Change status of a PublicScreen

Call:

```
ResultFlags = PubScreenStatus( Screen, StatusFlags )
D0            -552(A6)         A0      D0

UWORD  ResultFlags,StatusFlags
STRUCT Screen *Screen
```

Function: Change status flags of own PublicScreen.

Parameters: Screen Own PublicScreen

StatusFlags
Current flags

Result: Bit 0 0: Screen was not public or could not be made private.

QueryOverscan	Query overscan area

Call:

```
QueryOverscan( DisplayID, Rect, OScanType )
-474(A6)       A0         A1    D0

ULONG  DisplayID
STRUCT Rectangle *Rect
WORD   OScanType
```

Function: Fills a Rectangle structure with the dimensions of an overscan type corresponding to the 32 bit display mode.

Parameters: DisplayID 32 bit display mode

Rect Rectangle structure to be filled.

OScanType
OSCAN_...

Result: 0 MonitorSpec for the ID does not exist.

RemakeDisplay	Recalculate the display

Call:

```
RemakeDisplay()
-384(A6)
```

Function: Complete recalculation of all screens (ViewPorts) and the ViewLord (Intuition View). This function should be avoided (MakeScreen()+RethinkDisplay() will usually do the job).

RethinkDisplay	Global display reconstructure

Call:

```
RethinkDisplay()
-390(A6)
```

Function: Global reconstruction of the display. MakeVPort() should be called first.

ScreenToBack — **Move screen to the background**

Call:
```
ScreenToBack(Screen)
-246(A6)      A0

STRUCT Screen *Screen
```

Function: Moves the given screen to the back of the display.

Parameters: Screen   Screen structure

ScreenToFront — **Move screen to the foreground**

Call:
```
ScreenToFront(Screen)
-252(A6)       A0

STRUCT Screen *Screen
```

Function: Moves the given screen to the foreground.

Parameters: Screen   Screen structure

SetDefaultPubScreen — **Set standard PublicScreen**

Call:
```
SetDefaultPubScreen( Name )
-540(A6)             A0

APTR Name
```

Function: Sets the default PublicScreen.

Parameters: Name   Name of the PubScreen or 0 (=Workbench).

SetPubScreenModes — **Set global mode for PublicScreen**

Call:
```
OldModes = SetPubScreenModes( Modes )
D0         -546(A6)          D0

UWORD OldModes, Modes
```

Function: Sets the global mode for PublicScreen.

Parameters: Modes — New flags: SHANGHAI - Workbench windows are opened with the default PublicScreen; POPPUBSCREEN - PublicScreen moves to the foreground when opened.

Result: Old global flags.

ShowTitle — **Activate (or deactivate) a screen's title bar**

Call:

```
ShowTitle(Screen, ShowIt)
-282(A6)  A0      D0

STRUCT Screen *Screen
BOOL   ShowIt
```

Function: Manipulates the SHOWTITLE flag of a screen and refreshes the display.

Parameters: Screen — Screen structure

ShowIt — Boolean: TRUE (title bar visible) or 0

UnlockPubScreen — **Free a PublicScreen**

Call:

```
UnlockPubScreen( Name, Screen )
-516(A6)         A0    A1

APTR   Name
STRUCT Screen *Screen
```

Function: Undo a LockPubScreen() call.

Parameters: Name — Name of PublicScreen or 0.

Screen — Screen address if Name=0.

UnlockPubScreenList — **Free PublicScreen list**

Call:

```
UnlockPubScreenList()
-528(A6)
```

Function: Undo a LockPubScreenList() call.

ViewAddress	Get the address of the ViewLord

Call:

```
ViewLord = ViewAddress()
D0         -294(A6)

STRUCT View *ViewLord
```

Function: Gets the address of the Intuition View structure ViewLord.

Result: View structure

WBenchToBack	Move Workbench to the background

Call:

```
Success = WBenchToBack()
D0         -336(A6)

BOOL Success
```

Function: Moves the Workbench screen behind all other screens.

Result: 0 Workbench was not open.

WBenchToFront	Move Workbench to foreground

Call:

```
Success = WBenchToFront()
D0         -342(A6)

BOOL Success
```

Function: Move the Workbench screen in front of all other screens.

Result: 0 Workbench was not open.

```
Dec Hex STRUCTURE IntuitionBase,0
  0  $0 STRUCT ib_LibNode,LIB_SIZE
 34 $22 STRUCT ib_ViewLord,v_SIZEOF ;Intuition View
 52 $34 APTR   ib_ActiveWindow      ;active window
 56 $38 APTR   ib_ActiveScreen      ;that window's screen
 60 $3C APTR   ib_FirstScreen       ;first screen
 64 $40 ULONG  ib_Flags             ;private
 68 $44 WORD   ib_MouseY            ;INCOMPATIBLE!
 70 $46 WORD   ib_MouseX            ;
 72 $48 ULONG  ib_Seconds           ;time
```

```
 76 $4C ULONG  ib_Micros           ;

DRI_VERSION = 1

Dec Hex STRUCTURE DrawInfo,0
  0  $0 UWORD  dri_Version      ;structure version
  2  $2 UWORD  dri_NumPens      ;>= numDrIPens
  4  $4 APTR   dri_Pens         ;PenArray
  8  $8 APTR   dri_Font         ;ScreenDefaultFont
 12  $C UWORD  dri_Depth        ;(initial) BitMap depth
 14  $E UWORD  dri_ResolutionX  ;1/velocity
 16 $10 UWORD  dri_ResolutionY  ;
 18 $12 ULONG  dri_Flags
 22 $16 STRUCT dri_longreserved,28

DRIF_NEWLOOK = 1, DRIB_NEWLOOK = 0

PenArray Indices:
detailPen      = 0 ;old draw color
blockPen       = 1 ;
textPen        = 2 ;text on background (0)
shinePen       = 3 ;bright 3D edges
shadowPen      = 4 ;dark 3D edges
hifillPen      = 5 ;fill color for current window
hifilltextPen  = 6 ;text in current window border
backgroundPen  = 7 ;background color (0)
hilighttextPen = 8 ;highlighted text
numDrIPens     = 9

Dec  Hex STRUCTURE Screen,0
  0   $0 APTR   sc_NextScreen   ;next screen
  4   $4 APTR   sc_FirstWindow  ;first window
  8   $8 WORD   sc_LeftEdge     ;position
 10   $A WORD   sc_TopEdge      ;
 12   $C WORD   sc_Width        ;size
 14   $E WORD   sc_Height       ;
 16  $10 WORD   sc_MouseY       ;mouse position
 18  $12 WORD   sc_MouseX       ;
 20  $14 WORD   sc_Flags        ;
 22  $16 APTR   sc_Title        ;current title
 26  $1A APTR   sc_DefaultTitle ;title
 30  $1E BYTE   sc_BarHeight    ;size of title bar
 31  $1F BYTE   sc_BarVBorder   ;
 32  $20 BYTE   sc_BarHBorder   ;
 33  $21 BYTE   sc_MenuVBorder  ;size of menu border
 34  $22 BYTE   sc_MenuHBorder  ;
 35  $23 BYTE   sc_WBorTop      ;size of window border
 36  $24 BYTE   sc_WBorLeft     ;
```

```
 37  $25 BYTE   sc_WBorRight             ;
 38  $26 BYTE   sc_WBorBottom            ;
 39  $27 BYTE   sc_KludgeFill00
 40  $28 APTR   sc_Font                  ;font
 44  $2C STRUCT sc_ViewPort,vp_SIZEOF    ;ViewPort
 84  $54 STRUCT sc_RastPort,rp_SIZEOF    ;RastPort
184  $B8 STRUCT sc_BitMap,bm_SIZEOF      ;BitMap
224  $E0 STRUCT sc_LayerInfo,li_SIZEOF   ;LayerInfo
326 $146 APTR   sc_FirstGadget           ;gadgets
330 $14A BYTE   sc_DetailPen             ;for gadgets
331 $14B BYTE   sc_BlockPen              ;
332 $14C WORD   sc_SaveColor0            ;for Beeping
334 $14E APTR   sc_BarLayer              ;title Layer
338 $152 APTR   sc_ExtData               ;extended data
342 $156 APTR   sc_UserData              ;
346 $15A LABEL  sc_SIZEOF                ;not necessarily end of structure!

SCREENTYPE       =    $F ;mask
WBENCHSCREEN     =     1 ;Workbench screen
PUBLICSCREEN     =     2 ;PublicScreen
CUSTOMSCREEN     =    $F ;other screens
SHOWTITLE        =   $10 ;show title bar
BEEPING          =   $20 ;beep the screen
CUSTOMBITMAP     =   $40 ;User BitMap
SCREENBEHIND     =   $80 ;open screen behind
SCREENQUIET      =  $100 ;forbid drawing
SCREENHIRES      =  $200 ;HiRes gadgets
NS_EXTENDED      = $1000 ;extended screen structure
AUTOSCROLL       = $4000 ;for oversized screens

STDSCREENHEIGHT = -1   ;NewScreen.Height
STDSCREENWIDTH  = -1   ;NewScreen.Width

SA_Left          = TAG_USER+33
SA_Top           = TAG_USER+34
SA_Width         = TAG_USER+35
SA_Height        = TAG_USER+36
SA_Depth         = TAG_USER+37
SA_DetailPen     = TAG_USER+38
SA_BlockPen      = TAG_USER+39
SA_Title         = TAG_USER+40
SA_Colors        = TAG_USER+41
SA_ErrorCode     = TAG_USER+42
SA_Font          = TAG_USER+43
SA_SysFont       = TAG_USER+44
SA_Type          = TAG_USER+45
SA_BitMap        = TAG_USER+46
SA_PubName       = TAG_USER+47
```

```
SA_PubSig        = TAG_USER+48
SA_PubTask       = TAG_USER+49
SA_DisplayID     = TAG_USER+50
SA_DClip         = TAG_USER+51
SA_Overscan      = TAG_USER+52
SA_Obsolete1     = TAG_USER+53
SA_ShowTitle     = TAG_USER+54
SA_Behind        = TAG_USER+55
SA_Quiet         = TAG_USER+56
SA_AutoScroll    = TAG_USER+57
SA_Pens          = TAG_USER+58
SA_FullPalette   = TAG_USER+59

OSERR_NOMONITOR     = 1 ;monitor not available
OSERR_NOCHIPS       = 2 ;old CustomChips
OSERR_NOMEM         = 3 ;not enough memory
OSERR_NOCHIPMEM     = 4 ;not enough ChipMem
OSERR_PUBNOTUNIQUE  = 5 ;PublicScreen already exists
OSERR_UNKNOWNMODE   = 6 ;unknown display mode

Dec Hex STRUCTURE NewScreen,0
  0  $0 WORD   ns_LeftEdge     ;position
  2  $2 WORD   ns_TopEdge      ;
  4  $4 WORD   ns_Width        ;size
  6  $6 WORD   ns_Height       ;
  8  $8 WORD   ns_Depth        ;
 10  $A BYTE   ns_DetailPen    ;preset color
 11  $B BYTE   ns_BlockPen     ;
 12  $C WORD   ns_ViewModes    ;display mode (old)
 14  $E WORD   ns_Type         ;screen type
 16 $10 APTR   ns_Font         ;font TextAttr
 20 $14 APTR   ns_DefaultTitle ;screen title
 24 $18 APTR   ns_Gadgets      ;0
 28 $1C APTR   ns_CustomBitMap ;own BitMap
 32 $20 LABEL  ns_SIZEOF
        STRUCTURE ExtNewScreen,ns_SIZEOF
 32 $20 APTR   ens_Extension   ;TagItem field
 36 $24 LABEL  ens_SIZEOF

OSCAN_TEXT     = 1 ;everything visible
OSCAN_STANDARD = 2 ;right up to the edge of the screen
OSCAN_MAX      = 3 ;large as possible
OSCAN_VIDEO    = 4 ;more than is possible

Dec Hex STRUCTURE PubScreenNode,LN_SIZE
 14  $E APTR   psn_Screen ;screen
 18 $12 UWORD  psn_Flags  ;private?
 20 $14 WORD   psn_Size   ;structure size+name
```

```
 22 $16 WORD    psn_VisitorCount ;number of VisitorWindows
 24 $18 APTR    psn_SigTask      ;control task
 28 $1C UBYTE   psn_SigBit       ;signal bit
 29 $1D UBYTE   psn_Pad1
 30 $1E LABEL   psn_SIZEOF

PSNF_PRIVATE    = 1

MAXPUBSCREENNAME = 139 ;max. name length

SHANGHAI        = 1     ;Workbench window on PubScreen
POPPUBSCREEN    = 2     ;PubScreen to front
```

2. Windows

ActivateWindow	**Activate a window**

Call:

```
success = ActivateWindow(Window)
d0        -450(A6)      A0

STRUCT Window *Window
```

Function: Activates a window (for input).

Parameters: Window   Window structure

Result: 0   Okay

BeginRefresh	**Prepare window for a refresh**

Call:

```
BeginRefresh(Window)
-354(A6)      A0

STRUCT Window *Window
```

Function: The window's layer is locked for other programs (such as "input.device"->Intuition()). During this time, only functions that don't handle other tasks may be called (never Intuition).

Parameters: Window   Window to be refreshed

Example: Refresh several regions in a window:

```
...
move.l   im_Class(a0),d0          ;IntuiMessage class
cmpi.l   #IDCMP_REFRESHWINDOW,d0 ;refresh window?
beq      _WindowRefresh
...

_WindowRefresh
movea.l im_IDCMPWindow(a0),a2     ;get window
movea.l wd_WScreen(a2),a0         ;screen address
lea     sc_LayerInfo(a0),a0       ;get LayerInfo
movea.l a0,a3                     ;and save
movea.l _LayersBase,a6            ;Layers library
jsr     _LVOLockLayerInfo(a6)     ;lock access

movea.l a2,a0                     ;Window
movea.l wd_WLayer(a0),a0          ;Layer
movea.l _Region1,a1               ;1. Region
jsr     _LVOInstallClipRegion(a6) ;new Clipping
move.l  d0,d2                     ;save old Region
movea.l _IntuiBase,a6             ;Intuition
movea.l a2,a0                     ;Window
jsr     _LVOBeginRefresh(a6)      ;start refresh
...                               ;refresh first Region
movea.l _IntuiBase,a6             ;Intuition
movea.l a2,a0                     ;Window
moveq   #0,d0                     ;NOT DONE
jsr     _LVOEndRefresh(a6)        ;end

movea.l _LayersBase,a6            ;LayersBase
movea.l a2,a0                     ;Window
movea.l wd_WLayer(a0),a0          ;Layer
movea.l _Region2,a1               ;2nd Region
jsr     _LVOInstallClipRegion(a6) ;new Clipping
movea.l _IntuiBase,a6             ;Intuition
movea.l a2,a0                     ;Window
jsr     _LVOBeginRefresh(a6)      ;start refresh
...                               ;refresh second Region
movea.l _IntuiBase,a6             ;Intuition
movea.l a2,a0                     ;Window
moveq   #0,d0                     ;NOT DONE
jsr     _LVOEndRefresh(a6)        ;end

...                               ;refresh additional Regions

movea.l _LayersBase,a6            ;LayersBase
movea.l a2,a0                     ;Window
movea.l wd_WLayer(a0),a0          ;Layer
movea.l _RegionN,a1               ;nth Region
```

```
jsr      _LVOInstallClipRegion(a6) ;new Clipping
movea.l _IntuiBase,a6              ;Intuition
movea.l a2,a0                      ;Window
jsr      _LVOBeginRefresh(a6)      ;start refresh
...                                ;refresh nth Region
movea.l _IntuiBase,a6              ;Intuition
movea.l a2,a0                      ;Window
moveq    #-1,d0                    ;DONE
jsr      _LVOEndRefresh(a6)        ;process Damagelist

movea.l _LayersBase,a6             ;LayersBase
movea.l a2,a0                      ;Window
movea.l wd_WLayer(a0),a0           ;Layer
movea.l d2,a1                      ;old Region
jsr      _LVOInstallClipRegion(a6) ;restore Clipping
movea.l a3,a0                      ;LayerInfo
jsr      _LVOUnlockLayerInfo(a6)   ;unlock

...
```

ChangeWindowBox — Change position and size of a window

Call:

```
ChangeWindowBox( Window, Left, Top, Width, Height)
-486(A6)         A0      D0    D1   D2     D3

STRUCT Window *Window
WORD Left,Top,Width,Height
```

Function: Changes the position and size of a window simultaneously (MoveWindow() and SizeWindow() combined).

Parameters: Window — Window to be changed

Left, Top, Width, Height — New position and size

ClearMenuStrip — Remove menus from a window

Call:

```
ClearMenuStrip(Window)
-54(A6)        A0

STRUCT Window *Window
```

Function: Removes the MenuStrip from the given menu (waits if the menu is active).

Parameters: Window   Window with menu

ClearPointer	**Reset appearance of mouse pointer**

Call:

```
ClearPointer(Window)
-60(A6)       A0

STRUCT Window *Window
```

Function: If you have defined a custom mouse pointer for a window, this function will reset it to the standard mouse pointer.

Parameters: Window   Window

CloseWindow	**Close a window**

Call:

```
CloseWindow(Window)
-72(A6)      A0

STRUCT Window *Window
```

Function: Closes a window. If there are messages in the UserPort, these are also set free. If there is a MenuStrip, it must first be deleted with ClearMenuStrip(). If it is a "Visitor Window", the PublicScreen's counter is decremented.

Parameters: Window   Window to be closed.

Example: Close a window with a UserPort not created by Intuition. You can, for example, assign one UserPort to 10 windows, since the window that is to receive the IntuitionMessage is visible from the im_IDCMPWindow. When such a window is opened, a value of 0 must be given as the IDCMPFlag. After entering the common MsgPort, the desired value can be set with ModifyIDCMP(). Closing such a window is somewhat more complicated:

```
 movea.l _SysBase,a6                  ;ExecBase
 addq.b  #1,TDNestCnt(a6)             ;Multitasking off

 movea.l _Window,a2                   ;get window
 movea.l wd_UserPort(a2),a0           ;UserPort
 clr.l   wd_UserPort(a2)              ;delete entry
 move.l  MP_MSGLIST+LH_HEAD(a0),d2    ;ListNode
.Loop
 movea.l d2,a1                        ;message
 move.l  (a1),d2                      ;LN_SUCC
 beq.s   .StopMsgs                    ;LH_TAIL reached?
 cmpa.l  im_IDCMPWindow(a1),a2        ;Msg for this window?
 bne.s   .Loop                        ;if not
 move.l  a1,d3                        ;save Msg
 jsr     _LVORemove(a6)               ;remove Msg
 movea.l d3,a1                        ;get Msg
 jsr     _LVOReplyMsg(a6)             ;send back
 bra.s   .Loop                        ;continue

.StopMsgs
 movea.l _IntuiBase,a6                ;Intuition
 movea.l a2,a0                        ;window
 moveq   #0,d0                        ;no Flags
 jsr     _LVOModifyIDCMP(a6)          ;prevent further Msgs

 movea.l _SysBase,a0                  ;Exec
 subq.b  #1,TDNestCnt(a6)             ;Multitasking on

 movea.l a2,a0                        ;window
 jsr     _LVOCloseWindow(a6)          ;close
```

EndRefresh — End screen refresh

Call:

```
EndRefresh(Window, Complete)
-366(A6)   A0      D0

STRUCT Window *Window
BOOL   Complete
```

Function: Unlocks a window that was locked by BeginRefresh() (if Complete=TRUE). If you program a routine that continues when Complete=0, the system may easily crash.

Parameters: Window Window

Complete Boolean, indicates if refresh has ended.

ItemAddress — **Get address of a MenuItem**

Call:

```
Item = ItemAddress(MenuStrip, MenuNumber)
D0     -144(A6)    A0         D0

STRUCT MenuItem *ItemAddress
STRUCT Menu *MenuStrip
UWORD  MenuNumber
```

Function: Gets the MenuItem or SubItem belonging to the given menu code.

Parameters: MenuStrip Address of the first menu in a MenuStrip.

MenuNumber
Bit-packed menu code

Result: MenuItem address or 0 (MenuNumber = MENUNULL)

ModifyIDCMP — **Change IDCMP flags**

Call:

```
success  ModifyIDCMP(Window, IDCMPFlags)
D0         -150(A6)    A0      D0

STRUCT Window *Window
ULONG  IDCMPFlags
BOOL success
```

Function: Changes the status of the IDCMP (Intuition Direct Communication Message Port). Warning: the MsgPort is freed whenever the flags are set to 0 (FreeMem()).

Parameters: Window Window address

IDCMPFlags = MsgPort status

MoveWindow — **Move a window**

Call:

```
MoveWindow(Window, DeltaX, DeltaY)
-168(A6)   A0      D0      D1
```

```
STRUCT Window *Window
WORD   DeltaX, DeltaY
```

Function: Moves a window within a screen after first checking the coordinates.

Parameters:

Window	Window to be moved
DeltaX	Horizontal delta value
DeltaY	Vertical delta value

MoveWindowInFrontOf — **Change order of windows**

Call:

```
MoveWindowInFrontOf( Window, BehindWindow)
-480(A6)             A0      A1

STRUCT Window *Window,*BehindWindow
```

Function: Moves a window in front of the other windows.

Parameters:

Window	Window to be placed in front of 'BehindWindow'.
BehindWindow	Window that will end up behind 'Window'.

OffMenu — **Turn menu off**

Call:

```
OffMenu(Window, MenuNumber)
-180(A6) A0     D0

STRUCT Window *Window
UWORD  MenuNumber
```

Function: Turn off SubItem, MenuItem, or an entire menu. Whatever is turned off may not be selected.

Parameters:

Window	Window with MenuStrip
MenuNumber	Bit-packed menu code

OnMenu	**Turn on menu, item, or subitem**

Call:

```
OnMenu(Window, MenuNumber)
-192(A6) A0     D0

STRUCT Window *Window
UWORD  MenuNumber
```

Functions, Parameters:

Opposite of OffMenu(). Allows selection again.

OpenWindow	**Open a window**

Call:

```
Window = OpenWindow(NewWindow)
D0        -204(A6)    A0

STRUCT Window *Window
STRUCT (Ext)NewWindow *NewWindow
```

Function: Opens a window given a NewWindow or ExtNewWindow structure. If WBENCHSCREEN is given as the screen type, the window is opened on the Workbench screen or a SHANGHAI screen.

Parameters: NewWindow

NewWindow or ExtNewWindow

New IDCMP flags: IDCMP_IDCMPUPDATE for custom and BoopsiGadgets; IDCMP_CHANGEWINDOW for changes to the window (position, size); IDCMP_MENUHELP for printing menu HELP.

New type: PUBLICSCREEN (preset PublicScreen)

Tags: Old functions (s. NW): WA_Left, WA_Top, WA_Width, WA_Height, WA_DetailPen, WA_BlockPen, WA_IDCMP, WA_Flags, WA_Gadgets, WA_Checkmark, WA_Title, WA_CustomScreen, WA_SuperBitMap, WA_MinWidth, WA_MinHeight, WA_MaxWidth, WA_MaxHeight.

Bool tags for flags: WA_SizeGadget, WA_DragBar, WA_DepthGadget, WA_CloseGadget, WA_Backdrop, WA_ReportMouse.

WA_ScreenTitle: screen title.

WA_AutoAdjust: (BOOL) adjust to screen dimensions.

WA_InnerWidth, WA_InnerHeight: dimensions of the region.

WA_PubScreenName: name of PublicScreens for the window.

WA_PubScreen: Screen structure of the PublicScreen.

WA_PubScreenFallBack: default PublicScreen if the one requested is not available (BOOL).

WA_WindowName: not yet implemented.

WA_Colors: palette for the window.

WA_Zoom: field with alternative size (ZoomGadget).

WA_MouseQueue, WA_RptQueue: limits for IntuiMessages of the types IDCMP_MOUSEMOVE and repeated IDCMP_RAWKEY.

WA_BackFill: LayerHook for backfill.

Result: Window address or 0

Example: Display AmigaDOS requesters in the window entered in the Process structure of the currently running program. This is normally a Workbench window. Programs that open a custom screen should change this pointer to a window that is in the custom screen. Otherwise, the Workbench screen will brought to the foreground with every return query.

```
...
jsr      _LVOOpenWindow(a6)                ;open window
move.l   d0,_Window                        ;save address
beq      _Zerror                           ;in case of error

movea.l $4.w,a0                            ;ExecBase
movea.l ThisTask(a0),a0                    ;Process structure
move.l   pr_WindowPtr(a0),_SavedWindowPtr ;save old value
move.l   d0,pr_WindowPtr(a0)               ;enter custom window

...                                        ;Program code

movea.l $4.w,a0                            ;ExecBase
movea.l ThisTask(a0),a1                    ;Process structure
movea.l pr_WindowPtr(a1),a0                ;our window
move.l   _SavedWindowPtr,pr_WindowPtr(a1) ;restore old value
movea.l _IntuiBase,a6                      ;Intuition
jsr      _LVOCloseWindow(a6)               ;close window
...
```

Proceed as follows to open a window on a PublicScreen:

```
LockPubScreen()       - lock PubScreen
 Get information and modify window
 OpenWindow()         - open window
 UnlockPubScreen()    - free PublicScreen

 ...

 CloseWindow()        - close window
```

OpenWindowTagList	Open window

Call:

```
Window = OpenWindowTagList( NewWindow, TagItems)
D0       -606(A6)           A0         A1

STRUCT Window *Window
STRUCT NewWindow *NewWindow
STRUCT TagItem *TagItems
```

Function: Like OpenWindow() with ExtNewWindow, except the TagItem field is passed as a parameter.

Parameters: NewWindow
Optional NewWindow structure

TagItems   Optional TagItem field

Result: Window address or 0

RefreshWindowFrame   Refresh the window frame

Call:

```
RefreshWindowFrame(Window)
-456(A6)           A0

STRUCT Window *Window
```

Function: Refreshes the window frame of the given window.

Parameters: Window   Window

ResetMenuStrip   Super-fast SetMenuStrip()

Call:

```
Success = ResetMenuStrip(Window, Menu)
D0        -702(A6)       A0      A1

BOOL   Success
STRUCT Window *Window
STRUCT Menu *Menu
```

Function: If a MenuStrip has been removed from a window and not changed in the meantime, it can be activated again with this function without having to recalculate JazzX, JazzY, BeatX, and BeatY.

Parameters: Window   Window

Menu   First menu of the MenuStrip.

Result: TRUE

Example: Turn off MenuStrip while a program is executing:

```
movea.l _IntuiBase,a6          ;must happen first
movea.l _Window,a0
lea     _MenuStrip,a1
jsr     _LVOSetMenuStrip(a6)
...
```

```
     ;jumps to '_Routine' possible from here
 ...
 movea.l _IntuiBase,a6        ;Intuition
 movea.l _Window,a0           ;window
 jsr     _LVOClearMenuStrip(a6) ;turn off menu
 ...


_Routine
 ...                          ;save registers, etc.
 movea.l _IntuiBase,a6        ;Intuition
 movea.l _Window,a0           ;window
 jsr     _LVOClearMenuStrip(a6) ;turn off menu
 ...                          ;now Flags such as
                              ;CHECKED or ITEMENABLED can be changed

 ...                          ;Routine

 movea.l _IntuiBase,a6        ;Intuition
 movea.l _Window,a0           ;window
 lea     _MenuStrip,a1        ;old menu
 jsr     _LVOResetMenuStrip(a6) ;reactivate
 ...                          ;restore registers, etc.
 rts
```

SetMenuStrip	**Set MenuStrip in a window**

Call:

```
Success = SetMenuStrip(Window, Menu)
D0        -264(A6)      A0      A1

BOOL   Success
STRUCT Window *Window
STRUCT Menu *Menu
```

Function: Installs a MenuStrip in a window and calculates the menu boxes (JazzX/.Y, BeatX/.Y).

Parameters: Window Window

Menu First Menu structure

Result: TRUE

SetMouseQueue **Set maximum number of mouse movement messages**

Call:

```
old = SetMouseQueue( Window, QueueLength )
D0    -498(A6)       A0      D0

STRUCT Window *Window
UWORD  QueueLength
LONG   old
```

Function: Sets the maximum number of mouse movement messages that may lie unanswered in the window's MessagePort (only meaningful with slow languages such as C, BASIC, etc.).

Parameters: Window Window

QueueLength
Number of MouseMove messages.

Result: Old queue length or -1 (unknown window).

SetPointer **Set the mouse pointer**

Call:

```
SetPointer(Window, Pointer, Height, Width, XOffset, YOffset)
-270(A6)   A0      A1       D0      D1     D2       D3

STRUCT Window *Window
APTR   Pointer
WORD   Height, Width, XOffset, YOffset
```

Function: Sets the mouse pointer for a window.

Parameters: Window Window

Pointer Sprite data

Height Sprite height

Width Width (1...16)

XOffset, YOffset
Offset from selection point.

SetWindowTitles — **Set title bar text**

Call:

```
SetWindowTitles(Window, WindowTitle, ScreenTitle)
-276(A6)        A0      A1           A2

STRUCT Window *Window
APTR   WindowTitle,ScreenTitle
```

Function: Defines the text displayed in the title bar of the active window and screen.

Parameters: Window Window

WindowTitle
Window title, 0 (empty), or -1 (do not change)

ScreenTitle
Screen title, 0 (empty) or -1 (do not change)

SizeWindow — **Change window size**

Call:

```
SizeWindow(Window, DeltaX, DeltaY)
-288(A6)   A0      D0      D1

STRUCT Window *Window
WORD   DeltaX, DeltaY
```

Function: Change the size of a window (only within the limits of MinWidth-MinHeight, MaxWidth-MaxHeight, and if the window has a Size gadget).

Parameters: Window Window

DeltaX Delta value for width

DeltaY Delta value for height

ViewPortAddress — Get a ViewPort

Call:

```
ViewPort = ViewPortAddress(Window)
D0            -300(A6)          A0

STRUCT Window *Window
STRUCT ViewPort *ViewPort
```

Function: Returns the address of the ViewPort containing the given window.

Parameters: Window Window

Result: Screen ViewPort of the window.

WindowLimits — Define limits for window size

Call:

```
Success = WindowLimits(Window, MinWidth, MinHeight, MaxWidth, MaxHeight)
D0        -318(A6)     A0      D0        D1         D2        D3

BOOL   Success
STRUCT Window *Window
WORD   MinWidth, MinHeight
UWORD  MaxWidth, MaxHeight
```

Function: Sets minimum and maximum values for window size.

Parameters: Window Window

MinWidth, MinHeight, MaxWidth, MaxHeight
New size limits, 0 (do not change) or -1 (full screen)

Result: 0 Error

WindowToBack — Move window behind all other windows

Call:

```
WindowToBack(Window)
-306(A6)      A0

STRUCT Window *Window
```

Function: Moves a window to the back.

Parameters: Window Window

WindowToFront **Moves a window in front of all others**

Call:
```
WindowToFront(Window)
-312(A6)      A0

STRUCT Window *Window
```

Function: Move window to the front.

Parameters: Window Window

ZipWindow **Activate alternative window size and position**

Call:
```
ZipWindow( Window)
-504(A6)   A0

STRUCT Window *Window
```

Function: Like ZoomGadget: the window is moved to the alternative position with the alternative size.

Parameters: Window Window

```
Dec Hex STRUCTURE Window,0
  0  $0 APTR   wd_NextWindow   ;next window
  4  $4 WORD   wd_LeftEdge     ;position
  6  $6 WORD   wd_TopEdge      ;
  8  $8 WORD   wd_Width        ;size
 10  $A WORD   wd_Height       ;
 12  $C WORD   wd_MouseY       ;relative mouse position
 14  $E WORD   wd_MouseX       ;
 16 $10 WORD   wd_MinWidth     ;minimum size
 18 $12 WORD   wd_MinHeight    ;
 20 $14 WORD   wd_MaxWidth     ;maximum size
 22 $16 WORD   wd_MaxHeight    ;
 24 $18 LONG   wd_Flags        ;Flags see below
 28 $1C APTR   wd_MenuStrip    ;first menu
 32 $20 APTR   wd_Title        ;title string
 36 $24 APTR   wd_FirstRequest ;active Requester
 40 $28 APTR   wd_DMRequest    ;double-menu Requester
```

```
 44 $2C WORD   wd_ReqCount      ;number of Requesters
 46 $2E APTR   wd_WScreen       ;Screen
 50 $32 APTR   wd_RPort         ;RastPort
 54 $36 BYTE   wd_BorderLeft    ;border size
 55 $37 BYTE   wd_BorderTop     ;
 56 $38 BYTE   wd_BorderRight   ;
 57 $39 BYTE   wd_BorderBottom  ;
 58 $3A APTR   wd_BorderRPort   ;border RastPort
 62 $3E APTR   wd_FirstGadget   ;first Gadget
 66 $42 APTR   wd_Parent        ;window (activation)
 70 $46 APTR   wd_Descendant    ;
 74 $4A APTR   wd_Pointer       ;mouse data
 78 $4E BYTE   wd_PtrHeight     ;
 79 $4F BYTE   wd_PtrWidth      ;
 80 $50 BYTE   wd_XOffset       ;
 81 $51 BYTE   wd_YOffset       ;
 82 $52 ULONG  wd_IDCMPFlags    ;IDCMP Flags
 86 $56 APTR   wd_UserPort      ;IDCMP
 90 $5A APTR   wd_WindowPort    ;ReplyPort (Intuition)
 94 $5E APTR   wd_MessageKey    ;Msg
 98 $62 BYTE   wd_DetailPen     ;no longer meaningful
 99 $63 BYTE   wd_BlockPen      ;
100 $64 APTR   wd_CheckMark     ;Image with new check mark
104 $68 APTR   wd_ScreenTitle   ;screen title
108 $6C WORD   wd_GZZMouseX     ;mouse position within screen
110 $6E WORD   wd_GZZMouseY     ;
112 $70 WORD   wd_GZZWidth      ;size of region
114 $72 WORD   wd_GZZHeight     ;
116 $74 APTR   wd_ExtData       ;extension structure
120 $78 APTR   wd_UserData      ;NOT available
124 $7C APTR   wd_WLayer        ;Layer
128 $80 APTR   wd_IFont         ;TextFont
132 $84 ULONG  wd_MoreFlags     ;new system Flags
136 $88 LABEL  wd_Size          ;size definition
136 $88 LABEL  wd_SIZEOF        ;not necessarily end of structure!

WINDOWSIZING   =      1 ;Size gadget available
WINDOWDRAG     =      2 ;movable window
WINDOWDEPTH    =      4 ;window overlapping gadget
WINDOWCLOSE    =      8 ;close gadget
SIZEBRIGHT     =    $10 ;Size gadget on the right
SIZEBBOTTOM    =    $20 ;bottom Size gadget
REFRESHBITS    =    $C0 ;refresh type
SMART_REFRESH  =      0 ;incremental save
SIMPLE_REFRESH =    $40 ;manual refresh
SUPER_BITMAP   =    $80 ;buffer everything
OTHER_REFRESH  =    $C0 ;other methods
BACKDROP       =   $100 ;background window
```

```
REPORTMOUSE     =      $200 ;report mouse movements
GIMMEZEROZERO   =      $400 ;window with border
BORDERLESS      =      $800 ;window without border
ACTIVATE        =     $1000 ;activate upon opening
WINDOWACTIVE    =     $2000 ;currently active window
INREQUEST       =     $4000 ;Requesters available
MENUSTATE       =     $8000 ;menus displayed
RMBTRAP         =    $10000 ;no menu with right mouse button
NOCAREREFRESH   =    $20000 ;no refresh messages
NW_EXTENDED     =    $40000 ;extended NewWindow structure
WINDOWREFRESH   =  $1000000 ;window being refreshed
WBENCHWINDOW    =  $2000000 ;Workbench window
WINDOWTICKED    =  $4000000 ;window received time impulse
VISITOR         =  $8000000 ;Visitor window
ZOOMED          = $10000000 ;zoomed window
HASZOOM         = $20000000 ;window with Zoom gadget
SUPER_UNUSED    = $C0F80000 ;unused bits

DEFAULTMOUSEQUEUE    = 5    ;number of unanswered Msgs

Dec Hex STRUCTURE NewWindow,0
  0  $0 WORD   nw_LeftEdge    ;position
  2  $2 WORD   nw_TopEdge     ;
  4  $4 WORD   nw_Width       ;size
  6  $6 WORD   nw_Height      ;
  8  $8 BYTE   nw_DetailPen   ;meaningless
  9  $9 BYTE   nw_BlockPen    ;
 10  $A LONG   nw_IDCMPFlags  ;IDCMP Flags
 14  $E LONG   nw_Flags       ;Flags (see window)
 18 $12 APTR   nw_FirstGadget ;Gadgets
 22 $16 APTR   nw_CheckMark   ;menu check mark
 26 $1A APTR   nw_Title       ;title
 30 $1E APTR   nw_Screen      ;screen
 34 $22 APTR   nw_BitMap      ;SuperBitMap
 38 $26 WORD   nw_MinWidth    ;min. size
 40 $28 WORD   nw_MinHeight   ;
 42 $2A WORD   nw_MaxWidth    ;max. size
 44 $2C WORD   nw_MaxHeight   ;
 46 $2E WORD   nw_Type        ;screen type
 48 $30 LABEL  nw_SIZE
 48 $30 LABEL  nw_SIZEOF
        STRUCTURE ExtNewWindow,nw_SIZE
 48 $30 APTR   enw_Extension  ;TagItem field
 52 $34 LABEL  enw_SIZEOF

Tags:
WA_Left                  = TAG_USER + 100
WA_Top                   = TAG_USER + 101
```

```
WA_Width              = TAG_USER + 102
WA_Height             = TAG_USER + 103
WA_DetailPen          = TAG_USER + 104
WA_BlockPen           = TAG_USER + 105
WA_IDCMP              = TAG_USER + 106
WA_Flags              = TAG_USER + 107
WA_Gadgets            = TAG_USER + 108
WA_Checkmark          = TAG_USER + 109
WA_Title              = TAG_USER + 110
WA_ScreenTitle        = TAG_USER + 111
WA_CustomScreen       = TAG_USER + 112
WA_SuperBitMap        = TAG_USER + 113
WA_MinWidth           = TAG_USER + 114
WA_MinHeight          = TAG_USER + 115
WA_MaxWidth           = TAG_USER + 116
WA_MaxHeight          = TAG_USER + 117
WA_InnerWidth         = TAG_USER + 118
WA_InnerHeight        = TAG_USER + 119
WA_PubScreenName      = TAG_USER + 120
WA_PubScreen          = TAG_USER + 121
WA_PubScreenFallBack  = TAG_USER + 122
WA_WindowName         = TAG_USER + 123
WA_Colors             = TAG_USER + 124
WA_Zoom               = TAG_USER + 125
WA_MouseQueue         = TAG_USER + 126
WA_BackFill           = TAG_USER + 127
WA_RptQueue           = TAG_USER + 128
WA_SizeGadget         = TAG_USER + 129
WA_DragBar            = TAG_USER + 130
WA_DepthGadget        = TAG_USER + 131
WA_CloseGadget        = TAG_USER + 132
WA_Backdrop           = TAG_USER + 133
WA_ReportMouse        = TAG_USER + 134
WA_NoCareRefresh      = TAG_USER + 135
WA_Borderless         = TAG_USER + 136
WA_Activate           = TAG_USER + 137
WA_RMBTrap            = TAG_USER + 138
WA_WBenchWindow       = TAG_USER + 139
WA_SimpleRefresh      = TAG_USER + 140
WA_SmartRefresh       = TAG_USER + 141
WA_SizeBRight         = TAG_USER + 142
WA_SizeBBottom        = TAG_USER + 143
WA_AutoAdjust         = TAG_USER + 144
WA_GimmeZeroZero      = TAG_USER + 145

Dec Hex STRUCTURE ColorSpec,0 ;18 bit color value (2.0)
  0  $0 WORD   cs_ColorIndex  ;index or -1
  2  $2 UWORD  cs_Red
```

```
  4  $4 UWORD  cs_Green
  6  $6 UWORD  cs_Blue
  8  $8 LABEL  cs_SIZEOF

Dec Hex STRUCTURE Menu,0    ;menu
  0  $0 APTR  mu_NextMenu  ;next menu
  4  $4 WORD  mu_LeftEdge  ;position
  6  $6 WORD  mu_TopEdge   ;
  8  $8 WORD  mu_Width     ;box size
 10  $A WORD  mu_Height    ;
 12  $C WORD  mu_Flags     ;see below
 14  $E APTR  mu_MenuName  ;menu text
 18 $12 APTR  mu_FirstItem ;first MenuItem
 22 $16 WORD  mu_JazzX     ;box with all MenuItems
 24 $18 WORD  mu_JazzY     ;
 26 $1A WORD  mu_BeatX     ;
 28 $1C WORD  mu_BeatY     ;
 30 $1E LABEL mu_SIZEOF

MENUENABLED = 1    ;menu can be selected
MIDRAWN =  $100    ;Items are drawn

Dec Hex STRUCTURE MenuItem,0    ;MenuItem, SubItem
  0  $0 APTR  mi_NextItem      ;next MenuItem
  4  $4 WORD  mi_LeftEdge      ;position
  6  $6 WORD  mi_TopEdge       ;
  8  $8 WORD  mi_Width         ;size
 10  $A WORD  mi_Height        ;
 12  $C WORD  mi_Flags         ;see below
 14  $E LONG  mi_MutualExclude ;exclude
 18 $12 APTR  mi_ItemFill      ;Image, IntuiText or 0
 22 $16 APTR  mi_SelectFill    ;Image, IntuiText or 0
 26 $1A BYTE  mi_Command       ;key code
 27 $1B BYTE  mi_KludgeFill00
 28 $1C APTR  mi_SubItem       ;SubItem (only with MenuItems)
 32 $20 WORD  mi_NextSelect    ;when selected
 34 $22 LABEL mi_SIZEOF

CHECKIT      =     1 ;check when selected
ITEMTEXT     =     2 ;...Fill points to IntuiText
COMMSEQ      =     4 ;with Amiga key code
MENUTOGGLE   =     8 ;toggle when selected
ITEMENABLED =    $10 ;selection possible
HIGHFLAGS    =   $C0 ;display mode for Flags
HIGHIMAGE    =     0 ;mi_SelectFill when activated
HIGHCOMP     =   $40 ;compliment Item region
HIGHBOX      =   $80 ;draw border around item
HIGHNONE     =   $C0 ;do not react
```

```
CHECKED     =  $100 ;if CHECKIT: Item is checked
ISDRAWN     = $1000 ;SubItems are drawn
HIGHITEM    = $2000 ;Item is activated
MENUTOGGLED = $4000 ;Item has been toggled

NOMENU   =   $1F
NOITEM   =   $3F
NOSUB    =   $1F
MENUNULL = $FFFF

CHECKWIDTH     = 19
COMMWIDTH      = 27
LOWCHECKWIDTH  = 13
LOWCOMMWIDTH   = 16

Dec Hex STRUCTURE IntuiMessage,0
  0  $0 STRUCT im_ExecMessage,MN_SIZE
 20 $14 LONG   im_Class         ;IDCMP event
 24 $18 WORD   im_Code          ;associated data (key code, etc.)
 26 $1A WORD   im_Qualifier     ;copy of InputEvent
 28 $1C APTR   im_IAddress      ;object address
 32 $20 WORD   im_MouseX        ;mouse coordinates
 34 $22 WORD   im_MouseY        ;
 36 $24 LONG   im_Seconds       ;time
 40 $28 LONG   im_Micros        ;
 44 $2C APTR   im_IDCMPWindow   ;window
 48 $30 APTR   im_SpecialLink   ;internal link
 52 $34 LABEL  im_SIZEOF

SIZEVERIFY      =         1 ;before change in size
NEWSIZE         =         2 ;new window size
REFRESHWINDOW   =         4 ;refresh window
MOUSEBUTTONS    =         8 ;mouse buttons
MOUSEMOVE       =       $10 ;mouse movements
GADGETDOWN      =       $20 ;gadget selected
GADGETUP        =       $40 ;gadget released
REQSET          =       $80 ;Requester appeared
MENUPICK        =      $100 ;menu selection
CLOSEWINDOW     =      $200 ;close window
RAWKEY          =      $400 ;raw key code
REQVERIFY       =      $800 ;before Requester
REQCLEAR        =     $1000 ;Requester cleared
MENUVERIFY      =     $2000 ;prior to menu display
NEWPREFS        =     $4000 ;preferences changed
DISKINSERTED    =     $8000 ;disk inserted
DISKREMOVED     =    $10000 ;disk removed
WBENCHMESSAGE   =    $20000 ;for Open/CloseWorkbench
ACTIVEWINDOW    =    $40000 ;active window
```

```
INACTIVEWINDOW =    $80000 ;window deactivated
DELTAMOVE      =   $100000 ;relative mouse movement
VANILLAKEY     =   $200000 ;ASCII characters and strings
INTUITICKS     =   $400000 ;1/50 second impulse
IDCMPUPDATE    =   $800000 ;for BOOPSI Gadgets
MENUHELP       =  $1000000 ;HELP with menu selection
CHANGEWINDOW   =  $2000000 ;window position/size changed
LONELYMESSAGE  = $80000000 ;invalid message (internal)

MENUHOT     = 1 ;check MENUCANCEL (MENUVERIFY)
MENUCANCEL  = 2 ;cancel menu operation? (MENUVERIFY)
MENUWAITING = 3 ;waiting for ReplyMsg()

OKOK        = MENUHOT    ;does not matter
OKABORT     = 4          ;aha, draw the window
OKCANCEL    = MENUCANCEL ;aha, cancel

WBENCHOPEN  = 1 ;for WBENCHMESSAGE
WBENCHCLOSE = 2

SELECTUP    = (IECODE_LBUTTON+IECODE_UP_PREFIX)
SELECTDOWN  = (IECODE_LBUTTON)
MENUUP      = (IECODE_RBUTTON+IECODE_UP_PREFIX)
MENUDOWN    = (IECODE_RBUTTON)
ALTLEFT     = (IEQUALIFIER_LALT)
ALTRIGHT    = (IEQUALIFIER_RALT)
AMIGALEFT   = (IEQUALIFIER_LCOMMAND)
AMIGARIGHT  = (IEQUALIFIER_RCOMMAND)
AMIGAKEYS   = (AMIGALEFT+AMIGARIGHT)

CURSORUP    = $4C
CURSORLEFT  = $4F
CURSORRIGHT = $4E
CURSORDOWN  = $4D
KEYCODE_Q   = $10
KEYCODE_X   = $32
KEYCODE_N   = $36
KEYCODE_M   = $37
KEYCODE_V   = $34
KEYCODE_B   = $35
KEYCODE_LESS    = $38
KEYCODE_GREATER = $39
```

3. Requesters

AutoRequest — **Display and query requester**

Call:

```
Response = AutoRequest(Window, BodyText, PosText, NegText, PosFlags, NegFlags)
D0         -348(A6)    A0      A1        A2       A3       D0        D1

BOOL     Response
STRUCT Window *Window
STRUCT IntuiText *BodyText, *PosText, *NegText
ULONG  PosFlags, NegFlags
```

Function: Opens a window and displays the Okay-Cancel requester. Both gadgets can be activated by clicking or by incoming IDCMP events. Warning: prior to AmigaOS 2.0, the size of the requester window must be given - WORD width (D2), height (D3).

Parameters:

Window	Window structure of the window to be locked.
BodyText	IntuiText structure(s) of the requester.
PosText	IntuiText structure for 'Okay' or 0.
NegText	IntuiText structure for 'Cancel'.
PosFlags	IDCMP flags for 'Okay'.
NegFlags	IDCMP flags for 'Cancel'.

Result: 0 'Cancel'

BuildEasyRequestArgs — **Create system requester**

Call:

```
ReqWindow    BuildEasyRequestArgs( RefWindow, easyStruct, IDCMP, Args )
D0           -594(A6)              A0         A1          D0     A3

STRUCT Window *ReqWindow,*RefWindow
STRUCT EasyStruct *easyStruct
ULONG  IDCMP
APTR   Args
```

Function: Displays a requester in a new window.

Parameters:

Window	Window locked by the requester.
easyStruct	EasyStruct of the requesters.
IDCMP	Flags of the requester window.
Args	See EasyRequest()

Result: The address of the requester window or 0 (error, cancel) or 1 (error, continue).

BuildSysRequest **Create system requester (old)**

Call:

```
ReqWindow = BuildSysRequest(Window, BodyText, PosText, NegText, IDCMPFlags)
D0          -360(A6)        A0      A1        A2       A3       D0

STRUCT Window *ReqWindow,*Window
STRUCT IntuiText *BodyText,*PosText,*NegText
ULONG IDCMPFlags
```

Function: Displays a system requester. Warning: prior to AmigaOS 2.0 the window size must be given (WORD Width,Height D2/D3).

Parameters:

Window	Window to be locked
BodyText	Requester text
PosText	Positive gadget text
NegText	Negative gadget text
IDCMPFlags	Flags for the requester window.

Result: Window of the requesters or 0 (error) or 1 (pre-OS 2.0).

ClearDMRequest — **Clear double menu requester**

Call:

```
Response = ClearDMRequest(Window)
D0         -48(A6)        A0

BOOL   Response
STRUCT Window *Window
```

Function: Attempts to remove the requester that appears when the right mouse button is double-clicked.

Parameters: Window — Window with DMRequest

Result: 0 — Requester is active and could not be removed.

DisplayAlert — **Display and query alert message**

Call:

```
Response = DisplayAlert(AlertNumber, String, Height)
D0         -90(A6)      D0           A0      D1

BOOL  Response
ULONG AlertNumber
APTR  String
WORD  Height
```

Function: Displays the text defined in the alarm string on a black display using the Topaz/8 font. DeadEnds are in red and Recoverables are in amber. The alarm string is constructed as follows:

- 16 bit X coordinate
- 8 bit Y coordinate
- String ending with 0
- Byte flag for another string (1 or 0 (=end))

Parameters: AlertNumber
Exec alert code (only bit 31 is important)

String — Alarm string address

Height — Required display height

Result: 0 DeadEnd or right mouse button, TRUE = left mouse button.

Example: Display a multi-line alarm message:

```
...
movea.l _IntuiBase,a6
moveq   #0,d0                ;Recoverable Alert
lea     _Meldung(pc),a0      ;special string
moveq   #38,d1               ;height
jsr     _LVODisplayAlert(a6) ;display
tst.l   d0
bne     _Okay
...

_Meldung
 dc.b   1,40               ;x coordinate: 1*256+40
 dc.b   7+10               ;y coordinate
 dc.b   'Hallo!',0         ;text
 dc.b   1                  ;not ended yet

 dc.b   0,20
 dc.b   7+20
 dc.b   'Links      HALLO!',0
 dc.b   1

 dc.b   1,248
 dc.b   7+20
 dc.b   'Rechts     ?#!$ ',0
 dc.b   0                   ;end
```

EasyRequestArgs — Query with requester

Call:

```
GGNum = EasyRequestArgs( Window, easyStruct, IDCMP_ptr, ArgList )
D0        -588(A6)         A0        A1           A2         A3

STRUCT Window *Window
APTR   IDCMP_ptr,Args
STRUCT EasyStruct *easyStruct
LONG   GGNum
```

Function: Display system requester with desired gadgets and formattable text in such a way that the requester is optimized with respect to the screen resolution and the font size.

Parameters: Window — Parent window or 0 (default PublicScreen, usually Workbench)

IDCMP_ptr — Address of IDCMP flags for ending.

easyStruct — EasyStruct structure:

es_StructSize:	EasyStruct_SIZEOF
es_Flags:	0
es_Title:	Window title or 0 (title of A0)
es_TextFormat:	RawDoFmt() format for query (also '\n')
es_GadgetFormat:	Format string for gadgets; gadgets are separated by '|'.

Args — Arguments for TextFormat, then for GadgetFormat.

Result: Selected gadget number (numbering: 1,2,...,x,0) or -1 (IDCMP: event in *IDCMP_ptr).

Example: Security prompt for a word processor when the ' LOAD TEXT' menu item is selected:

```
tst.b    _FileName          ;load file?
beq      _LoadIt

movea.l _IntuiBase,a6
movea.l _Window,a0
lea      _EasyStruct(pc),a1
lea      _IDCMP(pc),a2
lea      _FileName,a3
jsr      _LVOEasyRequestArgs(a6)
tst.l    d0
bne      _LoadIt
...
```

```
_LoadIt
 ..

_IDCMP
 dc.l 0

_EasyStruct
 dc.l es_SIZEOF,0,_Title,_Fmt,_Buttons

_Title
 dc.b 'load text',0
_Fmt
 dc.b 'the file %s has not yet been saved.',0
_Buttons
 dc.b 'Load|Return',0

 ...

_FileName
 ds.b 256        ;contains current file name
```

EndRequest	**Remove requester**

Call:

```
EndRequest(Requester, Window)
-120(A6)   A0         A1

STRUCT Requester *Requester
STRUCT Window *Window
```

Function: Removes the currently active requester (makes it inactive).

Parameters:

Requester   Requester to be removed

Window   Locked window

FreeSysRequest	**Free a system requester**

Call:

```
FreeSysRequest(Window)
-372(A6)       A0

STRUCT Window *Window
```

Function: Frees a system requester that was created with BuildSysRequest() or BuildEasyRequest().

Parameters: Window — Window of the requester (result of the Build...Request() functions). The values 0 and 1 have no effect.

InitRequester — **Initialize requester structure**

Call:

```
InitRequester(Requester)
-138(A6)       A0

STRUCT Requester *Requester
```

Function: Initializes a requester structure. After the call, the structure must be loaded with user data. This is a C routine, which is seldom or never used because of the speed of execution.

Parameters: Requester — Requester structure

Request — **Display requester**

Call:

```
Success = Request(Requester, Window)
D0        -240(A6) A0        A1

BOOL   Success
STRUCT Requester *Requester
STRUCT Window *Window
```

Function: Displays a requester in a window (POINTREL now works).

Parameters: Requester — Requester to be displayed

Window — Window

Result: 0 — Error

SetDMRequest — **Define double menu requester**

Call:

```
success = SetDMRequest(Window, DMRequester)
D0        -258(A6)     A0      A1

BOOL   success
STRUCT Window *Window
STRUCT Requester *DMRequester
```

Function: Attempts to define the DMRequester. This will not work if another DMRequester is active. POINTREL works.

Parameters: Window Window

DMRequester
Requester for menu key double-click.

Result: 0 Error, DMRequest now active.

SysReqHandler **Query system requester**

Call:

```
num  SysReqHandler( Window, IDCMPFlagsPtr, WaitInput )
D0   -600           A0      A1             D0

STRUCT Window *Window
CPTR   IDCMPFlagsPtr
BOOL   WaitInput
```

Function: Queries a system requester.

Parameters: Window Result from Build...Request().

IDCMPFlagsPtr
Address of the IDCMP flags.

WaitInput Boolean: wait for input.

Result: Like EasyRequest(), -2 also possible (no input).

```
Dec Hex STRUCTURE EasyStruct,0
  0  $0 ULONG   es_StructSize    ;es_SIZEOF
  4  $4 ULONG   es_Flags         ;0
  8  $8 APTR    es_Title         ;Requester title
 12  $C APTR    es_TextFormat    ;format string for BodyText
 16 $10 APTR    es_GadgetFormat  ;format string for Gadgets
 20 $14 LABEL   es_SIZEOF        ;size of structure

Dec Hex STRUCTURE Requester,0
  0  $0 APTR    rq_OlderRequest  ;older Requester
  4  $4 WORD    rq_LeftEdge      ;position
```

```
  6  $6 WORD   rq_TopEdge     ;
  8  $8 WORD   rq_Width       ;size
 10  $A WORD   rq_Height      ;
 12  $C WORD   rq_RelLeft     ;position relative to mouse
 14  $E WORD   rq_RelTop      ;
 16 $10 APTR   rq_ReqGadget   ;Gadgets
 20 $14 APTR   rq_ReqBorder   ;border
 24 $18 APTR   rq_ReqText     ;IntuiTexts
 28 $1C WORD   rq_Flags       ;see below
 30 $1E UBYTE  rq_BackFill    ;Requester color
 31 $1F BYTE   rq_KludgeFill00
 32 $20 APTR   rq_ReqLayer    ;Layer
 36 $24 STRUCT rq_ReqPad1,32
 68 $44 APTR   rq_ImageBMap   ;BitMap with complete Requester
 72 $48 APTR   rq_RWindow     ;window
 76 $4C APTR   rq_ReqImage    ;v2.0: Images after Backfill
 80 $50 STRUCT rq_ReqPad2,32
112 $70 LABEL  rq_SIZEOF

POINTREL      =     1 ;display relative to mouse or center of window
PREDRAWN      =     2 ;graphic from ImageBMap
NOISYREQ      =     4 ;do not filter input
SIMPLEREQ     =   $10 ;with SIMPLEREFRESH Layer (2.0)
USEREQIMAGE   =   $20 ;with Images after Backfill, before GGs
NOREQBACKFILL =   $40 ;do not fill background
REQOFFWINDOW  = $1000 ;Gadget component outside Requester
REQACTIVE     = $2000 ;Requester is active
SYSREQUEST    = $4000 ;Requester generated by system
DEFERREFRESH  = $8000 ;Requester stops Refresh

ALERT_TYPE     = $80000000 ;mask
RECOVERY_ALERT =         0
DEADEND_ALERT  = $80000000 ;crash
```

Example: Display and query EasyRequest. The requester that follows indicates insufficient ChipMem until enough can be reserved after a 'Retry' or 'Cancel' is selected. The result is the memory block allocated with AllocVec():

```
_AskForHelp
 movem.l a2-a4/a6,-(a7)

 movea.l _IntuiBase,a6
 movea.l _Window,a0
 lea     _EasyRequest(pc),a1
 lea     _IDCMP(pc),a2
 lea     _NeededMem(pc),a3
```

```
 jsr      _LVOBuildEasyRequestArgs(a6)
 subq.l   #1,d0
 ble.s    _Failure
 addq.l   #1,d0
 movea.l d0,a4

_Loop
 movea.l a4,a0
 lea      _IDCMP(pc),a1
 moveq    #-1,d0
 jsr      _LVOSysReqHandler(a6)
 tst.l    d0
 beq.s    _Break
 bpl.s    _Retry
 addq.l   #2,d0
 beq.s    _Loop
 bra.s    _Break

_Retry
 movea.l _SysBase,a6
 move.l   (a3),d0
 move.l   #MEMF_CLEAR!MEMF_CHIP,d1
 jsr      _LVOAllocVec(a6)
 movea.l _IntuiBase,a6
 tst.l    d0
 beq.s    _Loop

_Break
 movea.l d0,a3
 movea.l a4,a0
 jsr      _LVOFreeSysRequest(a6)
 move.l   a3,d0
 bra.s    _Exit

_Failure
 moveq    #0,d0

_Exit
 movem.l (a7)+,a2-a4/a6
 rts

_NeededMem
 dc.l 256000

_IDCMP
 dc.l 0

_EasyRequest
```

```
 dc.l es_SIZEOF,0,_Title,_Fmt,_Buttons

_Title
 dc.b 'Not enough memory',0
_Fmt
 dc.b 'I need %ld byte chipmem!',0
_Buttons
 dc.b 'Retry|Cancel',0
```

4. Gadgets

ActivateGadget — Activate string or GadTools gadget

Call:

```
Success = ActivateGadget(Gadget, Window, Request)
D0        -462(A6)       A0      A1      A2

BOOL   Success
STRUCT Gadget *Gadget
STRUCT Window *Window
STRUCT Requester *Request
```

Function: Activates a string or CustomGadget.

Parameters:

Gadget	String or CustomGadget
Window	Active window with the gadget
Requester	Requester if GTYP_REQGADGET

Result:

0	Gadget not activated

AddGadget — Add gadget to the window list

Call:

```
RealPosition = AddGadget(Window, Gadget, Position)
D0             -42(A6)   A0      A1      D0

UWORD  RealPosition,Position
STRUCT Window *Window
STRUCT Gadget *Gadget
```

Function: Adds a gadget to a window list.

Parameters: Window — Window structure of the window.

Gadget	Gadget structure
Position	Position in the list (0...65535).

Result: Position at which the gadget was inserted in the list (0...65535).

AddGList — Add gadgets to the window list

Call:

```
RealPosition = AddGList(Window, Gadget, Position, Numgad, Requester)
D0             -438(A6) A0      A1      D0        D1      A2

UWORD  RealPosition,Position,Numgad
STRUCT Window *Window
STRUCT Gadget *Gadget
STRUCT Requester *Requester
```

Function: Adds liked gadgets to a window list.

Parameters:

Window	Window structure of the window.
Gadget	First gadget to insert.
Position	Position in the list (0...65536 NumGads).
Numgad	Number of gadgets or -1 (all).
Requester	Requester, if GTYP_REQGADGET.

Result: Position at which the gadgets were inserted in the list (0...65535).

GadgetMouse — Gadget-relative mouse position

Call:

```
GadgetMouse( Gadget, GInfo, MousePoint )
-570(A6)     A0      A1     A2

STRUCT Gadget *Gadget
STRUCT GadgetInfo *GInfo
STRUCT Point *MousePoint
```

Function: Calculates the gadget-relative mouse position (completely meaningless, since this information is always available).

Parameters: GInfo GadgetInfo structure for the hook routine.

MousePoint
Address of two words for the position.

Gadget Desired gadget

ModifyProp — Modify proportional gadget

Call:

```
ModifyProp(Gadget, Window, Requester, Flags, HorizPot, VertPot,
HorizBody, VertBody)
-156(A6)    A0      A1      A2      D0      D1      D2      D3
D4

STRUCT Gadget *Gadget
STRUCT Window *Window
STRUCT Requester *Requester
UWORD  Flags,HorizPot,VertPot,HorizBody, VertBody
```

Function: Changes the contents of a PropGadget and executes a complete refresh of the gadget and all following gadgets.

Parameters: Gadget PropGadget

Window Window of the gadgets

Requester Requester or 0

Flags,...Pot,...Body
Values for PropInfo

NewModifyProp — Change proportional gadget

Call:

```
NewModifyProp(Gadget, Window, Requester, Flags, HorizPot, VertPot, HorizBody,
VertBody, NumGad)
-468(A6)    A0      A1      A2      D0      D1      D2      D3
D4      D5

STRUCT Gadget *Gadget
```

```
STRUCT Window *Window
STRUCT Requester *Requester
UWORD  Flags,HorizPot,VertPot,HorizBody,VertBody
WORD   NumGad
```

Function: Like ModifyProp(), but this function allows you to specify how many gadgets should be refreshed. A value of 1 will cause only the knob to be refreshed.

Parameters: NumGad — Number of gadgets to refresh, -1 (all), or 1 (knob only).

... — See ModifyProp()

ObtainGIRPort — Allocate RastPort for a CustomGadget

Call:

```
RPort = ObtainGIRPort( GInfo )
D0      -558(A6)       A0

STRUCT RastPort *RPort
STRUCT GadgetInfo *GInfo
```

Function: Allocates the RastPort of a CustomGadget and initializes it for a hook routine.

Parameters: GInfo — GadgetInfo structure of the CustomGadget.

Result: RastPort or 0

OffGadget — Turn gadget off

Call:

```
OffGadget(Gadget, Window, Requester)
-174(A6)  A0      A1      A2

STRUCT Gadget *Gadget
STRUCT Window *Window
STRUCT Requester *Requester
```

Function: Turns a gadget off. The gadget is displayed as a ghost and cannot be selected. Also refreshes all gadgets.

Parameters: Gadget — Gadget to turn off

Window	Gadget window
Requester	Requester if GTYP_REQGADGET

OnGadget — **Turn gadget on**

Call:

```
OnGadget(Gadget, Window, Requester)
-186(A6) A0      A1      A2

STRUCT Gadget *Gadget
STRUCT Window *Window
STRUCT Requester *Requester
```

Functions, Parameters:
Opposite of OffGadget(). Allows selection of the gadget again.

RefreshGadgets — **Refresh gadgets**

Call:

```
RefreshGadgets(Gadgets, Window, Requester)
-222(A6)       A0       A1      A2

STRUCT Gadget *Gadgets
STRUCT Window *Window
STRUCT Requester *Requester
```

Function: Refreshes all GGs starting with the given gadget.

Parameters:

Gadgets	Address of the first gadget.
Window	Gadget window
Requester	Requester or 0

RefreshGList — **RefreshGList**

Call:

```
RefreshGList(Gadgets, Window, Requester, NumGad)
-432(A6)     A0       A1      A2         D0

STRUCT Gadget *Gadgets
STRUCT Window *Window
STRUCT Requester *Requester
```

```
WORD    NumGad
```

Function: Similar to RefreshGadgets(), but only allows you to specify the number of gadgets to refresh.

Parameters: NumGad — Number of gadgets, -1 (all) or -2 (all requester gadgets).

... — See RefreshGadgets()

ReleaseGIRPort	**Free CustomGadget RastPort**

Call:

```
ReleaseGIRPort( RPort )
-564(A6)        A0

STRUCT RastPort *RPort
```

Function: Free a RastPort allocated with ObtainGIRPort().

Parameters: RPort — Result of ObtainGIRPort() or 0.

RemoveGadget	**Remove a gadget from the window list**

Call:

```
Position = RemoveGadget(Window, Gadget)
D0         -228(A6)     A0      A1

UWORD  Position
STRUCT Window *Window
STRUCT Gadget *Gadget
```

Function: Removes a gadget from the window list. If it's active, it is first deactivated.

Parameters: Window — Gadget window

Gadget — Gadget to be removed

Result: Position of the gadget or -1 if it was not in the list (or gadget #65535).

RemoveGList — **Remove gadgets from the window list**

Call:

```
Position = RemoveGList(Window, Gadget, Numgad)
D0         -444(A6)    A0      A1      D0

UWORD  Position
STRUCT Window *Window
STRUCT Gadget *Gadget
WORD   Numgad
```

Function: Removes several gadgets from the window list and clears gg_NextGadget for the last gadget removed. If the active gadget is included, it is first deactivated.

Parameters, Result:

Numgad — Number of gadgets or -1 (all)

... — See RemoveGadget()

ReportMouse — **Change ReportMouse flag**

Call:

```
ReportMouse(Window,Boolean)
-234(A6)    A0     D0

BOOL   Boolean
STRUCT Window *Window
```

Function: Changes the ReportMouse flag of the window and the FollowMouse flag of the active gadget. If a gadget is active when the call is made, the change is only good at the time of gadget activation. C compilers often make errors with this function because the order of the two parameters is often switched.

Parameters: Window — Window

Boolean — TRUE or 0 (bit status)

SetEditHook — **Set StringGadget hook**

Call:

```
OldHook = SetEditHook( Hook)
D0        -492(A6)     A0
```

```
STRUCT Hook *OldHook,*Hook
```

Function: Defines the global editor hook for StringGadgets. This does not just include its own gadgets; this should be used only in highly optimized Assembler code.

Parameters: Hook — Hook with editor routine for ALL StringGadgets.

Result: Hook of the previous editor routine.

Warning: Since this routine has not been tested by Commodore yet, you should not use it.

SetGadgetAttrsA — Set gadget attributes of a BoopsiGadget

Call:

```
Result = SetGadgetAttrsA( Gadget, Window, Requester, TagList )
D0   -660(A6)          A0      A1      A2         A3

STRUCT Gadget *Gadget
STRUCT Window *Window
STRUCT Requester *Requester
STRUCT TagItem *TagList
LONG Result
```

Function: Like SetAttrs(), with context information for CustomGadgets.

Parameters:

Gadget — Boopsi object

Window — Object's window

Requester — For REQGADGETs

TagList — TagItem field

Result: Not 0: Gadget must be refreshed to display the new attributes.

```
Dec Hex STRUCTURE GadgetInfo,0
  0  $0 APTR   ggi_Screen
  4  $4 APTR   ggi_Window
  8  $8 APTR   ggi_Requester
 12  $C APTR   ggi_RastPort
 16 $10 APTR   ggi_Layer
 20 $14 STRUCT ggi_Domain,ibox_SIZEOF
 28 $1C STRUCT ggi_Pens,2
 30 $20 APTR   ggi_DrInfo

Dec Hex STRUCTURE IBox,0
  0  $0 WORD   ibox_Left
  2  $2 WORD   ibox_Top
  4  $4 WORD   ibox_Width
  6  $6 WORD   ibox_Height
  8  $8 LABEL  ibox_SIZEOF

Dec Hex STRUCTURE Gadget,0
  0  $0 APTR   gg_NextGadget    ;next Gadget
  4  $4 WORD   gg_LeftEdge      ;position
  6  $6 WORD   gg_TopEdge       ;
  8  $8 WORD   gg_Width         ;size
 10  $A WORD   gg_Height        ;
 12  $C WORD   gg_Flags         ;see below
 14  $E WORD   gg_Activation    ;see below
 16 $10 WORD   gg_GadgetType    ;see below
 18 $12 APTR   gg_GadgetRender  ;Border, Image or Null
 22 $16 APTR   gg_SelectRender  ;Border, Image or Null
 26 $1A APTR   gg_GadgetText    ;IntuiText or Null
 30 $1E LONG   gg_MutualExclude ;CustomGadget Hook
 34 $22 APTR   gg_SpecialInfo   ;according to GadgetType
 38 $26 WORD   gg_GadgetID      ;User ID
 40 $28 APTR   gg_UserData      ;User data
 44 $2C LABEL  gg_SIZEOF

GADGHIGHBITS =     3 ;selection Flags
GADGHCOMP    =     0 ;complement
GADGHBOX     =     1 ;box
GADGHIMAGE   =     2 ;use SelectRender
GADGHNONE    =     3 ;no reaction
GADGIMAGE    =     4 ;...Render is Image structure
GRELBOTTOM   =     8 ;coordinates relative to bottom
GRELRIGHT    =   $10 ;coordinates relative to right edge
GRELWIDTH    =   $20 ;width relative to window width
GRELHEIGHT   =   $40 ;height relative to window height
SELECTED     =   $80 ;Gadget is in selected mode
GADGDISABLED =  $100 ;Gadget is disabled
LABELMASK    = $3000 ;meaning of gg_GadgetText
```

```
LABELITEXT    =     0 ;GadgetText is IntuiText
LABELSTRING   = $1000 ;GadgetText is string
LABELIMAGE    = $2000 ;GadgetText is BoopsiImage

RELVERIFY      =     1 ;activation: only within Box
GADGIMMEDIATE =      2 ;activate immediately
ENDGADGET      =     4 ;ends Requester
FOLLOWMOUSE    =     8 ;ReportMouse during selection
RIGHTBORDER    =   $10 ;right border
LEFTBORDER     =   $20 ;left border
TOPBORDER      =   $40 ;title bar
BOTTOMBORDER   =   $80 ;bottom border
BORDERSNIFF    = $8000 ;private
TOGGLESELECT   =  $100 ;toggle when selected
BOOLEXTEND     = $2000 ;BoolInfo in gg_SpecialInfo
STRINGCENTER   =  $200 ;center StringGG contents
STRINGRIGHT    =  $400 ;right justify StringGG contents
LONGINT        =  $800 ;StringGadget for integer values
ALTKEYMAP      = $1000 ;StringGadget with another KeyMap
STRINGEXTEND   = $2000 ;StringGadget extended
ACTIVEGADGET   = $4000 ;Gadget is active

GADGETTYPE    = $FC00 ;global GadgetTypes
SYSGADGET     = $8000 ;operating system Gadget
SCRGADGET     = $4000 ;screen Gadget
GZZGADGET     = $2000 ;Gadget for window borders
REQGADGET     = $1000 ;Requester Gadget
SIZING        =   $10 ;sizing Gadget
WDRAGGING     =   $20 ;movable title bar
SDRAGGING     =   $30 ;same for Screens
WUPFRONT      =   $40 ;window to front
SUPFRONT      =   $50 ;screen to front
WDOWNBACK     =   $60 ;
SDOWNBACK     =   $70
CLOSE         =   $80 ;close Gadget
BOOLGADGET    =     1 ;BoolGadget
GADGET0002    =     2
PROPGADGET    =     3 ;PropGadget
STRGADGET     =     4 ;StringGadget
CUSTOMGADGET =      5 ;CustomGadget

Dec Hex STRUCTURE BoolInfo,0
  0  $0 WORD   bi_Flags     ;BOOLMASK
  2  $2 APTR   bi_Mask      ;bit mask, image
  6  $6 LONG   bi_Reserved ;0
 10  $A LABEL  bi_SIZEOF

BOOLMASK      =     1 ;mask
```

```
Dec Hex STRUCTURE PropInfo,0
  0  $0 WORD   pi_Flags      ;Flags s.u.
  2  $2 WORD   pi_HorizPot   ;position
  4  $4 WORD   pi_VertPot    ;
  6  $6 WORD   pi_HorizBody  ;slider size
  8  $8 WORD   pi_VertBody   ;
 10  $A WORD   pi_CWidth     ;container size
 12  $C WORD   pi_CHeight    ;
 14  $E WORD   pi_HPotRes    ;slider resolution
 16 $10 WORD   pi_VPotRes    ;
 18 $12 WORD   pi_LeftBorder ;border size
 20 $14 WORD   pi_TopBorder  ;
 22 $16 LABEL  pi_SIZEOF

AUTOKNOB       =     1 ;old AutoKnob
FREEHORIZ      =     2 ;horizontally movable
FREEVERT       =     4 ;vertically movable
PROPBORDERLESS =     8 ;no border
KNOBHIT        =  $100 ;selected Knob

KNOBHMIN       =     6 ;minimal horizontal size
KNOBVMIN       =     4 ;minimal vertical size
MAXBODY        = $FFFF ;maximum Knob size
MAXPOT         = $FFFF ;maximum position

Dec Hex STRUCTURE StringInfo,0
  0  $0 APTR   si_Buffer     ;buffer for the contents
  4  $4 APTR   si_UndoBuffer ;buffer for the Undo function
  8  $8 WORD   si_BufferPos  ;character position in buffer
 10  $A WORD   si_MaxChars   ;buffer size including 0 byte
 12  $C WORD   si_DispPos    ;offset of the first displayed character
 14  $E WORD   si_UndoPos    ;position in Undo buffer
 16 $10 WORD   si_NumChars   ;length of string in buffer
 18 $12 WORD   si_DispCount  ;number of visible characters
 20 $14 WORD   si_CLeft      ;offset in Gadget
 22 $16 WORD   si_CTop       ;
 24 $18 APTR   si_Extension  ;extension structure (2.0)
 28 $1C LONG   si_LongInt    ;value for integer Gadgets
 32 $20 APTR   si_AltKeyMap  ;custom key map
 36 $22 LABEL  si_SIZEOF

Dec Hex STRUCTURE StringExtend,0
  0  $0 APTR   sex_Font          ;TextFont (open)
  4  $4 STRUCT sex_Pens,2        ;colors: text, background
  6  $6 STRUCT sex_ActivePens,2  ;colors when activated
  8  $8 ULONG  sex_InitialModes  ;Flags
 12  $C APTR   sex_EditHook      ;edit Hook
```

```
 16 $10 APTR   sex_WorkBuffer   ;StringInfo.buffer length
 20 $14 STRUCT sex_Reserved,16  ;0
 36 $24 LABEL  sex_SIZEOF


Dec Hex STRUCTURE SGWork,0
  0  $0 APTR   sgw_Gadget      ;Gadget
  4  $4 APTR   sgw_StringInfo  ;StringInfo
  8  $8 APTR   sgw_WorkBuffer  ;Intuition's result
 12  $C APTR   sgw_PrevBuffer  ;previous contents
 16 $10 ULONG  sgw_Modes       ;current Flags
 20 $14 APTR   sgw_IEvent      ;InputEvent
 24 $18 UWORD  sgw_Code        ;character
 26 $1A WORD   sgw_BufferPos   ;CursorPosition
 28 $1C WORD   sgw_NumChars    ;number of characters
 30 $1E ULONG  sgw_Actions     ;what Intuition wants to do
 34 $22 LONG   sgw_LongInt     ;value for integer Gadget
 38 $26 APTR   sgw_GadgetInfo  ;GadgetInfo
 42 $2A UWORD  sgw_EditOp      ;editor operation
 44 $2C LABEL  sgw_SIZEOF      ;current structure size

EditOps:
EO_NOOP        =  1 ;nothing
EO_DELBACKWARD =  2 ;number of characters to delete (0 allowed)
EO_DELFORWARD  =  3 ;number of characters under/before cursor to delete
EO_MOVECURSOR  =  4 ;move cursor
EO_ENTER       =  5 ;ENTER or LF
EO_RESET       =  6 ;undo
EO_REPLACECHAR =  7 ;replace character
EO_INSERTCHAR  =  8 ;insert character
EO_BADFORMAT   =  9 ;bad input (IntegerGadget)
EO_BIGCHANGE   = 10 ;text completely changed
EO_UNDO        = 11 ;other Undo operations
EO_CLEAR       = 12 ;clear string
EO_SPECIAL     = 13 ;special functions

SGM_REPLACE     = 1 ;modes
SGMB_REPLACE    = 0
SGMF_REPLACE    = 1

SGM_FIXEDFIELD  = 2
SGMB_FIXEDFIELD = 1
SGMF_FIXEDFIELD = 2

SGM_NOFILTER    = 4 ;do not filter control
SGMB_NOFILTER   = 2
SGMF_NOFILTER   = 4
```

```
SGA_USE       = 1     ;take contents from SGWork
SGAB_USE      = 0
SGAF_USE      = 1

SGA_END       = 2     ;end
SGAB_END      = 1
SGAF_END      = 2

SGA_BEEP      = 4     ;DisplayBeep()
SGAB_BEEP     = 2
SGAF_BEEP     = 4

SGA_REUSE     = 8     ;reuse InputEvent
SGAB_REUSE    = 3
SGAF_REUSE    = 8

SGA_REDISPLAY    = $10 ;Gadget appearance changed
SGAB_REDISPLAY   =   4
SGAF_REDISPLAY   = $10

SGH_KEY       = 1   ;process keystroke
SGH_CLICK     = 2   ;process mouse click
```

5. Output Functions

DisplayBeep	**Cause display to blink**

Call: `DisplayBeep(Screen)`
`-96(A6)      A0`

`STRUCT Screen *Screen`

Function: Causes the entire display or a given screen to blink. This function may be patched.

Parameters: Screen   Screen to blink or 0

DrawBorder	**Draw a border**

Call: `DrawBorder(RastPort, Border, LeftOffset, TopOffset)`
`-108(A6)   A0         A1      D0          D1`

```
STRUCT RastPort *RastPort
STRUCT Border *Border
WORD LeftOffset, TopOffset
```

Function: Draws the border(s) defined in the given Border structure(s).

Parameters:

RastPort	RastPort
Border	Border structure
...Offset	Position added to the border vectors.

DrawImage **Draw an image**

Call:

```
DrawImage(RastPort, Image, LeftOffset, TopOffset)
-114(A6)  A0        A1     D0          D1

STRUCT RastPort *RastPort
STRUCT Image *Image
WORD LeftOffset, TopOffset
```

Function: Copies one or more bit BitImages to the given RastPort.

Parameters:

RastPort	RastPort
Image	Image structure
...Offset	Position added to the image position.

DrawImageState **Draw extended image**

Call:

```
DrawImageState(RPort, Image, LeftOffset, TopOffset, State, DrawInfo)
-618(A6)       A0     A1     D0          D1         D2     A2

STRUCT RastPort *RPort
STRUCT Image *Image
WORD  LeftOffset,TopOffset
ULONG  State
STRUCT DrawInfo *DrawInfo
```

Function: Draws a bit image of the desired type:

```
IDS_NORMAL         = as with DrawImage()
IDS_SELECTED       = as in the selected Gadget
IDS_DISABLED       = as with disabled Gadgets
```

```
IDS_BUSY              = not yet supported
IDS_INDETERMINANT     = not yet supported
IDS_INACTIVENORMAL    = for Gadgets in window borders
IDS_INACTIVESELECTED  = for Gadgets in window borders
IDS_INACTIVEDISABLED  = for Gadgets in window borders
```

Parameters: RPort RastPort

Image Image, CustomImage, etc.

...Offset Position (offset)

State IDS_...

DrawInfo Information on how to display the image.

EraseImage — Erase an image

Call:

```
EraseImage( RPort, Image, LeftOffset, TopOffset )
-630(A6)    A0     A1     D0          D1

STRUCT RastPort *RPort
STRUCT Image *Image
WORD   LeftOffset, TopOffset
```

Function: Removes an image, usually using graphics/EraseRect(). For custom images, it depends on the image type.

Parameters: RPort RastPort

Image Image or CustomImage

LeftOffset,RightOffset
Image position offset

IntuiTextLength — TextLength for IntuiText structure

Call:

```
length = IntuiTextLength(IText)
D0       -330(A6)        A0

LONG   length
STRUCT IntuiText *IText
```

Function: Gets the output width of an IntuiText structure in pixels.

Parameters: IText IntuiText structure

Result: Output width

PrintIText **Output IntuiText**

Call:

```
PrintIText(RastPort, IText, LeftOffset, TopOffset)
-216(A6)   A0        A1     D0          D1

STRUCT RastPort *RastPort
STRUCT IntuiText *IText
WORD   LeftOffset, TopOffset
```

Function: Outputs the text(s) defined in the given IntuiText structure(s) at the given position (offset).

Parameters:

RastPort RastPort structure

IText IntuiText structure(s)

LeftOffset, TopOffset
Position

```
Dec Hex STRUCTURE IntuiText,0
  0  $0 BYTE   it_FrontPen      ;foreground color
  1  $1 BYTE   it_BackPen       ;background color
  2  $2 BYTE   it_DrawMode      ;draw mode
  3  $3 BYTE   it_KludgeFill00
  4  $4 WORD   it_LeftEdge      ;relative position
  6  $6 WORD   it_TopEdge       ;
  8  $8 APTR   it_ITextFont     ;TextAttr structure
 12  $C APTR   it_IText         ;string
 16 $10 APTR   it_NextText      ;next IntuiText structure
 20 $14 LABEL  it_SIZEOF

AUTOFRONTPEN    = 0
AUTOBACKPEN     = 1
AUTODRAWMODE    = RP_JAM2
AUTOLEFTEDGE    = 6
AUTOTOPEDGE     = 3
AUTOITEXTFONT   = 0
AUTONEXTTEXT    = 0
```

```
Dec Hex STRUCTURE Border,0
  0  $0 WORD    bd_LeftEdge   ;relative position
  2  $2 WORD    bd_TopEdge    ;
  4  $4 BYTE    bd_FrontPen   ;foreground color
  5  $5 BYTE    bd_BackPen    ;background color
  6  $6 BYTE    bd_DrawMode   ;draw mode
  7  $7 BYTE    bd_Count      ;number of vectors
  8  $8 APTR    bd_XY         ;vector table (2 Words each)
 12  $C APTR    bd_NextBorder ;next Border structure
 16 $10 LABEL   bd_SIZEOF

Dec Hex STRUCTURE Image,0
  0  $0 WORD    ig_LeftEdge   ;relative position
  2  $2 WORD    ig_TopEdge    ;
  4  $4 WORD    ig_Width      ;size
  6  $6 WORD    ig_Height     ;
  8  $8 WORD    ig_Depth      ;
 10  $A APTR    ig_ImageData  ;Bitplanes
 14  $E BYTE    ig_PlanePick  ;destination plane to be used
 15  $F BYTE    ig_PlaneOnOff ;what happens with the others
 16 $10 APTR    ig_NextImage  ;next Image structure
 20 $14 LABEL   ig_SIZEOF
```

6. Other Functions

AddClass — Add IClass

Call:

```
AddClass( Class )
-684(A6)  A0

STRUCT IClass *Class
```

Function: Adds an IClass from MakeClass() to the system list.

Parameters: Class — Result from MakeClass()

AllocRemember — Allocate and remember memory block

Call:

```
MemBlock = AllocRemember(RememberKey, Size, Flags)
D0         -396(A6)     A0           D0    D1

APTR   MemBlock
STRUCT Remember **RememberKey
ULONG  Size,Flags
```

Function: Uses AllocMem() to allocate a memory block. The position and size of the reserved block is held in a RememberNode which is added to a list so that all blocks can be freed with FreeRemember() later.

Parameters: RememberKey
Address of a longword that contains the address of the first RememberNode. The first time the function is called, this longword must be initialized with the value 0.

Size,Flags Arguments for exec/AllocMem(Size,Flags).

Result: Address of the allocated memory block or 0.

Example: Allocate and free memory block(s):

```
movea.l _IntuiBase,a6
clr.l   -(a7)                  ;RememberKey=Null

movea.l a7,a0                  ;RememberKey
moveq   #rp_SIZEOF,d0          ;buffer size
move.l  #MEMF_CLEAR!MEMF_PUBLIC ;memory type
jsr     _LVOAllocRemember(a6)  ;allocate
tst.l   d0                     ;test
beq     _Zerror                ;if error

...                            ;use the memory

movea.l a7,a0                  ;RememberKey
moveq   #bm_SIZEOF,d0          ;buffer size
move.l  #MEMF_CLEAR!MEMF_PUBLIC ;memory type
jsr     _LVOAllocRemember(a6)  ;allocate
tst.l   d0                     ;test
beq     _Zerror                ;if error

...                            ;use memory

_Zerror
movea.l a7,a0                  ;RememberKey
moveq   #-1,d0                 ;clear all
jsr     _LVOFreeRemember(a6)   ;free
addq.l  #4,a7                  ;restore stack

...
```

CurrentTime — **Get the current time**

Call:

```
CurrentTime(Seconds, Micros)
-84(A6)       A0       A1

APTR Seconds,Micros
```

Function: Writes the current time to the longword at the given address. The microsecond value is not exact.

Parameters:

Seconds — Address of longword for seconds.

Micros — Address of longword for microseconds.

DisposeObject — **Delete an object**

Call:

```
DisposeObject( Object )
-642(A6)       A0

APTR Object
```

Function: Deletes the given object including its data and subobjects.

Parameters:

Object — Result of NewObject()

DoubleClick — **Compare two mouse clicks to the double click period**

Call:

```
IsDouble = DoubleClick(StartSecs, StartMicros, CurrentSecs, CurrentMicros)
D0         -102(A6)    D0         D1           D2           D3

BOOL IsDouble
ULONG StartSecs, StartMicros, CurrentSecs, CurrentMicros
```

Function: Compares the time difference between two mouse clicks to the length of the double-click period.

Parameters:

Start... — Time of the first mouse click.

Current... — Time point of the second mouse click.

Result:

0 — Time points too far apart for a double-click.

Example: Evaluate mouse click for IDCMP messages of type MOUSEBUTTONS:

```
**
** IntuiMessage in a1
**
** Result: d0 >< 0 for double-click
**
 ...
_MouseButtons
 ...
 movem.l d2-d3/a1/a6,-(a7)
 movea.l _IntuiBase,a6
 movem.l im_Seconds(a1),d2-d3
 lea     _OldValues(pc),a0
 movem.l (a0),d0-d1
 movem.l d2-d3,(a0)
 jsr     _LVODoubleClick(a6)
 tst.l   d0
 movem.l (a7)+,d2-d3/a1/a6
 ...
_OldValues
 ds.l    2
 ...
```

FreeClass **Free IClass**

Call:

```
success = FreeClass( ClassPtr )
D0        -714(A6)   A0

STRUCT IClass *ClassPtr
```

Function: Attempts to free the result of a MakeClass() call.

Parameters: ClassPtr IClass structure

Result: 0 IClass could not be freed.

FreeRemember **Free memory and/or Remember structures**

Call:

```
FreeRemember(RememberKey, ReallyForget)
-408(A6)     A0           D0

STRUCT Remember **RememberKey
```

```
BOOL   ReallyForget
```

Function: Frees only the Remember structures (ReallyForget = 0) or the associated memory blocks.

Parameters: RememberKey

Address of the longword containing the address of the first Remember structure.

ReallyForget

Flag that indicates whether memory blocks should also be set free.

Example: Allocate several memory blocks that can only be used if no errors occur. bit-planes are not much good without bit-maps, and bit-maps can't be used without RastPorts:

```
 movea.l _IntuiBase,a6
 clr.l   -(a7)                      ;RememberKey = 0
 moveq   #-1,d2                     ;error, free everything

 movea.l a7,a0                      ;RememberKey
 moveq   #rp_SIZEOF,d0              ;RastPort size
 move.l  #MEMF_CLEAR!MEMF_PUBLIC,d1 ;memory type
 jsr     _LVOAllocRemember(a6)      ;allocate
 move.l  d0,d3                      ;save result
 beq.s   .Zerror                    ;if error

 movea.l a7,a0                      ;RememberKey
 moveq   #bm_SIZEOF,d0              ;BitMap size
 move.l  #MEMF_CLEAR!MEMF_PUBLIC,d1 ;memory type
 jsr     _LVOAllocRemember(a6)      ;allocate
 move.l  d0,d4                      ;save result
 beq.s   .Zerror                    ;if error

 ...

 moveq   #0,d2                      ;no errors, just free Remember structures

.Zerror
 movea.l a7,a0                      ;RememberKey
 move.l  d2,d0                      ;Remember structures or everything
 jsr     _LVOFreeRemember(a6)       ;free
 move.l  d2,d0                      ;return error code

 ...
```

GetAttr **Get object attributes**

Call:
```
GetAttr( AttrID, Object, StoragePtr )
-654(A6) D0      A0      A1

ULONG result,AttrID
APTR  Object,StoragePtr
```

Function: Returns the attribute values for the given object.

Parameters:

AttrID Attribute ID

Object Object address

StoragePtr Address of longword for result.

GetDefPrefs **Get default Preferences**

Call:
```
Prefs = GetDefPrefs(PrefBuffer, Size)
D0      -126(A6)    A0          D0

STRUCT Preferences *Prefs,*PrefBuffer
WORD Size
```

Function: Copies the default Preferences structure to a buffer.

Parameters:

PrefBuffer Buffer for the Preferences structure.

Size Buffer size

Result: Buffer address

GetPrefs **Get the current Preferences**

Call:
```
Prefs = GetPrefs(PrefBuffer, Size)
D0      -132(A6) A0          D0

STRUCT Preferences *Prefs,*PrefBuffer
WORD   Size
```

Function: Copies the current Preferences structure to a buffer.

Parameters: PrefBuffer Buffer for the Preferences structure.

Size Buffer size

Result: Buffer address

LockIBase **Lock IntuitionBase**

Call:

```
Lock = LockIBase(LockNumber)
D0     -414(A6)  D0

ULONG Lock,LockNumber
```

Function: Locks one or more Intuition functions. This is required for operations such as dynamic entries in the IntuiBase structure.

Parameters: LockNumber
Number of the internal SignalSemaphore or 0 (almost all SSs).

Result: Number of the allocated SignalSemaphore or 0 (almost all).

MakeClass **Define object class**

Call:

```
IClass = MakeClass( ClassID, SuperClassID, SuperClassPtr, InstanceSize, Flags)
D0       -678(A6)   A0       A1            A2             D0            D1

STRUCT IClass *IClass,*SuperClassPtr
APTR   ClassID,SuperClassID
UWORD  InstanceSize
ULONG  Flags
```

Function: Defines a new object class. The object class must be registered with Commodore.

Parameters: ClassID PublicClass name or 0 (PrivateClass)

SuperClassID
Superclass name or 0 (PrivateClass)

SuperClassPtr
Private SuperClass address

InstanceSize
Object data structure size

Flags 0

Result: IClass or 0

NewObjectA — **Create a new object**

Call:

```
Object = NewObjectA( class, classID, tagList )
D0       -636(A6)    A0     A1       A2

APTR   Object,classID
STRUCT IClass *class
STRUCT TagItem *tagList
```

Function: Create a Boopsi class object (Boopsi = Basic object-oriented Programming System for Intuition).

Parameters:

class	BoopsiClass from MakeClass()
classID	Name if class=0
tagList	TagItems for the object

Result: Object that may be used, for example, as a gadget or image.

NextObject — **Get the next object**

Call:

```
Object = NextObject ( objectPtrPtr )
D0       -666(A6)     A0

APTR Object,objectPtrPtr
```

Function: Gets the next object entered in a list by OM_ADDMEMBER.

Parameters: objectPtrPtr Address of the list or an object.

Result: Object or 0

PointInImage — **Checks to see if a point is in an Image**

Call:

```
DoesContain = PointInImage( Point, Image )
D0            -624(A6)      D0     A0

BOOL DoesContain
STRUCT Point Point (LONG)
STRUCT Image *Image
```

Function: Checks to see if a point at the given coordinates in the Image is set (for BOOLMASK, etc.).

Parameters: Point — X<<16!Y (packed coordinates)

Image — Image or CustomImage

Result: 0 — Point not set

RemoveClass — **Remove Boopsi class from system list**

Call:

```
RemoveClass( classPtr )
-708(A6)     A0

STRUCT IClass *classPtr
```

Function: Removes an IClass from the system list.

Parameters: ClassPtr — Result from MakeClass()

SetAttrsA — **Set object attributes**

Call:

```
result = SetAttrsA( Object, TagList )
D0       -648(A6)   A0      A1

APTR   Object
STRUCT TagItem *TagList
ULONG result
```

Function: Defines a set of attributes for a Boopsi object.

Parameters:	Object	Object
	TagList	TagItem field

Result: Not 0 if the object is a gadget and should be refreshed in order for the new attributes to be displayed.

SetPrefs	**Change the Preferences settings**

Call:

```
Prefs = SetPrefs(PrefBuffer, Size, Inform)
D0      -324(A6) A0          D0    D1

STRUCT Preferences *Prefs, *PrefBuffer
LONG   Size
BOOL   Inform
```

Function: Changes the default Preference settings and informs (optional) all windows. The Preferences structure no longer contains all the defaults. This routine should never be used.

Parameters:	PrefBuffer	Custom settings
	Size	Size of custom structure
	Inform	Boolean - Inform windows

Result: PrefBuffer

UnlockIBase	**Free IntuitionBase**

Call:

```
UnlockIBase(Lock)
-420(A6)    A0

ULONG Lock
```

Function: Frees the SignalSemaphore(s) locked with LockIBase().

Parameters: Lock SignalSemaphore number or 0 (almost all).

```
Dec Hex STRUCTURE Remember,0
  0  $0 APTR    rm_NextRemember
  4  $4 LONG    rm_RememberSize
```

```
  8   $8 APTR    rm_Memory
 12   $C LABEL   rm_SIZEOF

FILENAME_SIZE = 30          ;file name size
POINTERSIZE   = (1+16+1)*2  ;mouse pointer size
TOPAZ_EIGHTY  = 8
TOPAZ_SIXTY   = 9

Dec Hex STRUCTURE Preferences,0 ;Anachronism!
  0  $0 BYTE    pf_FontHeight
  1  $1 BYTE    pf_PrinterPort
  2  $2 WORD    pf_BaudRate
  4  $4 STRUCT  pf_KeyRptSpeed,TV_SIZE
 12  $C STRUCT  pf_KeyRptDelay,TV_SIZE
 20 $14 STRUCT  pf_DoubleClick,TV_SIZE
 28 $1C STRUCT  pf_PointerMatrix,POINTERSIZE*2
 64 $40 BYTE    pf_XOffset
 65 $41 BYTE    pf_YOffset
 66 $42 WORD    pf_color17
 68 $44 WORD    pf_color18
 70 $46 WORD    pf_color19
 72 $48 WORD    pf_PointerTicks
 74 $4A WORD    pf_color0
 76 $4C WORD    pf_color1
 78 $4E WORD    pf_color2
 80 $50 WORD    pf_color3
 82 $52 BYTE    pf_ViewXOffset
 83 $53 BYTE    pf_ViewYOffset
 84 $54 WORD    pf_ViewInitX
 86 $56 WORD    pf_ViewInitY
 88 $58 BOOL    EnableCLI
 90 $5A WORD    pf_PrinterType
 92 $5C STRUCT  pf_PrinterFilename,FILENAME_SIZE
122 $7A WORD    pf_PrintPitch
124 $7C WORD    pf_PrintQuality
126 $7E WORD    pf_PrintSpacing
128 $80 WORD    pf_PrintLeftMargin
130 $82 WORD    pf_PrintRightMargin
132 $84 WORD    pf_PrintImage
134 $86 WORD    pf_PrintAspect
136 $88 WORD    pf_PrintShade
138 $8A WORD    pf_PrintThreshold
140 $8C WORD    pf_PaperSize
142 $8E WORD    pf_PaperLength
144 $90 WORD    pf_PaperType
146 $92 BYTE    pf_SerRWBits
147 $93 BYTE    pf_SerStopBuf
148 $94 BYTE    pf_SerParShk
```

```
149 $95 BYTE   pf_LaceWB
150 $96 STRUCT pf_WorkName,FILENAME_SIZE
180 $B4 BYTE   pf_RowSizeChange
181 $B5 BYTE   pf_ColumnSizeChange
182 $B6 UWORD  pf_PrintFlags
184 $B8 WORD   pf_PrintMaxWidth
186 $BA UWORD  pf_PrintMaxHeight
188 $BC UBYTE  pf_PrintDensity
189 $BD UBYTE  pf_PrintXOffset
190 $BE UWORD  pf_wb_Width
192 $C0 UWORD  pf_wb_Height
194 $C2 UBYTE  pf_wb_Depth
195 $C3 UBYTE  pf_ext_size
196 $C4 LABEL  pf_SIZEOF

LACEWB           =    1

PARALLEL_PRINTER =    0
SERIAL_PRINTER   =    1

BAUD_110         =    0
BAUD_300         =    1
BAUD_1200        =    2
BAUD_2400        =    3
BAUD_4800        =    4
BAUD_9600        =    5
BAUD_19200       =    6
BAUD_MIDI        =    7

FANFOLD          =    0
SINGLE           =  $80
PICA             =    0
ELITE            = $400
FINE             = $800
DRAFT            =    0
LETTER           = $100
SIX_LPI          =    0
EIGHT_LPI        = $200
IMAGE_POSITIVE   =    0
IMAGE_NEGATIVE   =    1
ASPECT_HORIZ     =    0
ASPECT_VERT      =    1
SHADE_BW         =    0
SHADE_GREYSCALE  =    1
SHADE_COLOR      =    2
US_LETTER        =    0
US_LEGAL         =  $10
N_TRACTOR        =  $20
```

```
W_TRACTOR          =  $30
CUSTOM             =  $40
CUSTOM_NAME        =    0
ALPHA_P_101        =    1
BROTHER_15XL       =    2
CBM_MPS1000        =    3
DIAB_630           =    4
DIAB_ADV_D25       =    5
DIAB_C_150         =    6
EPSON              =    7
EPSON_JX_80        =    8
OKIMATE_20         =    9
QUME_LP_20         =   $A
HP_LASERJET        =   $B
HP_LASERJET_PLUS =     $C

SBUF_512         =      0
SBUF_1024        =      1
SBUF_2048        =      2
SBUF_4096        =      3
SBUF_8000        =      4
SBUF_16000       =      5

SREAD_BITS     = $F0 ; pf_SerRWBits
SWRITE_BITS    =  $F

SSTOP_BITS     = $F0 ; pf_SerStopBuf
SBUFSIZE_BITS =   $F

SPARITY_BITS   = $F0 ; pf_SerParShk
SHSHAKE_BITS   =  $F

SPARITY_NONE     =      0
SPARITY_EVEN     =      1
SPARITY_ODD      =      2

SHSHAKE_XON      =      0
SHSHAKE_RTS      =      1
SHSHAKE_NONE     =      2

CORRECT_RED      =      1
CORRECT_GREEN    =      2
CORRECT_BLUE     =      4
CENTER_IMAGE     =      8

IGNORE_DIMENSIONS    =   0
BOUNDED_DIMENSIONS   = $10
ABSOLUTE_DIMENSIONS = $20
```

```
PIXEL_DIMENSIONS     = $40
MULTIPLY_DIMENSIONS = $80

INTEGER_SCALING      = $100

ORDERED_DITHERING    =    0
HALFTONE_DITHERING   = $200
FLOYD_DITHERING      = $400

ANTI_ALIAS           = $800
GREY_SCALE2          =$1000 ;for A2024 monitor

CORRECT_RGB_MASK     = (CORRECT_RED+CORRECT_GREEN+CORRECT_BLUE)
DIMENSIONS_MASK      =
(BOUNDED_DIMENSIONS+ABSOLUTE_DIMENSIONS+PIXEL_DIMENSIONS+MULTIPLY_DIMENSIONS)
DITHERING_MASK       = (HALFTONE_DITHERING+FLOYD_DITHERING)

Dec Hex STRUCTURE ICLASS,0
  0  $0 STRUCT cl_Dispatcher,h_SIZEOF ;Hook
 20 $14 ULONG  cl_Reserved            ;0
 24 $18 APTR   cl_Super
 28 $1C APTR   cl_ID                  ;string
 32 $20 UWORD  cl_InstOffset
 34 $22 UWORD  cl_InstSize
 36 $24 ULONG  cl_UserData            ;User data for the Class
 40 $28 ULONG  cl_SubclassCount       ;number of subclasses
 44 $2C ULONG  cl_ObjectCount         ;number of objects
 48 $30 ULONG  cl_Flags

CLB_INLIST = 0, CLF_INLIST = 1 ;Class in PublicClassList

Dec Hex STRUCTURE _Object,0
  0  $0 STRUCT o_Node,MLN_SIZE
  8  $8 APTR   o_Class
 12  $C LABEL  _object_SIZEOF

Dec Hex STRUCTURE Msg,0
  0  $0 ULONG  msg_MethodID ;data to follow
  4  $4 ... ;according to ID (see below)

OM_NEW       = $101 ;parameter is really a Class
OM_DISPOSE   = $102 ;self-deleting (no parameters)
OM_SET       = $103 ;set attributes (list)
OM_GET       = $104 ;read attributes
OM_ADDTAIL   = $105 ;add self to list
OM_REMOVE    = $106 ;remove self from list (no parameters)
OM_NOTIFY    = $107 ;notify self
OM_UPDATE    = $108 ;NotifyMsg
```

```
OM_ADDMEMBER = $109 ;
OM_REMMEMBER = $10A ;

Dec Hex STRUCTURE opSet,4 ;OM_NEW, OM_SET
        ...
  4  $4 APTR   ops_AttrList ;new attributes
  8  $8 APTR   ops_GInfo    ;0 for OM_NEW

Dec Hex STRUCTURE opUpdate,4 ;OM_UPDATE
        ...
  4  $4 APTR   opu_AttrList ;attributes
  8  $8 APTR   opu_GInfo
 12  $C ULONG  opu_Flags

OPUB_INTERIM = 0, OPUF_INTERIM = 1

Dec Hex STRUCTURE opGet,4 ;OM_GET
        ...
  4  $4 ULONG  opg_AttrID
  8  $8 APTR   opg_Storage

Dec Hex STRUCTURE opAddTail,4 ;OM_ADDTAIL
        ...
  4  $4 APTR   opat_List

Dec Hex STRUCTURE opMember,4 ;OM_...MEMBER
        ...
  4  $4 APTR   opam_Object

GA_Dummy            = TAG_USER+$30000 ;Gadget attributes
GA_LEFT             = TAG_USER+$30001
GA_RELRIGHT         = TAG_USER+$30002
GA_TOP              = TAG_USER+$30003
GA_RELBOTTOM        = TAG_USER+$30004
GA_WIDTH            = TAG_USER+$30005
GA_RELWIDTH         = TAG_USER+$30006
GA_HEIGHT           = TAG_USER+$30007
GA_RELHEIGHT        = TAG_USER+$30008
GA_TEXT             = TAG_USER+$30009
GA_IMAGE            = TAG_USER+$3000A
GA_BORDER           = TAG_USER+$3000B
GA_SELECTRENDER     = TAG_USER+$3000C
GA_HIGHLIGHT        = TAG_USER+$3000D
GA_DISABLED         = TAG_USER+$3000E
GA_GZZGADGET        = TAG_USER+$3000F
GA_ID               = TAG_USER+$30010
GA_USERDATA         = TAG_USER+$30011
GA_SPECIALINFO      = TAG_USER+$30012
```

```
GA_SELECTED          = TAG_USER+$30013
GA_ENDGADGET         = TAG_USER+$30014
GA_IMMEDIATE         = TAG_USER+$30015
GA_RELVERIFY         = TAG_USER+$30016
GA_FOLLOWMOUSE       = TAG_USER+$30017
GA_RIGHTBORDER       = TAG_USER+$30018
GA_LEFTBORDER        = TAG_USER+$30019
GA_TOPBORDER         = TAG_USER+$3001A
GA_BOTTOMBORDER      = TAG_USER+$3001B
GA_TOGGLESELECT      = TAG_USER+$3001C
GA_SYSGADGET         = TAG_USER+$3001D
GA_SYSGTYPE          = TAG_USER+$3001E
GA_PREVIOUS          = TAG_USER+$3001F
GA_NEXT              = TAG_USER+$30020
GA_DRAWINFO          = TAG_USER+$30021
GA_INTUITEXT         = TAG_USER+$30022
GA_LABELIMAGE        = TAG_USER+$30023

PGA_Dummy            = TAG_USER+$31000 ;PropGadget attributes
PGA_FREEDOM          = TAG_USER+$31001
PGA_BORDERLESS       = TAG_USER+$31002
PGA_HORIZPOT         = TAG_USER+$31003
PGA_HORIZBODY        = TAG_USER+$31004
PGA_VERTPOT          = TAG_USER+$31005
PGA_VERTBODY         = TAG_USER+$31006
PGA_TOTAL            = TAG_USER+$31007
PGA_VISIBLE          = TAG_USER+$31008
PGA_TOP              = TAG_USER+$31009

STRINGA_Dummy           = TAG_USER+$32000 ;StringGadget attributes
STRINGA_MaxChars        = TAG_USER+$32001
STRINGA_Buffer          = TAG_USER+$32002
STRINGA_UndoBuffer      = TAG_USER+$32003
STRINGA_WorkBuffer      = TAG_USER+$32004
STRINGA_BufferPos       = TAG_USER+$32005
STRINGA_DispPos         = TAG_USER+$32006
STRINGA_AltKeyMap       = TAG_USER+$32007
STRINGA_Font            = TAG_USER+$32008
STRINGA_Pens            = TAG_USER+$32009
STRINGA_ActivePens      = TAG_USER+$3200A
STRINGA_EditHook        = TAG_USER+$3200B
STRINGA_EditModes       = TAG_USER+$3200C
STRINGA_ReplaceMode     = TAG_USER+$3200D
STRINGA_FixedFieldMode  = TAG_USER+$3200E
STRINGA_NoFilterMode    = TAG_USER+$3200F
STRINGA_Justification   = TAG_USER+$32010
STRINGA_LongVal         = TAG_USER+$32011
STRINGA_TextVal         = TAG_USER+$32012
```

```
SG_DEFAULTMAXCHARS  = 128 ;default buffer length

LAYOUTA_Dummy       = TAG_USER+$38000 ;Layout
LAYOUTA_LAYOUTOBJ   = $38001
LAYOUTA_SPACING     = $38002
LAYOUTA_ORIENTATION = $38003

LORIENT_NONE        = 0 ;orientation
LORIENT_HORIZ       = 1
LORIENT_VERT        = 2

GM_HITTEST          = 0 ;send Hook commands to GMR_GADGETHIT
GM_RENDER           = 1 ;draw self
GM_GOACTIVE         = 2 ;Gadget activated
GM_HANDLEINPUT      = 3 ;process input
GM_GOINACTIVE       = 4 ;Gadget inactivated

Dec Hex STRUCTURE MsgHeader,0 ;again for structures
  4  $4 ULONG   MethodID
  8  $8 LABEL   methodid_SIZEOF

Dec Hex STRUCTURE gpHitTest,methodid_SIZEOF
  4  $4 APTR   gpht_GInfo
  8  $8 WORD   gpht_MouseX
 10  $A WORD   gpht_MouseY

GMR_GADGETHIT = 4 ;not hit = 0

Dec Hex STRUCTURE gpRender,methodid_SIZEOF
  4  $4 APTR   gpr_GInfo   ;GadgetContext
  8  $8 APTR   gpr_RPort
 12  $C LONG   gpr_Redraw

GREDRAW_UPDATE  = 2   ;update with new attributes
GREDRAW_REDRAW  = 1   ;refresh
GREDRAW_TOGGLE  = 0   ;toggle

Dec Hex STRUCTURE gpInput,methodid_SIZEOF ;also GM_GOACTIVE
  4  $4 APTR   gpi_GInfo
  8  $8 APTR   gpi_IEvent
 12  $C APTR   gpi_Termination
 16 $10 WORD   gpi_MouseX
 18 $12 WORD   gpi_MouseY

Dec Hex STRUCTURE gpGoInactive,methodid_SIZEOF
  4  $4 APTR    gpgi_GInfo
  8  $8 ULONG   gpgi_Abort ;V37 and up!
```

```
GMR_MEACTIVE = 0
GMR_NOREUSE  = 2
GMR_REUSE    = 4
GMR_VERIFY   = 8

GMRB_NOREUSE = 1, GMRF_NOREUSE = 2
GMRB_REUSE   = 2, GMRF_REUSE   = 4
GMRB_VERIFY  = 3, GMRF_VERIFY  = 8

ICM_SETLOOP   = $402
ICM_CLEARLOOP = $403
ICM_CHECKLOOP = $404

ICA_Dummy       = $40000
ICA_TARGET      = ICA_Dummy+1
ICA_MAP         = ICA_Dummy+2
ICSPECIAL_CODE = ICA_Dummy+3

ICTARGET_IDCMP = -1 ;$ffffffff

CUSTOMIMAGEDEPTH = -1 ;depth for CustomGadgets

IMAGE_ATTRIBUTES = TAG_USER+$20000
IA_LEFT            = IMAGE_ATTRIBUTES+$01
IA_TOP             = IMAGE_ATTRIBUTES+$02
IA_WIDTH           = IMAGE_ATTRIBUTES+$03
IA_HEIGHT          = IMAGE_ATTRIBUTES+$04
IA_FGPEN           = IMAGE_ATTRIBUTES+$05
IA_BGPEN           = IMAGE_ATTRIBUTES+$06
IA_DATA            = IMAGE_ATTRIBUTES+$07
IA_LINEWIDTH       = IMAGE_ATTRIBUTES+$08
IA_PENS            = IMAGE_ATTRIBUTES+$0E
IA_RESOLUTION      = IMAGE_ATTRIBUTES+$0F
IA_APATTERN        = IMAGE_ATTRIBUTES+$010
IA_APATSIZE        = IMAGE_ATTRIBUTES+$011
IA_MODE            = IMAGE_ATTRIBUTES+$012
IA_FONT            = IMAGE_ATTRIBUTES+$013
IA_OUTLINE         = IMAGE_ATTRIBUTES+$014
IA_RECESSED        = IMAGE_ATTRIBUTES+$015
IA_DOUBLEEMBOSS    = IMAGE_ATTRIBUTES+$016
IA_EDGESONLY       = IMAGE_ATTRIBUTES+$017

SYSIA_Size      = IMAGE_ATTRIBUTES+$0B ;system IClass
SYSIA_Depth     = IMAGE_ATTRIBUTES+$0C
SYSIA_Which     = IMAGE_ATTRIBUTES+$0D
SYSIA_DrawInfo = IMAGE_ATTRIBUTES+$018
```

```
SYSIA_Pens       = IA_PENS
IA_SHADOWPEN     = IMAGE_ATTRIBUTES+$09
IA_HIGHLIGHTPEN = IMAGE_ATTRIBUTES+$0A

SYSISIZE_MEDRES = 0
SYSISIZE_LOWRES = 1
SYSISIZE_HIRES  = 2

DEPTHIMAGE  = 0 ;SYSIA_Witch values
ZOOMIMAGE   = 1
SIZEIMAGE   = 2
CLOSEIMAGE  = 3
SDEPTHIMAGE = 5
LEFTIMAGE   = $A
UPIMAGE     = $B
RIGHTIMAGE  = $C
DOWNIMAGE   = $D
CHECKIMAGE  = $E
MXIMAGE     = $F

IM_DRAW       = $202 ;draw self
IM_HITTEST    = $203 ;TRUE=hit
IM_ERASE      = $204 ;delete self
IM_MOVE       = $205 ;redraw
IM_DRAWFRAME  = $206 ;draw within Box
IM_FRAMEBOX   = $207
IM_HITFRAME   = $208
IM_ERASEFRAME = $209

IDS_NORMAL            = 0
IDS_SELECTED          = 1 ;active
IDS_DISABLED          = 2 ;cannot be selected
IDS_BUSY              = 3
IDS_INDETERMINATE     = 4
IDS_INACTIVENORMAL    = 5 ;within window border
IDS_INACTIVESELECTED = 6 ;
IDS_INACTIVEDISABLED = 7 ;
IDS_INDETERMINANT     = IDS_INDETERMINATE

Dec Hex STRUCTURE impFrameBox,4
  4  $4 APTR   impf_ContentsBox
  8  $8 APTR   impf_FrameBox
 12  $C APTR   impf_DrInfo
 16 $10 LONG   impf_FrameFlags

FRAMEB_SPECIFY  = 0, FRAMEF_SPECIFY  = 1

Dec Hex STRUCTURE impDraw,4
```

```
  4  $4 APTR    impd_RPort
  8  $8 WORD    impd_OffsetX
 10  $A WORD    impd_OffsetY
 12  $C ULONG   impd_State
 16 $10 APTR    impd_DrInfo
 20 $14 WORD    impd_DimensionsWidth
 22 $16 WORD    impd_DimensionsHeight

Dec Hex STRUCTURE impErase,4
  4  $4 APTR     impe_RPort
  8  $8 WORD     impe_OffsetX
 10  $A WORD     impe_OffsetY
 12  $C WORD     impe_DimensionsWidth
 14  $E WORD     impe_DimensionsHeight

Dec Hex STRUCTURE impHitTest,4
  4  $4 WORD     imph_PointX
  6  $6 WORD     imph_PointY
  8  $8 WORD     imph_DimensionsWidth
 10  $A WORD     imph_DimensionsHeight
```

3.1.12 The Layers Library

The "layers.library" is responsible for the complex Clipping and Refresh of overlapping software levels. The base address of the function is expected in A6.

Functions of the Layers Library

BeginUpdate
BehindLayer
CreateBehindHookLayer
CreateBehindLayer
CreateUpfrontHookLayer
CreateUpfrontLayer
DeleteLayer
DisposeLayerInfo
EndUpdate
InstallClipRegion
InstallLayerHook
LockLayer
LockLayerInfo
LockLayers
MoveLayer
MoveLayerInFrontOf
MoveSizeLayer
NewLayerInfo
ScrollLayer
SizeLayer
SwapBitsRastPortClipRect
UnlockLayer
UnlockLayerInfo
UnlockLayers
UpfrontLayer

Description of the routines

BeginUpdate — **Begin layer update**

Call:

```
result = BeginUpdate( l )
d0       -78(A6)      a0

LONG   result
STRUCT Layer *l
```

Function: Converts the DamageList to a ClipRectList and adds it to the layer.

Parameters: l — Layer

Result: 0 — Error (EndUpdate(l,0) call)

BehindLayer — **Put layer in the background**

Call:

```
result = BehindLayer( l )
d0       -54(A6)      a1

LONG   result
STRUCT Layer *l
```

Function: Moves the given layer behind all other layers.

Parameters: l — Layer

Result: 0 — Error

CreateBehindHookLayer — **Create layer with backfill hook**

Call:

```
result = CreateBehindHookLayer(li,bm,x0,y0,x1,y1,flags,hook,bm2)
d0       -192(A6)              a0 a1 d0 d1 d2 d3 d4    a3   a2

STRUCT Layer *Result;   STRUCT Layer_Info *li
STRUCT BitMap *bm,*bm2
LONG   x0,y0,x1,y1,flags
STRUCT Hook *hook
```

Function: Creates a new layer in the background and installs a backfill hook.

Parameters:

li	LayerInfo
bm	Screen bit-map
x0,y0	Upper left corner
x1,y1	Lower right corner
flags	Layer type
hook	BackFill hook
bm2	SuperBitMap or 0

Result: Layer or 0

CreateBehindLayer — Create layer in background

Call:

```
result = CreateBehindLayer(li,bm,x0,y0,x1,y1,flags,bm2)
d0       -42(A6)           a0 a1 d0 d1 d2 d3 d4    a2

STRUCT Layer *result
STRUCT Layer_Info *li
STRUCT BitMap *bm,*bm2
LONG   x0,y0,x1,y1,flags
```

Function: Creates a new layer behind all other layers.

Parameters:

li	LayerInfo
bm	Screen bit-map
x0,y0	Upper left corner
x1,y1	Lower right corner
flags	Layer type

bm2	SuperBitMap or 0

Result: Layer or 0

CreateUpfrontHookLayer — Create foreground layer with hook

Call:

```
result = CreateUpfrontHookLayer(li,bm,x0,y0,x1,y1,flags,hook,bm2)
d0       -186(A6)              a0 a1 d0 d1 d2 d3 d4   a3   a2

STRUCT Layer *result
STRUCT Layer_Info *li
STRUCT BitMap *bm,*bm2
LONG   x0,y0,x1,y1,flags
STRUCT Hook *hook
```

Function: Creates a new layer in the foreground and installs a backfill hook.

Parameters:

li	LayerInfo
bm	Screen bit-map
x0,y0	Upper left corner
x1,y1	Lower right corner
flags	Layer type
hook	BackFill hook
bm2	SuperBitMap or 0

Result: Layer or 0

CreateUpfrontLayer — Create a foreground layer

Call:

```
result = CreateUpfrontLayer(li,bm,x0,y0,x1,y1,flags,bm2)
d0       -36(A6)              a0 a1 d0 d1 d2 d3 d4   a2

STRUCT Layer *result
STRUCT Layer_Info *li
STRUCT BitMap *bm,*bm2
LONG   x0,y0,x1,y1,flags
```

Function: Creates a new layer in the foreground.

Parameters:

li	LayerInfo
bm	Screen bit-map
x0,y0	Upper left corner
x1,y1	Lower right corner
flags	Layer type
bm2	SuperBitMap or 0

Result: Layer or 0

DeleteLayer **Free a layer**

Call:

```
result = DeleteLayer( l )
d0       -90(A6)      a1

LONG   result
STRUCT Layer *l
```

Function: Frees the given layer and its memory blocks.

Parameters: l Layer

Result: 0 Error

DisposeLayerInfo **Free the LayerInfo**

Call:

```
DisposeLayerInfo( li )
-150(A6)          a0

STRUCT Layer_Info *li
```

Function: Free LayerInfo and its memory.

Parameters: li LayerInfo

EndUpdate — **End update and normalize clipping**

Call:

```
EndUpdate( l, flag )
-84(A6)    a0 d0

STRUCT Layer *l
UWORD  flag
```

Function: Return normal ClipRects to the layer.

Parameters:

l	Layer
flag	TRUE: Update completely ended.

InstallClipRegion — **Install clipping**

Call:

```
oldclipregion = InstallClipRegion( l, region )
d0              -174(A6)           a0 a1

STRUCT Region *oldclipregion,*region
STRUCT Layer *l
```

Function: Installs a new clipping region in layer.

Parameters:

l	Layer
region	New ClipRegion

Result: Previous ClipRegion or 0

InstallLayerHook — **Install backfill hook**

Call:

```
oldhook = InstallLayerHook( layer, hook )
d0        -198(A6)          a0     a1

STRUCT Hook *oldhook,*hook
STRUCT Layer *layer
```

Function: Installs a new backfill hook in a layer.

Parameters: layer — Layer

hook   New backfill hook

Result: Previous backfill hook

LockLayer — **Lock layer**

Call:
```
LockLayer( l )
-96(A6)    a1

STRUCT Layer *l
```

Function: Lock a layer from other programs.

Parameters: l   Layer

LockLayerInfo — **Lock LayerInfo**

Call:
```
LockLayerInfo( li )
-120(A6)       a0

STRUCT LayerInfo *li
```

Function: Lock LayerInfo from other programs.

Parameters: li   LayerInfo

LockLayers — **Lock all layers of a LayerInfo**

Call:
```
LockLayers( li )
-108(A6)    a0

STRUCT LayerInfo *li
```

Function: Locks an entire layer system from other programs.

Parameters: li   LayerInfo

MoveLayer — **Move a layer**

Call:
```
result = MoveLayer( l, dx, dy )
d0       -60(A6)    a1 d0  d1
```

```
LONG   result,dx,dy
STRUCT Layer *l
```

Function: Move a layer relative to its current position.

Parameters: l Layer

dx Relative X position

dy Relative Y position

Result: 0 Error

MoveLayerInFrontOf — Move layer in front of another layer

Call:

```
result = MoveLayerInFrontOf( layertomove, targetlayer )
d0       -168(A6)            a0           a1

LONG   result
STRUCT Layer *layertomove,*targetlayer
```

Function: Moves one layer in front of another.

Parameters: layertomove
Layer to move in front of targetlayer.

targetlayer Layer that layertomove will overlay.

Result: 0 Error

MoveSizeLayer — Change layer size and position

Call:

```
result = MoveSizeLayer( layer, dx, dy, dw, dh )
d0       -180(A6)       a0     d0  d1  d2  d3

LONG   result,dx,dy,dw,dh
STRUCT Layer *layer
```

Function: Move upper left and lower right corners.

Parameters: layer Layer

	dx,dy	Relative position
	dw,dy	Relative size
Result:	0	Error

NewLayerInfo — **Get LayerInfo**

Call:

```
result = NewLayerInfo()
d0       -144(A6)

STRUCT LayerInfo *result
```

Function: Creates a new LayerInfo structure.

Result: LayerInfo or 0

ScrollLayer — **Scroll layer contents**

Call:

```
ScrollLayer( l, dx, dy )
-72(A6)      a1 d0  d1

STRUCT Layer *l
LONG   dx,dy
```

Function: Scrolls the contents of a layer.

Parameters:	l	Layer
	dx	Delta value X
	dy	Delta value Y

SizeLayer — **Change layer size**

Call:

```
result = SizeLayer( l, dx, dy )
d0       -66(A6)    a1 d0  d1

LONG   result,dx,dy
STRUCT Layer *l
```

Function: Changes the size of a layer relative to its current size.

Parameters: 1 Layer

dx,dy Relative size change

Result: 0 Error

SwapBitsRastPortClipRect — **Switch contents of a bit-map and ClipRect**

Call:

```
SwapBitsRastPortClipRect( rp, cr )
-126(A6)                  a0  a1

STRUCT RastPort *rp
STRUCT ClipRect *cr
```

Function: Switches the contents of a ClipRect with the regions of a bit-map.

Parameters: rp RastPort

cr ClipRect

UnlockLayer — **Undo a LockLayer() call**

Call:

```
UnlockLayer( l )
-102(A6)     a0

STRUCT Layer *l
```

Function: Frees the layer for other programs to use again.

Parameters: l Layer

UnlockLayerInfo — **Undo LockLayerInfo() call**

Call:

```
UnlockLayerInfo( li )
-138(A6)         a0

STRUCT LayerInfo *li
```

Function: Frees the LayerInfo structure for other programs.

Parameters: li LayerInfo

UnlockLayers **Undo LockLayers() call**

Call:

```
UnlockLayers( li )
-114(A6)      a0

STRUCT LayerInfo *li
```

Function: Frees the entire layer system in the given LayerInfo list.

Parameters: li LayerInfo

UpfrontLayer **Move a layer to the front**

Call:

```
result = UpfrontLayer( l )
d0       -48(A6)       a1

LONG   result
STRUCT Layer *l
```

Function: Moves a layer in front of all other layers.

Parameters: l Layer

Result: 0 Error

3.1.13 The MathFFP, MathIEEESingBas, and MathIEEEDoubBas Libraries

The Amiga supports three different floating point formats: the international IEEE formats for 32 and 64 bit floating point numbers (which can be directly processed by the FPU 68882), and the FastFloatingPoint format.

The FFP format is the fastest 32 bit floating point format as long as you don't have an FPU, which will process the IEEE formats faster than any CPU.

Two libraries exist for each format. First, we will discuss the library for basic mathematical functions. The functions and their function offsets are the same for all three libraries.

MathFFP functions begin with 'SP' and expect 32 bit FFP values. MathIEEESingBas functions begin with 'IEEESP' and expect 32 bit IEEE values. MathIEEEDoubBas functions begin with 'IEEEDP' and expect 64 bit IEEEs.

The 64 bit numbers are always distributed across two registers (upper 32 bits/lower 32 bits).

Functions of the Base Libraries

Abs
Add
Ceil
Cmp
Div
Fix
Floor
Flt
Mul
Neg
Sub
Tst

Description of the functions

SPAbs/IEEESPAbs/IEEEDPAbs	Absolute value

Call:

```
x       =     ...Abs ( y )
              -54(A6)
d0              SPAbs  d0
d0          IEEESPAbs  d0
d0/d1       IEEEDPAbs  d0/d1
```

Function: Returns the positive value of 'y'.

SPAdd/IEEESPAdd/IEEEDPAdd **Add two values**

Call:

```
x       =      ...Add  ( y ,    z )
               -66(A6)
d0               SPAdd   d0     d1
d0           IEEESPAdd   d0     d1
d0/d1        IEEEDPAdd   d0/d1  d2/d3
```

Function: x = y + z

SPCeil/IEEESPCeil/IEEEDPCeil **Round up**

Call:

```
x       =      ...Ceil  ( y )
               -96(A6)
d0               SPCeil   d0
d0           IEEESPCeil   d0
d0/d1        IEEEDPCeil   d0/d1
```

Function: Rounds 'y' to the next whole number '>=y'.

SPCmp/IEEESPCmp/IEEEDPCmp **Compare values**

Call:

```
c       =      ...Cmp  ( y ,    z )
               -42(A6)
d0,cc            SPCmp   d1     d0
d0,cc        IEEESPCmp   d1     d0
d0,cc        IEEEDPCmp   d0/d1  d2/d3
```

Function: Compare two values.

Result: c = 1, cc = gt: y > z

c = 0, cc = eq: y = z

c = -1, cc = lt: y < z

SPDiv/IEEESPDiv/IEEEDPDiv **Division**

Call:

```
x       =      ...Div  ( y ,    z )
               -84(A6)
d0               SPDiv   d0     d1
d0           IEEESPDiv   d0     d1
```

```
d0/d1    IEEEDPDiv   d0/d1 d2/d3
```

Function: x = y / z

SPFix/IEEESPFix/IEEEDPFix — Convert float to 32 bit integer

Call:

```
x   =     ...Fix( y )
          -30(A6)
d0          SPFix  d0
d0      IEEESPFix  d0
d0      IEEEDPFix  d0/d1
```

Function: Converts floating point number into a 32 bit integer value.

SPFloor/IEEESPFloor/IEEEDPFloor — Round down

Call:

```
x =          ...Floor( y )
             -90(A6)
d0             SPFloor  d0
d0         IEEESPFloor  d0
d0/d1      IEEEDPFloor  d0/d1
```

Function: Rounds 'y' to the next whole number '<=y'.

SPFlt/IEEESPFlt/IEEEDPFlt — Convert long to float

Call:

```
x =          ...Flt( y )
             -36(A6)
d0             SPFlt  d0
d0         IEEESPFlt  d0
d0/d1      IEEEDPFlt  d0
```

Function: Converts a 32 bit integer to a floating point number.

SPMul/IEEESPMul/IEEEDPMul — Multiplication

Call:

```
x =          ...Mul( y ,   z )
             -78(A6)
d0             SPMul  d0    d1
d0         IEEESPMul  d0    d1
d0/d1      IEEEDPMul  d0/d1 d2/d3
```

Function: x = y * z

SPNeg/IEEESPNeg/IEEEDPNeg **Negation**

Call:

```
x =          ...Neg( y )
             -60(A6)
d0             SPNeg  d0
d0         IEEESPNeg  d0
d0/d1      IEEEDPNeg  d0/d1
```

Function: x = -y

SPSub/IEEESPSub/IEEEDPSub **Subtraction**

Call:

```
x =          ...Sub( y ,    z )
             -72(A6)
d0             SPSub  d0    d1
d0         IEEESPSub  d0    d1
d0/d1      IEEEDPSub  d0/d1 d2/d3
```

Function: x = y - z

SPTst/IEEESPTst/IEEEDPTst **Test a value**

Call:

```
c =          ...Tst( y )
             -48(A6)
d0,cc          SPTst  d0
d0,cc      IEEESPTst  d0
d0,cc      IEEEDPTst  d0/d1
```

Function: Compares a value with 0.

Result: c = 1, cc = gt: y > 0.0

c = 0, cc = eq: y = 0.0

c = -1, cc = lt: y < 0.0

3.1.14 The MathTrans, MathIEEESingTrans, and MathIEEEDoubTrans Libraries

Now we will look at the libraries for trigonometrical functions. What is true for the basic mathematical functions also applies to these functions.

Trigonometrical functions

Acos
Asin
Atan
Cos
Cosh
Exp
Fieee
Log
Log10
Pow
Sin
Sincos
Sinh
Sqrt
Tan
Tanh
Tieee

Description of the functions

SPAcos/IEEESPAcos/IEEEDPAcos	arc cosin

Call:

```
x =          ...Acos( y )
           -120(A6)
d0           SPAcos  d0
d0       IEEESPAcos  d0
d0/d1    IEEEDPAcos  d0/d1
```

Function: Returns the arc cos of 'y'.

SPAsin/IEEESPAsin/IEEEDPAsin	arc sin

Call:

```
x =          ...Asin( y )
           -114(A6)
```

```
d0          SPAsin  d0
d0      IEEESPAsin  d0
d0/d1   IEEEDPAsin  d0/d1
```

Function: Returns the arc sin of 'y'.

SPAtan/IEEESPAtan/IEEEDPAtan — arc tangent

Call:

```
x =         ...Atan( y )
            -30(A6)
d0          SPAtan  d0
d0      IEEESPAtan  d0
d0/d1   IEEEDPAtan  d0/d1
```

Function: Returns the arc tan of 'y'.

SPCos/IEEESPCos/IEEEDPCos — cosin

Call:

```
x =         ...Cos( y )
            -42(A6)
d0          SPCos  d0
d0      IEEESPCos  d0
d0/d1   IEEEDPCos  d0/d1
```

Function: Returns the cos of 'y'.

SPCosh/IEEESPCosh/IEEEDPCosh — hyperbolic cosin

Call:

```
x =         ...Cosh( y )
            -66(A6)
d0          SPCosh  d0
d0      IEEESPCosh  d0
d0/d1   IEEEDPCosh  d0/d1
```

Function: Returns the hyperbolic cos of 'y'.

SPExp/IEEESPExp/IEEEDPExp — Exponential function, base e

Call:

```
x =         ...Exp( y )
            -78(A6)
d0          SPExp  d0
d0      IEEESPExp  d0
```

```
d0/d1 IEEEDPExp  d0/d1
```

Function: x = e^y

SPFieee/IEEESPFieee/IEEEDPFieee — Convert IEEE single

Call:

```
x =       ...Fieee( y )
          -108(A6)
d0         SPFieee  d0
(d0    IEEESPFieee  d0)
d0/d1 IEEEDPFieee  d0
```

Function: Converts a 32 bit IEEE value to the format of the current library.

SPLog/IEEESPLog/IEEEDPLog — Natural logarithm

Call:

```
x =       ...Log( y )
          -84(A6)
d0         SPLog  d0
d0    IEEESPLog  d0
d0/d1 IEEEDPLog  d0/d1
```

Function: Returns the natural log of 'y'.

SPLog10/IEEESPLog10/IEEEDPLog10 — Logarithm, base 10

Call:

```
x =       ...Log10( y )
          -126(A6)
d0         SPLog10  d0
d0    IEEESPLog10  d0
d0/d1 IEEEDPLog10  d0/d1
```

Function: Returns the base 10 log of 'y'.

SPPow/IEEESPPow/IEEEDPPow — Exponential function

Call:

```
z =       ...Pow( x ,   y )
          -90(A6)
d0         SPPow  d0    d1
d0    IEEESPPow  d0    d1
d0/d1 IEEEDPPow  d0/d1 d2/d3
```

Function: z = x^y

SPSin/IEEESPSin/IEEEDPSin — sin

Call:

```
x =         ...Sin( y )
           -36(A6)
d0          SPSin  d0
d0      IEEESPSin  d0
d0/d1   IEEEDPSin  d0/d1
```

Function: Returns the sin of 'y'.

SPSincos/IEEESPSincos/IEEEDPSincos — Sin and Cosin

Call:

```
x =         ...Sincos( y,     z )
           -54(A6)
d0          SPSincos  d0      d1
d0      IEEESPSincos  d0      a0
d0/d1   IEEEDPSincos  d0/d1   a0
```

Function: x = ...Sin(y) AND (z)=Cos(y). 'z' is the address of the cos result.

SPSinh/IEEESPSinh/IEEEDPSinh — Hyperbolic sin

Call:

```
x =         ...Sinh( y )
           -60(A6)
d0          SPSinh  d0
d0      IEEESPSinh  d0
d0/d1   IEEEDPSinh  d0/d1
```

Function: Returns the hyperbolic sin of 'y'.

SPSprt/IEEESPSqrt/IEEEDPSqrt — Square root

Call:

```
x =         ...Sqrt( y )
           -96(A6)
d0          SPSqrt  d0
d0      IEEESPSqrt  d0
d0/d1   IEEEDPSqrt  d0/d1
```

Function: Returns the square root of 'y'.

SPTan/IEEESPTan/IEEEDPTan	**tangent**

Call:

```
x =       ...Tan( y )
          -48(A6)
d0          SPTan  d0
d0      IEEESPTan  d0
d0/d1   IEEEDPTan  d0/d1
```

Function: Returns the tangent of 'y'.

SPTanh/IEEESPTanh/IEEEDPTanh	**Hyperbolic tangent**

Call:

```
x =       ...Tanh( y )
          -72(A6)
d0          SPTanh  d0
d0      IEEESPTanh  d0
d0/d1   IEEEDPTanh  d0/d1
```

Function: Returns the hyperbolic tangent of 'y'.

SPTieee/IEEESPTieee/IEEEDPTieee	**Create an IEEE single**

Call:

```
x =     ...Tieee( y )
        -102(A6)
d0       SPTieee  d0
(d0  IEEESPTieee  d0)
d0   IEEEDPTieee  d0/d1
```

Function: Converts a value from the library format to a 32 bit IEEE value.

3.1.15 The Translator Library

The "translator.library" consists of only one routine. It is used to translate text into Phoneme codes for the Narrator device.

Description of function

Translate	**Generate Phoneme**

Call:

```
rtnCode = Translate(inString, inLength, outBuffer, outLength)
D0        -30(A6)   A0        D0        A1         D1
```

```
LONG rtnCode,inLength,outLength
APTR inString,outBuffer
```

Function: Translates text into Phoneme codes.

Parameters:	inString	Text
	inLength	Text length
	outBuffer	Buffer for Phonemes
	outLength	Buffer length
Result:	0	Okay, otherwise negative cancel offset from start of text.

3.1.16 The Utility Library

The "utility.library" contains helpful routines designed to make programming easier. One of the most important things you can do with these routines is construct TagItem fields from high level languages.

Functions of the Utility Library

AllocateTagItems
CloneTagItems
FilterTagChanges
FilterTagItems
FindTagItem
FreeTagItems
GetTagData
MapTags
NextTagItem
PackBoolTags
RefreshTagItemClones
TagInArray

Description of the functions

AllocateTagItems — **Allocate a TagItem field**

Call:

```
TagList = AllocateTagItems( NumItems )
D0        -66(A6)           D0

STRUCT TagItem *TagList
ULONG  NumItems
```

Function: Allocates a TagItem field for storing size, etc. Access to the tags is only available via NextTagItem().

Parameters: NumItems Number of usable slots.

Result: TagList or 0

CloneTagItems — **Copy a TagItem field**

Call:

```
NewTagList = CloneTagItems( TagList )
D0           -72(A6)        A0

STRUCT TagItem *NewTagList,*TagList
```

Function: Copies a complete TagItem list.

Parameters: TagList TagItem list to be copied.

Result: TagItem list or 0

FilterTagChanges — **Filter out changes to a TagItem field**

Call:

```
FilterTagChanges( ChangeList, OldValues, Apply)
-54(A6)           A0          A1         D0

STRUCT TagItem *ChangeList,*OldValues
LONG   Apply
```

Function: Replaces all unchanged TagItems in a change list with TAG_IGNORE.

Parameters: ChangeList New TagItems

OldValues Old TagItem list

Apply Boolean, indicates whether the old list should be used.

FilterTagItems **Remove certain TagItems**

Call:

```
nvalid    FilterTagItems(TagList, TagArray, Logic)
D0   -96(A6)       A0        A1         D0

STRUCT TagItem *TagList
APTR   TagArray
LONG   Logic
ULONG  nvalid
```

Function: Replaces with TAG_IGNORE the TagItems whose ti_Tag entry is in the given field.

Parameters: TagList TagItem field

TagArray Field with the tags to be deleted, ends with TAG_END.

Logic TAGFILTER_AND (delete items not given in the field) or TAGFILTER_NOT (delete items given in the field).

Result: Number of valid items remaining in the list.

FindTagItem **Find a TagItem**

Call:

```
TagItem = FindTagItem( TagVal, TagList )
D0          -30(A6)       D0       A0

STRUCT TagItem *TagItem,*TagList
LONG   TagVal
```

Function: Finds an item in a TagItem list with the given tag value.

Parameters: TagVal Tag to be found

TagList TagItem list

Result: TagItem or 0

FreeTagItems — Free a TagItem field

Call:

```
FreeTagItems( TagList )
-78(A6)       A0

STRUCT TagItem *TagList
```

Function: Frees a TagItem field allocated with AllocateTagItems() or CloneTagItems().

Parameters: TagList Field to be set free.

GetTagData — Get data on a TagItem

Call:

```
TagData = GetTagData( TagVal, Default, TagList )
D0        -36(A6)     D0      D1       A0

ULONG  TagData,TagVal,Default
STRUCT TagItem *TagList
```

Function: Returns the ti_Data entry of the given tag or the default value if no TagItem of this type could be found.

Parameters: TagVal Tag to be found

Default Default value

TagList List to be searched

Result: ti_Data or 'Default'

MapTags — Change tags

Call:

```
MapTags(TagList, MapList, IncludeMiss)
-60(A6) A0       A1       D0
```

```
STRUCT TagItem *TagList,*MapList
BOOL   IncludeMiss
```

Function: Tags that are to be replaced by the ti_Tag of the given list are given in a 'MapList' as ti_Data entries.

Parameters: TagList List with tags to be changed.

MapList List with changes

IncludeMiss
Boolean, whether items not in the MapList should remain unchanged (otherwise they are replaced with TAG_IGNORE).

NextTagItem — **Find the next normal TagItem**

Call:

```
TagItem = NextTagItem( TagItemPtr )
D0        -48(A6)      A0

STRUCT TagItem *TagItem,**TagItemPtr
```

Function: Returns the next TagItem, skipping over all system tags.

Parameters: TagItemPtr Address of a longword with the address of the first TagItem.

Result: TagItem or 0

PackBoolTags — **Combine BoolTags into a flag longword**

Call:

```
Flags = PackBoolTags( InitialFlags, TagList, BoolMap )
D0      -42(A6)       D0            A0       A1

ULONG Flags,InitialFlags
STRUCT TagItem *TagList,*BoolMap
```

Function: BoolTags are entered as bit flags in a longword. The tag flags are given as ti_Data in a TagItem field.

Parameters: InitialFlags Default result

TagList	TagItem field with BoolTags
BoolMap	TagItem field with flag longwords

Result: Changed flag longword

Example: Assume that we are managing the IDCMP Flags in a complex program using TagItems, and now we want to assemble the IDCMPFlag longword:

```
**
** Definition of the Tags
**
TAG_NEWSIZE          = TAG_USER+1
TAG_REFRESHWINDOW    = TAG_USER+2
TAG_MOUSEBUTTONS     = TAG_USER+3
TAG_MOUSEMOVE        = TAG_USER+4
TAG_GADGETDOWN       = TAG_USER+5
TAG_GADGETUP         = TAG_USER+6
TAG_REQSET           = TAG_USER+7
TAG_MENUPICK         = TAG_USER+8
TAG_CLOSEWINDOW      = TAG_USER+9
TAG_RAWKEY           = TAG_USER+10
TAG_REQCLEAR         = TAG_USER+11
TAG_NEWPREFS         = TAG_USER+12
TAG_DISKINSERTED     = TAG_USER+13
TAG_DISKREMOVED      = TAG_USER+14
TAG_ACTIVEWINDOW     = TAG_USER+15
TAG_INACTIVEWINDOW   = TAG_USER+16
TAG_DELTAMOVE        = TAG_USER+17
TAG_VANILLAKEY       = TAG_USER+18
TAG_INTUITICKS       = TAG_USER+19
TAG_MENUHELP         = TAG_USER+20
TAG_CHANGEWINDOW     = TAG_USER+21

**
** Example
**
 ...
 movea.l _MainWindow,a1
 lea     _Changes(pc),a0
 bsr     _SetIDCMP
 ...
_Changes
 dc.l    TAG_MOUSEMOVE,0      ;turn off
 dc.l    TAG_GADGETDOWN,0     ;turn off
```

```
 dc.l     TAG_DELTAMOVE,0        ;turn off
 dc.l     TAG_VANILLAKEY,-1      ;turn on
 dc.l     TAG_USER+40,-1         ;no meaning
 dc.l     TAG_RAWKEY,0           ;turn off
 dc.l     TAG_MOUSEBUTTONS,-1    ;turn on
 dc.l     TAG_USER+35,0          ;no meaning
 dc.l     TAG_DONE

**
** Change IDCMPFlags
**
** Input: a1=Window, a0=TagItems
**
_SetIDCMP
 movem.l a1/a6,-(a7)
 movea.l _UtilityBase,a6
 move.l  wd_IDCMPFlags(a1),d0
 lea     _BoolMap(pc),a1
 jsr     _LVOPackBoolTags(a6)
 movea.l (a7)+,a0
 movea.l _IntuiBase,a6
 jsr     _LVOModifyIDCMP(a6)
 movea.l (a7)+,a6
 rts

_BoolMap
 dc.l TAG_NEWSIZE,2
 dc.l TAG_REFRESHWINDOW,4
 dc.l TAG_MOUSEBUTTONS,8
 dc.l TAG_MOUSEMOVE,$10
 dc.l TAG_GADGETDOWN,$20
 dc.l TAG_GADGETUP,$40
 dc.l TAG_REQSET,$80
 dc.l TAG_MENUPICK,$100
 dc.l TAG_CLOSEWINDOW,$200
 dc.l TAG_RAWKEY,$400
 dc.l TAG_REQCLEAR,$1000
 dc.l TAG_NEWPREFS,$4000
 dc.l TAG_DISKINSERTED,$8000
 dc.l TAG_DISKREMOVED,$10000
 dc.l TAG_ACTIVEWINDOW,$40000
 dc.l TAG_INACTIVEWINDOW,$80000
 dc.l TAG_DELTAMOVE,$100000
 dc.l TAG_VANILLAKEY,$200000
 dc.l TAG_INTUITICKS,$400000
 dc.l TAG_MENUHELP,$1000000
 dc.l TAG_CHANGEWINDOW,$2000000
 dc.l TAG_DONE
```

RefreshTagItemClones — **Reset a copied TagItem field**

Call:

```
RefreshTagItemClones( CloneTagItems, OriginalTagItems )
-84(A6)                A0             A1

STRUCT TagItem *CloneTagItems,*OriginalTagItems
```

Function: Restores a list obtained with CloneTagItems() to the values of the original list.

Parameters: CloneTagItems
Result from CloneTagItems (original TagItems).

OriginalTagItems
Unchanged original list

TagInArray — **Check if a tag is present**

Call:

```
Bool = TagInArray( Tag, TagArray )
D0     -90(A6)     D0   A0

ULONG Tag
APTR  TagArray
```

Function: Checks for a certain value in a tag value field ending with TAG_END.

Parameters: Tag — Tag value to search for.

TagArray — Field with tag values, ends with TAG_END.

Result: 0 — Value not found.

```
Dec Hex STRUCTURE TagItem,0
  0  $0 ULONG   ti_Tag      ;ID (TAG_...)
  4  $4 ULONG   ti_Data     ;ID-specific data
  8  $8 LABEL   ti_SIZEOF

TAG_DONE   =        0 ;end of a TagItem field
TAG_IGNORE =        1 ;skip TagItem
TAG_MORE   =        2 ;next TagItem field

TAG_USER   = $80000000
```

3.1.17 The Workbench Library

The Workbench used to be a task module. Starting with AmigaOS 2.0, it is now a library. The functions of the "workbench.library" allow you to create menus and icons in the Workbench window.

Functions of the Workbench Library

AddAppIconA
AddAppMenuItemA
AddAppWindowA
RemoveAppIcon
RemoveAppMenuItem
RemoveAppWindow

Description of the functions

AddAppIconA	**Add custom icons to the Workbench**

Call:

```
AppIcon = AddAppIconA(id, userdata, text, msgport, lock, diskobj, taglist)
D0          -60(A6)    D0  D1        A0    A1       A2    A3       A4

STRUCT AppIcon *AppIcon
ULONG  id,userdata
APTR   text
STRUCT MsgPort *msgport
BPTR   lock
STRUCT DiskObject *diskobj
STRUCT TagItem *taglist
```

Function: Creates a custom icon and adds it to the Workbench. Two types of events are generated by the icon: a double-click on the icon (am_NumArgs=0) and dragging another icon across it (like WbStartup message).

Parameters:

id	Custom ID value
userdata	Custom data
text	Icon name
lock	File lock or 0

msgport MsgPort for AppMessages of type MTYPE_APPICON.

diskobj Address of a DiskObject structure.

taglist TagItem field or 0

Result: AppIcon structure or 0

AddAppMenuItemA — Add a MenuItem to the Tools menu

Call:

```
AppMenuItem = AddAppMenuItemA(id, userdata, text, msgport, taglist)
D0            -72(A6)          D0  D1        A0    A1       A2

STRUCT AppMenuItem *AppMenuItem
ULONG  id,userdata
APTR   text
STRUCT MsgPort *msgport
STRUCT TagItem *taglist
```

Function: Adds a menu item to the Tools menu of the Workbench.

Parameters:

id Custom ID value

userdata Custom data

text Item text

msgport MsgPort for AppMessages of type MTYPE_APPMENUITEM.

taglist TagItem field or 0

Result: AppMenuItem structure or 0

Example: Add a menu item to the Workbench Tools menu:

```
...
movea.l $4.w,a6                  ;ExecBase
jsr     _LVOCreateMsgPort(a6)    ;get MsgPort
move.l  d0,_MsgPort              ;and save
beq     _Zerror
```

```
 ...
 movea.l _WbenchBase,a6                    ;"workbench.library"
 moveq   #1,d0                             ;ID
 moveq   #0,d1                             ;User data
 lea     _ItemText(pc),a0                  ;menu item
 movea.l _MsgPort,a1                       ;MsgPort
 suba.l  a2,a2                             ;no Tags
 jsr     _LVOAddAppMenuItemA(a6)           ;add
 move.l  d0,_AppMenuItem                   ;save result:
 beq     _Zerror2
 ...
 movea.l $4.w,a6                           ;ExecBase
 movea.l _MsgPort,a0                       ;Port
 jsr     _LVOGetMsg(a6)                    ;get message
 tst.l   d0                                ;and test
 beq     _NoMessage
 movea.l d0,a1
 cmpi.w  #MTYPE_APPMENUITEM,am_Type(a1) ;menu selected?
 bne     _NextMessage
 moveq   #1,d0
 cmp.l   am_ID(a1),d0                      ;our ID?
 bne     _NextMenu
 jsr     _LVOReplyMsg(a6)

_MenuChoice
 ...

 ...
 movea.l _WbenchBase,a6
 movea.l _AppMenuItem,a0
 jsr     _LVORemoveAppMenuItem(a6)
 ...
 movea.l $4.w,a6
_Loop
 movea.l _MsgPort,a0
 jsr     _LVOGetMsg(a6)
 tst.l   d0
 beq.s   _DelPort
 movea.l d0,a1
 jsr     _LVOReplyMsg(a6)
 bra.s   _Loop
_DelPort
 movea.l _MsgPort,a0
 jsr     _LVODeleteMsgPort(a6)
 ...

_ItemText
 dc.b    'My Menu',0
```

```
 ...
_MsgPort
 ds.l    1
_AppMenuItem
 ds.l    1
```

AddAppWindowA — **Add a window to the Workbench**

Call:

```
AppWindow = AddAppWindowA(id, userdata, window, msgport, taglist)
D0          -48(A6)       D0  D1        A0      A1       A2

STRUCT AppWindow *AppWindow
ULONG  id,userdata
STRUCT Window *window
STRUCT MsgPort *msgport
STRUCT TagItem *taglist
```

Function: Adds a window to the Workbench list and sends notification of all objects placed in the window.

Parameters:

id	Custom ID value
userdata	Custom data
window	Window
msgport	MsgPort for AppMessages of type MTYPE_APPWINDOW.
taglist	TagItem field or 0

Result: AppWindow structure or 0

RemoveAppIcon — **Remove icon from the Workbench**

Call:

```
error = RemoveAppIcon(AppIcon)
D0      -66(A6)       A0

BOOL   error
STRUCT AppIcon *AppIcon
```

Function: Undo AddAppIconA().

Parameters: AppIcon Result from AddAppIconA().

Result: 0 Error

RemoveAppMenuItem **Remove item from the Tools menu**

Call:

```
error = RemoveAppMenuItem(AppMenuItem)
D0      -78(A6)            A0

BOOL   error
STRUCT AppMenuItem *AppMenuItem
```

Function: Undo AddAppMenuItemA().

Parameters: AppMenuItem
Result from AddAppMenuItemA().

Result: 0 Error

RemoveAppWindow **Remove window from Workbench**

Call:

```
error = RemoveAppWindow(AppWindow)
D0      -54(A6)         A0

BOOL   error
STRUCT AppWindow *AppWindow
```

Function: Undo AddAppWindowA().

Parameters: AppWindow
Result from AddAppWindowA().

Result: 0 Error

```
WBDISK    = 1 ;object types: diskette
WBDRAWER  = 2 ;              directory
WBTOOL    = 3 ;              program
WBPROJECT = 4 ;              file
WBGARBAGE = 5 ;              trash can
WBDEVICE  = 6 ;              device driver
WBKICK    = 7 ;              OS disk
```

```
WBAPPICON = 8 ;              user icon

Dec Hex STRUCTURE DrawerData,0
  0  $0 STRUCT   dd_NewWindow,nw_SIZE ;for OpenWindow()
 48 $30 LONG     dd_CurrentX          ;position
 52 $34 LONG     dd_CurrentY          ;
 56 $38 LABEL    OldDrawerData_SIZEOF ;previous structure size
 56 $38 ULONG    dd_Flags             ;Flags
 60 $3C UWORD    dd_ViewModes         ;view mode
 62 $3E LABEL    DrawerData_SIZEOF    ;size of structure starting with v36

DRAWERDATAFILESIZE = DrawerData_SIZEOF

Dec Hex STRUCTURE DiskObject,0
  0  $0 UWORD    do_Magic            ;start ID: $e310
  2  $2 UWORD    do_Version          ;version number of the structure
  4  $4 STRUCT   do_Gadget,gg_SIZEOF ;Gadget structure
 48 $30 UBYTE    do_Type
 49 $31 UBYTE    do_PAD_Byte
 50 $32 APTR     do_DefaultTool
 54 $36 APTR     do_ToolTypes
 58 $3A LONG     do_CurrentX
 62 $3E LONG     do_CurrentY
 66 $42 APTR     do_DrawerData
 70 $46 APTR     do_ToolWindow       ;only with Tools
 74 $4A LONG     do_StackSize        ;only with Tools
 78 $4E LABEL    do_SIZEOF

WB_DISKMAGIC         = $e310   ;ID
WB_DISKVERSION       =     1   ;version
WB_DISKREVISION      =     1   ;revision: lower 8 bits gg_Userdata
WB_DISKREVISIONMASK  =   $ff
GADGBACKFILL         =     1
NO_ICON_POSITION     = $80000000


Dec Hex STRUCTURE FreeList,0
  0  $0 WORD     fl_NumFree
  2  $2 STRUCT   fl_MemList,LH_SIZE
 16 $10 LABEL    FreeList_SIZEOF

MTYPE_PSTD         = 1 ;standard message
MTYPE_TOOLEXIT     = 2 ;ExitMessage from Tools
MTYPE_DISKCHANGE   = 3 ;disk change
MTYPE_TIMER        = 4 ;timer tick
MTYPE_CLOSEDOWN    = 5 ;not implemented
MTYPE_IOPROC       = 6 ;not implemented
MTYPE_APPWINDOW    = 7 ;Msg for application window
```

```
MTYPE_APPICON      = 8 ;Msg for application icon
MTYPE_APPMENUITEM = 9 ;Msg for application menu
MTYPE_COPYEXIT     =10 ;end of a copy process
MTYPE_ICONPUT      =11 ;Msg from icon.library/PutDiskObject()

AM_VERSION  = 1 ;version of following structure

Dec Hex STRUCTURE AppMessage,0
  0  $0 STRUCT am_Message,MN_SIZE ;StandardMessage
 20 $14 UWORD  am_Type            ;message type
 22 $16 ULONG  am_UserData        ;user data
 26 $1A ULONG  am_ID              ;
 30 $1E LONG   am_NumArgs         ;number of arguments
 34 $22 APTR   am_ArgList         ;arguments
 38 $26 UWORD  am_Version         ;AM_VERSION
 40 $28 UWORD  am_Class           ;message class
 42 $2A WORD   am_MouseX          ;mouse position
 44 $2C WORD   am_MouseY          ;
 46 $2E ULONG  am_Seconds         ;even time
 50 $32 ULONG  am_Micros          ;
 54 $36 STRUCT am_Reserved,8
 62 $3E LABEL  AppMessage_SIZEOF


 STRUCTURE AppWindow,0   ;PRIVATE

 STRUCTURE AppIcon,0     ;PRIVATE!

 STRUCTURE AppMenuItem,0 ;PRIVATE!

Dec Hex STRUCTURE WBStartup,0
  0  $0 STRUCT   sm_Message,MN_SIZE
 20 $14 APTR     sm_Process      ;Process
 24 $18 BPTR     sm_Segment      ;SegList
 28 $1C LONG     sm_NumArgs      ;number of arguments
 32 $20 APTR     sm_ToolWindow   ;window
 36 $24 APTR     sm_ArgList      ;argument field
 40 $28 LABEL    sm_SIZEOF

Dec Hex STRUCTURE WBArg,0
  0  $0 BPTR     wa_Lock ;directory lock
  4  $4 APTR     wa_Name ;file name
  8  $8 LABEL    wa_SIZEOF
```

Part 2

ARexx

Part 2 - Introduction

ARexx - by now it's a buzzword in the Amiga community. ARexx is a logical evolution of CLI and CLI commands. As a command language, it controls external applications. You can use ARexx to tell a word processor to format text and then tell the desktop publishing program (using ARexx) to import and print the text. ARexx was conceived as a commercial product. Starting with the AmigaOS 2.0, it is a component of the Amiga operating system. Compared to simple CLI commands, variable manipulation is easier, and because variable manipulation is simple, ARexx is at least as powerful as BASIC. But can BASIC indirectly control application programs?

Author: Christian Kuhnert

4. ARexx

ARexx is not new. Since 1987, the Amiga version of Rexx by William S. Hawes has been commercially available. Since then, ARexx has become the de-facto standard for external program control on the Amiga. No serious commercial program can afford not to access the ARexx-Port as part of the Amiga multitasking operating system.

Including ARexx in the Amiga 2.0, as a component of the operating system, was the next logical step for Commodore. This decision can also be interpreted as a decision against other models of processor communication, like the IPC project in the PD field, an approach that is not as complex, but also not as flexible.

This book is not intended to be a complete guide to programming in ARexx; our focus is how an interrupt directed to ARexx can make application programs configurable, expandable and sometimes enable connection to other programs.

An experienced user of structured programming languages like C, Modula, or Pascal (or any BASIC dialect with structured form) will understand ARexx immediately.

4.1 The ARexx Language

Rexx is the name of a programming language that was developed at two IBM research sites in England and the USA between 1979 and 1982. Its main characteristics are:

- Universal applicability: Rexx is not dedicated to a certain application (or application type). Many programming languages make this claim and ARexx is actually better suited to applications that value higher running speed over programming speed.
- "Type-less" data: all data are treated as character strings at first. No type classification takes place until a specific operation is performed. Defined data types such as integers, floating decimals, bytes and words are not natural limits, but machine terms. These are limitations Rexx developers intended to avoid.

- No declarations: variables must not, as in many programming languages, be declared before use; in this sense, Rexx is like BASIC or APL. Even very large data fields do not have to be previously dimensioned.
- Only a few basic commands: about 10 commands are sufficiently powerful to create complex programs. There are a total of about 30 commands.
- Easy string manipulation: the scope of the language includes many functions that perform string manipulation, which makes this aspect more developed than in other languages.
- Easy error trapping: the Rexx interpreter has a powerful TRACE function. The trace function also enables interruptions during program execution.
- "Human" Logic: instead of following firm syntax formalities, Rexx normally does what is intuitively right. This means that if you just think about the problem, you will usually come to the correct result without looking anything up.

4.2 The Functions of ARexx

Because it transmits input to the processor, Rexx is especially well-suited as a script or batch language for automatic control of an operating system or as a macro language. These operations are the same thing, but in the latter case, Rexx controls an application program.

Almost every operating system has a shell or batch language; each has its specific features and special functions. The same holds true for macro languages that are specially designed to configure and control an individual program, such as an editor or a database manager.

Rexx was developed with an eye toward becoming a universal command language. Rexx can pass commands to an external environment (or an operating system) and receive an answer in return.

Rexx is also capable of acting as a universal programming language, because it enables the creation of function libraries. These effectively expand the scope of the language itself. Specifically, Rexx makes program development and testing quick and easy.

4.3 An Overview of ARexx

All Rexx programs begin with a C-style comment. "/*" is expected at the beginning of the program by the interpreter. This convention encourages the programmer to document the purpose of the program with a short comment. A complete Rexx program would appear as follows:

```
/* A simple example program */
SAY "I am."
```

SAY is a Rexx command. It displays the following expression, which is a character string. You do not have to type commands in capital letters; the interpreter only differentiates between large and small letters within character strings. Double quotation marks (") or simple quotations (') define a character string.

The simplest counterpart to "say" is "pull":

```
/* Calculate body weight in engl. stones */
say "Please enter your body weight in Kg!"
pull weight
say "That equals" weight/6.348 "stones."
```

PULL waits for a user entry and assigns it to the variable ("weight"). As you can see, the variable name did not need to be specified. Even an error in user entry (like typing a letter) would have no consequences in the "Pull" line, since variables always contain character strings. Once the string is divided in the last line, the input must be interpreted as a number with a floating decimal. An error message does not appear until the division in the last line is impossible; then the program stops. Numbers can be written with a decimal point or in exponential notation. By using the NUMERIC setting, the number of decimal places can be set.

Although it doesn't appear to mean anything, the space character between the two expressions is also an operator for the SAY command. The space indicates that it should concatenate with a space inserted in the program.

The empty space can also be inserted within the two strings. Then the individual expressions are directly concatenated; this too forces concatenation, but without additional empty spaces.

An explicit concatenation operator also exists: "||" - it directly connects the contents of two variables, without an empty space.

Rexx has all the usual commands for program control. The most important and complex of these is the DO ...END group, which (like BEGIN...END in Pascal, or {...} in C) is a simple command grouping. It's used to control the formation of program loops: a sub-keyword FOREVER sets up unlimited repetition, a run-time variable can increment to a maximum value, or a BY can issue a step width. A FOR sets a maximum number of loops. Program termination conditions are WHILE or UNTIL. These commands can be combined in a meaningful manner with other commands that iterate and leave, go to the next step, or exit from loops.

There is also an IF...THEN...ELSE construction and a process like "switch" in C, called SELECT...WHEN...OTHERWISE...END. There is no "Goto" command. The SIGNAL command jumps to labels within a calculation, but not within command groups or loops. These always terminate with the SIGNAL command. Together, all of these lead to clearly structured programming.

```
/* Calculate factor */
ARG number              /* read in argument */
result=1                /* result(-ing) variable initialized */
if number<0             /* For negative entry */
   then return          /* cancel */
do n=1 to number        /* Loop with run-time variable n */
   result=result*n
   end n                /* the entry 'n' is optional */
say number"! =" result
```

Along with the use of a "do" loop, this example shows another possible method of data entry: ARG reads arguments listed after the Rexx command into variables.

If this program is started by entering RX FACULTY 7, the "number" receives the value "7". If the "if" query receives a negative number, it cancels the program with the RETURN command.

If the input is over 171, the result is not correct. 1.79769314E+308 is the upper limit for the program. An error message "Arithmetic overflow" is not implemented in ARexx 1.14.

ARexx requires a certain amount of care when dealing with very large or very small numbers, since false results do occur. The reason for such limitations may be ARexx's use of the mathieeedoubbas.library for all arithmetic calculations. Perhaps this will be changed in future versions.

In the next example, the program is defined as an internal function. For Rexx, functions are a part of the language, defined in programs or externally accessed in libraries.

```
/* example for function definition and call */
do n=1 to 9
   say n"! =" fak(n)      /* call fak */
   end n
           return         /* program end */
    fak: procedure        /* function name; local variable */
    ARG number            /* read argument */
           result=1
      if number<0
          then return "Error!"
      do n=1 to number
          result=result*n
          end n
    return result           /* end the function with value output*/
```

The function "fak" is defined in the lines after the label "fak:". The key word PROCEDURE is necessary because the main program uses the same run-time variable ('n') as this function. A separate variable environment is defined for the function, so the program works with local variables.

ARG reads arguments (given in the parentheses) at the function call and ends the function with a return. It outputs the background expression and replaces the call with that value.

4.4 ARexx - Rexx on the Amiga

ARexx is a version of Rexx that's used on the Amiga as a command and macro language. Rexx's rather unusual mathematic capabilities are considered less important than its program control features. ARexx is easy to use and many operating tasks can be automatically handled by the interpreter. Compared to the standard version, this version is slightly expanded. Unfortunately, because of changes in file operations, porting data is more difficult.

However, the expansions are used to adapt the Rexx language to the qualities of the Amiga. Although this makes the language easier to use, it also decreases its speed. Amiga BASIC is about six times faster than ARexx (Version 1.12). Although the running speed can be increased slightly, ARexx isn't suitable for large-scale programming.

Because all data are handled as strings, which require frequent internal conversions, and functions are called by runtime linking, this language offers limited programming possibilities. In the future, a compiler for ARexx may be available. If this happens, further applications may be possible. However, even compiled code would barely reach the speeds of C-compiled applications.

The core of the ARexx system is the Rexx master procedure, which manages function libraries and common data structures. This procedure waits in the background for the start of an ARexx program, which is often performed using the CLI program "rx". Any program can use the Rexx port to call a Rexx program. Rexx searches the current directory and then a logical device named Rexx: (if it is installed) for the desired program.

Every ARexx program starts a separate task that reads and executes the source code with the Rexxsyslib.library, which contains the actual interpreter. In this way, an unlimited number of ARexx programs can be run simultaneously, even with limited storage capacity.

4.5 A Sample Application

The following program can be used to experiment with the Rexx language. A simple line interpreter can be used to execute ARexx commands directly and interactively. It can also be expanded to become a complete and easy-to-use shell, fully replacing the CLI.

A Rexx program can be created using any editor. If you want to run it from CLI, give it the tag .Rexx and store it in the "Rexx:" directory. Call it from CLI by typing "rx program_name". If the Rexx master procedure is not already running, "rx" starts it and then executes the Rexx program.

```
/* interactive Rexx interface */
address command                /* command destination is the CLI */
options prompt "Rexx> "        /* A prompt for pull */
start:                         /* entry point at error */
signal on syntax               /* At error moves to the equivalent */
signal on error                /* label branching instead cancel */
do forever                     /* endless loop */
   parse pull input            /* Wait for entry */
   interpret input             /* execute entry as an ARexx line */
end                            /* next loop */

syntax:                        /* at syntax error output message */
say "Error" rc "in line" sigl":" errortext(rc)
signal start                   /* ...and so forth*/

error:                        /* at command error ...*/
say "Returncode:" rc          /* ...output Returncode */
signal start                  /* ...and so forth*/
```

The ADDRESS sets the destination for the external command. Rexx views all free-standing expressions (those that are not used by a Rexx command) as expressions that are to be transmitted to an external environment.

To specify DOS as the recipient, use the COMMAND address; otherwise the name of the Rexx Message Port would be the called program.

The OPTIONS prompt is a specific command for Rexx that results in an output of the strings defined as a user entry prompt when PULL is executed.

SIGNAL ON indicates that the error condition listed after it should not lead to a program stop, but to an equivalent label in the program. By doing this, errors can be trapped. When this is executed, all running "do" groups end and the corresponding Signal flag is turned off, as they are in the direct jump command using "SIGNAL Label". After error handling you are unable to continue the program, instead you must address a defined entry point. Special system variables contain the line number (SIGL) after interrupt conditions, or in this case the error code or the return code. The use of these variables becomes clear in the two blocks at the end of the program. ERRORTEXT() is a built-in Rexx function that outputs an appropriate (English) text for a given Rexx error number.

PARSE transmits character strings to variables. It also offers a powerful and simple procedure for string manipulation and cutting.

Although PARSE PULL waits for an entry from the console, there is no capitalization of the entry, as in the PULL command.

INTERPRET is a very simple but powerful command. The expression that follows is simply executed as a Rexx command.

Experiment with this program; it quickly gives you an understanding of how Rexx works.

5. ARexx Syntax

ARexx programs can contain all ASCII symbols. Either uppercase or lowercase letters can be used, since all symbols are automatically converted to capital letters.

An ARexx program must begin with a comment. The interpreter then searches for a clause, usually a single line, that is delimited by a semicolon (;), keywords or a colon (after an individual character). The tokens contained in the clause are then evaluated from left to right.

5.1 Using Tokens

Tokens are the smallest, self-contained units in the language, such as words in a sentence. They are separated by empty spaces (or, for operators, by their parameters). The interpreter differentiates among comments, symbols, strings, operators, and special characters.

5.1.1 ARexx Symbols

Symbols are characters (A..Z, a..z, 0..9), .!?$#@ and _ (Underscore). Alphabetical characters that appear in a symbol are converted to capital letters. There are four types of symbols:

- Constants begin with a number or a decimal point.
- Simple variables do not begin with a number and do not contain a period.
- Stem are like simple variables, but have a period at the end.
- Compound variables begin with a stem, followed by one or several constants or simple variables, each delimited by a period. The value of a constant symbol (that is not necessarily a number) is the name of the symbol, in capital letters. Other symbols are variables. They can be assigned a value during the program run. If a variable has not been given a value, it is an uninitialized variable and acts as a constant; its value is then its capitalized name. For example:

```
     47.11 /* a constant */
     7NewYorkers /* a constant, but not a number.
=> 7NEWYORKERS */
     Field. /* a stem symbol */
     Field.3.Where?/* a compound variable */
```

Stems and compound variables have special qualities that enable unusual programming techniques. The structure of a compound variable is "stem.n1.n2....ni". The name before the first period is the stem symbol and every other element, from "n1" to "ni", is either a constant or a simple variable. Whenever the interpreter finds a compound variable, the elements in it are evaluated. These strings can contain any characters, even spaces, and are not converted to capital letters. A new variable name is created and its contents are then calculated. For example, if "X" has the value 5 and "Y" has the value 2, then "a.x.y" creates the new name "A.5.2". By using the stem you can call or initialize an entire group of variables. If a stem is assigned a value, all combined variables that contain the stem also receive the same value.

Compound variables can also be used as addressable arrays or stacks. For example, if you wanted to show the area code of a city with the city name, you could create two fields "CITY" and "AREACODE". Paired values would be stored with the same index. The field "CITY" would be searched for the desired entry and "San Francisco" would be found with the index "415". In this case, "AREACODE.415" would contain the appropriate area code. In Rexx you can take another approach: the variable "CITY" could contain the name of the desired place and "AREACODE.CITY" would evaluate into "AREACODE.SanFrancisco", which would lead directly to the desired number. Although this process offers faster access to the data, it's not reversible. You cannot use the same field to look for the city name by area code, which is possible with the first method.

5.1.2 Character Strings in ARexx

Strings are character strings that begin and end with quotation marks ("). The quotation mark itself can be included by typing it twice (""). Single quotes can be used instead of quotation marks. Strings must be written on one line only. Empty character strings are called "null strings". A string followed by an open parenthesis "(" is assumed to be a function name. An "x" or "b" immediately following indicate hexadecimal or binary evaluation of the string. In this case, only the characters (0..9 and a..f for "x" and 0 or 1 for "b") can be contained in the string. (Empty spaces can be used to make the program readable.) Such character strings are immediately converted into strings with the equivalent ASCII symbols. Enter control codes or memory addresses in this way. For example:

```
'Is there a grammar'            /* simple example */
"Is it possible..."             /* => Is it possible... */
""                              /* Null-String*/
"Say ""It is true""!"           /* => Say "It is true"! */
"49 42 4d"x                     /* big blue in hex*/
"00110000"b                     /* binary for ASCII 0*/
```

5.1.3 The ARexx Operators

Operators are the characters ~+-*/=><&|^. Empty spaces (even between them) make no difference to the ARexx interpreter. The space character itself, placed between symbols or strings, is an operator. The execution of operators has a set order. Operators with equal priority are executed from left to right.

<table>
<tr><th>Operator</th><th>Priority</th><th>Meaning</th></tr>
<tr><td>~</td><td>8</td><td>logical negation</td></tr>
<tr><td>+</td><td>8</td><td>prefix: conversion</td></tr>
<tr><td>-</td><td>8</td><td>prefix: negation</td></tr>
<tr><td>**</td><td>7</td><td>exponentiation</td></tr>
<tr><td>*</td><td>6</td><td>multiplication</td></tr>
<tr><td>/</td><td>6</td><td>division</td></tr>
<tr><td>%</td><td>6</td><td>integer division</td></tr>
<tr><td>//</td><td>6</td><td>remainder</td></tr>
<tr><td>+</td><td>5</td><td>addition</td></tr>
<tr><td>-</td><td>5</td><td>subtraction</td></tr>
<tr><td>||</td><td>4</td><td>concatenation</td></tr>
<tr><td>(space)</td><td>4</td><td>concatenation with empty space</td></tr>
<tr><td>==</td><td>3</td><td>exact equality</td></tr>
<tr><td>~==</td><td>3</td><td>exact inequality</td></tr>
<tr><td>=</td><td>3</td><td>normal equality</td></tr>
<tr><td>~=</td><td>3</td><td>normal inequality</td></tr>
<tr><td>></td><td>3</td><td>greater than</td></tr>
<tr><td>>=,=<</td><td>3</td><td>greater than or equal</td></tr>
<tr><td><</td><td>3</td><td>less than</td></tr>
<tr><td><=,~></td><td>3</td><td>less than or equal</td></tr>
<tr><td>&</td><td>2</td><td>logical AND</td></tr>
<tr><td>|</td><td>1</td><td>logical OR</td></tr>
<tr><td>^,&&</td><td>1</td><td>logical exclusive OR</td></tr>
</table>

5.1.4 ARexx Special Characters

The special characters ";,:()" also have meaning. For example:

semicolon(;) A semicolon separates individual clauses. Normally, this is indicated by a line feed. Semicolons are used to put several clauses on one line.

comma(,) A comma prevents the automatic semicolon, if a clause extends over several lines. (Commas also separate the arguments of a function call from one another.)

colon(:) If there is a symbol in front of a colon, a branching (Label) is defined. A colon also implies a semicolon.

parentheses(()) A single open parenthesis, directly following a symbol, forces interpretation of the clause as a function name. Closed parentheses also form expression groups. This is used to alter the regular operator priority.

5.2 Expressions

Expressions consist of one or several terms, with or without operators. They can be strings, symbols, or function calls, perhaps grouped with parentheses. Between a pair of terms, there is always a dyadic element. There can be one or several prefix operators affecting the term. Strings are always interpreted as character strings, as are constant symbols (converted into capital letters). Variable symbols are replaced with their contents, or regarded as constant symbols. Function calls are recognized by an open parenthesis, followed by a symbol.

Arguments contained in parentheses are evaluated and passed to the function in place of the arguments. The calculated value is returned.

The value of an expression is determined in order by parentheses and operator priority. First the symbols that are contained in it are replaced. For example:

```
fak(n) "is the factorial of" n
```

This expression consists of the function call "fak(" with the argument expression "n", a concatenation operator (concatenation with empty spaces), the string "is a factorial of", a further chain, and the variable symbol "n".

First, the interpreter determines the function argument and then calls the function. The function, if it's not defined as a procedure, can assign a new value to the variable "n". At the second occurrence of "n", another content is calculated. The order of evaluation does not affect the calculation here, only its position. In short, symbols are always evaluated from left to right and replaced by the resulting value. If there is a function call, it's executed first, then the symbol value replacement process resumes. After this, the expression is evaluated under the priority rules. If the operator priority ranking is equal, an evaluation order is not defined. So, the analysis moves from left to right, as the Rexx language definition implies (and algebraic rules state and programmers expect). Exceptions to the rule have not been observed.

Both sides of logical operators are always evaluated, even if the result is already clear:

```
(2 = 2) | (120 = fak(5))
```

For example, "fak(" is always fully executed, even if analysis has already determined that the result of the procedure is 1.

Operators can be divided into four groups: arithmetic, concatenation, comparison, and logical operators.

5.2.1 Arithmetic Operators

Prefix conversion (+)

This operator acts as a prefix. The given number is converted to internal notation, rounded, and formatted according to the NUMERIC settings.

```
" 12.34 "        ==> '12.34'
"2.0009          ==> '2.001' (with DIGITS=3)
```

Prefix negation (-)

The single negation prefix changes the sign of the operand. It also has the same effects as the prefix for conversion (+).

```
-" 7.23 "        ==> '-7.23'
-3E3             ==> '-3000'
```

Exponentiation (**)

The left operand (base) is evaluated as the exponent of the right operand (exponent). The exponent must be an integer. The number of decimal places (for positive exponents) is the product of the exponent and the number of places given after the decimal point in the base number.

```
2**8             ==> 256
4**-1            ==> 0.25
0.1**3           ==> 0.001
```

Multiplication (*)

Calculates the product of the terms to the left and right of the operand. The number of decimal places is determined by the sum of the decimal places in the two terms.

```
4*7            ==> 49
0.5*1.50       ==> 0.750
```

Division (/) Determines the quotient of the two numbers. The number of decimal places is as large as necessary, and can be limited by the setting of NUMERIC DIGITS.

```
8/4            ==> 2
8/3            ==> 2.667 (with DIGITS=3)
```

Integer division (%) Calculates the quotient of two numbers. The integer portion of the result is returned.

```
8%3            ==> 2
```

Remainder (//) Returns the remainder of an integer division of the dividend terms. To determine the remainder of "a//b", "a-(a%b)*b" is calculated.

```
8//5           ==> 3
-7//3          ==> -1
3.7//0.2       ==> 1
```

Addition (+) Calculates the sum of two terms. The number of decimal places in the result is determined by the higher number of decimal places in one of the terms.

```
3+15           ==> 18
2.7+2.04       ==> 4.74
```

Subtraction (-) Calculates the difference between two terms. As in addition, the number of decimal places is determined by the higher number of places in one of the two.

5.2.2 Concatenation Operators in ARexx

These operators combine two strings into a new string. There are three such operators:

- The explicit concatenation operator (||) connects two strings without an empty space.

```
"BE"||"TA"          ==> BETA
```

- The direct concatenation operator, for example, a symbol and a string, specified right after one another. This results in a chain without an empty space.

```
be"TA"              ==> BETA
```

- The null concatenation operator is when two strings are specified with one or more spaces between them; an empty space is inserted between them in the concatenation.

```
"with" "empty" "space" ==> with empty space
```

5.2.3 Comparison Operators in ARexx

The are three different comparison modes:

- Exact includes empty spaces; strings of different length are never equal in exact comparison.
- String comparison disregards leading spaces and adds trailing spaces to fill a shorter string to equivalent length.
- Numeric transforms the operands to numeric notation, using the setting of NUMERIC DIGITS to determine the number of decimal places. Then an arithmetic comparison is made and NUMERIC FUZZ sets the specificity of that calculation.

With the exception of the exact equality and inequality operators, all comparison operators automatically differentiate between numeric and string comparisons.

If both terms are valid numbers, a numeric comparison is made; otherwise it is assumed to be a string comparison.

All comparison operators output a Boolean truth value: 0 for false or 1 for true. The comparisons ">", "<", ">=" and "<=" are used for strings, as defined in the ASCII code. This means:

```
"A"<"B" ==> 1
"A"<"a" ==> 1
```

5.2.4 Using Logical Operators

All four logical operators require two Boolean operators (a 0 or 1) and return a Boolean result. These operators cannot be used for bit-level logical combinations. (For such purposes, use the built-in BITxxx() functions.)

Logical NOT (~)
: Inversion: 0 becomes 1 and 1 becomes 0.

Logical AND (&)
: Returns a 1 if both operators are true.

Logical inclusive OR (|)
: Returns a 1 if one of the operators is true.

Logical exclusive OR (^ or &&)
: Returns a 1 if one of the operators (not both) is true.

5.3 ARexx Clauses

Clauses are the smallest executable units of the Rexx language. They can be divided into five groups:

5.3.1 Null Clauses

Lines that consist of empty space or comments are null clauses. They are also formed if two semicolons follow one another. These clauses are ignored by Rexx.

Comments are character strings of one or more lines that are contained in "/*" and "*/"; they can be set inside one another, but must appear in pairs (which, in ARexx, do not have to appear on one line). They hardly affect execution speed and can be used liberally. A first run through the interpreter removes them and their function is taken by an empty space.

5.3.2 ARexx Label Markers

A symbol immediately followed by a colon is a label marker. (The colon also implies a semicolon here.) Such markers serve as targets for CALL and SIGNAL commands and internal function calls. Several markers can follow one another.

If the same label marker appears twice in a program, only the first is located.

5.3.3 Assignments in ARexx

A variable symbol followed by an equal sign (=) is an assignment clause. In this case, the operator does not have its normal comparison function, instead it becomes an assignment operator. The terms to the right of the equal sign are analyzed as an expression and the result becomes the content of the variable symbol on the left.

5.3.4 ARexx Commands

A clause that starts with a command keyword is a command clause. Often a single command represents an executable action, or several commands (for example, SELECT groups) are combined to form clauses. They are not syntactically complete until all necessary commands are available.

5.3.5 Commands

An expression that cannot be assigned to any other type of clause is assumed to be external commands. The expression is then analyzed and the result is passed to the specified external environment. The address can be an external application (for example, an editor) or DOS ("COMMAND").

6. Instructions

Instructions consist of one or more words that are recognized as a key word. The keyword must appear as the first token in the clause. It cannot be preceded by a colon (:), it would then be a label, or an equal sign (=), which indicates a variable. Some key words call for further parameters of sub-keywords. We don't recommend that you define a variable "SAY" or a function "NUMERIC" because the readability of your program can suffer.

I/O Instructions

```
ARG
ECHO
PARSE
PULL
PUSH
QUEUE
SAY
```

Structural Instructions

```
BREAK
DO
ELSE
END
EXIT
IF
ITERATE
LEAVE
NOP
OTHERWISE
RETURN
SELECT
SIGNAL
THEN
WHEN
```

Control Instructions

```
ADDRESS
CALL
DROP
INTERPRET
NUMERIC
OPTIONS
PROCEDURE
SHELL
TRACE
UPPER
```

6.1 I/O Instructions

ARG

Represents "PARSE UPPER ARG". For more information, see that section.

ECHO

An ARexx synonym for SAY. For more information, see that section.

PARSE

Syntax: `PARSE [UPPER] source [template][,template...]`

Function: In Rexx, PARSE is the main input instruction. It takes data from various sources and passes it on to one or several variables, efficiently parsing character strings. The effect of the input character string can be selected using the following sub-keywords:

ARG Character strings passed to the program at the call or function are parsed. Each program usually receives one string; functions are capable of receiving up to 15, separated by commas, that are then parsed out according to templates.

EXTERNAL An entry is read into the function from "stderr". If "stderr" is not defined, the function returns the null string.

NUMERIC Current settings of NUMERIC are received as a string in the following order: DIGITS, FUZZ and FORM, each separated by an empty space.

PULL Reads a string from "stdin", that is usually input from the keyboard. If nothing is found in "stdin", program execution halts until something is entered. The function QUEUED can query how many lines have been saved in "stdin".

SOURCE Returns a string of data on how the program was called, in the form "{COMMAND|FUNCTION} {0|1} Type Result

Called Resolved Ext Host". The first word signals whether the call is a program or a function. Then a Boolean value indicates whether a result string has been requested. "Called" is the name with which the program was invoked, "resolved" is the full path and file extension of the program (usually ".rexx"). "Host" is the initial host address for external commands.

VALUE expression WITH
An expression that calls for the sub-keyword WITH. An expression is evaluated and the result is used as the parse input string. The keyword "WITH" is used to separate the expression and the template.

VAR Variable
The values of the given variables are used as the parse input sting. If several templates are entered, the current value of the variable is taken each time. (It can also change if the same variable appears in the template.)

VERSION Returns a string in the form "ARexx Version CPU MPU Video Freq". The value of "Version" is the interpreter revision (i.e. "V1.21"), "CPU" is the processor type ("680x0"), and "MPU" is the math-coprocessor ("6888x", if present, otherwise "NONE"). "Video" returns to video system ("PAL" or "NTSC") and "Freq" returns the network frequency ("50HZ" or "60HZ").

The UPPER keyword forces a translation of the source data into capital letters and it's used before the keyword, which indicates the source.

Templates can be assembled from symbols, strings, operators or parentheses. The function parses source strings into sub-strings that are assigned to the symbols in the template. The process ends when all variables have been assigned values. If a source string is completely evaluated, before all listed variables obtain a new value, the remaining variables are assigned the null string. There are three important template functions:

Parsing by words

If the variable names follow directly after one another delimited only by an empty space, the source string is parsed (using the spaces) into words, each of which is assigned to the next available variable. The last variable receives the remainder of the string. A period (.) can be used as a "placeholder" in a template, acting like a variable, but not actually receiving the corresponding part of the string. For example:

```
/* VERSION returned:"ARexx V1.15 68030 68882 PAL 50HZ" */
parse version . Revision CPU MPU .
say Revision CPU MPU
```

In this example the first word "ARexx" is not informative; it is to be deleted and so a period appears. After the parse is executed, the variable "Revision" contains " V1.15", CPU the value "68030" and MPU is "68882". The rest of the source string is uninteresting; it is absorbed by the second period, same as the first word.

Parsing by position

Absolute or relative positions of individual elements of the source strings can be specified using numeric values, between the symbols. Relative positions are differentiated from absolute positions by their prefixes (monadic "+" or "-" operators). For example:

```
Test = "1234567890"
parse var Test 3 a 5 b +3 4 c
say a b c               /* => 34 567 4567890 */
```

Parsing by pattern

If elements of the source string are separated by keywords or other specific characters, these can be searched and the parsing will follow the "pattern markers." The function also removes markers it finds from the source string; this means the string is changed during a pattern parse. An example:

```
/* The program was called with the argument "DRIVE dh0: name Bingo".*/
parse arg "DRIVE" Drive "NAME" Name
say Drive name
        /* => dh0: Bingo*/
```

See also: ARG, PULL, UPPER

PULL

PULL is short for "PARSE UPPER PULL". For further information, see that section.

PUSH

Syntax: `PUSH [expression]`

Function: PUSH prepares input lines for another program that expects entry from "stdin". During the function, a "Return" (ASCII 13) is attached to "expression" and the result is stored in the "stdin" channel along the "LIFO principle" (Last In First Out). The last line stored with PUSH is the first read from "stdin". The number of lines waiting in "stdin" can be queried with the function QUEUED.

Caution: This instruction should only be used with interactive DOS devices that are driven by a DOS handler that supports the "ACTION_STACK" instruction (or CON:, PIPE:, and similar input). This is especially important if you are attempting data redirection.

See also: QUEUE, QUEUED()

QUEUE

Syntax: `QUEUE [expression]`

Function: QUEUE prepares input lines for another program that expects entry from "stdin". The value of "expression" is a single "Return" (ASCII 13); the result is stored in the "stdin" channel along the "FIFO principle" (First In First Out). The first line stored with QUEUE is also the first read from "stdin". The number of previously stored lines in "stdin" can be queried with the function QUEUED.

Caution: This instruction should only be used with interactive DOS devices that are driven by a DOS handler with an "ACTION_QUEUE" instruction (or CON:, PIPE:, and others like them). This is important for scripts involving input redirection.

See also: PUSH, QUEUED()

SAY

Syntax: `SAY [expression]`

Function: The value of "expression" receives a single "Return" (ASCII 13) and is written to "stdout", which is usually the monitor, and displayed there.

6.2 Structured Instructions

BREAK

BREAK is only allowed within DO instructions. For more information see that section.

DO

Syntax:
```
DO [Iteration] [Condition]
      [Instructions]
      END [Symbol]
```

DO is used to group instructions together and possibly execute them again. The iteration takes the form:

```
[Symbol=ExprI] [TO ExprT] [BY ExprB]] [FOR ExprF]
   or: ExprR
   or: forever
```

All expressions that appear in the instruction must result in a number. ExprR and ExprF must be positive integers. The key words BY, TO, and FOR can appear with a matched expression (that is analyzed once at the beginning) in any order.

- The formal element "Symbol=ExprI" defines a run-time variable and supplies an initial value to it. It must follow the key word DO.
- BY ExprB: Determines the increment added to running variables with each iteration. If BY is not specified, it is assumed to be 1.
- TO ExprT: Sets the upper limit (or lower limit, depending on the increment) of the run-time variable. If this limit is overstepped during an iteration, the loop terminates and the program continues at the corresponding END.
- FOR ExprF: Specifies the maximum number of repetitions. When this value is reached, the DO loop terminates, regardless of the value of a run-time variable. If all you need is to specify the number of repetitions, you can use the ExprR form.
- FOREVER: If you do not need an index variable, this key word ensures repetition. This kind of loop ends with a LEAVE or BREAK. For example, the condition can read:

```
WHILE ExprW
  or: UNTIL ExprU
```

ExprW and ExprU must return "0" or "1".

- WHILE ExprW: This expression is evaluated at the beginning of each iteration. If it returns a "1", the loop continues; a "0" terminates it.
- UNTIL ExprU: Has the same function as WHILE, but with reverse logic. If it's a "0", the loop is terminated; if it's a "1", the loop continues.

The DO group is closed with the END instruction, after which a counting variable can be specified. This is helpful for debugging, since nesting errors during DO loops can be recognized by the interpreter. It also improves program readability.

Structural instructions in DO groups

```
BREAK, LEAVE [Symbol], ITERATE [Symbol].
```

BREAK terminates the inside DO group. Program execution continues after moving to the corresponding END. This is also the action an INTERPRET instruction forces in implicit DO groups.

In contrast, LEAVE is only allowed in DO loops. LEAVE terminates the inside DO group and execution continues after the corresponding END. A variable can be set here to terminate several embedded DO loops simultaneously, if your run-time variable is controlled in one of the outer loops.

ITERATE does not terminate the DO loop, but rather jumps back to the top of it. The value determined with BY is added to the run-time variable, all conditions are evaluated and, if appropriate, the next iteration starts. The variable specified after ITERATE acts analogously to the variable function in LEAVE.

ELSE

ELSE is only valid within an IF instruction. For further information, see that instruction.

END

END is an element of DO and SELECT groups. For more information, see that section.

EXIT

Syntax: `EXIT [expression]`

Function: This instruction terminates the program where it is read and passes (if indicated) a return value to the calling program. If a return string is requested, the result of "expression", a character string, is stored in the allocated storage block and a pointer to that block is set to RESULT_2. If no string was requested and the program was running as an instruction, EXIT tries to evaluate the result of "expression" with INT and report it as the return code. The EXIT instruction (without a return value) is also implied by the end of a program.

See also: RETURN

IF

Syntax:

```
IF expression
      THEN command
      [ELSE command]
```

Function: The IF instruction conditionally executes instructions. The expression after IF must return a Boolean result, a "0" or "1". If it's a "1", the instruction (or DO group) named after it is executed. If the expression is "0", the instruction behind ELSE is executed, if ELSE was specified.

An ELSE clause always refers to the last IF. Nested IF instructions make it impossible to use one of these branches. In this case, a dummy instruction (NOP) restores access to the next higher IF instruction. Just indicating an empty clause by typing two semicolons is not enough in Rexx, as it is for other programming languages. For example:

```
if 1+1=2 then                                    /* outer IF */
   if 2+2=4 then                                 /* inner IF */
```

```
        say "The world is still alright."
            else NOP                         /* belongs to the inner IF */
    else say "Nothing is happening anymore!"     /* belongs to the outer IF */
```

See also: NOP

ITERATE

ITERATE is only applicable within DO loops. For more information, see that description.

LEAVE

LEAVE is only applicable within DO loops. For more information, see that section.

NOP

Syntax: `NOP`

Function: Basically, "No OPeration" is a dummy instruction that does nothing. The instruction takes on meaning during IF instructions; it has no other function.

OTHERWISE

OTHERWISE is part of the SELECT instruction. For more information, see that section.

RETURN

Syntax: `RETURN [expression]`

Function: Used to end one function and continue program execution where it was called. "Expression" is analyzed and the result is the returned value of the function. Functions called from within an expression must return a result, or an error message appears. Placing a RETURN in the main program is not an error, it is equivalent to EXIT.

See also: EXIT

SELECT

Syntax:

```
SELECT
        WHEN expression [;] then [;] command
        [...]
        [OTHERWISE [;] [Instructions]]
        END
```

Function: The SELECT instruction is used to make a choice among several different possibilities. After it, a series of WHEN constructions can follow, each of which must contain an expression that returns a Boolean result, and a instruction (or DO group) that is to be executed if the result of the expression is "1". But only the first WHEN group whose Boolean expression returns a 1 is executed. If none of the expressions are true, instructions behind the key word OTHERWISE are executed. This can comprise an entire list of instructions. OTHERWISE and END shape a simple DO group in this respect.

The OTHERWISE construction is not required in a SELECT instruction. If no WHEN construction is executed, it's either called or an error message appears.

SIGNAL

Syntax:

```
SIGNAL {ON|OFF} Condition
    or: SIGNAL { [VALUE] expression | Label }
```

Function: There are two basic forms of the SIGNAL instruction. The first is used to switch error trapping flags on and off. The second is an expression used to transfer control; this one is evaluated. A control transfer should be used sparingly. It is primarily useful for resuming the program at a defined location after an error condition.

Error interrupts make it possible to recognize error conditions that would otherwise lead to program termination, and perhaps to catch them before they do. If "SIGNAL ON condition" is used to activate a specific error interruption, the program is not stopped, instead it branches to a label indicated by the key word of the corresponding condition. The following key words are available:

BREAK_C	Ctrl+C: Break has occurred.
BREAK_D	Ctrl+D: Break has occurred.
BREAK_E	Ctrl+E: Break has occurred.
BREAK_F	Ctrl+F: Break has occurred.
ERRORS	The return code passed by an external program was not "0".
FAILURE	The return code was greater than the FAILAT setting.
HALT	A HALT was encountered (after "hi", for example).
IOERR	DOS has reported an I/O operation error.
NOVALUE	An undefined variable was called.
SYNTAX	There have been syntax or execution errors.

THEN

This is only used within IF and SELECT instructions. For more information, see that section.

WHEN

WHEN is also only used within a SELECT instruction. For more information, see that section.

6.3 ARexx Control Instructions

ADDRESS

Syntax:

```
ADDRESS {Symbol | String} expression
ADDRESS {Symbol | String}
ADDRESS [VALUE] expression
ADDRESS
```

Function: ADDRESS defines the target of an external command. Its argument must provide the name of an ARexx message port, listed in the Public Port List of "exec". The first form shown does not change the current setting, it only sends a command to a certain ARexx port. The name of the port is specified as a symbol (that can be variable) or a string. Then the command character string to be passed to the external message port follows. The next two forms of ADDRESS set a new target command host. In the third form, the name of the target is an expression that must first be analyzed.

VALUE is only necessary if the expression starts with a symbol or character string. The interpreter also stores the indicated target; the last form of the examples shown (without parameters) toggles between two addresses.

The default setting is "REXX". The "COMMANDS" host is a special target address (it represents DOS). To query the current setting, use the internal function ADDRESS(). Any clause that only contains a single expression that the interpreter cannot manipulate in any way, is assumed to be an external command and passed to the appropriate port.

See also: SHELL, ADDRESS()

CALL

Syntax:

```
CALL {Symbol|String} [Exp.] [, Exp., ...]
```

Function: Calls an internal, built-in, or external function. The function name (as a string) can be a symbol that in turn can be a variable or a character string. Entering a character string bypasses the internal function search of the program. This instruction controls functions internal to the interpreter, or external functions, which are protected from re-definition in the running program. Following CALL, if necessary, one or several expressions, separated by commas, can present arguments for the function (for external functions, the maximum number is 15) that can then be accessed with ARG.

In contrast to the usage of the direct function call "Symbol (Arg,Arg,...)", which is analyzed to return a value immediately, CALL returns its result to the system variable RESULT. If no value results, RESULT remains un-initialized after the call.

DROP

Syntax: `DROP symbol [Symbol ...]`

Function: Variables are deleted with DROP. They are placed in an un-initialized state, in which the value of the variable is the variable name itself. It is not a mistake to use DROP on a previously un-initialized variable. If a stem (a symbol ending with a period) is specified as the variable to be dropped, all variables that use this stem are re-set.

INTERPRET

Syntax: `INTERPRET expression`

Function: The result of "expression" is interpreted as an ARexx program. By using this instruction, entire sections of the program can be evaluated only when the program is actually run. The result of "Expression" is executed as if it were surrounded by an implied DO ...; END" group. LEAVE or ITERATE instructions can only refer to DO loops also defined here. BREAK makes it possible to leave the INTERPRET instruction and continue the program. Branching instructions in the expression are ignored by the interpreter and you cannot define a function using INTERPRET.

NUMERIC

Syntax:
```
NUMERIC {DIGITS | FUZZ} [Expression]
NUMERIC FORM {SCIENTIFIC | ENGINEERING}
```

Function: NUMERIC sets the form of number representation and the precision of numeric operations.

DIGIT [Expression]

Sets significant places in arithmetic operations. "Expression" must be an integer between 1 and 14, larger than the current NUMERIC FUZZ setting. Small adjustments here should be made with care, since all operations, including run-time variables, are affected. If "Expression" is not included in the clause, the default setting is 9. The current setting can be queried with the function DIGITS().

NUMERIC FUZZ [Expression]

Returns the number of places (from the right) to be disregarded during numeric comparison and rounding operations. "Expression" must evaluate to an integer from 0 to 13 that is smaller than the current NUMERIC DIGITS setting. If it's omitted, 0 is the default setting. The current setting can be queried with the function FUZZ().

NUMERIC FORM {SCIENTIFIC | ENGINEERING}

Determines the type of display for results in exponential notation. Choices are the academic style (with numbers between 1 and 10 in front of the decimal point), the SCIENTIFIC (default) setting, or engineer's display in which the exponent is always a multiple of 3. The current setting can be called with the function FORM(). These NUMERIC settings are protected during a call to a function and set back after completing it.

See also: PARSE NUMERIC, OPTIONS

OPTIONS

Syntax:

```
OPTIONS FAILAT expression
OPTIONS PROMPT expression
OPTIONS [NO ]CACHE
OPTIONS [NO ]RESULTS
OPTIONS
```

Function: OPTIONS is the general instruction to set various default settings in the interpreter.

FAILAT Expression: Sets the limit after which return codes lead to a FAILURE report. The default setting is the FAILAT setting in the calling program (normally 10). "Expression" must return a positive integer value.

PROMPT Expression
Sets a character string to be used with PARSE PULL or PULL instructions as a user entry prompt. Normally, there is no prompt.

[NO]CACHE
Switches the internal instruction cache (in the interpreter) on or off. This switch, which increases function speed, is normally on.

[NO]RESULTS
Tells the interpreter whether or not it should request a result string when it executes an external instruction.

Use OPTIONS without any parameters to reset the default settings. The settings you make with OPTIONS are (such as NUMERIC settings) secured for the duration of function calls.

PROCEDURE

Syntax:

```
PROCEDURE [EXPOSE VSymbol [VSymbol ...]]
```

Function: Used within an internal function to define its variables (up to RETURN) as local variables. The function then has no access to the main program variables unless indicated with the (optional) key word EXPOSE. In the list that's placed after the EXPOSE variable stems or concatenated variables indicate variables that remain accessible. In this case, the order of exposure is important. For example, the variable Q has the value 45 in the main program. After "PROCEDURE expose Q RS.Q", the variables Q and RS.45 are still available to the function. If the instruction had been given as "PROCEDURE EXPOSE RS.Q Q", then RS.Q and Q would be exposed. Concatenated variables are evaluated from left to right.

SHELL

SHELL is an ARexx synonym for ADDRESS. For more information, see that section.

TRACE

Syntax: `TRACE [Symbol | String | [VALUE] Expression]`

Function: The TRACE instruction controls running ARexx programs and is most often used for error analysis.

Since you usually have to enter this instruction by hand, the syntax is kept short (the first letter suffices to name the key words). They are ALL, BACKGROUND, COMMANDS, ERRORS, INTERMEDIATES, LABELS, NORMAL, RESULTS, SCAN and OFF. If the result of the expression you enter does not display one of these sub-key words, the interpreter attempts to convert it into an integer value. If this is not possible, an error message occurs.

Two special characters precede the key words: "?" controls interactive tracing and "!" toggles execution of external commands.

Positive numeric entry forces a certain number of lines in the TRACE to elapse before the next display. Negative values indicate the number of lines to be skipped by the trace function. Negative values are only considered during interactive tracing.

See also: TRACE()

UPPER

Syntax: `UPPER VSymbol [VSymbol ...]`

Function: The content of the variables following UPPER are converted to capital letters. If a stem is specified, all variables with this stem are affected. Entering an undefined variable is not an error, instead it leads (if active) to a NOVALUE interrupt. Although you can use the built-in functions TRANSLATE() or UPPER(), the UPPER instruction is faster and easier, especially if several variables are to be converted.

See also: TRANSLATE(), UPPER()

6.4 Commands

The special quality of the Rexx language is that there is an entire class of syntactic units that are not evaluated by the interpreter. Instead, they are passed to an external environment. Each clause that contains an expression unknown to the interpreter is seen as a command meant for an external environment and passed on. These instructions are directed with ADDRESS. You can send a DOS (COMMAND) or an application program call using the ARexx interface. The expression is analyzed and passed on to the external environment as a character string. Then the external program executes your entry and passes back a return code that indicates whether or not the execution was successful.

The result can also be a character string. The advantage to this characteristic is that macro programs can easily be created to control and expand application programs. As indicated, a command is any expression that has no meaning for the interpreter. The command structure you type is entirely dependent on the external program for which it is intended. Often that is an alphanumeric name, followed by parameters. Commands can be written as strings or symbols. If you do not intend to pass the name as a variable parameter, it is safer to enter it as a literal (string), since then it will not be mistakenly read as an ARexx key word or redefined in a variable assignment. For example:

```
jumpto L+3 C
"select disk font" "ruby.font" 12
"end of file"
```

are all valid commands for "CygnusEd Professional 2". They can be executed by CygnusEd if "ADDRESS rexx_ced" indicates that the CygnusEd ARexx port is the target for the commands.

7. ARexx Functions

The basic idea of function definition is to indicate a program or a group of directions should always be carried out when the function name appears in an expression. In ARexx, a function can be part of a program (internal function), a built-in interpreter function, in an external function library, or a stand-alone program. The interpreter recognizes function calls when a symbol or string is followed directly by a left open parenthesis "(". The symbol or string indicates the function name and a list of arguments begins after the open parenthesis. There can be several expressions (that can contain functions) separated by commas to form arguments or none at all. The expression is analyzed from left to right and passed to the function. The following are valid function calls:

```
DIGITS()
"XRANGE"("A","Z")
showdir("dh0:fonts")
```

There is no limitation on the number of arguments passed to internal functions; the maximum arguments that can be passed to external functions is 15. The result is a string that's used in place of the function call. Functions are also retrieved with CALL (see Chapter 6). Use CALL when no result string is needed.

During a function ARexx searches for the function in a specific function order:

1. Internally: If the function name appears as a label in the script, the current state of interpretation is secured (including status information such as TRACE and NUMERIC settings). At the location where the function is found, execution continues. When it ends, with a RETURN clause, there must be a final argument. In other words, a function must return a value. If the function name was specified as a string, this step of the search is omitted.
2. Built-in: The built-in function library is searched for the given function. All names in the library are spelled with capital letters; they are described in the following sub-sections of this book.
3. External function libraries and environments: All available function libraries are stored in a list that simultaneously sets priorities for the search order. Each function library is searched with a separate offset

"QUERY" to check whether or not the given name is in the library. External function environments are called using a message protocol similar to a command syntax.

4. External ARexx programs: Finally, the interpreter tries to start an ARexx program with this name. The current directory is searched first, then the logical device "REXX:". Upper or lowercase spelling do not matter.

Internal and built-in functions do apply capitalization. For external functions it depends on the comparison algorithm being used in the QUERY routine. If you must use lowercase letters in a function name, the call to the function must be a written as a string, since symbols are always converted to uppercase and the lowercase distinctions are lost.

7.1 ARexx Internal Functions

During a call to an internal function, the interpreter creates a new storage environment for various internal status data. The following settings are saved:

- the NUMERIC setting
- the TRACE setting
- the SIGNAL setting
- the current and previous environment address
- the current prompt string, defined with the OPTIONS prompt

Although all previous variables remain accessible, this can be set to your needs with a procedure call. If a RETURN appears during the execution of the function, the function ends, all changes are discarded and the old settings are restored.

7.2 Built-in Functions

ARexx contains a sizable library of functions that contribute to the scope of the language. They have been optimized and should be preferred in most situations, over an interpreted function.

7.3 ARexx External Function Libraries

External function libraries can be used to expand the scope of the ARexx language. A function library contains one or several functions and a special "QUERY" access point with which you determine whether the function is in a library. Libraries have the same structure as normal Amiga libraries (with the exception of their significance for ARexx). Before an external function can be used, the corresponding library must be placed in the list of available function libraries in the Rexx master process, with the built-in function ADDLIB(...).

You can also set a search priority order in the function call; if priority ratings are equal, the order of mention determines the search order. During a search procedure, the Rexx master procedure opens and loads each listed library, unless the library has already been called. The query function is called with the desired function names as arguments. Normally, the entry point for this function is offset -30; other values can be set with ADDLIB(...). (CAUTION: false values here lead to a system crash.) If the function is not found, an error code is returned, the library is closed and the next library is searched. The offset of the function is returned if the search is successful. Then the function is accessed by the interpreter with the given arguments. It must return an error code (if successful, this is 0) and a result string.

The "rexxsupport.library" is included in ARexx and offers several Amiga-specific functions. There are also several Public Domain libraries, such as a math library, that makes more functions available.

External function environments are accessed by directing a function message to the appropriate Public Message Port. The program can do whatever is internally necessary with the function call; it must only answer the message at some time, sending back a return code and a result string.

The resident ARexx process itself is an example of a function environment; it's always available via its message port "REXX", to which program calls can be sent. It's located in the library list of the Rexx master tasks and it takes a priority of -60. If it receives a function call, it looks for a file with the appropriate name, the search path is the same as for Rexx sub-programs: the current directory, then the REXX: device. Each directory is searched first with the current extension (see PARSE SOURCE) and following that without the extension. By explicitly entering the search path within the function name, this process can be avoided. External programs are always started as separate processes. The calling program waits until its message is answered.

Built-in functions are internally set to DIGITS()=9 and FUZZ()=0 and are usually not influenced by settings in effect within the calling program. Lengths must be entered as positive integers (including 0) and positions cannot be 0.

Many functions process both necessary and optional arguments. Optional arguments are printed in square parentheses in the syntax descriptions following. If you leave these arguments out, a default setting is usually assumed.

If a function option can be selected with a single key word, usually the first letter will suffice. (Upper or lowercase characters don't matter.) If an empty string appears in that place, a default setting is used.

Some functions create and manipulate external DOS files. These files are called with a logical filename that was determined when the file was opened. This name is sensitive to upper and lowercase spelling. An unlimited number of files can be open simultaneously. Luckily, they don't all have to be individually closed; the interpreter handles the "housekeeping" at the end of each program.

I/O functions

```
CLOSE()
EOF()
EXISTS()
LINES()
OPEN()
READCH()
READLN()
SEEK()
WRITECH()
WRITELN()
```

String functions

```
ABBREV()
CENTER()
CENTRE()
COMPRESS()
COMPARE()
COPIES()
DATATYPE()
DELSTR()
DELWORD()
FIND()
INDEX()
INSERT()
LASTPOS()
```

LEFT()
LENGTH()
OVERLAY()
POS()
REVERSE()
RIGHT()
SPACE()
STRIP()
SUBSTR()
SUBWORD()
TRANSLATE()
TRIM()
UPPER()
VERIFY()
WORD()
WORDINDEX()
WORDLENGTH()
WORDS()
XRANGE()

Bit manipulation

BITAND()
BITCHG()
BITCLR()
BITCOMP()
BITOR()
BITSET()
BITTST()
BITXOR()

Numeric functions

ABS()
DIGITS()
FORM()
FUZZ()
MAX()
MIN()
RANDOM()
RANDU()
SIGN()
TRUNC()

Conversions

B2C()
C2B()
C2D()
C2X()
D2C()
D2X()
X2C()
X2D()

System functions

ADDLIB()
ADDRESS()
ARG()
DATE()
ERRORTEXT()
EXPORT()
FREESPACE()
GETCLIP()
GETSPACE()
HASH()
IMPORT()
PRAGMA()
REMLIB()
SETCLIP()
SHOW()
SOURCELINE()
STORAGE()
SYMBOL()
TIME()
TRACE()
VALUE()

7.4 I/O Functions

CLOSE()

Syntax: `CLOSE(name)`

Closes the file with the given logical filename. The function returns 1 if the file close was successful; otherwise (if the file was not open) a 0 is returned.

See also: OPEN()

Example: `SAY CLOSE("Datadump")        ==> 1`

EOF()

Syntax: `EOF(name)`

Returns a 1 if the end of the given logical file has been reached; otherwise a 0 is returned.

Example: `if EOF("Datadump") then CLOSE("Datadump")`

EXISTS()

Syntax: `EXISTS(DOSfilename)`

Determines if a file with the given name exists. If successful, the function returns a 1; otherwise a 0 is returned. Path names can precede the filename.

Example:

```
if EXISTS("dh0:Trashcan/LoadWB") then say,
"Look at the new Workbench 1.3, before you
empty the trash."
```

LINES()

Syntax: `LINES([name])`

Returns the number of lines listed in the entry buffer of the logical file "name" that must belong to an interactive device like CON: or SER:. If "name" is omitted, the number of lines "stdin" is returned.

Example:

```
say LINES("Pipeline")        ==> 3 (for example)
say LINES()                  ==> 1 (for example)
```

OPEN()

Syntax: `OPEN(name,DOS-filename[,"Append"|"Read"|"Write"])`

Opens a file for the given operation and gives it a logical filename ("name") that can later be called. "DOSfilename" is the name of the file to be opened and this can include device and directory names.

APPEND	Opens an existing file and sets the current position to the end of the file in order to add data.
READ	Opens an existing file and sets the current position to the beginning of the file.
WRITE	Opens a new file; if a file of the same name exists, it is erased.

To call these keywords, simply type the first letter. If nothing is entered, READ is assumed to be the function you are calling. When calling devices that do not support a "seek" function, such as CON: or SER:, the method of file access does not matter. The result of the function is Boolean. An unlimited number of files can be open simultaneously and they are all automatically closed when you leave the program.

See also: CLOSE(), READxx(), WRITExx(), SEEK()

Example:

```
Success = OPEN("Datastack","RAM:T/Testdata","W")
Success =  OPEN("Window","CON:200/100/200/100/RexxConsole")
```

READCH()

Syntax: `READCH(name,number)`

Reads the "number" of characters in the open logical file "name". This function returns the characters it reads as the result string, or fewer than requested if the end of file is reached. If you are reading from an interactive device like CON: or SER:, the function does not return anything until the necessary number of characters are in the buffer; execution halts until then. Reading from non-interactive devices is useless and leads to a false result.

Example: `data = READCH("Dataheap",5)`

READLN()

Syntax: `READLN(Name)`

Reads characters from the logical file "name" until a line feed (Hex 0A) or the end of the file is encountered. The line feed itself is removed and the entire line is returned as the result. If you are reading an interactive device like CON: or SER: the function does not return until a complete line is in the buffer; execution halts until then.

See also: READCH()

Example: `Entryline = READLN("Window")`

SEEK()

Syntax: `SEEK(name, offset[,"Begin"|"Current"|"End"])`

Sets the current position for calls to the open logical file "name". "Offset" determines the distance in characters from the current position. Whole numbers (including negative numbers) are allowed. By entering the keyword "Begin" the "offset" is set to the file beginning; "End" sets it to file end. You can overstep the limits of the file, but this is not recommended, since it can lead to some confusion and sometimes to errors. The result of SEEK is the current position in reference to the beginning of the open file. Using SEEK with interactive devices is senseless and has no effect.

Example:
```
say SEEK("Datahaystack",5,"B")     ==> 5
filelength = SEEK("Datahaystack",0,"E")
```

WRITECH()

Syntax: `WRITECH(name,string)`

Writes "string" to the logical file "name" and returns the number of characters written.

Example:
```
say WRITECH("Datahaystack","needle")    ==> 6
```

WRITELN()

Syntax: `WRITELN(name,string)`

Writes "string" to the logical file "name" and adds a line feed (Hex 0A). Returns the number of characters written, including the added line feed.

Example:
```
say WRITELN("Window","The rose is red.")     ==> 17
```

7.5 ARexx String Functions

ABBREV()

Syntax: `ABBREV(string1,string2[,length])`

Returns a 1 if "string2" is a permitted shorthand of "string1" and is not shorter than "length". The default for "length" is the length of "string2". An empty character string is a valid shorthand if nothing is specified in "length".

Example:

```
say ABBREV("Rosegarden","Rose")     ==> 1
say ABBREV("Rosegarden","R",4)      ==> 0
say ABBREV("Rosegarden","")         ==> 1
```

CENTER() or CENTRE()

Syntax: `CENTER(string,length[,pad])`
or: `CENTRE(string,length[,pad])`

Returns a character string of given length, in which the "string" is centered. Empty spaces to the left and right are replaced with spaces or pad (one character). If the "string" is too long, each side is cut. To avoid errors, both American and English spelling is permitted.

Example:

```
say CENTER("Hello",10)              ==> '   Hello  '
say CENTRE("0123456789",5)          ==> '23456'
say CENTER("TEST",10,">")           ==> '>>>TEST>>>'
```

COMPRESS()

Syntax: `COMPRESS(string[,list])`

If the second argument is omitted, this function removes all empty space from "string". In "list" one or several characters can be specified that are then removed instead of the spaces.

Example:

```
say COMPRESS(" Hey you! ")             ==> 'Heyyou!'
say COMPRESS("##AM++I#G+A++","#+")     ==> 'AMIGA'
```

COMPARE()

Syntax: `COMPARE(string1,string2[,pad])`

Returns the position of the first character of the two strings found not to be equivalent. If they agree, the result is 0. If necessary, a shorter string is filled with empty space to the right or an end of file marker, if that's found in the other string.

Example:
```
say COMPARE("Rose","Ross")          ==> 4
say COMPARE("abc","abc+-","+")      ==> 5
```

COPIES()

Syntax: `COPIES(string,number)`

Returns the "number" of repetitions of the "string". "Number" must be a whole number or zero.

Example:
```
say COPIES("Rose",3)        ==> 'RoseRoseRose'
say COPIES("Rose",0)        ==> ''
```

DATATYPE()

Syntax: `DATATYPE(string[,type])`

If only one parameter is specified, the function tests whether the argument is a valid ARexx number and returns "NUM". Otherwise, the result is "CHAR". If one of the following keywords is entered for "type", a test is executed and 1 is returned if "string" is that type; otherwise a 0 is returned. A null string only returns a 1 when tested for hexadecimal (X).

Available key words are:

Alphanumeric	A-Z, a-z and 0-9
Binary	valid binary string
Lowercase	a-z
Mixed	A-Z and a-z
Numeric	valid ARexx numbers
Symbol	valid ARexx symbols
Upper	A-Z
Whole	whole numbers
X	valid hexadecimal string

Example:

```
say DATATYPE("4711")                  ==> NUM
say DATATYPE("Rose","L")              ==> 0
say DATATYPE("52 6F 73 69"x,"X")      ==> 1
```

DELSTR()

Syntax: `DELSTR(string,n[,length])`

Returns the "string", after "length" characters from position "n" have been removed. If "length" is omitted, the rest of the character string is removed.

Example:

```
say DELSTR("The Rose is red",5,5)   ==> The is red
```

DELWORD()

Syntax: `DELWORD(string,n[,length])`

Returns the "string", after "length" words have been removed from and including word number "n". If "length" is omitted, the rest of the string is removed. Empty space in front of the first word that is not deleted remains.

Example:

```
say DELWORD("The Rose is red,3,1)   ==> The Rose red
```

FIND()

Syntax: `FIND(string,words)`

Searches for "words" in "string" and returns the word number of the first such agreement within "string". If "words" is not in "string", the function returns a 0.

Example:

```
say FIND("The Rose is red","Rose is")    ==> 2
```

INDEX()

Syntax: `INDEX(string,pattern[,start])`

Searches for the first appearance of "pattern" in "string" from the beginning of the string or from the optional position "start". The function returns either the position number or 0, if "pattern" does not appear.

Example:

```
say INDEX("The Rose is a Rose","Rose")          ==> 5
say INDEX("The Rose is a Rose","Rose",10)       ==> 15
say INDEX("The Rose is a Rose","Carnation")     ==> 0
```

Caution: This function is unique to ARexx and does not follow the typical order of arguments in Rexx syntax.

See also: LASTPOS() is similar to POS(), except it has reversed arguments.

INSERT()

Syntax: `INSERT(source,destin[,[start][,[length][,pad]]])`

Adds "source" after the "start" position to the "destin" string. "Source" is expanded with the character "pad" to the given "length". The default value for "start" is 0, for "length" the length of the "source", and the "pad" default is a space.

Example:

```
say INSERT("123","abcde")              ==> 123abcde
say INSERT("123","abcde",6,5,".")      ==> abcde.123..
```

LASTPOS()

Syntax: `LASTPOS(pattern,string[,start])`

Searches "string" backward for the first appearance of "pattern" and returns the equivalent index (or 0, if no agreement occurs). Normally, the search begins at the last character. If you want the process to start somewhere else, "start" indicates a position counted from the beginning.

Example:

```
say LASTPOS("Rose","The Rose is a Rose")        ==> 15
say LASTPOS("Rose","The Rose is a Rose",15)     ==> 5
say LASTPOS("Carnation","The Rose is a Rose")   ==> 0
```

LEFT()

Syntax: `LEFT(string,length[,pad])`

Returns a character string of the indicated "length", taken from the left side of the argument "string". If necessary, "string" is cut off at the right end or lengthened with "pad". Default for "pad" is a space character.

Example:

```
say LEFT("The Rose is red",8)   ==> The Rose
say LEFT("The Rose",10,":")     ==> The Rose::
```

LENGTH()

Syntax: `LENGTH(string)`

Returns the length of "string".

Example:

```
say LENGTH("The Rose")        ==> 8
say LENGTH("")                ==> 0
```

OVERLAY()

Syntax: `OVERLAY(new,old[,[start][,[length][,pad]]])`

Overlays the character string "old" with "new", beginning at the position "start". During the operation, "new" is cut to "length" or lengthened with the "pad" character. The default value of "start" is 1, if the value is greater than the length of "old", the extra space is filled with "pad". The default setting for "length" is the length of "new". The standard pad character is a space.

Example:

```
say OVERLAY("xx","The Rose")          ==> xxe Rose
say OVERLAY("Rose","The",7,6,"*")     ==> The***Rose**
```

POS()

Syntax: `POS(pattern,string[,start])`

Searches for the first appearance of "pattern" in "string" from the optional position "start". If no "start" is specified, it searches from the beginning of the file. It returns either the position at which the pattern is found or 0, if "pattern" does not occur at all.

Example:

```
say POS("Rose","The Rose is a Rose")         ==> 5
say POS("Rose","The Rose is a Rose",10)      ==> 15
say POS("Carnation","The Rose is a Rose")    ==> 0
```

REVERSE()

Syntax: `REVERSE(string)`

Reverses the order of characters in "string".

Example:

```
say REVERSE("esoR")             ==> Rose
```

RIGHT()

Syntax: `RIGHT(string,length[,pad])`

Returns a character string of "length" containing "string", starting from the right. "String" is cut off at the left side if necessary, or lengthened with the "pad" character. The default character for "pad" is a space.

Example:

```
say RIGHT("The Rose is red",3) ==> red
say RIGHT("The Rose",10,":")   ==> ::The Rose
```

SPACE()

Syntax: `SPACE(string,n[,pad])`

If "string" contains words separated by spaces, SPACE returns a character string with "n" spaces between the words. Empty spaces on the left and right are removed. The "pad" character can define another character to use instead of the space character.

Example:

```
say SPACE("The   Rose is red",1)          ==> The Rose is red
say SPACE("  The Rose is red",2)          ==> The  Rose  is  red
say SPACE(" The Rose is   red",1,"|")     ==> The|Rose|is|red
```

Caution: This function does not work correctly if the second argument is omitted. The default value for "n" is 0 (it should be 1). Omitting the second argument of this feature has not been documented; it cannot be recommended. Eventually this error will be corrected.

STRIP()

Syntax: `STRIP(string[,[{"B"|"L"|"T"}][,character]])`

If an argument is given, the function removes preceding and trailing spaces from "string". If "L" (for "leading") or "T" (for trailing) is indicated, only one or the other is removed. The third argument is used to specify the character to be removed.

Example:

```
say STRIP(" The Rose ")              ==> 'The Rose'
say STRIP(" The Rose ","T")          ==> ' The Rose'
say STRIP("--The-Rose--",,"-")       ==> 'The-Rose'
```

SUBSTR()

Syntax: `SUBSTR(string,start[,[length][,pad] ])`

Returns a sub-string of "string", from the position "start", for "length" and filled at the right side with the character "pad". Default for "length" is the remaining length of "string", the default pad character is a space.

Example:

```
say SUBSTR("abcde",3)          ==> cde
say SUBSTR("12345",3,2)        ==> 34
say SUBSTR("abcde",3,5,"#")    ==> cde##
```

SUBWORD()

Syntax: `SUBWORD(string,start[,length])`

Returns a sub-string of "string", starting with the word at "start" and containing the number of words set in "length". The default setting is the remainder of "string". The result contains no leading or trailing spaces, only the space between the selected words is preserved.

Example:

```
say SUBWORD("The Rose is red",3)   => is red
say SUBWORD("The Rose is red",2,2) => Rose is
```

TRANSLATE()

Syntax: `TRANSLATE(string[,[output][,[input][,pad]]])`

Replaces the characters in one string with the characters in the other and returns the new character string. TRANSLATE() has the same effect as UPPER() with a single argument. Default for "input" is a string with all characters from "00"x to "FF"x. Every character that occurs in "input" is replaced with the corresponding character in "output". If there is no such character in "output" (if "output" is shorter than "input"), an empty space or the "pad" character is returned. Characters that do not occur in "input" remain the same; the length of the "string" does not change. The tables can be as long as you want, but longer than 256 characters hardly makes sense, since within "input" only the first appearance of a character is noted. The final example shows the use of TRANSLATE() to rearrange a character string in any order. "String" determines the order and the second argument gives the specific working character string.

Example:

```
say TRANSLATE("The Rose")                 ==> THE ROSE
say TRANSLATE("xyz","wvu"","zyx")         ==> uvw
say TRANSLATE("12345","ab","123","-")     ==> ab-45
```

```
say TRANSLATE("312","abc","123")        ==> cab
```

TRIM()

Syntax: `TRIM(string)`

Removes trailing spaces from "string". Equivalent to the STRIP(string,"T") function.

Example:

```
say TRIM(" Rose ")            ==> " Rose"
```

UPPER()

Syntax: `UPPER(string)`

Converts "string" to capital letters. Equivalent to the TRANSLATE(string) function but a little bit faster with short character strings.

Example:

```
say UPPER("Rose")             ==> ROSE
```

VERIFY()

Syntax: `VERIFY(string,table[,[{M|N}][,start]])`

Checks if "string" only contains characters in "table". If so a 0 is returned; otherwise the position of the first character that does not appear in "table" is returned. The third argument can be "match" (default is "nomatch") to reverse the logic of the verification. The VERIFY() function in "match" mode returns the position of the first character that is contained in "table". Normally the search begins at the first character, but "start" can be used to define another entry point. If "string" is empty, or "start" is greater than the length of "string", the function always returns 0, regardless of the third argument.

Example:

```
say VERIFY("427","0123456789")          ==> 0
say VERIFY("4p7q1","0123456789")        ==> 2
say VERIFY("xx731","0123456789","M")    ==> 3
say VERIFY("4p7q1","0123456789",,3)     ==> 4
```

WORD()

Syntax: `WORD(string,n)`

Returns the "n"-th word in "string", or an empty string if "string" does not contain sufficient words. Equivalent to the SUBWORD(string,n,1) function.

Example: `say WORD("The Rose is red",2)    ==> Rose`

WORDINDEX()

Syntax: `WORDINDEX(string,n)`

Returns the position of the first character of the "n"-th word in the "string", or 0 if there are insufficient words.

Example: `say WORDINDEX("The Rose is red",2)    ==> 5`

WORDLENGTH()

Syntax: `WORDLENGTH(string,n)`

Returns the length of the "n"-th word in "string", or 0 if there are insufficient words in "string".

Example: `say WORDLENGTH("The Rose is red",2)    ==> 4`

WORDS()

Syntax: `WORDS(string)`

Returns the number of words in "string".

Example: `say WORDS("The Rose is red")    ==> 4`

XRANGE()

Syntax: `XRANGE([start][,end])`

Returns a character string containing all characters with ASCII codes ranging from the "start" to the "end" character. Default for "start" is "00"x and for "end" it is "FF"x. The order is always from high to low; if "start" is higher than "end", the order begins again after "FF"x at "00"x and continues until the "end" value.

Example:

```
say C2X(XRANGE())        ==> 000102 ... FDFEFF
say XRANGE("A","F")      ==> ABCDEF
say C2X(XRANGE(,"05"x)) ==> 000102030405
```

7.6 Bit Manipulation in ARexx

BITAND()

Syntax: `BITAND(string1[,[string2][,pad]])`

A logical AND function is performed with the two strings. The result has the length of the longer operand. Instead of breaking off the operation at the end of the shorter operand and appending the rest of the longer operand unchanged, the shorter operand is filled up to the right with "20"x (the space character) and the concatenation AND performed on the entire length of the string. The behavior described in the documentation can only be guaranteed if "pad" is always specified as "FF"x. The shorter operand is then filled with this value before the operation begins. If the second operand is omitted, "20"x is always filled in or the end of file marker is added.

Example:

```
say C2B(BITAND("00001111"b,"01010101"b))==> 00000101
say C2x(BITAND("FF"x,"FFFF"x)          ==> FF20
say C2x(BITAND("00"x,"AAAA"x,"FF"x)    ==> 00AA
say BITAND("Rose",,"11011111"b)        ==> ROSE
```

BITCHG()

Syntax: `BITCHG(string,bit)`

Inverts the given bit in "string". Bit 0 is the lowest value bit of the characters on the right side of "string".

Example:

```
say C2B(BITCHG("00001111"b,5))    ==> 00101111
```

BITCLR()

Syntax: `BITCLR(string,bit)`

Deletes the given bit in "string". Bit 0 is the lowest value bit on the right side of "string".

Example:

```
say C2B(BITCLR("00001111"b,2))    ==> 00001101
```

BITCOMP()

Syntax: `BITCOMP(string1,string2[,pad])`

Compares the bit patterns of the two strings, from bit number 0 going from right to left. The result is the number of first bits in which the strings are different, or -1 if they are equal. The shorter string is filled, before the operation at the left side with the "pad" character (the default "pad" is a space).

Example:

```
say BITCOMP("FF","FFFF"x)      ==> 8
say BITCOMP("FF","20FF"x)      ==> -1
```

BITOR()

Syntax: `BITOR(string1[,[string2][,pad]])`

A logical OR operation is performed on the two strings. The result is the length of the longer operand. Instead of breaking off the operation at the end of the shorter operand and adding the rest of the longer operand unchanged, the shorter operand is filled with "20"x (the space character) and the OR connection is carried out over the entire length of the string. The behavior that is described in the documentation is only possible if the "pad" is always specified as "00"x. The shorter operand is then filled up with this value before the logical operation takes place. If the second operand is omitted, it's filled with "20"x or the end of file marker.

Example:

```
say C2B(BITOR("00001111"b,"01010101"b))  ==> 01011111
say C2x(BITOR("FF"x,"0000"x)             ==> FF20
say C2x(BITOR("00"x,"AAAA"x,"FF"x)       ==> AAFF
say BITOR("Rose",,"00100000"b)           ==> rose
```

BITSET()

Syntax: `BITSET(string,bit)`

Sets a marker for the given bit in "string". Bit 0 is the lowest value bit of the characters from the right end of the "string".

Example: `say C2B(BITSET("00001111"b,5))    ==> 00101111`

BITTST()

Syntax: `BITTST(string,bit)`

Returns the given bit of "string". Bit 0 is the lowest value bit of the characters from the right end of the "string".

Example: `say BITTST("00001111"b,5)    ==> 0`

BITXOR()

Syntax: `BITXOR(string1[,[string2][,pad]])`

Performs a logical exclusive OR operation on the two strings. The result is the length of the longer operand. Instead of breaking off the operation at the end of the shorter operand and adding the rest of the longer operand unchanged, the shorter operand is filled with "20"x (the space character) on the right side, and the XOR operation is performed on the entire length. The documented behavior can only be achieved if the "pad" is always "00"x. The shorter operand is then filled with this value before the operation starts. If the second operand is omitted, it is always filled with "20"x or the end of file marker.

Example:

```
say C2B(BITXOR("00001111"b,"01010101"b))==> 01011010
say C2x(BITXOR("FF"x,"0000"x)          ==> FF20
say C2x(BITXOR("FF"x,"0000"x,"00"x)    ==> FF00
say BITXOR("Rose",,"00100000"b)        ==> rOSE
```

7.7 Numeric Functions

ABS()

Syntax: `ABS(number)`

Returns the absolute value of "number".

Example:
```
say ABS(-345)          ==> 345
say ABS(4.32)          ==> 4.32
```

DIGITS()

Syntax: `DIGITS()`

Returns the current NUMERIC DIGITS setting.

Example:
```
say DIGITS()           ==> 9
```

FORM()

Syntax: `FORM()`

Returns the current NUMERIC FORM setting.

Example:
```
say FORM()             ==> SCIENTIFIC
```

FUZZ()

Syntax: `FUZZ()`

Returns the current NUMERIC FUZZ setting.

Example:
```
say FUZZ()             ==> 0
```

MAX()

Syntax: `MAX(number,number[,number]...)`

Returns the largest of the given numbers.

Example:
```
say MAX(3,2,7,5)             ==> 7
say MAX(-3,-1,-8,-1)         ==> -1
```

MIN()

Syntax: `MIN(number,number[,number]...)`

Returns the smallest of the given numbers.

Example:
```
say MIN(3,2,7,5)             ==> 2
say MIN(-3,-1,-8,-1)         ==> -8
```

RANDOM()

Syntax: `RANDOM([min][,[max][,[startvalue]]])`

Returns pseudo-random integer values between "min" and "max". Default values are 0 and 999. The interval between "min" and "max" cannot be larger than 1000. The third value can be a different start value, in order to achieve a repetitive sequence. This start value should be specified at the first call and can be any number. The results of later calls to RANDOM() (without a start value) are repeated, if the random number generator is initialized to the same start value again. If no start value is specified, the random number generator is initialized with the system time at the first call. The start value is not secured for all routine calls, but rather globally, for an entire program.

Example:
```
say RANDOM(1,49)             ==> 17 ?
say RANDOM(,,4711)           ==> 365 ?
```

RANDU()

Syntax: `RANDU([startvalue])`

Returns evenly distributed pseudo-random numbers between 0 and 1. The number of places after the decimal point depends on the current NUMERIC DIGITS setting. Normally, the random number generator is initialized with the system time at the first call. By entering the optional "start value", the random number generator can be moved to a defined starting condition, in order to achieve repetitive pseudo-random sequences.

Example:

```
say RANDU()                ==> 0.018327461 ?
```

SIGN()

Syntax: `SIGN(number)`

Resembles the mathematic "sign" function. If "number" is negative, SIGN returns a -1, if the number is a 0, it returns a 0, and if "number" is positive, it returns a 1.

Example:

```
say SIGN(0.1)              ==> 1
say SIGN(0.0)              ==> 0
say SIGN(-5)               ==> -1
```

Caution: The SIGN() function should round the number according to the evaluation of the NUMERIC DIGITS setting. This is not implemented, so insignificant fractions are never reported as "0".

TRUNC()

Syntax: `TRUNC(number[,places])`

Returns the whole number portion of "number", followed by the desired number of places after the decimal, which is usually none. It does not round to the whole number. If needed, the number is filled with zeros. The result is never an exponential notation, so that "number" cannot require more places than are set in NUMERIC DIGITS. If necessary, the number is rounded according to the number of decimal places first.

Example:

```
say TRUNC(564.73294)           ==> 564
say TRUNC(564.73294,3)         ==> 564.732
say TRUNC(564.7,3)             ==> 564.700
```

7.8 Conversion Functions in ARexx

B2C()

Syntax: `B2C(string)`

Converts a string of binary symbols (0 and 1's) to the corresponding ASCII character string. Empty spaces are allowed in "string" but only at the byte limits, every 8th digit.

Example:

```
say B2C("01000001")          ==> A
```

C2B()

Syntax: `C2B(string)`

Converts an ASCII symbol string to an equivalent binary string.

Example:

```
say C2B("Rose")==>01010010011011110111001101100101
```

C2D()

Syntax: `C2D(string[,n])`

Converts "string" from a symbolic representation to the corresponding decimal number. The maximum "string" length is 4 bytes (32 bits). If "n" is given, the binary value of "string" is treated as a pair of length "n" bytes and transformed into a corresponding whole number (with a prefix if necessary). The "string" is cut off at the left side or filled with zeros if it's not the right length. No prefix evaluation takes place.

Example:

```
say C2D("0A"x)          ==> 10
say C2D("Rose")         ==> 1383035749
say C2D("FFFF"x,2)      ==> -1
```

C2X()

Syntax: `C2X(string)`

Converts "string" from symbolic representation to the corresponding hexadecimal number. The result contains capital letters for the numbers A-F and no empty spaces.

Example:

```
say C2X("Rose")          ==> 526F7365
say C2X("0A"x)           ==> 0A
```

D2C()

Syntax: `D2C(number1[,bytes])`

Converts decimal numbers into equivalent ASCII characters. If "bytes" is specified, the result takes that length; it's cut off at the left side or filled with "00"x on the right if necessary. Negative values, not otherwise permitted, can be expressed as a pair.

Example:

```
say D2C(65)              ==> A
say C2X(D2C(-1,4))       ==> FFFFFFFF
```

D2X()

Syntax: `D2X(number1[,nibbles])`

Converts whole decimal numbers to the corresponding hexadecimal notation. If "nibbles" is specified, negative numbers are converted into a number pair. The result has the corresponding number of places and, if necessary, is cut off at the left side or filled in with 0's at the right. For the numbers A-F it uses capital letters and no empty space is added.

Example:

```
say D2X(10)              ==> A
say D2X(10,2)            ==> 0A
say D2X(-1,5)            ==> FFFFF
```

X2C()

Syntax: `X2C(Xstring)`

Converts a string from hexadecimal notation to equivalent ASCII string. If necessary, a 0 is added to the left, in order to arrive at an even number of nibbles. At the byte limits, empty spaces can be added to improve readability. They are ignored by the program.

Example:

```
say X2C("4D 4E")                ==> MN
```

X2D()

Syntax: `X2D(Xstring[,nibbles])`

Converts a string from hexadecimal notation to the corresponding decimal number. If necessary, a single 0 is added on the left side, in order to arrive at an even number of nibbles. Empty spaces can be added at the byte limits to improve readability. They are ignored by the program. A maximum of 4 bytes (8 nibbles) are allowed. The NUMERIC DIGITS setting has no influence on this function.

Normally, X2D() returns positive numbers. If any value is entered for "nibbles", "Xstring" is assumed to be a pair and numbers with prefix signs are returned. If the number of nibbles in "Xstring" is not correct, it is simply filled with "0"x to the left or cut off, so that no prefix expansion takes place.

Example:

```
say X2D("0D")           ==> 13
say X2D("FFFF")         ==> 65535
say X2D("FFFF",4)       ==> -1
say X2D("FFFF",6)       ==> 65535
```

7.9 ARexx System Functions

ADDLIB()

Syntax: `ADDLIB(name,priority[,offset,version])`

Adds a function library or an external function environment to the library list that is managed by the Rexx Master process. "Name" is either the full name of a function library that is located on the logical device LIBS:, or the name of a Public Message Port that belongs to a function environment. "Priority" determines the search order for called functions and must be an integer between -100 and 100. Usually 0 is useful.

The arguments "offset" and "version" refer only to libraries and are necessary to open one. "Offset" indicates the entry point for the query function of the library (usually -30) and "version" takes a certain version number the library must minimally achieve (usually 0).

The function returns a Boolean result if everything is in order. This does not mean that the library is available and the program does not try to load it until the first command occurs. An equivalent Message Port is also not located until later.

Example:

```
if ADDLIB("rexxsupport.library",0,-30,0) then
say "OK!"
```

ADDRESS()

Syntax: `ADDRESS()`

Returns the name of the message port to which external commands can be sent. The function SHOW() can test if the port is available.

Example:

```
say ADDRESS()                ==> rexx
```

ARG()

Syntax: `ARG([number[,{"E"|"O"}]])`

Without arguments, ARG() returns the number of arguments that were passed to a program or a sub-routine. If a "number" is entered, the argument string is returned or, if that is not available, a null string.

If one of the options for "Exists" or "Omitted" is left out, the argument is tested for the other and a Boolean result is returned.

Example:

```
/* Arguments given: ("Rose",,-5) */
say ARG()                           ==> 3
say ARG(3)                          ==> -5
say ARG(2,"E")                      ==> 0
```

DATE()

Syntax: `DATE(option[,date[,{I|S}]])`

Returns the current system date in the desired form. (A "normal" format is used if the function is called without an argument.) Supported options are:

Base date:	Days since January 1, 0001
Century:	Days since the beginning of the century
Days:	Days since the beginning of the year
European:	Date in the form DD/MM/YY
Internal:	System days (since January 1, 1978)
Julian:	Date in the form YYDDD
Month:	Month in English (upper and lowercase letters)
Normal:	Date in the form DD MMM YYYY
Ordered:	Date in the form YY/MM/DD
Sorted:	Date in the form YYYYMMDD
USA:	Date in the form MM/DD/YY
Weekday:	The weekday in English (upper and lowercase)

A specific date can be requested. To do this, the argument "date" is given as system days or as a "sorted date" in the form YYYYMMDD; in the latter case a third argument "S" (for "sorted") must be supplied.

Example:

```
say DATE()                    ==> 22 Jan 1991
say DATE("W")                 ==> Tuesday
say DATE("W",DATE("I")+3)     ==> Friday
say DATE("J",19800517,"S")
```

Caution: In V1.14 no date before system date 0, January 1 1978 can be entered in this manner.

ERRORTEXT()

Syntax: `ERRORTEXT(number)`

Returns an error message for the given ARexx error number. If "number" is not a valid error number, the message "Undiagnosed internal error" is returned. Unfortunately, ARexx doesn't maintain the Rexx standard for error messages, but uses its own numbers.

Example:

```
say ERRORTEXT(15)    ==> Function not found
```

EXPORT()

Syntax: `EXPORT(address[,[string][,[length][,padpattern]]])`

Copies the given data from "string" to the 4-byte "address" in the storage space that must have previously been reserved with GETSPACE(). "Length" determines the maximum number of characters to be copied, "padpattern" (one byte) is used to fill up the string if it isn't long enough. The default value is "00"x. You can use this function to enter an address and length in order to delete from the storage area, or to initialize with "padpattern". The returned value is the number of characters actually copied.

Caution: This function can be used to overwrite any storage areas, which can lead to fatal error. Never use EXPORT() with a reserved stack unless you know exactly what you are doing. Secure your program scripts against the common error of overstepping reserved space. Also, during copy operations, task-switching is interrupted. With large amounts of data (if possible) copy several sub-strings, so multitasking operations aren't interrupted for too long.

Example:

```
say EXPORT("0024 DDB0"x,"The Rose is red")  ==> 15
say EXPORT("0006 0000"x,,640,"FF"x)          ==> 640
```

FREESPACE()

Syntax: `FREESPACE([address,length])`

Returns the storage area of the Rexx master procedure. If you specify the 4-byte address with which the block was designated using GETSPACE() earlier, its length (a multiple of 16) is returned. The function FREESPACE() with false entries (and sometimes, in V1.14, even correct ones) quickly allows the computer to get caught on the problem or run through endless loops. The returned value is not a Boolean result, as the documentation states, instead it's the size of the free space under the control of the Rexx master procedure (and that result often contains errors). A call without arguments returns the true size of the storage space being managed by the Rexx master procedure. Since the storage area is automatically returned after the program ends, calling FREESPACE() is only necessary when you may run out of space.

Example:

```
say FREESPACE("0002fa44"x,32)     ==> 848 ?
```

GETCLIP()

Syntax: `GETCLIP(name)`

Searches the Clip list for "name" and returns the corresponding character string. Upper and lowercase spelling are differentiated. If there is no entry, an empty string is returned.

See also: SETCLIP()

Example:

```
/* "DaData" contains "The Rose is flighty" */
say GETCLIP("DaData") ==> The Rose is flighty
```

GETSPACE()

Syntax: `GETSPACE(length)`

Reserves a stack of "length", managed by the Rexx master procedure. It returns a 4-byte address, indicating the beginning of the reserved storage area, which is not deleted. "Length" is rounded up to the next multiple of 16.

Stacks reserved with GETSPACE() are automatically returned to the Rexx master procedure at the end of the program, so external programs should not access this storage area. In the "rexxsupport.library", a function called ALLOCMEM() requests storage space directly from the system; it can be necessary in such cases.

Example:

```
say C2X(GETSPACE(64))          ==> 002937F8 ?
```

HASH()

Syntax: `HASH(string)`

Returns the hash value of "string" as a decimal number. The hash value is the lowest byte of the sum of all ASCII values contained in the string.

Example:

```
say HASH("A")                  ==> 65
say HASH("AAAA")               ==> 4
```

IMPORT()

Syntax: `IMPORT(address[,length])`

Reads data from the given 4-byte storage address. If no length is specified, the process ends at the first "00"x, which is practical for reading C strings.

Example:

```
say IMPORT("00FC0038"x,9)      ==> Amiga ROM
```

PRAGMA()

Syntax: `PRAGMA(option[,value])`

Various system-specific parameters of your own program can be determined. The options are:

Directory: A new current directory can be set for the running procedure. The function returns the full path name of the previously current directory; it can be saved in order to restore the old settings later. "Value" must be a valid Amiga DOS path name or be omitted. In the latter case, only the current setting is returned. If the path is not valid or not given, a null string is returned.

Id: Returns the 4-byte pointer to the Task Control Block structure of the current program as an 8-byte hexadecimal string. Using this address, you can create independent file or port names specific to the appropriate program call.

Priority: A new task priority can be given to the procedure with this option. The function then returns the previous priority setting. Its "value" must be a whole number between -128 and 127; no ARexx program should run with a higher priority than the ARexx main program, which is usually set at 4. "Value" must always be specified, which means that a priority cannot be queried without possibly changing it. If no area check is taking place, the lowest byte of the given number is used.

Window: This option changes the window pointer of the task control block in the running program. For "value", valid keywords are "Work Bench" and "Null". By entering "null", you can prevent requests from being sent the Workbench by DOS calls (such as Insert Volume ... etc.). At this point, only "null" is recognized; all others (including an omitted second argument) lead to the default setting "WorkBench". The function also always returns a 1 to indicate successful completion.

"*": Defines the given logical name "value" as the current ("*") console handler. This means you can open two data strings in one window. The result is a Boolean result.

Example:

```
say PRAGMA("P",-2)            ==> 0
say PRAGMA("D")               ==> Boot_2.X: ?
say PRAGMA("D","df0:c)        ==> ARexx1.14: ?
say PRAGMA("I")               ==> 0028FE08 ?
say PRAGMA("W","Null")        ==> 1
say PRAGMA("*","STDIN")       ==> 1
```

REMLIB()

Syntax: `REMLIB(name)`

Removes an entry with the given name from the library list managed by the ARexx master procedure. The function returns a 1 if the name is found and removed; otherwise it returns a 0. It does not differentiate between libraries and external function environments.

See also: ADDLIB()

Example:

```
REMLIB("rexxsupport.library")     ==> 1
```

SETCLIP()

Syntax: `SETCLIP(name[,value])`

Adds a "value" (any string) "name" to the Clip list being managed by the ARexx master procedure. If an entry already exists under that name, the contents are updated to the new value or, if no "value" is given, the entire entry is deleted. The result is a Boolean result.

Example:

```
say SETCLIP("Text1","No, no roses")     ==> 1
say SETCLIP("Text1")                    ==> 1
```

SHOW()

Syntax: `SHOW(option[,[name][,divider]])`

Returns the contents of various lists being managed or used by the ARexx master procedure. "Option" refers to one of the following key words:

Clip: Names in the Clip list.

Files: List of open logical filenames.

Internal: Internal port list.

Libraries: External library and function environment list.

Ports: List of Public Message Ports, managed by EXEC. An unnamed port is indicated by a question mark.

If no "name" is specified, the function returns a string with entries in the given list, separated by a space or the optional "divider". If "name" is specified, the corresponding list is searched for the entry and a Boolean result shows if it was found.

Example:

```
say SHOW("P",,";")        ==>REXX;DMouse;Workbench
say SHOW("C","Text1")     ==> 1
```

SOURCELINE()

Syntax: `SOURCELINE([line])`

Returns a string representing the given line of the current program. If "line" is omitted, the number of lines in the program is returned. The function can be used to display comment line: used as a help feature.

Example:

```
say SOURCELINE()    ==> 35 ?
say SOURCELINE(1)   ==> /* A test program */ ?
```

STORAGE()

Syntax: `STORAGE([address][,[string][,length[,pad]]])`

Writes "string", starting at the given address, directly to the main storage area. If "length" is specified, the actual length of the string is disregarded and only that number of bytes written; in this case the "string" is either shortened on the right or padded with empty space (or the given "pad"). The result string is the previous contents of the affected stack that can be saved and restored later.

If the function is entered without arguments, it returns the total available storage space.

Example:

```
say STORAGE()                ==>1846536 ?
before = STORAGE("00040000"x,after)
```

SYMBOL()

Syntax: `SYMBOL(name)`

Checks if the argument is a valid ARexx symbol. If not, it returns the string "BAD". If it is a valid but un-initialized symbol, the result is "LIT", and if the symbol has already been assigned a value, the answer is "VAR".

Example:

```
say SYMBOL("$%&")              ==> BAD
say SYMBOL("before")           ==> VAR
say SYMBOL("when")             ==> LIT
```

TIME()

Syntax: `TIME([option])`

Without an optional keyword, TIME returns the current system time in 24-hour format, in the form "hh:mm:ss". Possible options are:

Civil: American 12-hour format in the form"[h]h:mmxx", where "xx" is either "am" or "pm". The hour does not receive a leading zero, and the minute is the current minute, not (as is usually the case) the next minute.

Elapsed: The number of seconds and hundredths of a second that have passed since an initial call to the internal timer with "Elapsed" or "Reset".

Hours: The number of hours since midnight without a leading zero.

Minutes: The number of minutes since midnight without a leading zero.

Normal: Returns the default setting (the same result as calling the function without an argument).

Reset: Returns the number of seconds and hundredths of a second since an initial query to the internal timer using "Elapsed" or the last "Reset", and simultaneously resets the timer.

Seconds: The number of seconds since midnight without leading zeros.

Example:

```
say TIME()              ==> 18:35:22 ?
say TIME("R")           ==> 0 ?
say TIME("E")           ==> 2.12 ?
```

TRACE()

Syntax: `TRACE(option)`

With no argument, this function returns the current TRACE setting. All valid TRACE keywords can be specified as options (numbers are not allowed, but "?" and "!" are). The TRACE() function changes the TRACE mode, even during interactive tracings, when all other TRACE commands are ignored. The result is always the last setting that can thereby be saved and restored later.

Example:

```
say TRACE()             ==> N
```

VALUE()

Syntax: `VALUE(name)`

Returns the contents of the given ARexx symbol, which must be a valid symbol. This function is used when the variable name itself is a variable, as a whole or partially.

Example:

```
/* Situation: DROP q5, 155=8; n=5; Rose="n"*/
say VALUE("Rose")                       ==> n
say VALUE(Rose)                         ==> 5
say VALUE("q"n)                         ==> Q5
say VALUE("1"n||n)                      ==> 8
```

8. Special Features

Rexx contains several powerful special features that may be unfamiliar to users of other programming languages. The most important ones, parsing data and tracing programs, are discussed here.

8.1 Parsing Strings with Templates

The ARexx instruction PARSE (and its two abbreviations ARG and PULL) split an entry according to a "template" and direct the results to variables. This feature is especially useful when you are using ARexx as a script language on the Amiga, since many commands that were not conceived for automatic processing deliver cryptic return values that do not conform to any formatting standards. The CLI script language offers some help in parsing argument lines (with ".") and some command line syntheses (using "CLIs LFORMAT"), but both of them fail difficult parsing tasks.

The previous description of PARSE is a short explanation of its most important capabilities. The following is a complete process description:

A template consists of two elements, symbols which are assigned values during the operation, and markers to indicate a position within the source string. Valid markers are: strings, operators such as "+","-" and "=", closed parentheses, and commas. Using the template, a beginning and end position is determined within the source string for every target symbol. The corresponding portion of the string is then assigned to the symbol. There are three types of markers: "absolute", which indicate an exact position in the source string, "relative", which indicate a positive or negative offset from the present position, and "pattern" which indicates a position by comparing the given pattern to the source string. In a template, the target of the sub-string is a variable symbol or a specific goal (or a period); the corresponding value is not assigned to the target. Variables in a template always receive a new value, even if the source data do not contain enough words. Any remaining variables are set to 0.

Valid template elements

symbols: A symbol may be a target or a marker. If it immediately follows one of the valid operators ("+","-" or "="), its value (which in this case must be an integer) is interpreted as a relative or an absolute position. If a symbol appears in parentheses, its value is a comparison pattern. If neither condition is true, it must be a variable, to which a value is assigned.

strings: A string is always a comparison pattern.

parentheses: If a symbol appears in parentheses, it is a comparison pattern. Normally, a variable symbol is used; a constant value is easier to display within a string.

operators: The characters "+","-" and "=", followed by a symbol (which must represent an integer), indicate index positions in the source string. "+" and "-" indicate relative positions, "=" indicates an absolute position.

commas: A comma separates multiple templates. If several templates follow one another, the interpreter looks for a new source string. In some source options, it's identical to the last. With the options ARG, EXTERNAL and PULL, a new string is created; the same is true for the option VAR, if the contents of the variables has changed.

periods: A period serves as a dummy value and operates as a target for a sub-string which is to be discarded.

Each character in the source string has an index number, from 1, for the first character, to the length of the string plus 1 (the end of the string). If the limit is exceeded, the current position is set at the limit. A sub-string, defined by two indices, always contains the character of the first index and continues up to the second. The indices 3 and 8 would define a sub-string of the characters 3 to 7. If both indices are equal, or the second is smaller, the remainder of the source string is defined by the pair. The command:

```
PARSE value "bla bla bla" WITH 1 all 1 Word1 Word2 Word3
```

assigns the entire string to the variable "all", after which each word is parsed into equivalent variables. When a pattern is compared to the source string, the position of the first character matching the pattern is the new index and the pattern is removed from the source string. This means that the source string is altered in the process of this operation.

The evaluation goes from left to right in the template. At the beginning, the source string index is set to 1. Whenever a marker appears in the template, its position becomes the current one. Whenever a target is found, the program searches for the next object in order to determine the length of the sub-string the target expects. If the next object is a target, the source string is divided into words. The process does not end until the template has been completely evaluated. If the source string is fully parsed, remaining targets receive null strings.

8.1.1 Examples of Parsing

All of the following examples were given the source string.

```
"One believes,  one knows, but  know: one believes."
```

Please notice the double space after the first comma and after "but".

Comparison patterns

If there is a string in the template, the source string is scanned from left to right (after the first appearance of the sequence of characters). If it's found, it's removed from the source, and the index is placed on the first character after the sequence. If there is no matching string, the index is placed behind the last character of the source. Given the following template:

```
T1 "," T2 "," REST
```

the source string would be parsed as follows:

```
T1 = "One believes"
T2 = " one knows"
REST = " but  know: one believes."
```

The following template shows what happens if there is no agreement:

```
T1 "," T2 "," T3 "," REST
```

because no third comma is found, T3 receives the rest of the string and REST receives nothing.

```
T1 = "One believes"
T2 = " one knows"
T3 = " but  know: one believes."
REST = ""
```

If REST previously contained another value, it's now lost, since the variable received an empty string. Comparison patterns may be variable. In this case, the corresponding symbol must be indicated with closed parentheses. (In ARexx, this method always forces an analysis of a symbol, which makes the key word "VALUE" unnecessary in some situations, but not with the PARSE command). The corresponding variable can be previously defined (further to the left) in the same template. This is a possible application:

```
command = "\SEARCH\Typignmistake\CW"
parse var command divid 2 instruction (divid) string (divid) option
```

In this case, the first character of "command" is the separator used to parse the rest of the string.

Parsing into words

If several targets follow, the source string is parsed into words. (Or it could be a sub-string of the source, if it appears before or after the target patterns have been specified). Each target from left to right is assigned a word. Empty space between words is dropped. If several words are left over, the last target receives the remainder, including the empty space contained in it. For example:

```
W1 W2 W3 REST ":"
```

leads to the result:

```
W1 = "One"
W2 = "believes,"
W3 = "one"
REST = " knows, but  know"
```

As you can see, the remainder of the string contains the leading space in the source. (ARexx does not behave entirely according to Rexx specifications here: the space should be removed.) Please note that a template of the form:

```
W1 " " W2 " " W3 " " REST ":"
```

which refers to the empty space as a comparison pattern, leads to a different result:

```
W1 = "One"
W2 = "believes,"
W3 = ""
REST = "one knows, but know"
```

As expected, W1 received the first word, W2 the second word, between the first two spaces, but what about W3? In this example, it's assigned the entire string between the second and third empty spaces. A null string was correctly assigned, since they immediately follow one another. Since the comparison removes the empty space in front of "one", "REST" no longer contains a leading space.

A period has special meaning in parsing words: it works as a target, just as a variable symbol, but the value assigned to it is discarded. The period is used to ignore unnecessary words in the source string. The template:

```
. . . W4 .
```

would extract only the fourth word from the source string, in this case, "knows," and assign it to the variable W4.

Parsing by position

In this process, the source string is cut up at certain character positions. The appropriate index values are entered as whole numbers:

```
T1 10 T2 20 T3
```

returns from the original example string:

```
T1 = "One believ"
T2 = "es,  one "
T3 = "knows, but  know: one believes."
```

The target T1 receives the characters 1 to 9, T2 the characters from 10 to 19 and T3 is assigned the rest.

This example used absolute positions. Use prefix operators, ("+" or "-") to move the index position relative to the last position. For example:

```
numbers = "1234567890"
parse var numbers 2 Z1 +4 -1 Z2 -2 Z3 +5
```

leads to the following result:

```
Z1 = "2345"
Z2 = "567890"
Z3 = "34567"
```

First, Z1 receives four characters of input, starting from the second place. Then, the index is moved back by one character ("-1"), and the digit "5" reappears in Z2. From "-2", the absolute position 3 is calculated. The second index for Z2 is smaller than the first. This means that the rest of the source string is assigned to Z2. Finally, the Z3 target receives five characters ("+5") starting from the last position (3).

Using numeric position indicators, whether they are relative or absolute, you can read parts of the source string several times, if necessary. The following command string is also possible:

```
parse var numbers Z1 1 Z2 1 Z3
```

This command assigns the full contents of "numbers" to each of the three different variables.

A numeric position indicator can be a variable: for a relative position, add a "+" or "-" in front of the variable symbol. To indicate an absolute position place an equal sign "=" in the same place; this differentiates them from target symbols.

Combined parsing methods

If a comparison pattern is directly followed by a relative position indicator, you achieve a special effect. The pattern, if found, is not removed from the source string. The current position remains set at the first character of the pattern string.

8.2 Error Trapping with TRACE

What would be the advantage of using an interpreted language if there were no TRACE? A programmer can investigate the events during program execution; it makes the often difficult search for errors much easier, since even well-hidden, minor, logical, program errors become apparent. Rexx offers substantial support for this function. During TRACE, the interpreter displays certain program parts during their execution. A line number, the source text and additional information is displayed. Interpreter behavior is set with trace options, that determine which program parts should be displayed. Two flags control command suppression (!) and interactive tracing (?).

Because it uses "signals", the ARexx program can recognize certain synchronic events (i.e., a "syntax error") or asynchronic events (such as a "halt" request). Using these features, most error conditions can be handled by the program and program aborts can often be prevented.

8.2.1 Trace Options

The following modes are available:

ALL — Displays all clauses before execution.

BACKGROUND — Similar to OFF, except the tracing cannot be externally enabled with the "TS" command.

COMMANDS — Displays all command clauses before they are passed to the external environment. Also, displays return codes not equal to 0.

ERRORS — Displays commands that pass a return code not equal to 0 after execution.

INTERMEDIATES — Displays all clauses, sub-totals (including variable contents), a final form of concatenated symbols, and results of function calls.

LABELS Displays all jump markers.

NORMAL Displays commands with return codes that exceed the current failure level after their execution, and presents an error message. This is the default setting.

OFF Switches all tracing off.

RESULTS Displays all clauses before their execution and presents the result of every expression. Values assigned to variables with ARG, PARSE or PULL are also displayed.

SCAN Displays all clauses and checks them for errors, but does not actually execute them. This mode can be set on with the TRACE command or the internal function TRACE(). It can be engaged at appropriate spots in the program, so that previously tested parts are not re-tested. The RESULT option is usually effective for most error trapping situations.

8.2.2 TRACE Output

Each line is indented on the screen to represent the level of nesting applicable to the clause. At the beginning, there is the line number in which the clause appears in the program and then a three character marker, which shows the meaning of the displayed line. Sub-totals or expressions appear in quotation marks so prefixes and spaces are easily recognized.

Code	Meaning of code
-	program text of a clause
+++	command or syntax error
>C>	expanded form of a compound symbol
>F>	result of a function call
>L>	jump marker (literal or constant value)
>O>	result of a dyadic operation
>P>	result of a prefix operation
>U>	uninitialized variable
>V>	value of a variable
>>>	result of an expression
>.>	value of the place holder

If the data stream is defined, TRACE output is directed by the interpreter to "STDERR"; otherwise it goes to STDOUT, in addition to the display and normal program output.

In some cases "STDOUT" is not defined, for example, if a macro program is started without opening an I/O window. To enable tracing for such programs, a global trace window can be opened in the Rexx master procedure. For every program in which the "STDERR" is not defined, this window becomes the output target for "STDERR".

With the commands TCO and TCC, a global tracing window is opened and closed. Before it's closed, all output of all programs must be returned to its beginning status. The tracing window can also be directed with messages from application programs. During interactive tracing, this window is used for keyboard entry. Since all active programs share one window for trace output, following more than one simultaneous executing program is not recommended, since the result might be confusing.

8.2.3 Command Suppression in ARexx Programs

Suppressing commands is useful when an ARexx program should not pass commands to external environments without prior testing. If one uncontrolled program starts sending unnecessary commands to DOS (for example, to delete files), there may be disastrous results.

ARexx includes a trace mode in which these commands are only displayed. The return code is zero (which would usually be returned if the command was successful) and the program continues. Commands entered during interactive tracing are always executed, but they do not affect the value of the return code.

Command suppression is controlled using exclamation marks, either alone or in front of a trace option, to toggle these functions on and off. If the trace option "OFF" or "BACKGROUND" has been selected, then command suppression is disabled.

8.2.4 Interactive Tracing

During interactive tracing you may enter single clauses during program execution in order to test variable contents, to change them to enter commands or to direct branching and loops. You can enter as many commands as you want with the same limitations as interpreter commands, for example, DO-END constructions must appear in one line.

Any trace mode can be used interactively. The interpreter waits after each displayed clause and requests information with the message ">+>". As a programmer you have three options:

- Press [Enter], entering an empty line, and the program proceeds to the next trace output. The "ALL" mode executes the program step by step by pressing the [Enter] key.
- Enter an equal sign (=) and the last clause is repeated. This is only useful if a change has been made, a correction of variables; otherwise the result will always remain the same.
- Another command, which will be immediately executed if it's ARexx code. Where the program is interrupted depends on the trace mode you have chosen; the interpreter only stops after the clauses it's asked to display. There are some commands that cannot be executed a second time, at which the interpreter will not stop. They are: CALL, DO, ELSE, IF, THEN and OTHERWISE. Also, the interpreter will not stop after a clause that causes an error.

Interactive tracing is controlled with the question mark, which can occur alone or in front of a trace option. Each appearance of the question mark toggles interactive tracing on or off. For example, the command "TRACE ?I", to activate interactive tracing and set "INTERMEDIATES" tracing on, begins a sub-total display. During interactive tracing further instructions that call trace are ignored, so you cannot accidentally exit trace mode.

Errors in the execution of lines entered interactively are displayed but do not lead to a program stop. Also, during interactive tracing, SIGNAL interrupts are blocked. This is to avoid a command error or prevent another SIGNAL condition from immediately branching out to an equivalent label. Such a jump cannot be un-done and would normally prevent a programmer from taking interactive measures when an error occurs, thus creating an uninterruptable infinite loop. If a command with the form "SIGNAL Label" is interactively entered, the jump is executed, and further interactive entries are discarded.

Individual interrupt flags can still be set using the SIGNAL command, or they can be deleted; they will not work until normal program execution resumes.

The trace mode you enter last is retained through sub-routines that you are unable to see. At the beginning of an uninteresting sub-routine tracing with "RESULTS", enter "TRACE OFF". When the sub-routine ends, the old setting is automatically restored. The Rexx master procedure manages the "external" trace flag, with which running programs can be externally set to interactive tracing.

This flag is set with the CLI command "TS". All running programs not set to interactive tracing immediately start to trace, even programs that start after the command. The trace option defaults to "RESULTS" if the modes "INTERMEDIATES" or "SCAN" were not previously set; otherwise they remain unchanged. This flag can control programs that have run out of control, are caught in endless loops, or will not accept any entry. Set the display to interactive mode from the outside and perhaps you can recognize the problem and fix it more quickly. The disadvantage to this arrangement is that this flag influences all ARexx programs. If other programs do not have their own IO channels, and the global tracing window is used to do the trace, the output to this window is hard to interpret. The tracing flag is set off with the CLI command TE. When individual programs notice that the trace mode is no longer on, they also change the trace mode to "OFF". Programs whose trace mode has been set to "BACKGROUND" do not respond to the global tracing flag at all.

8.2.5 SIGNAL Interrupts and Error Handling

ARexx offers a mechanism which makes it easy to recognize errors and special program situations during execution, and to react to them without halting the program. If an interrupt is enabled and the condition occurs, program execution continues at the appropriate label. Deciding factors can be synchronic (for example, syntax errors) or asynchronic (for example, pressing Ctrl+C). These are called "interrupts" and are handled by ARexx; they have nothing to do with microprocessor "interrupt" channels.

The following events are handled by ARexx: the description of the event is the name of the label to which the program branches if the event occurs. A "BREAK_C" interrupt branches to a label of the form "BREAK_C:". An interrupt can be toggled on or off with the command SIGNAL. If the corresponding label is not defined and the condition that has been enabled occurs, the program will still interrupt and display an error message.

BREAK_C	Ctrl+C break detected by DOS. If the interrupt is off, the program immediately ends, with the message "Execution halted" and a return code of 2.
BREAK_D	Ctrl+D break detected by DOS. This is ignored if the appropriate interrupt is switched off.
BREAK_E	Ctrl+E break detected by DOS. This is ignored if the appropriate interrupt is switched off.
BREAK_F	Ctrl+F break detected by DOS. This is ignored if the appropriate interrupt is switched off.
ERROR	The return code of an external program is not "0".
FAILURE	The return code is greater than the FAITAT setting.
HALT	A HALT command appeared (for example, after "hi"). If the interrupt is switched off, the program ends immediately displaying the message "Execution halted" and a return code of 2.
IOERR	DOS has detected an error in an I/O operation.
NOVALUE	An attempt was made to access an un-initialized variable.
SYNTAX	A syntax or execution error has been encountered. Not all such errors can be caught. Certain errors, occurring before a program begins to execute commands, and errors that are not recognized by the external ARexx interface, belong to this group.

When the corresponding jump occurs as a result of the interrupt condition, all active command areas, (DO groups, loops etc.) are dissolved, and the corresponding interrupt is switched off again. This is necessary to avoid endless interrupt loops. Interrupts within a function or a sub-program do not effect the main program.

The interpreter also sets special variables when an interrupt appears. The variable SIGL contains the current line number at the moment the interrupt appeared. The variable RC is set to the appropriate error code during an "ERROR" or "SYNTAX" interrupt.

On an "ERROR", a command code is returned, which can usually be read as an error level. For "SYNTAX", the appropriate ARexx error code appears, which the internal function ERRORTEXT() translates into English.

The main purpose of interrupts is to make error handling easier. After an error, you can branch, to give more information, or get to the root of the condition. Error handling is often very important with the INTERPRET command.

9. ARexx on the Amiga

ARexx runs on any Amiga running Kickstart V1.1 or higher. It uses the IEEE math library on the Amiga and for double precision the "mathieeedoubbas.library", which must be on the logical device "LIBS:". The interpreter itself is in a library named "rexxsyslib.library", which must also be available there. ARexx programs can be named any way you want, but there are some rules meant to ensure a clear overview of library contents. It is customary for ARexx programs that are started directly from the CLI with "rx" to end with the characters ".rexx". Macro programs that are to be started from certain application programs should end with a set of characters specific to the application. For example, ARexx programs that control CygnusEd normally end with ".ced". ARexx uses its own logical device: the ARexx directory. ARexx searches for programs first in the current directory, then in the REXX: directory, if that was defined with the CLI command "ASSIGN".

After V2.X, ARexx is part of the Amiga operating system and the Rexx master procedure is started in the normal startup sequence; it runs in the background.

9.1 Commands

Several CLI commands belong to ARexx and must be located in the c: directory or in the Arexxc: directory that is in the command path. There are various available control functions, all of which depend on sending the corresponding message to the Rexx master procedure. Equivalent functions could be provided by an application program that works with ARexx.

HI	**(HaltInterpretation)**

Syntax: `HI`

Sets the global "Halt" flag, so that all active ARexx programs receive an external "Halt" request. All programs are immediately interrupted, unless caught with SIGNAL ON HALT. Then a subroutine branch would also eventually interrupt (possibly after some clean-up work). When all running programs have received the "Halt" command, the flag is reset.

RX **(RexxeXecute)**

Syntax:

```
RX name [arguments]
RX string [arguments]
```

Starts an ARexx program. If "name" includes a path name, ARexx only looks for the program there; otherwise it searches the current directory and then the REXX: directory. If the Rexx master procedure is not running, it's started first. Arguments are passed to the program and can be queried with ARG. The second form previously listed allows you to enter a complete argument as a string. Observe correct usage of string delimiters. If you want to define a string with this program, you must use the appropriate other string delimiter, or enter the same delimiter twice.

RX can also be started with a tool or project icon from the workbench. A project icon for an ARexx program can be defined as the default tool. If you are using RX in a tool icon you can enter an argument line under tool types with the flag "CMD=". In both cases "CONSOLE=" can specify a window.

RXSET

Syntax:

```
RXSET name [value]
```

Adds a name and a corresponding string "value" to the clip list. If "name" already exists, the old contents is discarded and "value" becomes the new contents. If there is no second argument the corresponding entry on the clip list is deleted.

RXC **(RexxClose)**

Syntax: `RXC`

Ends the Rexx master procedure. The "REXX" port is immediately deleted from the list of active public message ports and the task is complete as soon as the last active program ends.

TCC **(TracingConsoleClose)**

Syntax: `TCC`

Closes the global tracing window as soon as no active program is using it.

TCO **(TracingConsoleOpen)**

Syntax: `TCO`

Opens the global tracing window. All trace output is automatically routed to this window. It can be closed with TCC. Only one program should be in a trace mode, since the output is otherwise very confusing.

TE **(TraceEnd)**

Syntax: `TE`

Cancels the global "Trace" flag; all active ARexx programs are switched to the trace mode "OFF".

TS **(TraceStart)**

Syntax: `TS`

Sets the global external "Trace" flag, putting all active ARexx programs into interactive trace mode. The programs then produce trace output and wait after the next clause. The command is useful if an ARexx program is out of control and needs to be brought back into line. The "Trace" flag remains set until it is deleted with the "TE" command, so programs that are called later also go into trace mode.

WAITFORPORT

Syntax: `WAITFORPORT [-immediate] Portname`

This command waits up to 10 seconds for a message port with the given name to appear. (Caution: Upper and lowercase spelling is observed here.) WAITFORPORT returns a 0 if the port is available, otherwise a 5 (WARN). This is the best way to check for a port to become available for use by an application you just started or by the Rexx master. The option "-immediate" overrides the waiting interval and simply searches for the port once.

9.2 Exchanging Data with the Clip List

The "clip list" contains character strings and a corresponding name for each. This is useful for data exchange between different ARexx programs with the functions SETCLIP() and GETCLIP(). To avoid name conflicts, clip names should be specific to a certain program, perhaps by using a specific name that is related to the program name. There is no limit to the number of clips that can be saved, except for system storage capacity. Beyond data exchange, clips can also be used in other ways. Since ARexx does not support Includes, as other high level languages do, the clip list can be used to emulate this feature, for more flexibly and can be applied simultaneously to several programs. For example, flags that control several running programs could be filed in the clip list. A line named "Presets" with the following contents, for example:

```
quiet=1; speed=5; prompt="Hi >"
```

could with the command:

```
INTERPRET GETCLIP("Presets")
```

be called by each program and used as a series of commands, simple assignments in this example.

The Rexx master procedure manages the clip list and makes sure that a name only appears once in it. In searching for an entry, upper and lowercase letters are distinguished; the name must always be spelled exactly the same way. Entries remain available until a SETCLIP() without the second argument deletes them. When the Rexx master procedure ends, the clip list is discarded.

9.3 The rexxsupport.library

An external function library named "rexxsupport.library", contains several functions specifically intended for the Amiga. It has the same format as the EXEC function libraries, but contains additional code that is used by the interpreter to determine whether a function is in the library and then its offset. This is the QUERY function. If you want to access one of these functions, you must first add the library to the list of libraries. The function ADDLIB("rexxsupport.library",0,-30,34) performs this task; the corresponding file must be in the LIBS: directory. The priority can be set to another value, but this does not make sense unless there are several external libraries. -30 is the customary offset for the query function and a version number (not the revision number, only the whole number portion) must also be specified in order to make sure that the function is in the library. The following documentation refers to Version 34.9.

EXEC Functions

ALLOCMEM()
CLOSEPORT()
DELAY()
FORBID()
FORWARD()
FREEMEM()
GETARG()
GETPKT()
NEXT()
NULL()
OFFSET()
OPENPORT()
PERMIT()
REPLY()
SHOWLIST()
TYPEPKT()
WAITPKT()

DOS Functions

BADDR()
DELETE()
MAKEDIR()
RENAME()
SHOWDIR()

9.3.1 EXEC Functions

ALLOCMEM()

Syntax: `ALLOCMEM(Length[,Flags])`

Reserves a memory area of the indicated length from the list of free blocks managed by EXEC and returns the beginning address as a four byte string. "Length" is rounded up to the next multiple of 8. In addition, a 4 byte string can specify attributes of the memory area as follows:

ANY	"00000000"x	any memory area
PUBLIC	"00000001"x	hard disk, freely accessible
CHIP	"00000002"x	ChipRAM
FAST	"00000004"x	FastRAM
CLEAR	"00010000"x	deleted memory

If necessary, several flags can be combined by adding the values, for example, "00010003"x for PUBLIC, CHIP and CLEAR. The default is "PUBLIC". If the call fails (e.g., if there is no space) an error message is generated.

See also: FREEMEM()

Example:

```
say C2X(ALLOCMEM(256,"00000003"))     ==> 0001DE48
```

CLOSEPORT()

Syntax:

```
CLOSEPORT(Name)
```

Closes the message port of the given name. The port must have been initialized with a call to OPENPORT() by the same ARexx program before CLOSEPORT has effect. If result messages have arrived and have not been handled yet, they are automatically answered with a return code 10. The result is boolean.

See also: OPENPORT()

Example:

```
say CLOSEPORT("Delaware")        ==> 1
```

DELAY()

Syntax:

```
DELAY(Ticks)
```

Waits the given number of 50ths of a second (ticks) and then returns. You should always use this function when an ARexx program should wait a specific length of time.

Until the length of time is passed, the procedure is moved to a status of "waiting" and does not use the processor. Timed loops are generally not seen as useful for this purpose.

Example: `say DELAY(200)      ==> 1 (4 seconds later)`

FORBID()

Syntax: `FORBID()`

Toggles task switching off and returns the current nesting level in the previous call to FORBID() -1 (0 after the first FORBID(), 1 after the second, etc).

Since FORBID() only refers to the running task, it doesn't matter if a program ends before task switching is enabled with the PERMIT() function. Before manually calling STORAGE(), EXPORT() and IMPORT() to the EXEC list or to data areas of other programs from ARexx programs, you should always execute FORBID(), especially if you access the task several times. Following these operations, immediately execute PERMIT().

See also: PERMIT()

Example: `say FORBID()      ==> 0`

FORWARD()

Syntax: `FORWARD(Address,n)`

Not documented.

FREEMEM()

Syntax: `FREEMEM(Address,Length)`

Releases a storage area previously reserved with ALLOCMEM(). "Address" is normally the 4 byte string passed by the equivalent call. "Length" determines the size of the released area. The command FREEMEM() cannot be used to release memory space that was reserved with the internal function GETSPACE() through the Rexx master procedure. The function returns a boolean result.

Caution: False arguments immediately lead to program crash.

Example:

```
say FREEMEM("0001DE48"x,256)        ==> 1
```

GETARG()

Syntax:

```
GETARG(Message[,Entry])
```

Reads a command or function name from a message at a 4 byte address located with GETPKT(), given as "Message". The optional "Entry" can be used with a function message to read individual argument strings (max. 15).

Example:

```
command = GETARG(Packet)
function = GETARG(Packet,0)
Arg1 = GETARG(Packet,1)
```

GETPKT()

Syntax:

```
GETPKT(PortName)
```

Checks if the message port with the given PortName has received a report and returns the address of the oldest message or "0000 0000"x, if nothing has arrived. The port must first have been opened by the same program with OPENPORT().

The function immediately returns a value, even if there is no report. If a program doesn't have anything to do, it's not good to keep "running to the mailbox", which keeps the processor working overtime. Use WAITPKT() and let the program sleep until EXEC hears the mailbox opening.

Example:

```
Packet = GETPKT("Delaware")
```

NEXT()

Syntax: `NEXT(Address[,Offset])`

Returns the 4 byte value, found at the given address, after adding "Offset" (a positive integer). Use NEXT(Address) to move forward though a chained EXEC list, or NEXT(Address,4) to move in the opposite direction.

Example:

```
ExecBase = NEXT("00000004"x)
WaitingList = NEXT(ExecBase,420)
```

NULL()

Syntax: `NULL()`

Returns a 4 byte Amiga pointer with the value "0000 0000"x.

Example:

```
say C2X(NULL())          ==> 00000000
```

OFFSET()

Syntax: `OFFSET(Address,Amount)`

Calculates, from a 4 byte Address and a (prefixed) whole number Amount, a new address.

A convenient method of calculating the address of a particular entry in a structure; this function avoids doing various type conversions.

Example:

```
WaitListPtr = OFFSET(ExecBase,420)
```

OPENPORT()

Syntax: `OPENPORT(Name)`

Creates a public message port with the given name. The result is boolean. The function fails (except in the case of immediate lack of disk space) if a port of the same name has already been named or no further signal bit could be reserved. (16 are available, one is for communication with the master procedure.) The port created is bound to the global data structure of the program. When a program ends, all open ports are automatically closed and outstanding messages are answered with a return code of 10.

See also: CLOSEPORT()

Example: `say OPENPORT("Delaware")      ==> 1`

PERMIT()

Syntax: `PERMIT()`

Toggles task switching back on. The result code is the current nesting level of the previous FORBID() call -1, after executing the function. It returns -1, if task switching is actually permitted again.

Example: `say PERMIT()          ==> -1`

REPLY()

Syntax: `REPLY(Message,Returncode)`

Answers a message at a 4 byte address with "Returncode", an integer error code as Result1. Result2 (the result value) is deleted. The result is boolean.

Example: `say REPLY(Packet,10)       ==> 1`

SHOWLIST()

Syntax: `SHOWLIST(Option[,[Name][,[Pad][,"Address"]]])`

Shows entries in various system lists selected by options. Options are:

Assign:	DOS list of logical devices
Devices:	EXEC list of physical devices
Handlers:	DOS list of device drivers
Interrupts:	EXEC list of interrupts
Libraries:	EXEC list of open libraries
Memory:	EXEC list of free storage areas
Ports:	EXEC list of public message ports
TaskReady:	TaskReady list in EXEC
Resources:	EXEC list of resources
Semaphores:	EXEC list of semaphores
Waiting:	TaskWait list in EXEC
Volumes:	DOS list of storage media

If the first argument is given, the names of the nodes of that list are calculated and returned in a string delimited by an empty space. If the second argument specifies a name, the function returns a boolean result, showing whether the name is in the list. Upper and lowercase writing are distinguished in this search. The "Pad" argument can specify another character, instead of a space, to separate the entries in the result string. The key word "Address", in combination with a name, causes the address of the specific node to be returned, as a 4 byte pointer. If the name is not found, the pointer reads "0000 0000"x. The addresses of DOS nodes are calculated in machine addresses (APTR's), so you do not have to deal with BCPL pointers here.

Example:

```
say SHOWLIST("P")                      ==> rexx ARexx IDCMP
say SHOWLIST("P","REXX")               ==> 1
say C2X(SHOWLIST("P","REXX",,"A"))     ==> 0023485A
say SHOWLIST("P",,"*")                 ==> REXX*AREXX*IDCMP
```

TYPEPKT()

Syntax: `TYPEPKT(Message)`

Returns the 4 byte address of the pointer of a message sender to the global task structure. "Message" is the address of the message, calculated with GETPKT().

Example: `say C2X(TYPEPKT(Packet))    ==> 0026542E ?`

WAITPKT()

Syntax: `WAITPKT(Name)`

Waits for a message to arrive at the given message port name. The port itself must first have been created in the same program with the command OPENPORT().

The boolean result shows whether a report was actually received; normally the result is 1, since the function does not return otherwise. The message must then be retrieved with GETPKT() and should be answered with REPLY(), so that the sender can resume control over the storage area.

Example: `call WAITPKT "Delaware"`

9.3.2 DOS Functions

BADDR()

Syntax: `BADDR(BPTR)`

Re-calculates the BCPL pointer "BPTR" from a normal 4 byte machine address (APTR) by multiplying it with 4.

Example: `say C2X(BADDR("0000 0002"x))   ==> 00000008`

DELETE()

Syntax: `DELETE(Filename)`

Deletes a file or directory. "Filename" is a complete DOS path. The boolean result shows if the entry was found and deleted. Only one file at a time and only empty directories are deleted; wildcard characters (* or &) are not permitted.

Example: `say DELETE("T:Rose.bak")     ==> 1 ?`

MAKEDIR()

Syntax: `MAKEDIR(DirName)`

Creates a directory. "DirName" is a complete DOS path. A boolean result shows whether the processor was able to create the directory.

Example: `say MAKEDIR("RAM:Rosegarden")    ==> 1`

RENAME()

Syntax: `RENAME(AlterName,NewName)`

Renames a file or directory and/or moves it within the same medium and returns a boolean result.

Example: `say RENAME("DF0:Rose","DF0:Tulip")    ==> 1`

SHOWDIR()

Syntax: `SHOWDIR(DirName[,[{"All"|"File"|"Dir"}][,Pad]])`

Returns a string with the entries contained in the directory "DirName", delimited by an empty space. The second argument is the keyword, used to show all entries, only files, or only directories. The "Pad" can be used to put a different character between the entries.

Example: `say SHOWDIR("DF0:c")       ==> rx ts te`

9.4 Creating ARexx Function Libraries

You can always enlarge the scope of ARexx with supplementary function libraries. There are several good reasons to do this. In the simplest case, you could have a desire to take advantage of the mathematical or Amiga-specific options in ARexx with new functions you put together in a library. A function library created for this purpose could contain all the necessary code, or open other Amiga libraries to perform functions. Or you could write a library that works closely with a specific application program, enabling certain program operations to avoid reference to commands and work only with functions. This has its advantages, because the application program doesn't have to interpret commands or receive and answer messages. A library can contain more than entry points for entire and specific operations; it can contain code that is used directly by the application program.

As indicated, function libraries can act as bridges to other system libraries or application libraries. If an ARexx program controls "Intuition", a corresponding function library could recognize appropriate function names, calculate the needed offset, if necessary, convert individual parameters, and then call the corresponding function in the intuition.library. Also, ARexx can be applied as a test platform for new functions; it's easier to manage than a C program, which must be re-compiled after each change and offers no tracing functions.

Whatever the task, function libraries all have the same structure. They contain a portion of the normal EXEC system library with the basic functions OPEN, CLOSE and EXPUNGE as well as a reserved vector. There must also be a QUERY function that can compare the name delivered by ARexx with the names of the functions it contains and then call the correct one. Normally this is the first function after the system functions and has an offset of -30. Function libraries should be fully re-enterable, since many ARexx programs can run simultaneously and use the same functions. If this is not possible because of other constraints, the query function must contain a mechanism that prevents the function from being called more than once.

Function calls

The QUERY function is accessed by the interpreter with the address of a message in A0 and a LIBRARYBASE in A6. The message has the same structure as all Rexx messages, and is not passed by a message port yet. In ARG0, a pointer indicates the function name it's searching for in the table. If this name is not found, an error code of 1 ("program not found") must be returned in D0. The library is then closed and the search continues. The message itself should not be changed, since it must be passed from one library to the next until the function is found.

Parameter conversion

If the called function is found, sometimes the hierarchically higher parameters must be converted to the form the function is expecting. Depending on the structure of the functions, it could be enough to move the pointer; but sometimes parameters or pointers must be supplied in specific registers. Arguments are always passed as ARG strings that can be treated as normal strings supplied with 0's. Other attributes of strings have a negative pointer offset that can be useful.

Numeric values are passed as strings of ASCII symbols and must be converted into integers or variable decimal format in order to perform arithmetic calculations. The ARexx system library contains several functions that are useful for these purposes.

The number of arguments can be determined with the lowest value bytes of the action code. The function name in ARG0 is not counted here, but it's counted for arguments that are set to zero and are used as default values.

The parameter block of the message (with ARG0 to ARG15) is structured just like the argument array (argc,argv) function of a C program. This makes it easy to incorporate a C program into a function library: the query function simply calculates the address of the function you want, the address of the parameter block and the number of arguments that must be placed on the program stack before the function is started.

Returned values

Each function in a library must return an error code and a result string. The error code must be located in D0; if it's 0, A1 must contain an ARG string pointer. The routine that creates the correct returned values can be part of the query function, so that all functions return via this path.

10. The ARexx Interface

Using ARexx, there are two methods of communication with independent external programs:

The command interface

With message system commands, sent to the address of an initialized message port corresponding to an application program, from which answering messages are in turn expected.

The external function environment

Messages are exchanged with another task; a call to a function name still follows, accessed from a specified library list and function environments. Both argument and returned values must conform to ARexx conventions.

The Rexx master procedure is the common communications carrier for ARexx and external applications. It opens the public message port "REXX" and handles many administrative tasks, and also acts as a "host". As a host, it starts ARexx programs and manages global resources. The task structures of all running ARexx programs are maintained in a list, the contents of this list is available to external programs.

The interpreter is located in an operating system library and offers many entry points that are useful for the implementation of ARexx interfaces in other programs. It contains functions that are able to create ARexx structures, such as a RexxMessage or arg string, to manipulate and delete them. These functions should always be used, since future expansions can cause problems. Available functions are documented in more detail later.

10.1 Essential Data Structures

In most applications, the programmer uses two structures with ARexx. The ARexx "arg structure" is used for all strings handled by the interpreter. Normally, they are passed as arg strings, with pointers that indicate the string. The Rexx "msg structure" is used for all communication with external programs and is structurally an expansion of the EXEC message form.

Arg strings: all strings in ARexx are stored as Rexx arg structures, created for each string in an equivalent length. Strings are passed as arg strings (i.e., a pointer to the area where data is located in the Rexx arg structure). The data always ends with a zero in order to allow treatment as normal C strings in other programs. Additional data such as length, hash value etc. can then be accessed with negative offset of the arg string pointer.

```
struct RexxArg {
   LONG ra_Size;        /* reserved total length of the structure */
   UWORD ra_Length;     /* length of the string */
   UBYTE ra_Flags;      /* attribute of a string */
   UBYTE ra_Hash;       /* hash value */
   BYTE ra_Buff[8];     /* data area (where the arg string points) */
     }                  /* minimum size: 16 bytes*/
```

There are library functions used to create arg strings (CreateArgstring()) and to delete them (DeleteArgstring()), as well as converting from whole numbers into this format.

Message Packets

All communication between ARexx and external programs takes place with RexxMsg structures. There is a function in the ARexx system library that lets you create them (CreateRexxMsg()) and one to delete them (DeleteRexxMsg()).

Messages sent by ARexx, for example, to pass a command to an application program, have the same form as those that move in another direction to start a macro program. You can distinguish one from the other because all messages that are sent by ARexx contain a pointer to the string "REXX" in the name slot of the node. This can be useful in distinguishing messages when a port receives them from several sources.

```
struct RexxMsg {
  STRUCT Message rm_Node;      /* and EXEC message structure */
    APTR rm_TaskBlock;         /* pointer to the sender's task structure */
    APTR rm_LibBase;           /* pointer to RexxSysBase */
    LONG rm_Action;            /* action codes */
    LONG rm_Result1;           /* primary result (Returncode) */
    LONG rm_Result2;           /* secondary result  */
    STRPTR rm_Args[16];        /* pointers to arguments 0-15*/
                               /* the expanded area */
  STRUCT MsgPort *rm_PassPort;/* pointer to the next port*/
    STRPTR rm_CommAddr;        /* name of its own port */
    STRPTR rm_FileExt;         /* file name extension */
    LONG rm_Stdin;             /* file handle of the input data-flow*/
    LONG rm_Stdout;            /* file handle of the output data-flow */
    LONG rm_avail;             /* for future expansion */
     }                         /* size: 128 bytes*/
```

Resource Nodes

A further useful structure is often used by ARexx to set up resource lists: the Rexx "rsrc structure". It has a variable length, that is entered in the structure, along with the address of the function used to remove the structure. This means that heterogeneous lists can be set free by calling RemRsrcList().

10.2 Requirements for a Command Interface

An application program that wants to communicate with ARexx only needs a public message port and a program input that can process the commands received there. Usually this isn't too much to manage, since many programs often already have several message ports receiving keyboard and menu operations. For a program that's directed by commands, it processes the incoming commands easily and reacts accordingly. With menu-driven programs, more work is necessary once commands do more than just activate individual menu options. Which commands are recognized, and the syntactic form of each, depends on the programmer.

An application program sends a command call message to the Rexx master procedure, usually in direct response to user entry. As soon as the report is received, a new DOS procedure starts, that examines the command line, takes the first word, and searches for an equivalent macro program file (possibly with an application-specific extension that was passed with the filename). When a file of the same name is found, the program is executed. Usually the program sends back one or more commands to the public port of the calling program. While one is being executed, the macro program waits until it receives a return code from the command. If an error is encountered, it should be able to handle it logically. Finally, the macro program should end and pass the command call message back to the application program with an appropriate return code.

Error trapping in macro programs is an important feature of communication. Macro programs must be able to recognize whether a command was executed correctly, or if something went wrong in the process in order to react intelligently to whatever happens.

Normally, a command call message is not answered if the error status that followed the command is known. Programs that receive commands from a message port, from user input and handle both with the same routines, must be able to differentiate between the two input modes. A flag indicates what happens in case of an error. In the first case, an appropriate error code can be returned and in the second case, with direct input, an error message should also display on the screen.

Return codes that appear in the result slot of the message should also report the severity of the error. Small whole numbers would indicate relatively harmless errors, and large numbers would appear with major errors. This enables a programmer to set a "failure level" in order to ignore small errors and report those that exceed it. Other than this convention, a programmer has free choice of error codes.

Every program meant to support the command interface must open a public message port. If a command is to be sent to the program, it receives a Rexx message with the rm_Action entry "RXCOMM" and an arg string pointer to the command line in ARG0, at this port. The other ARG entries are not used with commands. There are two pointer entries that could be interesting for the program: rm_TaskBlock points to the sender's task structure and rm_LibBase points to the base address of the ARexx system library. With the exception of the result code in rm_Result1 and possibly also rm_Result2, the program should not change the message.

These appear when the corresponding command has been completed. rm_Result1 receives an error code, 0 if the command was carried out with no errors. This long word is later assigned to the variable RC in the macro program.

If the macro program expects a result string (indicated with the RXFB_RESULT bit in the command code), the corresponding arg string pointer in rm_Result2 should be returned. A result string should only be returned when it's requested and if rm_Result1 is zero; otherwise a zero must be entered in rm_Result2. If this convention is not followed, a loss of memory capacity results. An unexpected result string can lead to a program crash if memory areas become free without being assigned (or at least not with an arg string).

Many application programs support simultaneous work on several data files: most word processors let users open windows with separate files in them. If an ARexx macro program is called by the editor, it must be clear to which file the returned commands apply.

ARexx supports this distinction with the entry rm_CommAddr, in which the opening ADDRESS setting for a (new) macro can be entered. The word processor can then assign a separate message port (for example, "xyEdit1", "xyEdit2", etc.) for each file, and report the appropriate name when macro calls are encountered.

Application programs can open several ARexx ports that can also be used to differentiate command classes, each of which is then sent to the correct port with the ADDRESS command.

ARexx program calls are made by sending a corresponding report to the Rexx master procedure. Programs can be called as commands or as functions; the command mode is generally easier and more free, since only a few fields of the message must be completed.

When an ARexx message structure is created, all entries are first set to 0. Entries that are filled by the sending program are never changed by ARexx so that this structure can be re-used after the initial message is answered. For this reason, only one structure is necessary, which must then only be partially changed for new calls.

In the rm_Action slot of the message, the mode of the call is determined. For command mode, RXCOMM is entered; for function mode, RXFUNC.

In addition, certain flags can be set, to enable options that are described later.

Command strings, function names, and arguments must be entered as arg strings. Normal strings can be comfortably created with the CreateArgstring() function. Returned arg strings can usually be treated as normal strings, since the pointer refers to the data area (a string that ends with a 0). Because the corresponding strings are not changed in the course of an ARexx program, a program may have to build up many of these structures. The pointer that is returned by CreateArgstring() is placed in the equivalent slot of the message: ARG0 for the command string or function name, ARG1 to ARG15 for function arguments. When the message is answered, extra arg strings can be deleted with DeleteArgstring().

When all the necessary fields are filled, the report is sent to the public Port "REXX" using the EXEC function, PutMsg(). Its address must first be determined with the function FindPort(), but this value should not be saved by the program because the port can be closed at any time. To ensure against program crash, you must bracket the calls to FindPort() and PutMsg() with Forbid() and Permit().

After sending the message, the application program can resume its own tasks and the macro program runs as a separate task. It's often useful to prevent further user input for the duration of the ARexx macro so that data accessed by the macro is not changed by the user.

10.2.1 Command Calls

Command mode returns a command string to the calling program. The string consists of a macro name, an empty space and arguments in whatever form necessary. ARexx takes the name, usually the name of the executing program, and tries to start it. Normally the rest of the command string is a single argument the program uses. The RXFB_TOKEN flag can adjust behavior: if it's set, then the rest of the string is parsed into several arguments. In this process, words are separated as they would be with PARSE. The number of arguments is not limited in this case, since they don't have to fit into the 15 available message slots. In order to prevent spaces that represent arguments from being divided, they can be enclosed in quotation marks [" "]. If such a section contains quotation marks, use single quotes; the two types of quotation symbols can be used alternately. Double entry of one of the symbols doesn't work here. At the end of the string, no quotation mark is necessary.

For example, the call:

```
test.rexx "The first argument" second "'one more'
```

would mean that the command:

```
parse arg A1,A2,A3; say A1; say A2; say A3
```

would output as follows:

```
The first argument
second
'one more'
```

If the first element of a command string is already in quotation marks, it's assumed not to be a program name, but rather as a single word. This is an easy method for starting very short ARexx programs (its length is not limited in any way). If RXFB_TOKEN wasn't specified, only the first section that appears in quotation marks is examined, and the rest is discarded. The rxfb_string flag defines the entire command string as an

ARexx program text. In this case, no parsing takes place and the program is immediately executed. Calls usually don't expect a result string. The flag RXFB_RESULT can request it. The calling program must delete the string, which is hierarchically higher than itself, when it's no longer necessary.

10.2.2 Function Calls

Function calls pass a function name and up to 15 arguments as strings to the application program. The function name is used for access. The actual number of arguments (not counting the name) must be written to the lowest value byte of the command code.

This form is normally used when a result is expected (but this does not require the use of a function call) or when several argument strings are already available. A result is again requested with the RXFB_RESULT flag. After the function is completed, if no error took place, and Result1 is zero, the pointer in Result2 should be set to the equivalent string.

10.2.3 ARexx Program Search Order

Again, ARG0 can contain a complete program instead of a filename. It's signaled with the rxfb_string flag.

Searching for program files is a two-step process, in which the current name extension (".rexx", if nothing else is specified in the message) is attached to the filename, if not previously specified. If the search is unsuccessful, the un-expanded name is used for a new search.

If the name contains a path, the program only looks there; otherwise the current directory is searched first (possible with both name variations) and then the REXX: directory. A command call with "RAM:t/examples" would be searched for in RAM:t under the names "examples.rexx" and then "examples". Without the path name, the search order would be "examples.rexx", "examples", "REXX:examples.rexx" and "REXX:examples".

If a program is still not found, one more possibility exists: If the message path rm_PassPort was filled in, the message is simply passed to the port specified there. This means that one command can be passed to several programs, until one of them can do something with it. If there is a 0 the message is answered with an error code 1 ("program not found").

10.2.4 Expanded RexxMsg Structure Areas

Entries in this area of the message can adjust various default settings. If no settings are being changed, these can be left at zero.

Application programs should enter values for the appropriate name extension and the name of their own ports. The name extension is useful to identify macro programs for specific applications from other program files and should be specific to each program. Entering the port for the program is done so the addressed port is already set at the beginning of the macro. Since one program can have several ports and the macro must know where it should direct its commands, this is very important. Use the application program name or an abbreviation of it.

PassPort

In the rm_PassPort slot, a further message port address can be entered. The report is sent to this port if no corresponding program file is found. This port should be a secured resource so that it cannot be removed until the message has been passed. It does not have to be a public port; for this reason it's not possible to make sure it's available before the message is passed on.

Host address

An entry in the rm_CommAddr slot can indicate the ADDRESS setting for an ARexx program that is to be started. The entry includes a pointer to a string that closes with a zero and contains the name of the public message port to which commands are to be directed. This option is very important for application programs that allow work on several files simultaneously and open a separate message port for each file. The name of the correct ports are then passed to a macro when it's called. If such an entry is not found, "REXX" is the default setting.

File Extension

The entry for rm_FileExt changes the default value of ".rexx" for file name extensions. Application programs should enter a specific extension here, common to all its macro programs. If it is a pointer to a string it is terminated with a zero.

Input and output data flow

Default values for data input and output of an ARexx program are directly taken from the procedure structure of the calling application program, as it's a DOS procedure. One or both data streams can be diverted by entering corresponding DOS file handles in the rm_Stdin and rm_Stdout slots. The data flow cannot be closed as long as the program is running. Both values are entered directly in the procedure structure of the calling program.

The output stream is simultaneously the pre-set target for trace output by the program. If interactive tracing is used, the output stream should always be defined to an interactive device like CON:, since user entry is also expected.

If an ARexx program is called by an EXEC task, these entries are the only way to control input and output.

10.2.5 Result Entries

A message that is started by an ARexx program is answered as soon as it's completed. Two result entries then contain either error codes or a possible result string.

If the primary result in rm_Result1 is zero, the program ran without errors and the pointer in rm_Result2 indicates a result string, if requested. If the primary result is not zero, two things may have happened: either the secondary result is zero, meaning that the return code was passed with "EXIT rc", or it's "RETURN rc". This can be an error code or a result; how this return code is handled depends on the calling program. If the secondary result is not zero, then the primary result is an error level indicating the severity of the error and Result2 is an ARexx error code. This should be reported to the user. In order to translate the error code to an equivalent text, the function ErrorMsg() is provided.

Result strings are the responsibility of the calling program and must be deleted with the DeleteArgstring() function when they are no longer needed.

10.3 The Rexx Master Procedure

All communication with the Rexx master procedure takes place using the message structure previously described. It contains a command entry that indicates which operation is to be carried out and entries for the appropriate or necessary parameters. Messages received are immediately handled, either being answered or, in the case of program calls, passed on. The structure contains two result entries with which error codes or result strings are transmitted. In the parameter portion of the structure, either whole numbers of the "long" type or pointers to arg strings can be entered.

10.3.1 Action Codes

Valid command codes are described here. The commands are listed in order of their mnemonics, followed by the permitted flags. The resulting code is formed by a logical OR of the action code and all necessary flags. This code is entered in the rm_Action slot.

RXADDCON [RXFB_NONRET]

Adds an entry to the cliplist. ARG0 points to the name, ARG1 points to the data and ARG2 contains the length of the data. This is not required to be an arg string. The name should be a string that closes with a zero, but the data itself can contain null bytes; its length is explicitly indicated.

RXADDFH [RXFB_NONRET]

Adds a function environment to the library list. The first argument, ARG0, points to a name string closed with a zero along with a port. The argument ARG1 contains the search priority. A priority can be specified as an integer ranging from -100 to 100. If a previous entry of the same value exists, the message is returned with a warning and the appropriate error code. No check is made to verify existence of the port.

RXADDLIB [RXFB_NONRET]

Adds an entry to the library list. The argument ARG0 points to a name string that ends with a zero, with the name of the function library or of the function environment port. The search priority is set with ARG1, a whole number between -100 and 100; the remaining area is reserved for later expansion. The offset for the "query" function, specified in ARG2 and ARG3, contains the version number. If a previous entry of the same name exists, the message is returned with a warning and the error code. Otherwise, the new entry is accepted and the library or function environment is available to ARexx programs. There is no check for actual availability of the library, nor whether it can be opened.

RXCOMM [RXFB_TOKEN] [RXFB_STRING] [RXFB_RESULT] [RXFB_NOIO]

Calls an ARexx program in command mode. ARG0 must contain an arg string pointer to the command string. The flag RXFB_TOKEN specifies how the command string is to be parsed into several arguments. Or RXFB_STRING indicates that the command string itself contains the program. This call usually does not deliver a result string; RXFB_RESULT can be used to request one, but the calling program must then make sure this string is deleted after use. The argument RXFB_NOIO prevents the input and output of the called program from being used by the caller.

RXFUNC [RXFB_RESULT] [RXFB_STRING] [RXFB_NOIO] Number args

Calls to a function. A pointer in ARG0 refers to the function name. ARG1 to ARG15 point to arguments. All of them must be arg strings. The lowest value byte of the action code is the number of arguments (not counting the function name). For function calls, RXFB_RESULT is used to request a result string, but this is not required. RXFB_STRING shows whether the entire command string contains the program. Finally, RXFB_NOIO prevents the input and output of the called program from being used by the caller.

RXREMCON [RXFB_NONRET]

Removes an entry from the cliplist. ARG0 is a string that closes with a zero and points to the name to be removed. The cliplist is searched for an entry with the desired name. If it's found, the entry is removed from the list and the storage area it occupied is released. If the name is not found, the message is returned with an error code.

RXREMLIB [RXFB_NONRET]

Removes an entry from the library list. ARG0 is a string that closes with a zero and points to the name to be removed. The library list is searched for an entry with the desired name, whether it's a function environment or a system library. If it's found, the entry is removed from the list and the storage area it occupied is released. If the name is not found, the message is returned with an error code. The entry is not removed if an ARexx program is in the process of calling it.

RXTCCLS [RXFB_NONRET]

Closes the global Trace window. If no ARexx program is waiting for entry from the Trace window, it's immediately closed; otherwise the program waits until the active programs are no longer using it.

RXTCOPN [RXFB_NONRET]

Opens the global Trace window. After this instruction, the Trace output from all active ARexx programs is redirected to the Trace window. User entry, for interactive tracing, is also expected there. There can only be one open Trace window at a time; if it's already open, the message is returned with a warning.

10.3.2 Action Code Control Flags

In addition to the command codes, individual bits can be inserted in the action code to activate special functions. In the individual commands, only certain flags are accepted, all others are ignored.

RXFB_NOIO — With the command code RXCOMM or RXFUNC, this flag prevents automatic transfer of input and output data to the calling program.

RXFB_NONRET

Determines that the recipient will not respond to the report. This also means that it doesn't matter to the sender whether or not the operation was successful, since there is no other way to inform it about success or failure. The message is transferred of the receiver and must be released by it with DeleteRexxMsg().

RXFB_RESULT

With RXCOMM or RXFUNC, this flag controls the transfer of a result string. If the program or the function ends with EXIT (or RETURN) and passes on an expression, the calling program receives this expression as an arg string. If this result is no longer needed, the calling program must remove it with DeleteArgstring().

RXFB_STRING

With RXCOMM or RXFUNC, this flag indicates that ARG0 does not contain a filename, but that a complete ARexx program was passed (which then does not have to be within a set of quotation marks).

RXFB_TOKEN

Demands, in connection with the command code RXCOMM, that the data following the program name not be passed as a complete argument, but instead parsed into words and transformed into several arguments. Areas enclosed in quotation marks are not parsed, so that spaces are possible. At the end of the command strings, no additional quotation marks are necessary.

10.3.3 Managing the Results

The Rexx master procedure conforms to Amiga code conventions for the result that's passed in rm_Result1. This is an error level set for "warning" at 5 (WARN) and, for more serious errors, reads as 10 (ERROR) or 20 (FAIL). The value in rm_Result2 is then either zero or an ARexx error number, if an appropriate one is available.

10.4 Functions in rexxsyslib.library

The ARexx interpreter is part of the Amiga operating system library "rexxsyslib.library". Many of the functions in it are only used by the interpreter and are not documented. Others can be of use to other programs that use ARexx.

System library functions are meant to be called from assembly language programs and generally only affect registers A0 and A1, as well as D0 and D1. Many functions return values in several registers in order to reduce code. In addition, the functions control the status register CCR, if appropriate. Usually CCR refers to the value returned in D0.

The function offsets are defined in the file rexx/rxslib.i, included after Kickstart 2.0 and should be linked in matching assembler source code. It can also be called from C programs, if appropriate code is included in the link.

Overview of available functions

I/O Functions

There are two groups of I/O functions: the low level uses DOS file handles directly, while the higher level works with lists of I/O buffer structures and supports logical filenames.

CloseF()	close file buffer
CreateDOSPkt()	create a DOS standard packet structure and initialize it
DeleteDOSPkt()	delete a DOS standard packet structure
DOSRead()	read from a DOS file
DOSWrite()	write to a DOS file
ExistF()	test whether a file exists
FindDevice()	test whether a DOS device exists
OpenF()	open a file buffer
QueueF()	queue a line in a file buffer
ReadF()	read a character from a file buffer
ReadStr()	read a string from a file buffer
SeekF()	moves the access pointer to a specific position
StackF()	adds a line to the file buffer
WriteF()	writes characters to a file buffer

String Manipulation

ARexx treats all data as strings. These functions perform common string operations.

CmpString()	compare string structures
LengthArgstring()	calculate the length of an argument string
StcToken()	select a token
StrcmpN()	compare strings
StrcpyA()	copy a string and convert to ASCII
StrcpyN()	copy a string
StrcpyU()	copy a string, converted to capital letters
StrflipN()	transpose a string
Strlen()	determine the length of a string

Conversions

CVa2i()	convert ASCII to INT
CVc2x()	convert CHAR to HEX or BIN
CVi2a()	convert INT to ASCII
CVi2arg()	convert INT to an ASCII arg string
CVi2az()	convert INT to ASCII with leading zeros
CVs2i()	convert string structure to INT
CVx2c()	convert HEX or BIN to CHAR
ErrorMsg()	calculate error number from an error message
ToUpper()	convert ASCII to capital letters

ARexx Resource Handling

AddClipNode()	assign a clip node
AddRsrcNode()	assign a resource node
ClearMem()	delete a storage area
ClearRexxMsg()	delete arg strings from a message
ClosePublicPort()	release a port resource node
CreateArgString()	create an arg string structure
CreateRexxMessage()	create an ARexx message structure
CurrentEnv()	determine pointer position in the current storage environment
DeleteArgstring()	release an arg string structure
DeleteRexxMsg()	release an ARexx message structure
FillRexxMsg()	fill arg strings in a Rexx message
FindRsrcNode()	find a resource node
FreePort()	close a message port
FreeSpace()	release internal storage
GetSpace()	reserve internal storage
InitList()	initialize a list header structure

InitPort()	initialize a message port
IsRexxMsg()	check a message
LengthArgstring()	calculate the length of an arg string
ListNames()	list node names in an arg string
LockRexxBase()	protect a global resource from data write calls
OpenPublicPort()	create a port resource node
RemClipNode()	release a clip node
RemRsrcList()	release a resource list
RemRsrcNode()	remove a resource node
UnlockRexxBase()	release a resource

10.4.1 I/O Functions

CloseF()	Closes file buffer

Syntax:

```
success = CloseF(IoBuff)
D0                A0
```

Releases the IoBuff structure and closes the matching DOS file. An entire list of IoBuff structures can be deleted with a single call to RemRsrcList(); each individual structure is then processed with an automatic CloseF().

CreateDOSPkt ()	Creates and initializes a DOS standard packet structure

Syntax:

```
packet = CreateDOSPkt()
D0              A0
        (CCR)
```

Reserves a storage area for a DOS standard packet structure and initializes it by linking it to the EXEC message and DOS packet sub-structures. A ReplyPort is not automatically added, since entries are normally filled in immediately before sending the message.

See also: DeleteDOSPkt()

DeleteDOSPkt() — Releases a DOS standard packet structure

Syntax:

```
DeleteDOSPkt(message)
             A0
```

Releases a DOS standard packet structure, that has normally been created earlier with a call to CreateDOSPkt().

See also: CreateDOSPkt()

DOSRead() — Reads from a DOS file

Syntax:

```
count = DOSRead(filehandle,buffer,length)
D0                 A0        A1     D0
          (CCR)
```

Reads characters from the DOS "filehandle" into the "buffer". "Length" is the maximum number of characters to be read, "count" returns the actual number of characters read after the call, or -1, if an error was encountered.

DOSWrite() — Writes to a DOS file

Syntax:

```
count = DOSWrite(filehandle,buffer,length)
D0                 A0        A1      D0
         (CCR)
```

Writes characters from "buffer" to the given DOS filehandle. "Length" is the maximum number of characters to be written, "count" returns the number of characters actually written, or -1 if an error was encountered.

ExistF() — Tests whether a file exists

Syntax:

```
success = ExistF(filename)
D0                 A0
           (CCR)
```

Verifies whether a file exists by trying to receive a Read-Lock for the file. The result determines whether the operation was successful and the lock is released.

FindDevice()	**Tests whether a DOS device exists**

Syntax:

```
device = FindDevice(devicename,type)
D0                          A0          D0
         A0
        (CCR)
```

Searches in the DOS DeviceList for a device node of equivalent type, whose name equals "devicename". Available "types" are the constants DLT_DEVICE, DLT_DIRECTORY or DLT_VOLUME, that are defined in DOS includes. The name is converted to capital letters before the comparison. The argument "device" is a pointer to the device node, or 0 if nothing was found.

OpenF()	**Opens a file buffer**

Syntax:

```
IoBuff = OpenF(list,filename,mode,logical)
D0                  A0      A1      D0     D1
         A0
         (CCR)
```

Attempts to open a DOS file. The parameter "mode" is one of the constants RXIO_READ, RXIO_WRITE or RXIO_APPEND, that are defined in ARexx Includes. If it's successful, an IoBuff structure is created and added to the "list". The "list" must be a pointer to a regular EXEC list header. The argument "logical" is a pointer to the logical filename, or 0 if such a value is not needed.

See also: CloseF()

QueueF() — **Adds a line to a file buffer**

Syntax:

```
count = QueueF(IOBuff,buffer,length)
D0              A0      A1     D0
```

Adds a line to a data stream that belongs to the given IoBuff structure. This data stream must be directed by a driver that recognizes the ACTION_QUEUE command. The parameter "buffer" is a pointer to the data to be added and "length" indicates the number of bytes to be added. The "count" indicates how many characters were actually transferred, or shows -1 if an error was encountered.

See also: StackF()

ReadF() — **Reads characters from a file buffer**

Syntax:

```
count = ReadF(IoBuff,buffer,length)
D0             A0      A1     D0
     (CCR)
```

Reads characters from a file that belongs to the IoBuff structure. The value in "buffer" is a pointer to the target for the data to be read and "length" indicates the number of characters to be read. The "count" reports how many characters were actually transferred.

ReadStr() — **Reads a string from a file buffer**

Syntax:

```
(count,pointer) = ReadStr(IoBuff,buffer,length)
  D0     A1                A0     A1      D0
```

Reads characters from a file that belongs to the IoBuff structure until a line-feed (ASCII 10) occurs. The line-feed characters are discarded. "Buffer" is a pointer to the target of the data read and "length" is the maximum number of data to be read. The "count" relays how many characters are actually taken, or -1 if an error was encountered.

SeekF() — Moves the access pointer to a specific location

Syntax:

```
position = SeekF(IoBuff,offset,anchor)
D0                 A0     D0     D0
```

Moves the access pointer to a specific location, indicated by "offset", a relative byte position, given in reference to the "anchor" point. It can be set using "anchor" to the beginning (-1), the current position (0) or to the end of the file (1). The "position" returned is the new position in reference to the beginning of the file.

StackF() — Adds a line to the file buffer

Syntax:

```
count = StackF(IoBuff,buffer,length)
D0              A0     A1     D0
```

Adds a line to the data stream belonging to the given IoBuff structure. This data stream must be controlled by a driver that can process an ACTION_STACK command. The "buffer" points to the data location, and "length" is the number of bytes to be added. "Count" reports how many characters were actually transferred or appears as -1 if an error was encountered.

WriteF() — Writes characters into a file buffer

Syntax:

```
count = WriteF(IoBuff,buffer,length)
D0              A0     A1     D0
      (CCR)
```

Writes characters to the file that belongs to the IoBuff structure. "Buffer" is a pointer to the source for the data to be written and "length" is the number of data. The "count" indicates how many characters were actually transferred or reads as -1 in the case of an error.

10.4.2 String Manipulation

ARexx treats all data as strings. These functions fulfill the more common string operations.

CmpString()	Compares string structures

Syntax:

```
test = CmpString(ss1,ss2)
D0          A0          A1
     (CCR)
```

Compares two ARexx string structures whose pointers form the arguments. The structures also contain the length and hash value of the strings; if there is no agreement in these, the comparison ends. The function returns -1 (TRUE) if they agree or otherwise a 0 (FALSE).

lengthargstring()	Calculates the length of an ARexx arg string

Syntax:

```
length = LengthArgstring(argptr)
D0                        A0
```

Determines the length of the argument string at the given address.

StcToken()	Pulls out one token

Syntax:

```
(quote,length,scan,token) = StcToken(string)
 D0      D1    A0    A1                A0
```

Searches a string for the next token, delimited by an empty space, and returns a pointer to the first character of this token. The value "quote" contains the quotation mark character (" or ') or 0; spaces found within quotation marks are not located with this function. "Length" is the length of the token found, including quotation marks, if applicable. "Scan" is a pointer to the position after the token that was found, which prepares the following call.

StrcmpN() Compares strings

Syntax:

```
test = StrcmpN(string1,string2,length)
D0                 A0      A1      D0
        (CCR)
```

The strings at addresses "string1" and "string2" are compared character by character, until "length" is reached or a deviation is recognized. "Test" is -1 if the first string was shorter, 1 if it was larger, or 0 if the two strings were exactly equal.

StrcpyA() Copies a string and converts to ASCII

Syntax:

```
hash =  StrcpyA(destination,source,length)
D0                  A0          A1     D0
```

Copies the string at the location "source" to "destination". In the process, the data's MSB is deleted and projected onto the lower 128 characters in the ASCII table. The string can contain "00"x, which is why "length" is necessary (USHORT). The result is the hash byte of the copied string.

StrcpyN() Copies a string

Syntax:

```
hash = StrcpyN(destination,source,length)
D0                 A0          A1     D0
```

Copies the string found at "source" to "destination". The string can contain zeros "00"x, so "length" is necessary (USHORT). The result is the hash byte of the copied string.

StrcpyU() — Copies a string and converts to capital letters

Syntax:

```
hash = StrcpyU(destination,source,length)
D0             A0          A1     D0
```

Copies the string found at "source" to "destination". The string can contain zeros "00"x, so "length" is necessary (USHORT). The result is the hash byte of the copied string.

StrflipN() — Reverses a string

Syntax:

```
StrflipN(string,length)
A0              D0
```

Transposes the order of characters in a string at the given storage location.

Strlen() — Determines the length of a string

Syntax:

```
length = Strlen(string)
D0              A0
      (CCR)
```

Determines the number the characters in the string (closed with "00"x) at the given storage location.

10.4.3 Conversion Functions in ARexx

CVa2i() — Converts ASCII to INT

Syntax:

```
(digits,value) = CVa2i(buffer)
  D1     D0            A0
```

Converts from ASCII symbols at the given storage location to an equivalent 4-byte integer value (LONG). The function reads ASCII characters until a character appears that is not a number or until an overflow occurs. The function returns the integer value and the number of ASCII characters read.

CVc2x() — Converts CHAR to HEX or BIN

Syntax:

```
error = CVc2x(outbuff,string,length,mode)
D0               A0       A1       D1     D0
```

Changes "length" number of bytes from the storage location "string" to a string of equivalent hexadecimal or binary characters. The "mode" is either -1 for hex conversion or 0 for binary conversion.

CVi2a() — Converts INT to ASCII

Syntax:

```
(length,pointer) = CVi2a(buffer,value,digits)
  D0      A0               A0     D0    D1
```

Changes the prefixed integer value in D0 to the corresponding decimal number. The "buffer" is the target for the resulting string and "digits" is the maximum number of characters to be written. The function returns "length", the actual number of characters copied, and "pointer" the new "buffer" pointer.

CVi2arg() — Converts INT to ASCII arg string

Syntax:

```
argstring = CVi2arg(value)
D0                  D0
          A0
          (CCR)
```

Changes the "LONG" value in D0 to a string and creates a Rexx arg structure. The return value is a pointer to the arg string structure; or 0 if an error occurred. The structure created with this manipulation can be released with DeleteArgstring().

CVi2az() — Converts INT with leading zeros to ASCII

Syntax:

```
(length,pointer) = CVi2az(buffer,value,digits)
   D0       A0                A0     D0    D1
```

Changes the prefixed integer value in D0 to the corresponding decimal number. The "buffer" is the target for the resulting string and "digits" is the maximum number of characters to be written. If necessary, zeros are added to the left in order to reach the number of characters to be written. The function returns "length", the actual number of characters copied, and "pointer" the new "buffer" pointer.

CVs2i() — Converts string structure to INT

Syntax:

```
(error,value) = CVs2i(ss)
D0      D1            A0
```

"ss" is a pointer to a string structure. The function returns the value of the string as "LONG" in D1. "Error" is 47 if an error occurs; this is the code for "arithmetic conversion error".

CVx2c() — Converts HEX or BIN to CHAR

Syntax:

```
error = CVx2c(outbuff,string,length,mode)
D0            A0      A1     D0     D1
```

Changes "string", which must be a valid hexadecimal or binary number to the corresponding character. If "mode" is -1, hex is to be expected; or, if it's 0, binary numbers. There can be spaces, but only at the byte limits. "Length" indicates the number of bytes to be written to "outbuff". "Error" is 47 if an error occurs; this is the error code for "arithmetic conversion error".

ErrorMsg() — Calculates the error message from the error number

Syntax:

```
(bool,ss) = ErrorMsg(code)
D0    A0             D0
```

Returns an English error message for the given error code as a pointer to a string structure. "Bool" is -1 if "code" is not a valid ARexx error code; otherwise it's a 0. Undefined error codes return the ominous "undiagnosed internal error".

ToUpper() — Converts ASCII to capital letters

Syntax:

```
upper = ToUpper(char)
D0              D0
```

Converts ASCII symbols to capital letters, working only with D0.

10.4.4 ARexx Resource Handling

AddClipNode() — Sets up a clip node

Syntax:

```
node = AddClipNode(list,name,length,value)
D0                 A0    A1    D0     D1
     A0
     (CCR)
```

Creates a clip node and binds it to the list specified with the header "list". "Name" is a pointer to a name string that ends with zero; "value" is a pointer to the storage area. The result is a pointer to the newly created node, or zero if something went wrong.

The RemClipNode() function deletes a node created with this function. Clip nodes can be held in a resource list, mixed with other nodes that are all dissolved with RemRsrcList().

AddRsrcNode() — Adds a resource node

Syntax:

```
node = AddRsrcNode(list,name,length)
A0                 A0    A1    D0
     A0
     (CCR)
```

Creates an ARexxRsrc structure and binds it to the list indicated by the header "list". "Name" is a pointer to a string closed with a zero, which is set up as a copy of the NodeName slot in the structure. The "length" is the size of the entire node entered into the structure so that it can be removed later with RemRsrcNode(). The result is a pointer to the newly created node, or zero if an error occurred.

ClearMem() — **Deletes a storage area**

Syntax:

```
ClearMem(address,length)
         A0         D0
```

Deletes a storage area from "address" and "length" is number of bytes. The value for "address" must be even and "length" must be a multiple of four. A0 is preserved.

ClearRexxMsg() — **Deletes arg strings from a message**

Syntax:

```
ClearRexxMsg(msgptr,count)
           A0         D0
```

Releases one or several arg strings from a Rexx message and deletes their entries. "Count" is the number of entries to be released and can be set to values smaller than 16 in order to save some entries for your own use.

ClosePublicPort() — **Releases a port resource node**

Syntax:

```
ClosePublicPort(node)
             A0
```

Closes a message port and releases its resource node that must have been created with OpenPublicPort().

CreateArgString() — **Creates an arg string structure**

Syntax:

```
argstring = CreateArgString(string,length)
D0                            A0       D0
             A0
            (CCR)
```

Generates an ARexx arg structure and copies the given string into it. The "argstring" is a pointer to the string buffer of the structure and can be treated like a normal string pointer, since it also contains information about string length, structure length, and the hash value with negative offsets in front of the string.

CreateRexxMessage() — Creates an ARexx message structure

Syntax:

```
msgptr = CreateRexxMessage(replyport,extension,host)
D0              A0                    A1        D0
         A0
         (CCR)
```

Generates an ARexx message structure that is a normal EXEC message structure with additional entries for function arguments and return values. The "replyport" is a pointer to a public or private message port and must be specified so the message can be answered. The "extension" and "host" are pointers to strings, values for file recognition and the address of the external environment.

Additional entries in the structure can be inserted later. The interpreter only alters the entries for result1 and result2.

CurrentEnv() — Calculates a pointer to the current active environment

Syntax:

```
envptr = CurrentEnv(rxtptr)
D0                  A0
```

Returns a pointer to the current storage environment that belongs to the given ARexx program. The value "rxptr" is a pointer to the Rexx task structure of the corresponding program and can, for example, be calculated from a message that was sent by this program.

DeleteArgstring() — Releases an arg string structure

Syntax:

```
DeleteArgstring(argstring)
                A0
```

Deletes an ARexx arg structure. The structure contains its own length with a negative offset arg string pointer.

DeleteRexxMsg() — Deletes an ARexx message structure

Syntax:

```
DeleteRexxMsg(packet)
      A0
```

Releases an ARexx message structure. The value contained in the structure is used to determine its size. All arg strings in it must already have been released before the call to this function.

FillRexxMsg() — Fills in arg strings in a Rexx message

Syntax:

```
bool = FillRexxMsg(msgptr,count,mask)
D0                     A0       D0    D1
       (CCR)
```

Converts up to 16 arguments and fills them into the Rexx message "msgptr". The structure must already be initialized and the argument slots must either be pointers to strings closed with zeros or integer values. The "count" indicates the number of argument slots to be converted (usually all of them, except special purpose slots); the bits, 0-15, in "mask" determine if a pointer (0) or an integer (1) is in the slot. The result is -1 if all arguments were successfully converted. If an error occurred, all previously installed arg strings are released and 0 is returned.

FindRsrcNode() — Finds a resource node

Syntax:

```
node = FindRsrcNode(list,name,type)
D0                  A0        A1   D0
      A0
      (CCR)
```

Searches in the given "list" for the first node with the desired "name". The value of "list" must be a pointer to an EXEC list header and "name" must be a string that ends with a zero. If "type" is 0, not all nodes are examined, only those of the given type. The result is a pointer to the node, or 0 if the name was not found.

FreePort() — Closes a message port

Syntax:

```
FreePort(port)
A0
```

Releases all signal bits that belong to a port and closes it. A port must be closed by the same task that opened it, since the arrangement of signal bits is task specific and only available in the task control block. The storage area that belongs to the port is not released.

FreeSpace() — Releases internal storage area

Syntax:

```
FreeSpace(envptr,block,length)
      A0          A1     D0
```

Returns storage areas reserved with GetSpace() to the interpreter. The "envptr" points to the current disk storage environment and can be queried with CurrentEnv().

GetSpace() — Reserves internal storage area

Syntax:

```
block = GetSpace(envptr,length)
D0                 A0       D0
      A0
      (CCR)
```

Reserves a storage area of the interpreter. This storage area is managed by the interpreter and returned to the operating system at the end of the program. "Envptr" points to the current disk storage environment for the program.

This function is also used by the interpreter to obtain small storage areas for string contents; it's always useful for small storage areas that are only needed until the end of the program. The programmer does not have to worry about releasing these storage areas until they get too large.

InitList() — **Builds a list header structure**

Syntax:

```
InitList(list)
    A0
```

Initializes an EXEC list header structure.

InitPort() — **Initializes a message port**

Syntax:

```
(signal,port) = InitPort(port,name)
D0        A1                A0     A1
```

Initializes a message port structure that was previously created. The task ID of the calling program is used in the MP_SIGTASK slot and a signal bit is used. "Signal" is the used bit, or -1 if none were free. The "port" is a pointer to the message port structure and "name" is a pointer to a string that is to be used in the MP_NAME slot. The port address is after the call to A1, which is practical if you want to execute the EXEC function AddPort() in order to make the port public.

See also: FreePort()

IsRexxMsg() — **Tests a message**

Syntax:

```
bool = IsRexxMsg(msgptr)
D0                A0
```

Determines if the message that "msgptr" is pointing to is actually a Rexx message. This is determined by its name: Rexxmessages have a pointer to a hard-coded string containing "REXX" in the LN_NAME slot. The returned value is -1 if it's a Rexx message, otherwise 0.

IsSymbol() — Is the string a valid symbol?

Syntax:

```
(code,length) = IsSymbol(string)
D0     D1                 A0
```

Investigates the given string. If it's a valid ARexx symbol, the corresponding code is returned in D0, or 0 is returned if the string began with an invalid character. The value "length" returns the length of the symbols found.

LengthArgstring() — Calculates the length of an arg string

Syntax:

```
length = LengthArgstring(argptr)
D0                       A0
```

This is the recommended method to determine the length of an arg string. "Argptr" points to the arg string structure; "length" is the length of the strings in it.

ListNames() — Lists node names in an arg string

Syntax:

```
argstring = ListNames(list,separator)
D0                    A0       D0
            A0
            (CCR)
```

Goes through the given list and copies all nodes in it to an arg string structure. The "list" must point to an EXEC list header. The "separator" is an ASCII character inserted between the individual names. While the list is investigated, task switching is shut off with Forbid(). This ensures control of the structures, even for global and system lists. Arg string structures can be released with DeleteArgstring().

LockRexxBase() — Protects a global resource from data write calls

Syntax:

```
LockRexxBase(resource)
             D0
```

Protects the given resource from any data write access. "Resource" is a constant that shows what lock is requested:

RRT_ANY	0	All
RRT_LIB	1	Function libraries
RRT_PORT	2	Public ports
RRT_FILE	3	File IO Buffer (IOBuff)
RRT_HOST	4	External function environment
RRT_CLIP	5	The Clip list

Writing access to global resources are normally handled via the Rexx master procedure which runs with higher priority in order to ensure complete control. This is another reason not to run ARexx programs with higher priority than the Rexx master procedure.

See also: UnlockRexxBase()

OpenPublicPort() **Creates a port resource node**

Syntax:

```
node = OpenPublicPort(list,name)
D0                    A0        A1
      A1
      (CCR)
```

Opens a public message port with the name given in "name" and binds it to the list shown by the header in "list". The message port is also added to the system list of ports. See also: ClosePublicPort().

RemClipNode() **Releases a clip node**

Syntax:

```
RemClipNode(node)
       A0
```

Cuts the given clip node from the clip list and releases the storage area assigned to it. This function is automatically carried out by RemRsrcNode() and RemRsrcList() for a clip node.

See also: AddClipNode(), RemRsrcNode(), RemRsrcList()

RemRsrcList() — **Releases a resource list**

Syntax:

```
RemRsrcList(list)
            A0
```

Releases all nodes in the given list, all of which must be RexxRsrc structures. For each node, the "auto-delete" function is called.

RemRsrcNode() — **Removes a resource node**

Syntax:

```
RemRsrcNode(node)
            A0
```

Removes the given node from its list. If an "auto-delete" function is specified, it's executed first. The name string in it is also released.

UnlockRexxbase() — **Releases a global resource**

Syntax:

```
UnlockRexxBase(resource)
               D0
```

Releases the given resource. Each call to LockRexxBase() should be followed by this counterpart. The definition of the resource constants is explained in the section on LockRexxBase().

10.5 The RexxBase Lists

All structures managed by the Rexx master procedure are noted in the basic structure of the ARexx system library and can be found by other programs. The task list in RexxBase contains a pointer to the global structures for all currently running ARexx programs. Individual task structures are linked by the message ports in them.

The Rexx task structure is the global data structure for an ARexx program and its initial storage environment. All other storage areas are added to the lists contained here. By doing this, the internal data of each ARexx program can be reached using the RexxBase pointer.

There are two functions of the ARexx system library, LockRexxBase() and UnlockRexxBase(). The base structure should always be protected from access with a lock before looking at a list and reading data.

Usually, it's not necessary to access these structures directly, since there are corresponding functions in the ARexx system library for all necessary operations which should be used for that purpose. Direct control is not recommended.

10.6 ARexx Error Messages

If the ARexx interpreter discovers a program error, an error code is returned indicating the nature of the problem. Normally an error code displays the program line in which it was encountered, and a short descriptive error message. If the SYNTAX interrupt was not enabled, the program ends. The SYNTAX interrupt can catch most errors so that the program itself can take counter-measures. Some errors still develop in areas outside of the ARexx jurisdiction and cannot be caught.

There is a value attached to each error code showing the error level that's returned as the primary result. The error code itself appears as the secondary result.

Error code: 1	**Error level: 5**	**Message: Program not found**

The given program could not be found or is not an ARexx program. ARexx programs must always start with "/*". This cannot be trapped with the SYNTAX interrupt.

Error code: 2	**Error level: 10**	**Message: Execution halted**

The program ended because a Ctrl+C break or an external HALT request was given. This error can be caught with the HALT interrupt.

Error code: 3	**Error level: 20**	**Message: Insufficient memory**

The interpreter was unable to receive enough memory space for an operation. Since all operations of the interpreter usually need some storage access, this error cannot usually be caught with the SYNTAX interrupt.

Error code: 4	Error level: 10	Message: Invalid character

Invalid characters were located in the source code. Control codes and other special characters can only be used in hexadecimal or binary strings within a program. This error cannot be caught with the SYNTAX interrupt.

Error code: 5	Error level: 10	Message: Unmatched quote

A string delimiter (' or ") is omitted. Each string must be enclosed with the same character with which it began. This error cannot be caught with the SYNTAX interrupt.

Error code: 6	Error level: 10	Message: Unterminated comment

The characters ("*/") that indicate the end of a comment, were not found. Please note that comments can be nested, so every "/*" must be followed by a "*/". This error cannot be caught with the SYNTAX interrupt.

Error code: 7	Error level: 10	Message: Clause too long

A clause was too long to be written to the interpreter's internal interim storage area. The maximum length (without multiple spaces and commentaries) is 800 characters. The questionable clause should be divided into two or more parts. This error cannot be caught with the SYNTAX interrupt.

Error code: 8	Error level: 10	Message: Invalid token

An invalid token was encountered or a clause could not be classified. This error cannot be caught with the SYNTAX interrupt.

Error code: 9	Error level: 10	Message: Symbol or string too long

An attempt was made to generate a string with more than 65,535 characters.

Error code: 10	Error level: 10	Message: Invalid message packet

In a message received by the Rexx master procedure, an invalid action code was encountered. It was returned with no changes. This error is externally created and cannot be caught with the SYNTAX interrupt.

Error code: 11	Error level: 10	Message: Command string error

A command string was incorrect. This error is externally created and cannot be caught with the SYNTAX interrupt.

Error code: 12	Error level: 10	Message: Error return from function

An external function returned an error code not equal to zero. It's possible that the parameters were not correctly passed.

Error code: 13	Error level: 10	Message: Host environment not found

The message port indicated by an address was not found. If the name is correctly written (including capitalization), is the desired function environment active?

Error code: 14	Error level: 10	Message: Requested library not found

The program was not able to open a library entered in the library list. If ADDLIB() was called with the correct name, was the correct version number called? Is the library in the LIBS: directory?

Error code: 15	Error level: 10	Message: Function not found

A function was called that was not in any of the libraries added with ADDLIB() and also not found as an external program. Is the spelling correct? Was the library bound with ADDLIB() to the list?

Error code: 16	**Error level: 10**	**Message: Function did not return value**

A function was completed without delivering a result string and without encountering an error. Was the function correctly programmed? If it was accessed with CALL this can be avoided.

Error code: 17	**Error level: 10**	**Message: Wrong number of arguments**

A function expecting more or fewer arguments was called. This error also occurs if a built-in or an external function is called with more arguments than the message can contain (max. 15).

Error code: 18	**Error level: 10**	**Message: Invalid argument to function**

An argument that does not agree with the function was passed, or a necessary argument was omitted.

Error code: 19	**Error level: 10**	**Message: Invalid procedure**

A procedure call occurred at the wrong location. Either it was not in an internal function, or it occurred twice in a function.

Error code: 20	**Error level: 10**	**Message: Unexpected THEN or WHEN**

A THEN or WHEN command occurred at the wrong location. The WHEN command is only valid within the area of a SELECT command and THEN must directly follow an IF or WHEN.

Error code: 21	**Error level: 10**	**Message: Unexpected ELSE or OTHERWISE**

An ELSE or OTHERWISE command occurred at the wrong location. An OTHERWISE command is only valid within the area of a SELECT command. ELSE is only available after a THEN branch of an IF command.

Error code: 22	**Error level: 10**	**Message: Unexpected BREAK, LEAVE, or ITERATE**

The BREAK command is only valid in a DO group or in commands that are executed with INTERPRET. Commands to LEAVE or ITERATE are only valid in a DO loop.

Error code: 23	**Error level: 10**	**Message: Invalid statement in SELECT**

In the area of a SELECT command, an illegal construction was encountered. Only WHEN-THEN and OTHERWISE constructions are valid.

Error code: 24	**Error level: 10**	**Message: Missing or multiple THEN**

A THEN clause was expected, but not found, or a THEN appeared without IF or WHEN.

Error code: 25	**Error level: 10**	**Message: Missing OTHERWISE**

No WHEN clause in the area of a SELECT command was successful and no OTHERWISE was found.

Error code: 26	**Error level: 10**	**Message: Missing or unexpected END**

The source text ended without closing a DO or SELECT group with END, or an END clause was found outside such a group.

Error code: 27	**Error level: 10**	**Message: Symbol mismatch**

The symbol specified with an END, ITERATE, or LEAVE command did not agree with the index variable of the appropriate DO group.

Error code: 28	Error level: 10	Message: Invalid DO syntax

The interpreter found an error in a DO command: If TO or BY are specified, the index variable must be initialized and the expression after FOR must evaluate to a positive integer.

Error code: 29	Error level: 10	Message: Incomplete IF or SELECT

An IF or SELECT group ended before all of the necessary constructions were encountered. Perhaps a THEN, ELSE, or OTHERWISE construction is omitted.

Error code: 30	Error level: 10	Message: Label not found

A jump marker specified in a SIGNAL command or searched for with a SIGNAL interrupt, could not be found in the source code. Interactive commands or marks established in an interpreter command are usually not found.

Error code: 31	Error level: 10	Message: Symbol expected

At a location where only a symbol is appropriate, an invalid token was found. The commands DROP, END, LEAVE, ITERATE and UPPER can only be followed by symbols and create this message if anything but a symbol is found or a necessary symbol is omitted.

Error code: 32	Error level: 10	Message: Symbol or string expected

At a location where only a symbol or string is permitted, an invalid token was found.

Error code: 33	Error level: 10	Message: Invalid keyword

A symbol in a command was recognized as a key word but is not valid at this location.

Error code: 34	Error level: 10	Message: Required keyword missing

A certain keyword was expected by a command and was not found. This message occurs if none of the keywords for the individual interrupts (such as SYNTAX) follows a SIGNAL ON command.

Error code: 35	Error level: 10	Message: Extraneous characters

A seemingly correct command was executed but further characters were found following it.

Error code: 36	Error level: 10	Message: Keyword conflict

Two mutually exclusive keywords occurred in the same command or a key word was encountered twice.

Error code: 37	Error level: 10	Message: Invalid template

The template specified in an ARG, PARSE, or PULL command was invalid.

Error code: 38	Error level: 10	Message: Invalid TRACE request

The keyword for a TRACE command or an argument for the TRACE() function was not valid.

Error code: 39	Error level: 10	Message: Uninitialized variable

An attempt was made to read an uninitialized variable. This message appears only when the NOVALUE interrupt is enabled.

Error code: 40	Error level: 10	Message: Invalid variable name

An attempt was made to assign a value to a constant.

Error code: 41	Error level: 10	Message: Invalid expression

During an evaluation of an expression, an error occurred. Possibly an operator was not used correctly or invalid characters appeared. This error only appears when an expression is analyzed; expressions that are jumped over are not checked.

Error code: 42	Error level: 10	Message: Unbalanced parentheses

An expression was encountered that did not have the same number of open and close parentheses marks.

Error code: 43	Error level: 10	Message: Nesting limit exceeded

The number of nested sub-expressions was higher than 32. The expression should be divided into several partial expressions.

Error code: 44	Error level: 10	Message: Invalid expression result

The result of an expression was not valid. This error is created if an expression in a DO command does not lead to a numeric result.

Error code: 45	Error level: 10	Message: Expression required

An expression is omitted in a necessary location. An example is that after SIGNAL an expression must follow, unless ON or OFF was specified.

Error code: 46	Error level: 10	Message: Boolean value not 0 or 1

The result of an expression should be a Boolean result, but a value that is not 0 or 1 occurred.

Error code: 47	**Error level: 10**	**Message: Arithmetic conversion error**

During an operation that requires numeric operands, a non-numeric operand was encountered. A hex or binary string with errors also leads to this error message.

Error code: 48	**Error level: 10**	**Message: Invalid operand**

An operation was attempted with an invalid operand. This error occurs when dividing by 0 or when trying to display fractional Exponents (that are not supported by ARexx).

Part 3

A3000 Intern

11. The A3000 Hardware

The Amiga 3000 is the first completely new model Commodore has introduced since the Amiga 1000. Unlike the A500 and A2000, which hardly differed from their predecessor technologically, the A3000 is a truly new development, capable of holding its own against the Intel 80386-based IBM-compatible personal computers; in some areas it even surpasses them.

The most important innovation is the departure from the 68000 as central processor. Powerful as this chip was in comparison to its counterpart, Intel's 80286, a databus width of 32 bits has since become the standard. In fact, the first 64-bit microprocessor has already appeared -- the 80860 introduced in 1989 by Intel.

By its decision to base the new Amiga on the 68030 processor, Commodore has achieved the best possible compromise between price, performance and, perhaps most importantly, compatibility. Commodore's software developers don't have to worry about compatibility problems between the 68000 and the 68030.

It is safe to assume that the majority of existing software will run on the new machine. Any problems are more likely to be related to the new Kickstart 2.0 operating system than to the new hardware, and fortunately the A3000 will also run Kickstart 1.3.

"Dirty" programs unable to cope with the 6- to 8-fold increase in computing speed can always turn to the GURU generator. This same problem presented itself earlier, however, with the widespread use of 68020 and 68030 cards in the A2000. So programmers had enough time, before the A3000 was introduced, to correct bad programming habits learned on the C64 and adapt their software to the new generation of computers.

Linked with the new processor are the FPU 68881 and 68882 (in the 16 and 25 MHz models, respectively). These floating point processors speed mathematical routines and give the A3000 a "computing" power (in the truest sense of the word) that's suitable for a scientific workstation.

Another important improvement over the A2000 is the built-in hard disk and its SCSI bus, which also enables the addition of CD-ROM or tape drives. The 32-bit SCSI chip, developed by Commodore specifically for this purpose, offers adequate speed required for the operation of modern storage media.

The hard disk itself comes from Quantum. This company has succeeded in producing a drive that is not only fast and reliable, but also quiet. Any user who has dealt with the grinding, screeching or whistling of less adeptly engineered examples of mass storage technology will certainly appreciate this. (Unfortunately, the A3000 still has the old vacuum cleaner noise, although it does run at a whisper compared to the A2000.)

Technically speaking, little has changed in the area of graphics. However, there is one exception so crucial that it could be considered the A3000's single most important innovation. This is the flicker fixer, which alleviates the flickering that may occur when a screen is displayed in interlace mode by temporarily storing the individual half-pictures (frames). Many A2000 owners have purchased such a flicker fixer because it significantly improves the display quality.

Moreover, integration of the Enhanced Chip Set (ECS), which consists of a substantially improved Agnus chip and a new Denise, produces, even without the flicker fixer, a 640 * 480 screen with a refresh rate of 60 Hz, although this is with a maximum of only four colors.

All the new features mentioned above are explained in detail in the following sections, both from a hardware standpoint and from the programmer's point of view. The familiar A2000 features that have not changed in the A3000 are also discussed in detail.

11.1 Processor Generations

The heart of the A3000, the 68030 microprocessor, is the product of a continuing development process that began with the 68000 in 1979. So far this process has culminated in the 68040 and further advances are sure to come. Motorola has succeeded (or nearly so, at least) in maintaining software compatibility across this entire line of processors. The performance of the new models isn't significantly affected when running old programs. This is definitely a worthwhile accomplishment. Motorola's competitor, Intel, hasn't been able to do the same with the 86 processor series for the PC.

The Forefather: MC 68000

When it was introduced, the MC 68000 was a pioneering product. With astounding foresight, its developers gave it attributes that would make it the forefather of an entire processor family. Specifically, these attributes are:

A universal register structure

The entire 68000 family has eight data registers and eight address registers, all with a width of 32 bits. Except for the distinction between address and data registers, there is no connection between a register and the functions for which it may be used. This differs many other processors, where, for example, an accumulator is designated specifically for computation results, or an index register specifically for table addressing. With such a design, moving data from one register to another is largely eliminated. The greater register size also makes it unnecessary, in computations with integers, to divide a value over more than one register. Almost all computations can be executed in a single instruction. This structure was retained with the 68020, since it is fully adequate for a true 32-bit processor also.

Large linear address space

Although the address space of the 68000 is only 16 Megabytes, all its address registers are 32 bits wide. There is no addressing limitation, so that accommodating the 68020's address space of four Gigabytes (an amount still generous by today's standards) was a simple matter of bringing in eight more address lines. A data field can be quickly accessed by loading any address register with the desired base address and referencing the data by means of a 16-bit displacement value added to the base. This saves time because only 16 bits, not 32, must be loaded from memory. This scheme combines the advantages of a large linear (non-segmented) address space and quick access to contiguous data.

Many types of addressing

Besides the "normal" types of addressing, such as absolute, indirect or immediate, handled by almost all processors, the 68000 is capable of indirect addressing with displacement as well as PC-relative addressing, in which data is referenced relative to an instruction address. This also saves time, since again not all 32 address bits are required. Another type of addressing that distinguishes the 68000 from its competitors is postincrement/predecrement addressing. In this method, automatic increasing or decreasing of addresses with each data access allows any address register to function as a stack. When processing sequential data fields with postincrement or predecrement addressing, you save an instruction by not having to compute the next address.

Other types of addressing were also introduced with the 68020. These are primarily capable of speeding up programs written, for example, in C, with its improved compilers. A list of all the types of addressing is located in Section 11.2.

Team performance: 68020, FPU & MMU

Besides the widening of the address and data buses to their current 32 bits, another improvement in the 68020 was the addition of a universal coprocessor interface and the accompanying coprocessor, the FPU (Floating Point Unit) 68881. The 68020 completely takes over the addressing of instructions and data for its coprocessor. Machine-language instructions for the FPU are simply mixed with those of the main processor. From a software standpoint, the 68020 forms a closed unit with its coprocessor. The exact design and programming of this chip are described in more detail later in this chapter. The 68020/30 + FPU team is adept at screen processing and other computation-intensive tasks.

The 68020 system has another coprocessor in the form of the Paged Memory Management Unit (PMMU). This unit is responsible for controlling memory access of the various processes by creating a virtual address space for each one. You're probably already familiar with the GURU. Your experiences should help demonstrate the usefulness of this concept. Taking C as an example, suppose an uninitialized pointer is used to assign a value to a variable. It will erroneously point to an address in memory determined by the value in the stack where it was initially set by the compiler. If this is an area of memory being used by the operating system, the GURU will come to call.

The only protection the 68000 offers against such encroachment is the differentiation between supervisor and user mode. Memory can be divided by hardware means into two parts, one of which can be accessed only in supervisor mode. This technique, though, has two serious disadvantages. First, while the operating system is now safe, user programs can still be clobbered, which makes a multitasking, and obviously a multiuser system, impossible.

Secondly, such a technique requires a fixed and permanent dividing of memory, in which expensive RAM cannot be used to its fullest advantage. The maximum amount needed by the operating system must always be reserved for it in the supervisor area. Though part of this is used only occasionally, it is never available to user programs. In short, flexible memory management adapted to changing requirements is not possible.

For these reasons, little use has been made of this capability of the 68000. Even in the Amiga, supervisor and user memory are identical, with the disadvantage that any task can crash the system.

In the 68020 and 68030, this problem is solved by the PMMU. Switching between processor and memory, it checks every access, providing protection to all areas from even the most ill-mannered task.

Actual computing functions are also faster in the 68020. A barrel shifter was integrated for shift operations, making them equally fast, whether a register is shifted by a single bit or by 15 bits.

Since the 68020 functions at a clock frequency of 20 MHz and is internally faster, it processes significantly more data and instructions from main memory than the 68000. RAM must be very fast in order for the processor to work at maximum speed. You would also like RAM to be as large as possible. Unfortunately these two requirements can add up to considerable expense. For this reason, the 68020 includes a cache memory. This is a small, fast storage area (64 long words or 256 bytes) in which the most recently used instruction is saved. With a second reference to this instruction, for example another execution of a loop, the 68020 can fetch the instruction directly from the cache without having to access memory. This caching enables the processor to utilize less expensive memory chips with almost the same speed as would be attained with fast RAM.

11.2 The 68030

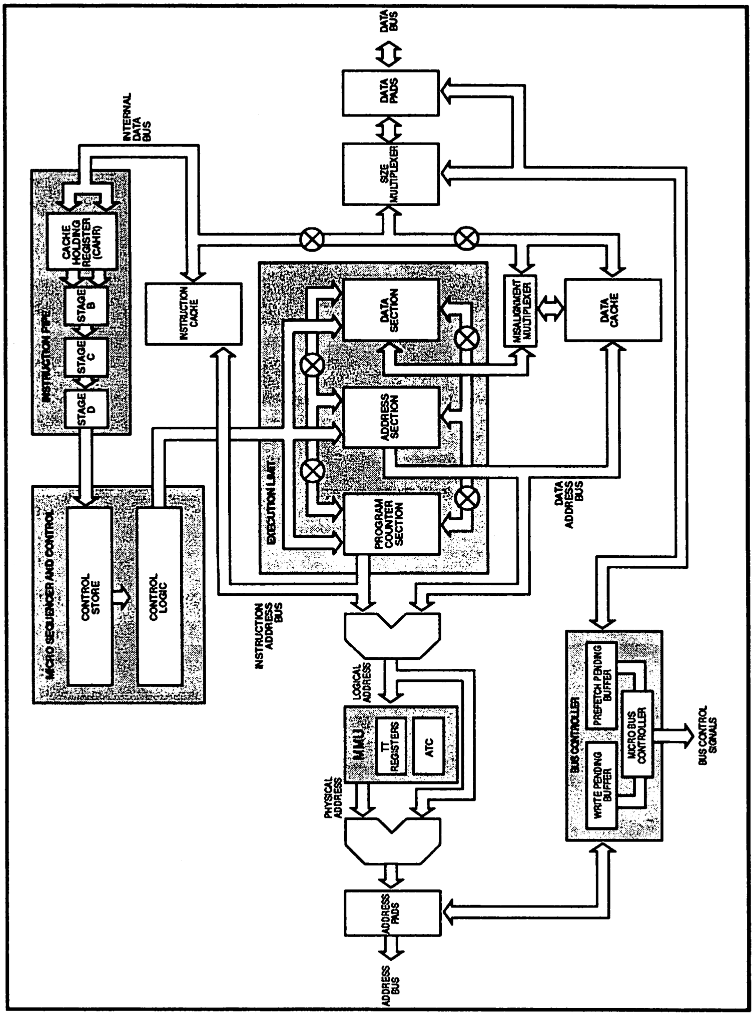
Because of the advances in semiconductor technology, more function blocks can be included on a single chip. The MMU, which was a coprocessor for the 68020, is now integrated into the 68030. Furthermore, the 256-byte instruction cache is accompanied in the 68030 by a data cache of the same size.

The bus controller, which manages communication between memory and the CPU, is also improved. It can now move data into the cache independent of the processor and, with sufficiently fast RAM, transfer data over the bus at a rate almost double that of the 68020 running at the same clock frequency.

This concept of parallel processing is actually what distinguishes the 68030 from its predecessor. Because of the separation of the address and data buses between the caches and the processing unit (Harvard Architecture), instructions can be processed partially in parallel. While the last operation's data is still being processed, the next instruction is decoded and prepared. The bus controller loads the data from RAM and the MMU translates and validates the addresses.

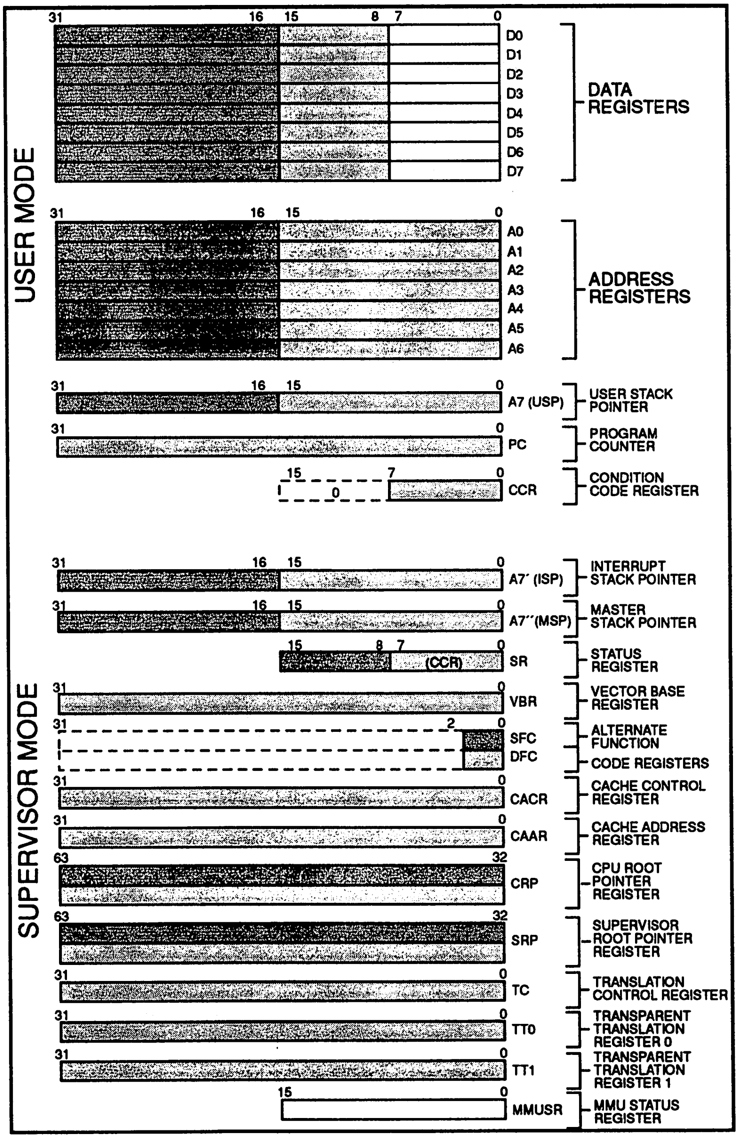
The following sections describe the various function blocks and their programming. However, since the subject of this book is the Amiga instead of the 68030, we won't provide detailed information. We will discuss primarily the differences and improvements that distinguish the 68030 from the 68000.

If you're not familiar with machine language, refer to one of the many books on programming the 68000. This should help you understand the instructions and addressing types that are new to the 68030.



The Architecture of the 68030

The Program Model



Program Model

The illustration shows all the registers of the 68030. Those under the heading 'User Mode Program Model' are identical to those of the 68000: eight each of the universal data and address registers (A7 is the stack pointer), the Program Counter and the Condition Code Register. These are the only registers that can be referenced in user mode.

Some new registers have been added for supervisor mode. The 68000 had two stack pointers, the User Stack Pointer (USP, A7) and the Supervisor Stack Pointer (SSP, A7'). In the 68030 the latter has been further divided into an Interrupt Stack Pointer (ISP, A7') and a Master Stack Pointer (MSP, A").

The following registers are new in the 68030:

VBR — Vector Base Register: With this register the base address of the Exception Vector Table can be set to any desired value (in the 68000 this was always 0).

SFC — Alternate Function Code Registers

DFC — SFC and DFC stand for Source and Destination Function Code, respectively. These registers permit explicit selection of the address region to be accessed by a MOVE instruction.

Five address regions are distinguished: User Program, Supervisor Program, User Data, Supervisor Data and Processor (the Processor address region is used for communication with the coprocessor and hardware, for example in fetching interrupt vectors, etc.).

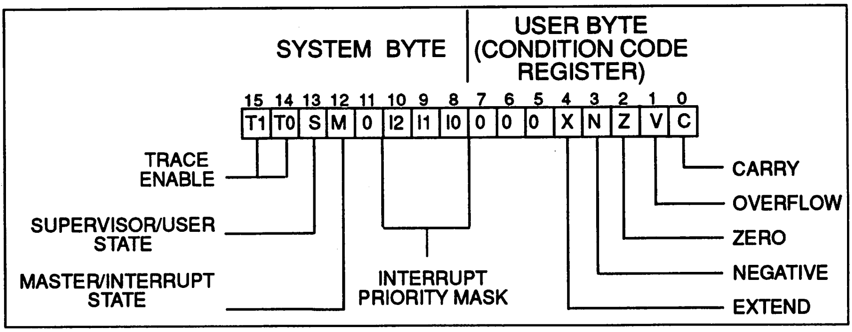
Data can be easily copied between the various address regions by means of MOVE instructions using the Alternate Function Code Registers.

CACR — Cache Control Register

CAAR — Cache Address Register

The previous two registers control the functioning of the integrated data and address caches.

CRP	CPU Root Pointer
SRP	Supervisor Root Pointer
TC	Translation Control Register
TT0	Transparent Translation Register 0
TT1	Transparent Translation Register 1
MMUSR	MMU Status Register
	The previous six registers belong to the MMU.
SR	Status Register



The Status Register

The lower byte of this register contains the condition code and is therefore referred to as the Condition Code Register (CCR). The CCR can be referenced in user as well as supervisor mode. The upper half of the Status Register, the System Byte, contains important system flags:

Interrupt Priority Mask

Like the 68000, the 68030 distinguishes even interrupt levels by priority, from a lowest priority of 1 to a highest of 7.

By means of the three bits in the Interrupt Priority Mask, all interrupts up to and including a certain priority can be disabled. Only interrupts of priority higher than the number in the mask will be executed. Level 7 is an exception. An interrupt of this priority is referred to as a Non-Maskable-Interrupt (NMI) and cannot be disabled.

Trace Enable

These two bits control the processor's trace mode (see Exceptions).

Supervisor Bit

When this bit is set, the 68030 is in supervisor as opposed to user mode. From within user mode this bit can be set only by an exception, since direct access to the Status Register's System Byte is not permitted in user mode.

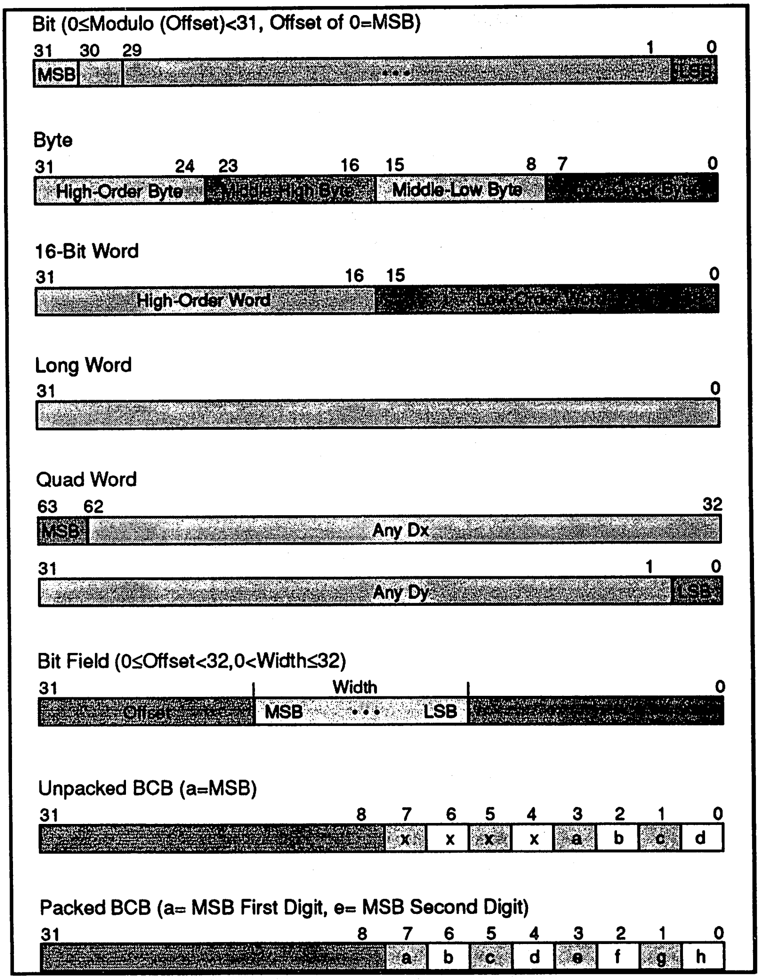
Master Bit

The Master Bit distinguishes between the two supervisor stack pointers. If this bit is NULL, the CPU uses the Interrupt Stack Pointer. If it is set, the CPU uses the Master Stack Pointer for all operations, provided that the Supervisor Bit is also set, and switches to the ISP only with an interrupt.

CCR The Condition Code Byte is the lower half of the Status Register. It contains the following five flag bits:

C	Carry	Carry from the MSB (most significant, or last, bit)
V	Overflow	Carry from the next to last bit
Z	Zero	Result equals zero
N	Negative	Result is negative (MSB = 1)
X	Extend	Like Carry, but only set in arithmetic operations

Every instruction that alters data sets these flags according to the result. These bits can be used as decision criteria for instructions that control program flow, as in the Bcc (Branch on condition code) instruction.



Data Types

The fundamental data type of the 680xx is the integer number. It can have a size of 8 bits (one byte), 16 bits (one word) or 32 bits (one long word). When placed in a data register (32 bits), an 8- or 16-bit operand is loaded into the lower half or quarter of the register.

Address registers can contain only word or longword data types. A 16-bit value written to an address register is expanded to 32 bits based on

Unlike the 68000, the 68030 can reference all data types on byte addresses. It is no longer necessary to align a word or longword operand to an address that corresponds to a multiple of its size. Nevertheless, it is always faster to align data fields by the number of bytes they occupy. In a hardware sense, the 68030 always reads one long word at a time (a transfer of 32 bits per bus cycle over the 32-bit data bus). The data bus is aligned, however, according to longword addresses. Two bus cycles are thus required to read a long word that begins on an odd word address (an address that is not a multiple of four).

Aligned:

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
Word0	Word2	Word4	Word6				
--	Longword0	--	--	Longword4	--		

Misaligned:

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
--	Word1	Word3	--	Word5--	Word7		
--	--	Longword1	--	--	Longword5		--

Therefore, although it is possible to work with misaligned data on the 68030, a trade-off in terms of reduced speed will likely result.

In addition to the integral data type there is also the bit. A bit can be read, set or cleared. An individual bit is referenced either by its byte address and bit number (0 through 7) or, when it is in a register rather than in memory, by the register name and bit number (here 0 through 31). The least significant bit is numbered 0.

As an extension of the bit data type, the bit field was introduced with the 68030. The bit field is a sequence of up to 32 bits, that don't have to begin on a byte boundary. Besides a memory address or register name, an offset is required to reference such a field. It indicates the beginning of the bit field relative to the most significant bit of the address or register. The offset is a signed quantity, meaning that the field can also begin before the base address.

The size of the bit field can be from one to 32 bits. Within the field, bits are numbered in the opposite direction from that of integral numbers. Bit number 0, whose position is determined by the offset, is the most significant bit of the field.

A new integer data type called the quad word has been added. This is simply a double long word (64 bits). Since registers are 32 bits, two of them are required for a quad word.

The quad word is used only in multiplication and division. It cannot be simply used in place of the other integer types. By means of the MOVEM instruction, however, it is possible to move a quad word between registers and memory. (Naturally, this is done in two normal MOVE.L instructions.)

One last data type exists, the binary coded decimal (BCD) numbers. A BCD number is stored in the base-ten system, using only values between 0 and 9. The 68030 distinguishes between packed and unpacked BCD values. In unpacked format, one byte contains one digit; the upper four bits are always null. In packed format, both half-bytes (nibbles) are used.

Types of Addressing

Addressing capabilities are greatly expanded in the 68030 as compared to the 68000, although the traditional types of addressing are still the most frequently used.

Register Direct
The desired operand is the register itself.

Syntax: Rn

(Rn is any address or data register number)

Absolute: The operand address is given immediately after the instruction.

Syntax: Address.W or Address.L

(With the Address.W (Address.Word) format, the address is expanded to 32 bits based on the sign of the 16-bit word value, thus referencing only the first or last 32K of memory.)

Immediate: The byte, word or long word following the instruction is the operand.

Syntax: #Operand

Address Register Indirect:

The value in the given register contains the address of the operand in memory.

Syntax: (An)

(An is any address register number)

Address Register Indirect with Postincrement or Predecrement:

In postincrement addressing, the address of the operand in the corresponding register is incremented by the size of the operand in bytes. This takes place AFTER the operand has been processed.

In predecrement addressing, the address is decremented by the size of the operand BEFORE the instruction is executed. Thus the address is incremented or decremented by one for byte, two for word and four for longword operands. (When this type of addressing is used for coprocessor data types, this value depends on their size.)

Syntax: (An)+ (Postincrement)
-(An) (Predecrement)

Address Register Indirect with Displacement:

A value specified with the instruction (the displacement) is added to the memory address from the address register. The result is the operand address in memory.

Since the displacement is a signed 16-bit number, this addressing method spans an area of 32K before and after the address contained in the register.

Syntax: (d_{16},An)
(d_{16} - 16-bit displacement)

Address Register Indirect with Index and 8-bit Displacement:

This method is slightly more complicated. First the displacement is added to the value in the address register.

Then the CPU multiplies the index, which may be in any address or data register, by a factor of 1, 2, 4 or 8, and finally adds the two results.

	Value from Address Register
+	Displacement
+	Value from Index Register (.W,.L) * 1, 2, 4 or 8
=	Address of Operand

Syntax: $(d_8$,An,Rn.SIZE * SCALE)

(Rn.SIZE * SCALE - any register number with SIZE of 16 or 32 bits, multiplied by SCALE value of 1,2,4 or 8.)

Address Register Indirect with Index and Base Displacement:

This method of addressing closely resembles the previous one, except that the displacement may be 16 or 32 bits. Also, some elements may be left out. Address register, base displacement and index register are optional. If the address register and displacement are omitted, for example, and a data register is used as the index register, the result is data register indirect addressing, which is normally not available.

Syntax: (Bd,An,Rn.SIZE * SCALE)
(Bd - base displacement)

Memory Indirect with Postindex:

Also new with the 68030 are the memory indirect modes. These methods first compute an address in memory, from which an operand address is then read. In this case there are two displacements: the base displacement and the outer displacement. The former is added to the address register in computing the memory address, the latter is added to the operand address. The index register is likewise added to the operand address after multiplication by a scaling value as described earlier. This is why the term postindex is used.

This method can also omit certain elements, and the CPU will assume them to be zeros.

	Value from Address Register
+	Base Displacement
=	Memory Address
	Value from above Memory Address
+	Value from Index Register (.W,.L) * 1, 2, 4 or 8
+	Outer Displacement
=	Operand Address

Syntax: ([Bd,An],Rn.SIZE*SCALE,Od)
(Bd - base displacement, Od - outer displacement)

The square brackets indicate the memory address from which the operand address is read.

Memory Indirect with Preindex:
This method is identical to the above, except that the index is added to the address register instead of the operand address.

	Value from Address Register
+	Value from Index Register (.W,.L) * 1,2,4 or 8
+	Base Displacement
=	Memory Address
	Value from above Memory Address
+	Outer Displacement
=	Operand Address

Syntax: ([Bd,An,Rn.SIZE*SCALE],Od)

PC-relative Addressing:
The following addressing methods can use the Program Counter (PC) instead of an address register as a base value:

- indirect with displacement
- indirect with index and 8-bit displacement
- indirect with index and base-displacement
- memory indirect with postindex
- memory indirect with preindex

All access then takes place not in the User or Supervisor Data region but in the corresponding Program region. These addressing methods are thus suited mainly for quick reference to nonvariable data, for example constants, which reside with the program in the code segment (see Operating System). PC-relative memory access always refers to data that resides a certain distance from the current instruction, independent of the instruction's memory address.

Instructions

The majority of instructions for the 68030 are those that applied to the 68000, with some enhancement of addressing capabilities or additional data types. For example, two long words can be multiplied to produce a quad word as the result.

New instructions are the bit field operations, some system control instructions, multiprocessor instructions and, of course, the cache, coprocessor and MMU commands.

Data Transfer Instructions

Instruction	Syntax	Size	Comments
EXG	Rn,Rn	32	Rn <- ->Rn
LEA	<ea>,An	32	<ea> -> An
LINK	An,<d>	16,32	Sp - 4 -> Sp: An -> (SP): SP ->atw12 An, SP + D -> SP
MOVE	<ea>,<ea>	8,16,32	source -> dest
MOVEA	<ea>,An	16,32 -> 32	
MOVEM	list,<ea>	16,32	Register list -> dest
	<ea>,list	16,32 -> 32	source -> Register list
MOVEP	Dn,(d16,An)	16,32	Dn(31:24) -> (An + d): Dn(23:16) -> An + d +2: Dn(15:8) -> (An + d + 4): Dn(7:0) -> (An + d + 6)
	(d16,An),Dn		(An+d) -> Dn(31:24): (An+d+2) -> Dn(23:16): (An + d + 4) -> Dn(15:8): (An + d + 6) -> Dn(7:0)
MOVEQ	#<data>,Dn	8 -> 32	direct data -> dest
PEA	<ea>	32	SP-4 -> SP: <ea> -> (SP)
UNLK	An	32	An -> SP: (SP) -> An: SP + 4 -> SP

Remarks:

- The MOVEP instruction (Move Peripheral) transfers data between the processor and peripheral components having only an 8-bit data bus. Although it isn't needed on the 68030, this instruction is retained for compatibility with the 68000.

  In the 68000 with its 16-bit bus width, two consecutive byte registers occupy consecutive word, not byte, addresses. This means that alternate bytes are unused. Four MOVE.B instructions are required to move a long word into as many consecutive registers. MOVEP increments the address to the next word with each byte and skips over the gaps. Thus a word or long word can be written to consecutive registers with a single instruction.

  The 68030's bus width is dynamic. With every bus cycle it determines whether the address it is about to access is 8, 16, or 32 bits long. Thus even 8-bit chip registers have consecutive addresses in memory.

- <data> in MOVEQ (Move Quick) represents an 8-bit value; this is expanded to 32 bits.

Arithmetic Operations

All arithmetic operations work with signed integers. ADDX, SUBX and NEGX computations include the X-Flag. By multiple executions of these instructions, computations can be performed on numbers larger than one long word.

Instruction	Syntax	Size	Comments
ADD	Dn,<ea>	8,16,32	source+dest -> dest
	<ea>,Dn	8,16,32	
ADDA	<ea>,Dn	16,32	
ADDI	#<data>,<ea>	8,16,32	direct data + dest -> dest
ADDQ	#<data>,<ea>	8,16,32	
ADDX	Dn,Dn	8,16,32	source + dest + X -> dest
	-(An),-(An)	8,16,32	
CLR	<ea>	8,16,32	0 -> dest
CMP	<ea>,Dn	8,16,32	dest - source
CMPA	<ea>,An	16,32	
CMPI	#<data>,<ea>	8,16,32	dest - direct data
CMPM	(An)+,(An)+	8,16,32	dest - source
CMP2	<ea>,Rn	8,16,32	
DIVS/DIVU	<ea>,Dn	32/16 -> 16:16	dest/source -> dest (with or without sign)
	<ea>,Dr:Dq	64/32 -> 32:32	
	<ea>,Dq	32/32 -> 32	
DIVSL/DIVUL	<ea>,Dr:Dq	32/32 -> 32:32	
EXT	Dn	8 -> 16	with sign extended dest -> dest
	Dn	16 -> 32	
EXTB	Dn	8 -> 32	
MULS/MULU	<ea>,Dn	16 x 16 -> 32	source x dest -> dest (with or without sign)
	<ea>,Dl	32 x 32 -> 32	
	<ea>,Dh:Dl	32 x 32 -> 64	
NEG	<ea>	8,16,32	0 - dest -> dest
NEGX	<ea>	8,16,32	0 - dest - X -> dest
SUB	<ea>,Dn	8,16,32	dest - source -> dest
	Dn,<ea>	8,16,32	
SUBA	<ea>,An	16,32	
SUBI	#<data>,<ea>	8,16,32	dest - direct data -> dest
SUBQ	#<data>,<ea>	8,16,32	
SUBX	Dn,Dn	8,16,32	dest - source - X -> dest
	-(An),-(An)	8,16,32	

Remarks:

- <data> in SUBQ and ADDQ must be 0 through 7.

- In multiplication and division (MUL, DIV) quad words can also be used (64 bits). In this case, two data registers are declared instead of one.

Logical Operations

These instructions perform the logical linking functions (And, Or, Exclusive-or and Negation).

The TST instruction subtracts 0 from an operand and sets the appropriate condition codes in the Status Register. This can be used to test a value in memory for zero.

Instruction	Syntax	Size	Comments
AND	<ea>,Dn	8,16,32	source / dest -> dest
	Dn,<ea>	8,16,32	
ANDI	#<data>,<ea>	8,16,32	Data / dest -> dest
EOR	Dn,<data> <ea>	8,16,32	source EOR dest -> dest
EORI	#<data>,<ea>	8,16,32	Data EOR dest -> dest
NOT	<ea>	8,16,32	dest -> dest
OR	<ea>,Dn	8,16,32	source OR dest -> dest
	Dn,<ea>	8,16,32	
ORI	#<data>,<ea>	8,16,32	Data OR dest -> dest
TST	<ea>	8,16,32	source - 0 to set cond.codes

Shift and Rotation Instructions

Shift and rotation instructions differ in whether or not bits displaced from one end of the operand are brought around to the other. ROXR and ROXL also include the X-Bit of the Status Register in the rotation.

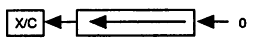
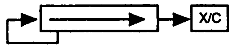
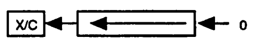
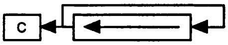
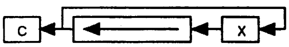
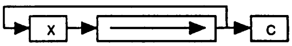
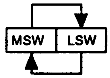
Shift instructions differ with regard to the MSB. In the arithmetic shift instruction it is interpreted as a sign and retained when shifting right, while in the logical variant it is replaced by zero.

When shifting left there is no difference between the arithmetic (ASL) and the logical (LSL) version of the instruction.

With all these instructions one declares the number of bits to be shifted, followed by a data register. Permissible values are 1 through 8 with immediate addressing and 1 through 63 when using a data register.

A memory location can also be shifted directly, but only as a word and only by one bit at a time.

The swap instruction switches the two words in a data register.

Instruction	Operand Syntax	Operand Size	Operation
ASL	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
ASR	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
LSL	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
LSR	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
ROL	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
ROR	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
ROXL	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
ROXR	Dn, Dn # (data), Dn (ea)	8, 16, 32 8, 16, 32 16	
SWAP	Dn	32	

Shift and Rotation Instructions

Bit Data Type Instructions

All bit manipulation instructions set the Zero-Flag according to the condition of the selected bit. Then the bit is either cleared (BCLR), set (BSET) or inverted (BCHG). The bit number can be declared immediately or in a data register.

Instruction	Operand Syntax	Operand Format/Size	Operation
BCHG	Dn,<ea>	8,32	- <bit number> from dest->Z->bit of dest
	#<data>,<ea>	8,32	
BCLR	Dn,<ea>	8,32	- <bit number> from dest ->Z;
	#<data>,<ea>	8,32	0->bit of dest
BSET	Dn,<ea>	8,32	- <bit number> from dest ->Z;
	#<data>,<ea>	8,32	1->bit of dest
BTST	Dn,<ea>	8,32	- <bit number> from dest ->Z
	#<data>,<ea>	8,32	

Bitfield Instructions

The bitfield commands transfer the MSB of the field to the N flag and set the Zero flag if all bits of the field are null. Then the corresponding operation is performed.

Instruction	Syntax	Size	Comments
BFCHG	<ea> (offset:length)	1-32	Field -> Field
BFCLR	<ea> (offset:length)	1-32	0 -> Field
BFEXTS	<ea> (offset:length),Dn	1-32	Field -> Dn ;extend sign
BFEXTU	<ea> (offset:length),Dn	1-32	Field -> Dn ;extend unsign (zero)
BFFFO	<ea> (offset:length),Dn	1-32	Searches for first set bit in field; offset -> Dn
BFINS	Dn,<ea> (offset:length)	1-32	Dn -> Field
BFSET	<ea> (offset:length)	1-32	1 -> Field
BFTST	<ea> (offset:length)	1-32	MSB -> N (OR all)-> Z

Binary Coded Decimal (BCD) Instructions

ABCD, SBCD and NBCD execute the corresponding arithmetic operations with packed BCD numbers. Converting between packed and unpacked format of BCD numbers is accomplished by using the PACK and UNPACK instructions.

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn	8	source10+dest10+X->dest
	-(An),-(An)	8	
NBCD	<ea>	8	0-dest10-X->dest
PACK	(An),-(An)	16->8	Unpacked source+Data->packed dest
	#<data>		
	Dn,Dn#<data>	16->8	
SBCD	Dn,Dn	8	dest10-source10-X->dest
	-(An),(An)	8	
UNPK	(An),-(An)	8->16	packed source->unpacked source
	#<data>		unpacked source+data->unpacked source
	Dn,Dn,#<data> 8->16		

Program Flow Control

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn	8	source10+dest10+X->dest
	-(An),-(An)	8	
NBCD	<ea>	8	0-dest10-X->dest
PACK	(An),-(An)	16->8	unpacked source+data->packed dest
	#<data>		
	Dn,Dn#<data>	16->8	
SBCD	Dn,Dn	8	dest10-source10-X->dest
	-(An),(An)	8	
UNPK	(An),-(An)	8->16	packed source->unpacked source
	#<data>		unpacked source+data->unpacked source
	Dn,Dn,#<data> 8->16		

Remarks:

- The variable 'cc' can be replaced by any of the following condition codes:

Code	Condition
T	true (not used with Bcc)
F	false (not used with Bcc)
HI	higher, logically
LS	lower or same, logically
CC(HS)	C(arry) flag cleared, logically higher or same
CS(LO)	C(arry) flag set, logically lower
NE	not equal, Z(ero) flag cleared
EQ	equal, Z(ero) flag set
VC	Overflow clear, V flag cleared
VS	Overflow set, V flag set
PL	Plus, N(egative) flag cleared
MI	Minus, N(egative) flag set
GE	greater or equal, arithmetically
LT	lower than, arithmetically
GT	greater than, arithmetically
LE	lower or equal, arithmetically

By testing the flags after a CMP or SUB operation (see arithmetic instructions), the relationship between the two operands can be determined. If, for example, the second number is greater than the first, the condition GT will be true.

Again there is a difference between logical and arithmetic operations in terms of the sign. Since, for example, the byte $FF can stand for either -1 or +255, comparing it with 0 will return $FF > 0 as a logical condition (without sign) and $FF < 0 as an arithmetic condition (with sign).

In other instructions, relational statements, such as greater than, are not definitive. Here the condition codes are used to test the condition of an individual bit. For example, after a bitfield instruction, BEQ checks whether the Zero flag was set, meaning that all bits in the field were cleared, instead of checking for equality.

Therefore both meanings are given in the above table.

- RTD fetches a return address from the stack and sets the stack pointer to the value of <data>.
- RTR fetches another word from the stack before the return address and writes it to the CCR.

System Control Instructions

This instruction set is comprised of the privileged instructions (executable only in supervisor mode), instructions that generate Trap Exceptions and those that write to the Condition Code Register (CCR).

Instruction	Syntax	Size	Comments
	Privileged		
ANDI	#<data>,SR	16	direct data AND SR -> SR
EORI	#<data>,SR	16	direct data EOR SR -> SR
MOVE	<ea>,SR	16	source -> SR
	SR,<ea>	16	SR -> dest
MOVE	USP,An	32	USP -> An
	An,USP	32	An -> USP
MOVEC	Rc,Rn	32	Rc -> Rn
	Rn,Rc	32	Rn -> Rc
MOVES	Rn,<ea>	8,16,32	Rn -> dest with DFC
	<ea>,Rn		source with SFC -> Rn
ORI	#<data>,SR	16	direct data V SR -> SR
RESET	none	none	
RTE	none	none	(SP) -> SR; SP + 2; (SP) -> PC; SP + 4 -> SP;
STOP	#<data>	16	direct data -> SR
	TRAP Generating		
BKPT	#<data>	none	
CHK	<ea>,Dn	16,32	if Dn < 0 or Dn > (ea), CHK is called
CHK2	<ea>,Rn	8,16,32	
ILLEGAL	none	none	SSP - 2 -> SSP; vector offset -> (SSP) SSP - 4 -> SSP; PC -> (SSP); SSP - 2 -> SSP; SR -> (SSP); vector address of illegal instruction -> PC
TRAP	#<data>	none	SSP - 2 -> SSP; SSP - 4 -> SSP; PC -> (SSP); SSP - 2 -> SSP; SR -> (SSP); vector address -> PC
TRAPcc	none	none	executes TRAP if cc is true
	#<data>	16,32	
TRAPV	none	none	
	Condition Code Register		
ANDI	#<data>,CCR	8	direct data AND CCR -> CCR

Instruction	Syntax	Size	Comments
EORI	#<data>,CCR	8	direct data EOR CCR -> CCR
MOVE	<ea>,CCR	16	source -> CCR
	CCR,<ea>	16	CCR -> dest
ORI	#<data>,CCR	8	direct data V CCR -> CCR

Multiprocessor Instructions

Multiprocessor instructions are all those that carry out a Read-Modify-Write bus cycle. What does this mean?

In a multiprocessor system (e.g., the Amiga with a PC/AT expansion card), CPUs running in parallel can access the same memory region. Let's assume there is a common data structure in which only one processor is allowed to work at any given time. Each CPU has a place in memory where it signals its activity to the other. This is done by means of the IBD bit (*Ich bin dran*, German for 'I am here'). In preparing to access the data structure in question, CPU A has just read CPU B's IBD bit and, seeing that it is free, is about to set its own bit on. At this time, CPU B makes the same check and also concludes that the data structure is available for access. The result is conflicting access by both CPUs.

There are software solutions for the mutual access exclusion problem, but the 68030 offers a faster and simpler one in its hardware. During a Read-Modify-Write cycle, it signals the hardware that controls RAM that this process may not be interrupted. Thus CPU A can complete its setting of the IBD bit before CPU B is allowed to check it.

Instruction	Operand Syntax Read-Modify-Write	Operand Size	Operation
CAS	Dc,Du,<ea>	8,16,32	dest-Dc->CC; If Z is set., then Du->dest else dest->Dc
CAS2	Dc1:Dc2 ,Du1:Du2 ,(Rn):(Rn)	8,16,32	two operands CAS
TAS	<ea>	8	dest-0, set cond.codes; 1->dest(7) Coprocessor
cpBcc	<Label>	16,32	If cpcc true, then pc+D->PC
cpDBcc	<Label>,Dn	16	If cpcc false, then Dn-1->Dn If Dn <> 1, then PC+d->PC
cpGEN	user def.	user def.	operand->coprocessor
cpRESTORE	<ea>	none	Restore coprocessor status from <ea>

Instruction	Operand Syntax Read-Modify-Write	Operand Size	Operation
cpSAVE	<ea>	none	Save coprocessor status to <ea>
cpScc	<ea>	8	If cpcc true, then 1's->dest, else 0's->dest
cpTRAPcc	none	none	If cpcc true, then generate TRAPcc Exception
	#<data>	16,32	

Exceptions

With an exception the processor interrupts the currently executing program and jumps to a routine that handles the error or carries out the desired action. An exception on the 68030, then, includes anything that interrupts normal program execution: reset, interrupts, software errors, bus errors, etc.

Exception handling takes place in several steps:

1. First the CPU makes an internal copy of the Status Register. Then the Supervisor Bit is set, the Trace Bits cleared and, if the exception was an interrupt, the Interrupt Priority Mask adjusted.
2. Now the vector number is ascertained, which depending on the exception type, is either already provided or must be read over the bus (interrupts, coprocessor).
3. With a Reset (which also is considered an exception), the processor saves its current internal position on the stack, so that the interrupted program can resume after exception processing. With an interrupt, if the Master bit is set, the CPU clears it and rewrites the information to the stack. However, this time it's written to the Interrupt Stack (ISP) rather than the Master Stack (MSP).

   The exact format of the stack data varies greatly among the various exception types. Always present are the vector offset, the old value of the Program Counter, and the Status Register (saved initially).
4. In the final step, the processor reads the exception routine address from the vector table and begins its execution. The remainder of the exception handling is then accomplished by the software.

Vector Number(n)	Vector Offset Hex	Meaning
0	000	Interrupt Stack Pointer after Reset
1	004	Program Counter (PC) after Reset
2	008	Bus error
3	00C	Address error
4	010	Illegal instruction
5	014	Division by zero
6	018	CHK-, CHK2-instruction
7	01C	cpTRAPcc-, TRAPcc-, TRAPV-instruction
8	020	Privilege violation
9	024	Single step (Trace)
10	028	Line 1010 emulation
11	02C	Line 1111 emulation
12	030	Reserved (not used)
13	034	Coprocessor protocol violation
14	038	Format error
15	03C	Uninitialized interrupt
16 through 23	040 Reserved 05C	
24	060	Spurious interrupt
25	064	Level 1 interrupt autovector
26	068	Level 2 interrupt autovector
27	06C	Level 3 interrupt autovector
28	070	Level 4 interrupt autovector
29	074	Level 5 interrupt autovector
30	078	Level 6 interrupt autovector
31	07C	Level 7 interrupt autovector
32 through 47	080 TRAP 0 - 15 instruction vectors 0BC	
48	0C0	FPCP-branching or Set by unregulated condition
49	0C4	FPCP Inexact result
50	0C8	FPCP Division by zero
51	0CC	FPCP Underflow
52	0D0	FPCP Operations error
53	0D4	FPCP Overflow
54	0D8	FPCP Signaled NAN
55	0DC	Reserved
56	0E0	MMU setting error
57	0E4	Provided for MC68851 (not used in MC68030)
58	0E8	Provided for MC68851 (not used in MC68030)
59 through 63	0EC Reserved 0FC	

Vector Number(n)	Vector Offset Hex	Meaning
64 through . 255	100 User-defined vectors 3FC	

Every exception is assigned a specific vector, except for interrupts, whose vectors are generated as required by the appropriate hardware. All vectors consist of a long word with the address of the appropriate exception handler. The Exception Vector Table has 256 entries, thus a size of 1K. Its address in memory can be determined by the Vector Base Register. After a reset it will be at address 0, the beginning of system memory.

Exception Types

Reset, Vectors 0 & 1

Reset is the only exception that does not permit resumption of the old program. The processor clears all internal registers without saving anything on the stack, and the caches and MMU are shut off. In the Status Register, the Supervisor Bit is set and the Interrupt Mask set to 7. It then begins program execution at the address in Vector 1. Vector 0 is written to the ISP as initialization value.

Bus Error, Vector 2

If a bus access cannot proceed because of error, for example, when an attempt is made to write to ROM, the hardware can inform the CPU by signaling a bus error. All processors of the 680x0 family have the /BERR line for this purpose.

In the Amiga, a bus error is caused only by expansion cards claiming the same or illegal addresses.

Address Error, Vector 3

An address error results when an instruction that must be at an even address is read at an odd address. This occurs much more frequently on the 68000, since there access of words and long words is also forbidden at odd addresses.

Illegal Instruction, Vector 4

Line-A- and Line-F-emulator, Vectors 10 & 11

These exceptions are assigned to the various bit patterns that do not represent a valid instruction (words with hexadecimal A ($Axxx) in the upper four bits, those with F ($Fxxx), and other illegal bit patterns.

Line-A-emulation can be used to implement proprietary instructions, which can then be executed over the exception handler. Distinguishing among various functions is done by means of the lower 12 bits.

Line F distinguishes whether bits 9 through 11 of the instruction word are unequal to zero. This indicates a coprocessor instruction. Only when this is not the case, that is, the hardware signals a bus error in accessing the instruction, does an F-line exception occur. Otherwise the coprocessor executes the instruction.

With this mechanism it is possible to emulate entire coprocessor functions via software in the exception handler, as if it were actually present. If the FPU is present, the software can be reused without change. In the A3000, since the coprocessor is completely integrated, there is no need to worry about its emulation.

If bits 9 through 11 equal zero, then, depending on the bit pattern in the lower half of the instruction word, either it is a valid MMU instruction or the F-line exception occurs. But since MMU instructions are permitted only in supervisor mode, an instruction word in the form $F0xx in user mode always results in a privilege violation.

Privilege Violation, Vector 8

A privilege violation occurs when one of the following instructions is executed in user mode:

ANDI to SR, EOR to SR, FRESTORE, FSAVE, MOVE from SR, MOVE to SR, MOVE USP, MOVEC, MOVES, ORI to SR, PFLUSH, PMOVE, PLOAD, PTEST, RESET, RTE, STOP

These instructions apply only in supervisor mode. Note: The instruction 'MOVE from SR' was permitted in user mode on the 68000. On the 68030 it is permitted only in supervisor mode.

Zero Divide, Vector 5

CHK and TRAP Instructions, Vectors 6 & 7

Instruction Trap Exceptions occur with the corresponding condition in an instruction. They are intentional and signal an arithmetic or logical error condition in the program:

- Division by zero
- TRAPcc instruction with valid condition
- CHK or CHK2 detect a partition overwrite

Trace Exception, Vector 9

The 68030 has two trace modes to facilitate debugging. They are selected using the Trace Bits in the Status Register:

T1	T0	Trace mode
0	0	Trace disabled
0	1	Trace every transfer of control (BRA, JMP etc.)
1	0	Trace every instruction
1	1	not implemented

Depending on the selected mode, a trace exception follows every instruction or only those that transfer program flow. Within the exception handling routines the trace bits are set to zero to inhibit subsequent trace exceptions.

Trace mode makes it possible to follow the execution of a program step by step.

Independent of the mode selected, an instruction that writes to the Status Register generates a trace exception.

Format Error, Vector 14

The 68030 places varying numbers of words on the stack depending on the type of exception. These are returned from the stack by the RTE instruction (Return from Exception) so that the interrupted program can continue processing. A format error exception occurs when the processor detects an illegal stack format and cannot restore the previous state.

This error is caused by programs that overwrite stack data.

Interrupts

A number of vectors are assigned to interrupts: the spurious interrupt vector, 24, the seven autovectors, 25 - 31, all user-defined vectors, 64 - 255, and the uninitialized interrupt vector, 15.

As was previously mentioned in the discussion of the Status Register, an interrupt exception is called only when the interrupt level in the SR is lower than that of the signal in the interrupt entry. If this is the case, the 68030 attempts to read the appropriate interrupt vector over the data bus. It executes for this purpose an Interrupt Acknowledge Cycle, by which the hardware detects that its interrupt request has been acknowledged and that it can make a vector available.

Now there are three possibilities:

- The hardware delivers a vector (in the range of 64 - 255).
- It signals an autovector interrupt with one of the seven autovectors (25 - 31), according to the interrupt level. This is the case with the Amiga.
- It responds with a bus error and the spurious interrupt (24) occurs.

The Uninitialized Interrupt (15) occurs when a peripheral chip tries to deliver a vector that has not yet been initialized by the processor. This vector is therefore not generated by the processor, but rather is read by the Interrupt Acknowledge Cycle when, due to a software error, the corresponding vector register in the chip that produced the interrupt has not been set.

11.2.1 The PMMU

With the Paged Memory Management Unit, the 68030 provides hardware-controlled memory management and protection for the operating system. The PMMU supports two mechanisms to implement this control:

1. The generation of a virtual address space for each task.
2. The validation of authorization for every memory access.

Virtual Memory

Formerly, any reference to address locations or memory in the Amiga referred to the physical memory, that is, the actual RAM chips. In earlier models than the A3000, every address transmitted over the address bus selects one precise memory location in the corresponding component. This fixed assignment of addresses from the processor to the various registers and memory locations is inherent in the hardware and normally cannot be changed. Although there are some exceptions to this rule (e.g., bank switching), they don't apply to the Amiga. So, for every address, there is only one memory location.

In a multitasking operating system, where two or more tasks can run concurrently, the tasks must utilize different addresses for their data. When a task occupies memory, it reduces the amount available to all other tasks. Eventually memory becomes so fragmented from continuos allocation and de-allocation, that sufficient contiguous memory for even a simple function is no longer available.

Without the PMMU, logical memory (the memory addresses used by the machine-language instructions of the various tasks) is identical to physical memory.

In a system with virtual memory management, every task is assigned its own logical address partition. The maximum size of each partition is theoretically the entire address space, or 4 Gigabytes with the 68030. This logical address partition does not exist physically. Each task has 4 Gigabytes of memory, but only in a "virtual" sense. A specific correlation must be established between physical memory (usually only 1 Megabyte in 68030-based computers) and the logical addresses of all the tasks. The PMMU does just that. For every memory access, it translates the logical address to the appropriate physical address and also validates the access authority of the task for the area in question.

The fundamental unit of memory on which these mechanisms operate is called a page. The entire logical address space is divided into equal-size pages (page size may be as small as 256 bytes or as large 32K). A logical page can be mapped to a physical page of the same size. Physical pages in memory are sometimes referred to as frames. Of course not all logical pages can map cumulatively to main memory frames, since the virtual address space far exceeds the physical. This fact is transparent, though, to executing tasks, which share memory without even "knowing" it. If a task requests a page that is not currently in memory, the MMU detects this condition, called a page fault, and generates a bus error. The operating system can now load the desired page to an available frame in RAM. If none is available, a page can be swapped to disk and temporarily saved to make room for the new logical page, and the fault can be satisfied.

Besides managing the transparent sharing of memory among tasks, the PMMU also solves the problem of memory fragmentation. Physically noncontiguous available fragments can be allocated by the PMMU in such a way as to logically satisfy a task's request for contiguous address space.

These capabilities are not implemented in the current version of the Amiga operating system. However, it is worthwhile to study the functions and programming of the processes previously described, since the PMMU is a fixed part of the Amiga 3000 hardware. Also, virtual addressing is used by the UNIX operating system, which Commodore also offers for the 3000. Finally, a knowledge of the PMMU will allow you to experiment with your own applications.

To perform the translation of logical into physical addresses, the PMMU naturally needs information about the allocation of pages requested by the operating system. A multi-level structure called a translation tree is constructed in main memory for this purpose. The translation tree indicates whether the desired logical page is contained in a physical page frame and if so, which one.

If the tree were to be scanned for every memory access, however, the CPU would have to sacrifice most of its processing power for this task alone. In reality the PMMU slows program execution by only 1 to 2%.

The solution lies again in a cache: the ATC (Address Translation Cache). Every time a valid allocation is read from the translation tree, the PMMU transfers it to the ATC. The ATC has room for up to 22 entries. Upon subsequent request for any of these pages, the cache signals their availability to the PMMU. As long as a time-consuming table search is avoided, memory access with PMMU address translation is just as fast as without it.

Only when a program references a page for the first time or when more than 22 pages are needed will the search be performed. Since this rarely happens, as mentioned earlier, the processing time is barely affected.

The program model of the PMMU consists of six registers:

- CPU (or User) Root pointer (CRP)
- Supervisor Root pointer (SRP)
- Translation Control Register (TC)
- Transparent Translation Register 1 (TT0)
- Transparent Translation Register 2 (TT1)
- MMU Status Register (MMUSR)

The root pointer registers hold translation tree starting addresses. You can choose between separate trees for user and supervisor programs or a single common tree. If only one tree is used, its address must be in the CPU root pointer.

The transparent translation registers make it possible to reference any two address regions without address translation. Their logical addresses then are the same as the physical, as though the MMU were disabled. This function is useful for larger areas which require quick access, for example screen memory. Since such an area consists of many pages that are accessed in no particular sequence, the 22 ATC entries are not sufficient to avoid frequent searching of the translation tree tables. It is more efficient to reference these areas without the MMU.

MMU Transparent Translation Registers TT0 and TT1:

BitNr.:	Function:
31-24	Address bits 31 through 24 of the transparent region
23-16	Mask for above address bits
15	E - Enable bit for transparent region
14-11	Unused
10	CI - Cache inhibit
9	R/W - Read/write
8	RWM - Read/write mask
7	Unused
6-4	Function code base of region
3	Unused
2-0	Function code mask

Address bits and mask

The transparent window has a minimum size of 16 Meg. The eight address bits in the TTx register specify its address. The masking bits can be used to define larger windows. When one of these is set, its corresponding address bit is ignored.

Enable bit (E)

Setting the E bit to 1 enables the transparent region defined by the TTx register.

Cache inhibit (CI)

The CI bit allows you to choose whether or not values from the transparent region should be placed in the data and instruction caches (1 = no caching).

R/W and RWM

The R/W bit specifies the type of access that should be transparent:

R/W = 0	Write access
R/W = 1	Read access

To grant both types, the RWM bit must be set:

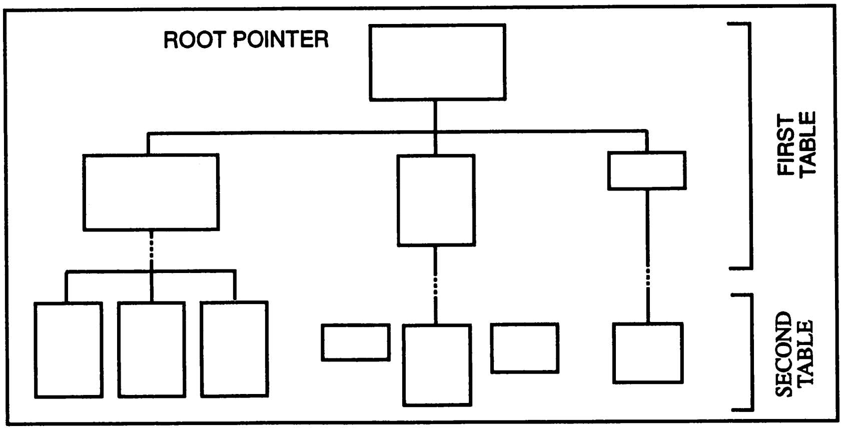
RWM = 0	Access according to R/W bit
RWM = 1	Ignore R/W bit

Function code base and function code mask

The base field determines the region's function code. If you want it to be transparent for more than one specific function code, you can use the mask to choose which bits of the base field should be ignored (mask = 1).

Translation Control Register (TC)

The TC register controls the construction of the translation tree. The root of this tree is found in the CPU root pointer register. That address points to the first table, or level, of the tree. Each entry in this highest table (level A) points to a table at the next level (level B). The tree continues to branch until the lowest level, which contains the actual page addresses, is reached.



Page Pointer Tables

The logical address consists of up to six fields, which are used to move through the various levels of tables. The fields (with the exception of the first one) represent indexes for successive branches of the translation tree.

Here is how the individual fields look in relation to the logical address:

Bits 31	to 31-IS	Ignored
Bits 31-IS	to 31-IS-TIA	Table A index
Bits 31-IS-TIA	to 31-IS-TIA-TIB	Table B index
Bits 31-IS-TIA-TIB	to 31-IS-TIA-TIB-TIC	Table C index
Bits 31-IS-TIA-TIB-TIC	to 31-IS-TIA-TIB-TIC-TID	Table D index
Bits 31-IS-TIA-TIB-TIC-TID	to bit 0	Page offset
Shown another way:		
Logical address:	31.........BitNr...........0 IS|TIA|TIB|TIC|TID|Pageoffset	

The IS (initial shift) field is a series of up to 15 bits, starting with bit 31, which are to be ignored by the MMU. Because the IS bits are not considered, the same physical address is allocated by the MMU regardless of their value, so the effect of the shift is to reduce the logical address space accordingly. Where IS = the number of bits in the initial shift field, the shift reflects a reduction of address space by a factor of 2^{IS}.

TIA through TID are table indexes A through B. Each index can be from 0 to 15 bits long, except for TIA, which must be at least 1 bit. With a TIA length of five bits, for example, the five most significant bits of the logical address (after the IS) are used as an index to the highest level of the translation tree. This table must then have 2^5, or 32 entries. Each entry points to a table at level B. The size of the TIB field (again as a power of 2) indicates the number of entries in the level B tables. This pattern continues down to level D. The remaining bits of the logical address form the page offset, the relative position of the desired memory location within the page.

It is not necessary to use all four levels of tables. The translation tree terminates when a null TIx field is reached. If TIC = 0, the tree is limited to levels A and B.

Even a fifth level can be used by setting the FCL (function code lookup) enable bit in the TC register. As the name suggests, the function code bits then become part of the table lookup algorithm. This is implemented with another table, consisting of eight entries, preceding the A level. The root pointer points to this table. The function code bits are used to select from the eight entries, which in turn point to tables of level A.

Translation Control Register Layout

BitNr.	Name	Function
31	E	Enables the MMU
30-26	-	Unused
25	SRE	Enables supervisor root pointer (separate translation trees for user and supervisor mode)
24	FCL	Function code lookup enable
23-20	PS	Pagesize: 1000 - 256 Bytes 1001 - 512 Bytes 1010 - 1 KByte 1011 - 2 KByte 1100 - 4 KByte 1101 - 8 KByte 1110 - 16 KByte 1111 - 32 KByte
19-16	IS	Initial Shift (0 to 15)
15-12	TIA	Table Index A (1 to 15)
11-8	TIB	Table Index B (0 to 15)
7-4	TIC	Table Index C (0 to 15)
3-0	TID	Table Index D (0 to 15)

PS, IS, TIA - TID must clearly establish the use of the individual bits of the logical address. Their sum must always be 32. Otherwise the MMU generates an MMU configuration exception.

Elements of the Translation Tree

The various tables of the translation tree can contain different types of entries called descriptors. A descriptor consists of a pointer to a memory page or to the next level of tables, as well as control information about the subsequent structure of the tables or about the memory that the descriptor addresses. Not every descriptor type contains all the following fields:

DT Descriptor type

This 2-bit field can contain the following:

Invalid, DT=0

If the MMU encounters an invalid descriptor, it terminates its search and reports a bus error to the CPU. Invalid descriptors allow the translation tree to start out in skeleton form (in which these descriptors point to as yet undefined locations), and to be completed only as needed (when bus errors occur).

Page descriptor, DT=1

A page descriptor also terminates the MMU's search, but signals a successful completion because it contains the address of the desired memory page.

Valid 4 bytes, DT=2

This designates a valid pointer to a table of the next lower level. All entries in this table must be 4 bytes long.

Valid 8 bytes, DT=3

Same as above, but the table being addressed must contain 8 byte entries.

U Used

This bit is automatically set when the MMU reads the descriptor. It can be used, for example, to determine whether a certain memory page has been accessed.

M Modified

This bit indicates a write access to the allocated memory page.

WP Write protect

If this bit is set in one of the descriptors read during a table search, a corresponding page may not be written. This is useful, for example, to prohibit subsequent changes after modifying Kickstart in RAM.

S Supervisor only

This bit designates a table or memory page that may be referenced by supervisor programs only. Reference by user programs results in a bus error.

CI Cache inhibit

Many addresses may not be cached. This is true primarily for the various peripheral chip registers or for areas of memory that can be changed independently of the CPU (the dual-ported RAM on the PC/AT plug-in board and chip memory).

Caching can be turned off for the corresponding memory pages using the CI bit. When the CI bit is set, the CPU does not transfer these pages to the cache as they are read.

LIMIT and L/U bit

The 15-bit limit field can be used to limit the index to the next table. The MMU checks to see if the index value contained in the logical address is higher than the limit when the L/U bit = 0, or lower than the limit when L/U = 1. The L/U (lower/upper) bit determines whether the index has a lower or an upper limit.

Page address

This 24-bit field contains the physical page address that corresponds to the requested logical address. Depending on the page size (see Translation Control Register), some of the bits are unused. With 4K pages, 12 bits are needed to address a location within the page. These are taken directly from the logical address (the page offset). Then only the upper 20 bits of the page address field are used to establish the address of the page.

Table address

This 28-bit field contains the base address of the descriptor table of the next lower level.

Descriptor address

This is a 30-bit pointer to another descriptor. It is used only by indirect descriptors (see the following).

The Various Descriptor Types

Bits labeled "unused" can be used to store values as needed (e.g., additional status information for memory management).

Root pointer (either CPU or Supervisor Root Pointer Register)

LW	BitNr.	Function
0	31	L/U bit for limit
0	30-16	Limit value
0	15-2	Must be zero
0	1-0	DT (descriptor type) of root pointer
1	31-4	Table address
1	3-0	Unused

Short Format Table Descriptor (4 bytes)

BitNr.	Function
31-4	Table address
3	U (Used)
2	WP (Write protect)
1-0	DT (Descriptor type) = 2

Long Format Table Descriptor (8 bytes)

LW	BitNr.	Function
0	31	L/U bit for limit
0	30-16	Limit value
0	15-10	Must be 1's
0	9	Must be zero
0	8	S (Supervisor only)
0	7-4	Must be zero
0	3	U (Used)
0	2	WP (Write protect)
0	1-0	DT (Descriptor type) = 3
1	31-4	Table address
1	3-0	Unused

Short Format Page Descriptor (4 bytes)

BitNr.	Function
31-8	Page address
7	Must be zero
6	CI (Cache inhibit)
5	Must be zero
4	M (Modified)
3	U (Used)
2	WP (Write protect)
1-0	DT (Descriptor type) = 1

Long Format Page Descriptor (8 bytes)

LW	BitNr.	Function
0	31	L/U bit for limit
0	30-16	Limit value
0	15-10	Must be 1's
0	9	Must be zero
0	8	S (Supervisor only)
0	7	Must be zero
0	6	CI (Cache inhibit)
0	5	Must be zero
0	4	M (Modified)
0	3	U (Used)
0	2	WP (Write protect)
0	1-0	DT (Descriptor type) = 1
1	31-8	Page address
1	7-0	Unused

Short Invalid Descriptor (4 bytes)

BitNr.	Function
31-2	Unused
1-0	DT (Descriptor type) = 0

Long Invalid Descriptor (8 bytes)

LW	BitNr.	Function
0	31-2	Unused
0	1-0	DT (Descriptor type) = 0
1	31-0	Unused

Short Format Indirect Descriptor (4 bytes)

BitNr.	Function
31-2	Descriptor address
1-0	DT (Descriptor type) = 2 or 3

Long Format Indirect Descriptor (8 bytes)

LW	BitNr.	Function
0	31-2	Unused
0	1-0	DT (Descriptor type) = 2 or 3
1	31-2	Descriptor address
1	1-0	Unused

Early Termination and Memory Blocks

Normally page descriptors are in the lowest level of the tree and table descriptors in the upper level. A page descriptor that occurs at a higher level is called an "Early Termination (ET)" page descriptor. It can be used to allocate a consecutive area of physical addresses to all logical addresses belonging to a table entry.

Assume that an ET page descriptor exists at the next-to-last table level, the last level contains 4096 entries, and the page size is 8K. A 32 Meg block (4096 x 8K) is then referenced by this ET descriptor. The descriptor's page address field contains the physical base address of the block. The logical base address is any bit combination that references the ET descriptor in the translation tree. The bits of the logical address that normally would select one of the 4096 entries of the lowest level table now select the referenced page directly within the 32 Meg block. The lowest level table can be eliminated.

Here is how the various bits in the logical address are used relative to the previous example:

Values in the Translation Control Register:
IS = 0, TIA = 7, TIB = 12, TIC & TID = 0, PS = 8 KByte

Logical address without ET descriptor:

BitNr. 31 ...	24 ...	12 ...	0
Pointer to Table A	Pointer to Table B	Address within the memory page	

Logical address with ET descriptor in Table A:

BitNr. 31 ...	24 ...	12 ...	0
Pointer to Table A	Page number in 32 Meg	Address within the memory page block	

An ET descriptor can also replace more than one level of tables if it occurs at a higher (previous) level of the translation tree. You can imagine this as a tree, where the root pointer is the trunk, the tables of the intermediate levels are the limbs and branches, and the page descriptors at the lowest (last) level are the leaves. An ET descriptor replaces, a branch or limb with a single "hyper-dimensional" leaf, which represents all the leaves this branch or limb carries.

Indirect Descriptors

Another special feature of the translation tree, whose structure has already been described, is the indirect descriptor. This descriptor occurs in place of a page descriptor at the lowest level of the translation tree and must point to a page descriptor in some other branch of the tree. It allows two different logical addresses to be assigned to the same physical page. This would apply to global data structures or when the tree is divided into supervisor and user sections.

The same effect is achieved simply by placing the same page address into two page descriptors at different parts of the tree. But in this case, the Used and Modified bits of both descriptors must always be checked to determine whether the page has been accessed. Indirect descriptors can provide significant time savings in the case of global data structures, where not just two, but a multiplicity of descriptors may point to the same physical address, since these structures can be accessed by every task.

The MMU Instructions

The MMU, like every coprocessor in a 68030 system, adds its own commands to the instruction set of the CPU.

The MMU recognizes the following instructions:

PMOVE

The PMOVE instruction accesses the registers of the MMU: TC, SRP, CRP, TT0 or TT1.

Syntax: PMOVE <effective address (ea)>, MMU-register
PMOVE MMU-register,<ea>

PMOVEFD Since changing the contents of an MMU-register usually invalidates the current values in the ATC (Address Translation Cache), the ATC is cleared with every PMOVE instruction that writes to one of these registers. This can be prevented with PMOVEFD (Flush Disable).

Syntax: PMOVEFD <ea>, MMU-register

PFLUSH

Clears an entry in the ATC.

Syntax: PFLUSH (An)

PFLUSHA

Clears all entries in the ATC.

Syntax: PFLUSHA

PLOAD

Performs a table search for the given logical address and function code and writes the physical address to the ATC. The search can be performed in advance for routines where time is critical.

The PLOAD instruction has two variants: PLOADR (read) and PLOADW (write). In PLOADW, the MMU sets the Modified bit in the page desciptor, as though a write access to this logical address had occurred.

The Used bit is always set in PLOAD instructions.

Syntax:

PLOADR <function code>,<ea>
PLOADW <function code>,<ea>

PTEST

The PTEST instruction can be used to find the cause of a bus error in the exception handler. The MMU performs a table search for the logical address (this can be fetched from the stack after a bus error), and sets the bits in the Status Register (see the following) to indicate whether an invalid descriptor was found, a LIMIT exceeded, etc.

Syntax: PTEST <function code>,<ea>,level,An

"level" is the maximum number of table levels which are to be searched.

level=0 - look only in ATC
level=7 - search all table levels

PTEST also has two variants, PTESTR and PTESTW. The distinction is pertinent only when PTEST level = 0, and one of the two transparent addresses (TTx registers) is referenced.

The MMU Status Register (MMUSR)

BitNr.	Name
15	B - Hardware bus error in table search
14	L - LIMIT error (invalid for PTEST level 0)
13	S - Supervisor only (invalid for PTEST level 0)
12	Must be zero
11	W - Write protected
10	I - Invalid
9	M - Modified
8 & 7	Must be zero
6	T - Transparent region (only for PTEST level 0)
5 - 3	Must be zero
2 - 0	Number of table levels searched (0 with PTEST level 0)

For PTEST with level = 0, the I bit signals that the desired logical address has no ATC entry (in which case, a level 7 PTEST will return additional information), or that a hardware bus error has occurred in searching the table (in which case the B bit also is set).

For PTEST with level > 0, the I bit indicates that an invalid descriptor was found, a LIMIT was exceeded (L bit = 1), or a bus error occurred (B bit = 1).

11.2.2 The Floating Point Coprocessor

The microprocessor in a desktop computer like the A3000 is normally responsible for the execution of all program instructions. But even the 68030, one of the most powerful processors, has its limits, namely in the processing of data types that are not directly supported in its hardware design.

Attempts must be made, then, to construct these from other available data types and handle their processing by using software. The data types that the 68030 supports directly were listed in the previous chapter. An example of one that is not directly supported, but which nevertheless can be effectively handled with the existing types, is the character string. This usually consists of a variable-length sequence of bytes terminated by a zero. Using a small loop and the "byte" data type, it is possible to perform all the elementary character string operations, such as copying, comparing and clearing.

Another data type not directly supported, and even more essential in the Amiga than the character string, is the bit-map graphic. Bit-map graphics are used to build a representation in memory of the visible contents of a screen. Elementary bit-map operations, such as drawing lines or filling surfaces, require multiple operations with individual memory bits. Unfortunately, the bit-by-bit manipulation of data is not very effective in a processor optimized for 32 bit integers.

For this reason, Commodore has equipped the Amiga with an integrated circuit called the Blitter. While not an actual coprocessor in Motorola's sense of the word, it can perform independent bit-map operations much faster than the 68000.

Functions

The FPU now places data types unknown to the 68030 but indispensable for a variety of computer applications. These are the floating-point numbers.

The 68030 recognizes only a limited range of integer numbers, which can be expressed in mathematical set notation as follows: $Z = \{X \mid -2^{31} <= X < 2^{31}\}$. This set of numbers may suffice for the normal tasks of an operating system and many application programs, but for the solution of mathematical problems it must be expanded.

The drawing of a circle presents a simple example. The formula for a point on the circumference is:

X-coordinate = sin angle * radius
Y-coordinate = cos angle * radius

The final result, the point's "x:" and "y" coordinates, must be whole numbers, since there is no such thing as half a pixel position in graphic memory. But computations with the trigonometric sine and cosine functions require a greater universe than whole numbers, since these functions return values from the set of real numbers. The real numbers, when expressed in decimal form, have a fractional component that either terminates, repeats periodically or repeats at random.

A binary format, that can represent the greatest possible subset of these numbers, must be found. We are at best limited to a subset, because any machine format is finite (i.e., limited to a certain number of places). The limitations imposed on our available universe are twofold, affecting both range and precision. First, there is a minimum and a maximum number that the format can represent. Secondly, the fractional digits of repeating decimal numbers (e.g., pi) must eventually (as justified by considerations of memory and computing time) be truncated.

Over time, the floating-point representation has gained general acceptance, since it is applied relatively simply to computers and it is adequate for almost all mathematical computations.

Floating-point format corresponds to exponential notation in mathematics, where a number greater than or equal to 1 and less than 10, called the mantissa (the decimal part of a logarithm), is multiplied by a power of 10, the exponent. Some examples are:

Normal notation	Exponential notation		
27	->2.7	x	10^{1}
1,500,000	->1.5	x	10^{6}
0.5	->5	x	10^{-1}
0.0000042	->4.2	x	10^{-6}
	^		^
	Mantissa		Exponent

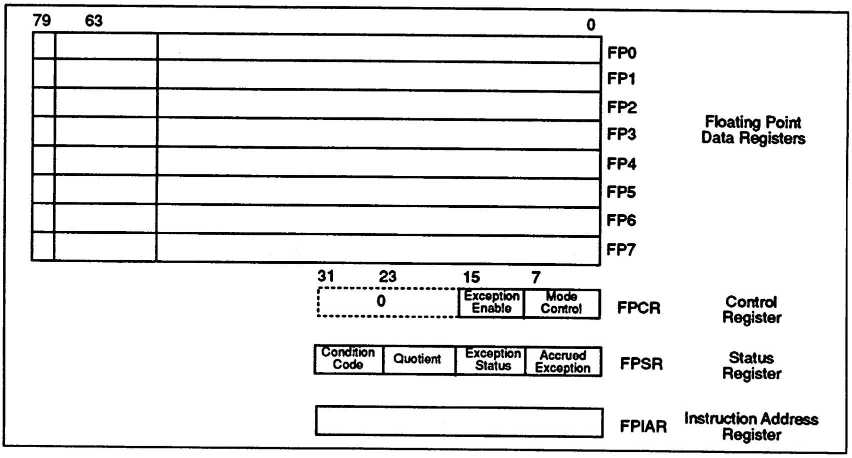
The exponent in floating-point machine format always refers to base 2.

This notation permits the representation of both very large and very small numbers without long strings of zeros. The FPU 68881/68882 can perform computations on these numbers directly, just as the 68030 does with integers.

Internal Design - The Program Model

From the programmer's point of view, the FPU has 11 registers (see the following illustration). Eight of them are universal floating-point data registers, designated FP0 - FP7. Each is 80 bits wide and can accept one number. In addition to these are the Control Register (FPCR), which controls the FPU's mode of operation, the Status Register (FPSR), which holds flag bits like its counterpart in the 68030, and the Instruction

Address Register (FPIAR). This last register stores the address of the current FPU instruction, for reasons that will be seen later.



Floating-point Data Format

Floating-point Data Formats

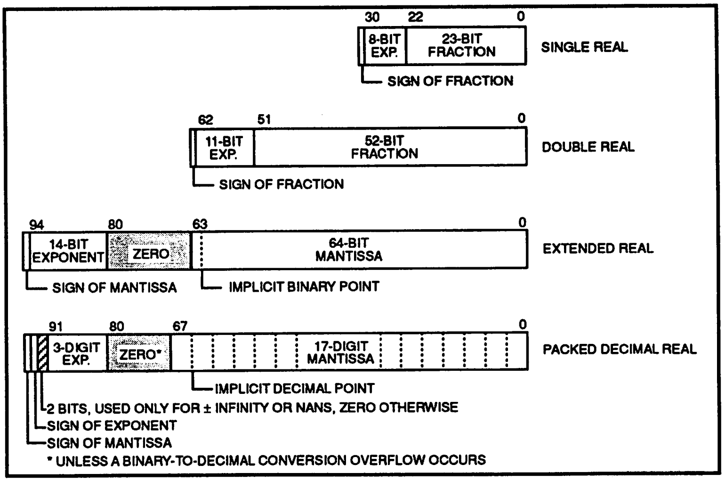
Internally the FPU operates exclusively on a single data type called extended real. This is the only data type that can be placed in one of the FPU data registers. It is, however, only one of seven types with which we may actually work. All others are automatically converted by the FPU to extended real format before being used. When a value is transferred from one of the eight FPU data registers to main memory, the corresponding conversion again takes place.

The byte, word, and longword integer formats are identical to those of the 68030, but are also handled internally as extended.

Single real and double real formats differ only in size, with one long word for single and two for double. Extended real format consists of three long words, but 16 bits are not used, since the FPU data registers are only 80 bits wide.

Packed decimal real format differs from the other six formats in that it represents numbers by the base-ten (decimal) rather than base-two (binary) system.

The mantissa undergoes special handling in single and double real formats. Basically all floating-point numbers are stored in normalized form. This means that the exponent is chose to produce a mantissa in the range 10^0 <= mantissa < 10^1 (1 <= mantissa < 10). One writes not 35 * 10^3 but 3.5 * 10^4, and 5 * 10^{-4} instead of 0.5 * 10^{-3}. There is exactly one place before the decimal point, whose position is established in all formats.



Real Formats

Similarly, a mantissa in the binary system would be governed as follows: 2^0 <= mantissa < 2^1 (1 <= mantissa < 2). The bit before the "decimal" point must always be 1. To save space, it is simply dropped in single and double format. The first bit of the mantissa represents the first place after the "decimal" point, with a leading 1 always implied.

In extended format this bit still exists and can be set to zero. The same is true of packed decimal format. Such a number is said to be unnormalized. It is normalized by the FPU before any operation on it is begun. The result of an FPU instruction is never an unnormalized number.

The exponent (except in packed decimal format) is constructed as an unsigned binary number, from which the sign is then reclaimed by subtracting an offset. Depending on the format, the offset is 127, 1023 or 16383. For a number in double real format, for example, the value 1023 in the exponent field signifies an actual exponent value of 0, 1022 is -1, and so on.

The reason for this lies in the 68030. By virtue of this special representation of the exponent and the fact that the mantissa sign appears in the MSB of the long word, two floating-point numbers can be compared by the CMP instruction, without having to call upon the FPU.

For this purpose they behave just like normal integers.

Status and Control Registers

The Status Register is subdivided into four function groups of one byte each:

- Condition Code Byte
- Quotient Byte
- Exception Byte
- Accrued Exception Byte

Condition Code Byte (FPCC - Floating Point Condition Code)

BitNr.:	31	30	29	28	27	26	25	24
Function:	0	0	0	0	N	Z	I	NAN

N	Negative result
Z	Result is 0
I	Result is + or - infinity
NAN	Result is not a number

These four bits fulfill the same purposes as the N, C, Z and X bits in the Status Register of the 68030. They are set according to the data type of the result following every computation operation. With the aid of these bits, you can use the FBcc instruction to build conditional jumps into a program, making program flow dependent upon FPU computation results.

Quotient Byte

BitNr.:	23	22	21	20	19	18	17	16
Function:	S	MSB..........Quotient.............LSB						

This byte is set after only two instructions: modulo and IEEE-REST. Both compute the remainder of a division. Instead of throwing away the inner of a computed quotient, it is placed in this byte. If it is greater than seven bits, the higher order ones are dropped. The sign of the quotient appears in the S bit.

Exception Byte

BitNr.:	15	14	13	12	11	10	9	8
Func:	BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2

BSUN	Branch / Set On Unordered
SNAN	Signaling Not A Number
OPERR	OPerand ERRor
OVFL	OVerFLow
UNFL	UNderFLow
DZ	Divide by Zero
INEX2	INEXact Operation
INEX1	INEXact decimal input

Just as the 68030 calls an exception routine in case of error, for example, a supervisor instruction executed in user mode, the same can be done by the coprocessor. In fact, computations with floating-point numbers can cause several types of errors.

These errors are reflected in the Exception byte.

Whether an exception routine will actually be called depends on the upper byte of the Control Register, the Exception Enable byte. It has exactly the same layout as the Exception byte in the Status Register. If a bit in the Enable byte is set and the corresponding exception occurs, the 68030 calls the assigned exception routine. If an exception is disabled by clearing the corresponding bit, checking the Exception byte will reveal whether an error has occurred, but the exception procedure will not be called.

Accrued Exception Byte

BitNr.:	7	6	5	4	3	2	1	0
Function:	IOP	OVFL	UVFL	DZ	INEX	-	-	-

IOP	Invalid OPeration (BSUN or SNAN or OPERR)
OVFL	OVerFLow (OVFL)
UNFL	UNderFLow (UNFL and INEX2)
DZ	Divide by Zero (DZ)
INEX	INEXact Operation (INEX1 or INEX2 or OVFL)

Sometimes you may prefer simply not to permit any exceptions, if for no other reason than to avoid having to program the exception handlers. In this case the Exception byte would have to be checked after every floating-point computation, since any bits set are cleared for each new operation.

Here the Accrued Exception byte is useful. If the FPU changes a bit in the Exception byte, the bits in the Accrued byte are set as previously shown. Thus IOP is activated when BSUN, SNAN or OPERR = 1.

Unlike in the Exception byte, the bits in the Accrued byte retain their status. Once set, they are cleared only by the explicit writing of a 0 in the Status Register.

This allows you to perform a series of computations and wait until their completion to determine whether any errors occurred.

Control Register - Mode Control Byte

BitNr.:	7	6	5	4	3	2	1	0
Function:	PREC		RND		0	0	0	0

PREC	Rounding Precision	
	0 0	Extended
	0 1	Single
	1 0	Double
RND - Rounding Mode		
	0 0	round to nearest - RN
	0 1	round toward zero - RZ
	1 0	round toward minus infinity - RM
	1 1	round toward plus infinity - RP

Precision bits are used to adjust the precision to which a result should be rounded. Normally these bits are set to Extended (both 0), since internal processing takes place on extended numbers. A result stored in single or double precision is rounded automatically, regardless of the PREC bits.

The rounding mode determines what the coprocessor will do when a number cannot be represented exactly with the available precision. Internally it computes with three additional bits for the mantissa. If these are not all zero as the result of a computation, rounding is required. The actual value lies between two choices, which are numbers that can be represented with the available number of bits in the mantissa. One of these must be chosen as the result. How the FPU proceeds in this choice is determined by the two RND bits:

Mode RN always rounds to the nearer of the two possibilities previously described. If they are equidistant, the FPU chooses the even one (LSB = 0).

Mode RZ rounds to the one with the smaller absolute value (i.e., the one closer to zero).

Mode RM always chooses the smaller number, RP chooses the larger.

The FPU Instruction Set

As we mentioned earlier, all FPU assembler instructions are simply considered extensions of the 68030 instruction set and can be mixed with 68030 instructions as desired. Processing occurs in parallel. A main processor instruction that follows an FPU instruction can begin executing as soon as the 68030 has given the FPU the data it needs. Only upon encountering another FPU instruction must the CPU wait for the first one to be completed.

The only exceptions are those FPU instructions that transfer data from the FPU to main memory. Here the 68030 must wait until the 68881 has finished making the data available.

The syntax of the instructions follows the same rules already familiar to the 68000 series processors. By now most assemblers on the Amiga can process 68030 and 68881 instructions.

All FPU instructions begin with the letter "F" (as all PMMU instructions begin with "P"), to distinguish them from those of the main processor.

The registers are designated as follows:

- FP0 - FP7 for the eight floating-point data registers
- FPCR, FPSR and FPIAR for the control registers (in general FPcr)

The following abbreviations are valid for the various data types:

.B, .W, .L
for byte, word or long word integer

.S, .D
for single or double precision real

.X for extended precision real

.P for packed decimal

All the same addressing modes can be used as for the 68030, except in the few instructions that permit only certain kinds of addressing. The syntax is also the same.

Data Transfer Instructions

Instruction Syntax	Operand Format	Operand	Operation
FMOVE	FPmm,FPn	X	Source->dest
	<ea>,FPn	B,W,L,S,D,X,P	
	FPm,<ea>	B,W,L,S,D,X	
	FPm,<ea>(#k)	P	
	FPm,<ea>(Dn)	P	
	<ea>,FPcr	L	
	FPcr,<ea>	L	
FMOVECR	#ccc,FPn	X	ROM constant -> FPn
FMOVEM	<ea>,<list>	L,X	Register list -> dest
	<ea>,Dn	X	
	<list>,<ea>	L,X	Source -> register list
	Dn,<ea>	X	

Remarks:

The FMOVE instruction with .P (packed decimal) as the destination format can automatically round to a desired number of places. The rounding value, which can be supplied immediately or in a data register, is specified as follows:

-64 <= k <= 0	Rounded to |k| places after the decimal point
0 < k <= 17	Mantissa is rounded to k places independent of the exponent

In the FMOVECR instruction, "ccc" is the number of a numerical constant from the ROM of the FPU:

Number	Constant
\$00	PI
\$0B	$Log_{10}(2)$
\$0C	e
\$0D	$Log_2(e)$
\$0E	$Log_{10}(e)$
\$0F	0.0
\$30	ln(2)
\$31	ln(10)
\$32	10^0
to	$10^1, 10^2, 10^4, \ldots 10^{2048}$
\$3F	10^{4096}

MOVEM transfers any combination of the eight data registers or one of the three control registers between memory and the FPU. If the list is in a processor data register, only a data register transfer is possible. The following format applies:

Type of addressing	Bit 7	--	Bit 0
Predecrement -(An)	FP7	--	FP0
All others	FP0	--	FP7

Dyadic Operations

Dyadic operations are functions performed on two operands, for example multiplication or addition. The first operand can be addressed with any addressing method, the second must always be one of the FPU data registers. The result of the function will be placed in this data register.

Instruction	Function
FADD	Add
FCMP	Compare
FDIV	Divide
FMOD	Modulo
FMUL	Multiply
FREM	IEEE remainder
FSCALE	Exponent scaling
FSGLDIV	Divide (single precision)
FSGLMUL	Multiply (single precision)
FSUB	Subtract

Remarks:

FSCALE adds the first value (whose fractional places are truncated) to the exponent of the second.

FREM delivers the remainder of a division according to the IEEE-definition:

```
X    INT(Operand2 / Operand1) rounded with 'round-to-nearest'!
     Operand2 - (Operand1 * X)
```

Monadic Operations

A monadic operation is a function performed on a single operand. The operand can be referenced with any addressing method. The result is always placed in an FPU data register.

Instruction	Function
FABS	Absolute value
FACOS	Arccosine
FASIN	Arcsine
FATAN	Arctangent
FATANH	Hyperbolic arctangent
FCOS	Cosine
FCOSH	Hyperbolic cosine
FETOX	e to the x
FETOXM1	e to the x-1
FGETEXP	Get exponent
FGETMAN	Get mantissa
FINT	Integer
FINTRZ	Integer round to zero
FLOGN	Logarithm of (x)
FLOGNP1	Logarithm of (x+1)
FLOG10	Log base 10 of x
FLOG2	Log base 2 of x
FNEG	Negate
FSIN	Sine
FSINH	Hyperbolic sine
FSQRT	Square root
FTAN	Tangent
FTANH	Hyperbolic tangent
FTENTOX	10^x
FTWOTOX	2^x

Instruction	Syntax	Format	Instruction
FSINCOS	<ea>,FPc:FPs FPm,FPc:FPs	B,W,L,S,D,X,P X	SIN(source) -> FPs; COS(source) -> FPc

Remarks:

FSINCOS produces two results, which are placed in separate data registers.

The unit of measurement of angles in trigonometric functions is the radian.

Program Control Instructions

This group of instructions allows control of program flow by using condition codes generated by the FPU. Their functions correspond to the 68030 instructions of the same name.

Instruction	Operand Syntax	Operand Format/Size	Operation
FBcc	<Label>	W,L	If true, then PC+d->PC
FDBcc	Dn,<Label>	W	If true, then no operation, else Dn-1->Dn; If Dn <> -1 then PC+d->PC
FNOP	None	None	No operation
FScc	<ea>	B	If true, then 1's->dest else 0's->dest
FTST	<ea> FPn	B,W,L,S,D,X,P X	FPSR Set cond.codes

The following mnemonics can be given for "cc":

With Exception (NAN bit set in Status Register):

GE	greater or equal
GL	greater or less
GLE	greater, less or equal
GT	greater
LE	less or equal
LT	less
NGE	not (greater or equal)
NGL	not (greater or less)
NGLE	not (greater, less or equal)
NGT	not (greater)
NLE	not (less or equal)
NLT	not less
SEQ	equal
SNE	unequal
SF	always false
ST	always true

Without Exception:

OGE	ordered and greater or equal
OGL	ordered and greater or less
OR	ordered
OGT	ordered and greater
OLE	ordered and less or equal
OLT	ordered less
UGE	unordered or greater or equal
UEQ	unordered or equal
UN	unordered
UGT	unordered or greater
ULE	unordered or less or equal
ULT	unordered or less
EQ	equal
NE	unequal
F	always false
T	always true

This list may seem confusing if you're used to the 68030 condition codes. What does ordered mean? Why is there a distinction between "greater or less" and "unequal"?

The reason is that a floating-point number can represent two special conditions that a normal number cannot:

1. + or - infinity
2. not-a-number (NAN)

A number with all 1's in the exponent and all 0's in the mantissa represents an infinity. The sign bit remains valid, so there are both plus and minus infinities.

An infinity is produced when the exponent of a result is greater than or equal to the greatest possible exponent that can be represented.

An invalid number (not-a-number) occurs when the exponent is all 1's and the mantissa is not all 0's. This can arise through a number of invalid operations, such as applying the square root function to a negative number or dividing two infinities.

A NAN can also be used to signal user-defined exception conditions. This is the purpose of the SNAN exception (signaling Not-A-Number), which occurs when a function is called with a "signaling" NAN as an operand. This intentionally supplied NAN is identified by a zero bit in the first fractional position of the mantissa. If the FPU produces a NAN as the result of a computation, it sets this bit, even if it was called with a signaling NAN (assuming the SNAN exception has been disabled).

The NAN presents a few problems in the handling of condition codes. Whereas plus infinity can be said to be greater than any natural number, this concept of comparison is not possible with a NAN. Is the number 1 greater or less than a NAN? There is no answer to this question. For this reason, in addition to greater, less and equal, the conditions ordered and unordered have been introduced. The unordered condition applies when the NAN bit in the Condition Code Register is set. All comparisons testing for greater or less check the NAN bit. If it is set, the first number is neither greater nor less than the second. Moreover, the numbers are not equal, either. Rather, they are unordered, and this is the only condition that will test true in such a case.

Only the conditions equal and not equal (EQ and NE) function the same as they do for integers in the 68030.

So that an additional branch instruction is not required to intercept the NAN condition, the 68881 has a built-in BSUN exception. If you use one of the instructions from the previous first list and allow this exception, the 68030 executes the exception procedure as soon as the NAN bit is set at the branch.

An important reminder: “not less” doesn't always mean “greater“; it could mean “unordered”.

System Control Instructions

This group consists of only three instructions:

FSAVE and FRESTORE cause the operating system to save and restore the internal state of the FPU and are mainly used when switching between two tasks, both of which want FPU access.

An FSAVE and subsequent MOVEM for all 11 registers preserves the momentary state of the FPU entirely, even if it was interrupted in the middle of a computation. (In this case the FSAVE transfers 184 bytes.) Any other FPU operation can now be allowed to execute, until, by means of the reverse MOVEM and FRESTORE, the FPU is restored to its earlier state and can resume the interrupted computation.

With FTRAPcc an exception can be generated regardless of condition code. A word or long word given with the FTRAPcc instruction is considered to be surrendered to the trap handler and will not be handled by the processor.

Instruction	Operand Syntax	Operand Size	Operation
FRESTORE	<ea>	None	State frame -> internal register
FSAVE	<ea>	None	Internal register -> state frame
FTRAPcc	None	None	If condition true, then exception
	#xxx	W,L	

The FPU Exceptions

As previously mentioned, there are eight exception types that can be generated by the FPU:

BSUN	Branch / Set On Unordered
SNAN	Signaling Not A Number
OPERR	OPerand ERRor
OVFL	OVerFLow
UNFL	UNderFLow
DZ	Divide by Zero
INEX2	INEXact Operation
INEX1	INEXact decimal input

BSUN has the highest priority and INEX2/1 the lowest priority. If multiple exceptions occur simultaneously, the 68030 executes the one with the highest priority, and the trap handler must worry about whether lower priority bits are also set.

The individual FPU exceptions are assigned to the following TRAP vectors on the 68030:

Vector Number	Vector Offset	Assignment
7	$01C	FTRAPcc instruction
11	$02C	F-Line emulator
13	$034	Protocol violation
48	$0C0	BSUN
49	$0C4	Inexact result
50	$0C8	Divide by zero
51	$0CC	Underflow
52	$0D0	Operand error
53	$0D4	Overflow
54	$0D8	SNAN

FTRAPcc — This exception is called if condition cc is valid on an FTRAP instruction.

F-Line — This exception indicates that an invalid FPU instruction was detected. The bit pattern of the current instruction does not match any known instruction.

Cop.Prot. — A protocol violation in the communication between main and coprocessor generates this exception. The cause is usually a hardware defect.

BSUN — As described in the last section, this exception occurs on certain branch conditions (also FTRAPcc, etc.), when the NAN bit in the Condition Code Register is set. The NAN bit must be reset within the trap routine, because at the conclusion of exception handling the FPU calls the instruction again.

INEX — There are two potential generators of an Inexact Result Exception: INEX1 is produced when a packed decimal number is converted internally to extended precision format.

INEX2 — Indicates the need for rounding in all other circumstances. All operations resulting in periodic or nonterminating binary numbers produce this exception.

DZ — A Divide-by-zero occurs upon dividing a number by zero or calling a transcendental function that has a perpendicular asymptote at this position, so that F(X) goes to infinity (e.g., tan(PI/2)).

If this exception is disabled, the FPU returns an infinity result.

UNFL — The Underflow exception is called when the result of a computation is too small to be represented internally, meaning the exponent is less than or equal to the least possible value.

It also occurs when a MOVE instruction converts an operand from extended precision to single or double, and the exponent is less than or equal to the least possible value.

Conversion to .B, .W or .L format does not cause an Underflow exception, but simply returns a zero value.

Despite the trap, in all Underflows, either the least possible result or zero is returned, depending on rounding mode.

OPERR — Operand Error refers to a variety of possible error conditions arising from the use of mathematical functions with inappropriate input or in a context that does not have a valid mathematical interpretation. Square root of a negative number is an example of such a condition.

OVFL — The same conditions apply to the Overflow exception as to Underflow, except that it is activated by the greatest possible exponent, and returns infinity or the greatest possible result, again depending on the rounding mode.

SNAN — This exception is called when an operand of a monadic or dyadic function is a signaling NAN.

Because the main processor and the math coprocessor work in parallel, the FPU exceptions are not recognized by the 68030 until it has already processed subsequent instructions and arrived at the next FPU command. A trap handler whose job is to redress an FPU error would like to know which instruction caused it. But the 68030 has already gone ahead in the program, and the last address it has placed on the stack is the next FPU instruction. Between this and the one that caused the exception is an unknown number of normal processor instructions.

The FPIAR Register (Floating Point Instruction Address Register) is designed to take care of this problem. It holds the address of the FPU instruction that the coprocessor is currently processing. If an exception occurs, the trap handler only needs to read the address from the FPIAR Register to find the offending instruction in main memory.

11.2.3 Differences Between the MC 68881 and 68882

The essential difference between the two floating-point coprocessors is in the manufacturing technology. Whereas the maximum cycle rate for the 68881 was 20 MHz, the fastest version of the 68882 manages 50 MHz. But the 68882's advantage in speed results not only from a higher cycle frequency. Even at the same frequency it runs 25% faster than the '881. This effect is linked to the improved facility for parallel processing. Both coprocessors have only a single APU, and can perform only one floating-point computation at a time. The fetching of new operands and conversion of numeric formats runs parallel to the work of the APU. Here's an example of programming that maximizes the efficiency of the 68882:

When the following loop:

```
FMOVE.X (A0),FP1        ; Fetch next entry
FADD.X (A1),FP1         ;  Add
FMOVE.X FP1,(A2)        ;  Store
```

is optimized for the 68882 it looks like this:

```
FMOVE.X (A0),FP1        ;  Fetch next entry
FMOVE.X (A3),FP2        ;  Fetch next + 1
FADD.X (A1),FP1         ;  First add
FADD.X (A4),FP2         ;  Add next + 1
FADD.X FP1,(A2)         ;  Store first result
FMOVE.X FP2,(A3)        ;  Store next +1
```

This simple example shows how the placement of instructions within a loop can help alleviate register conflict.

To do this, try to combine the fastest FMOVE's with the fastest arithmetic instructions, and the slowest FMOVE's with the slowest arithmetic instructions.

11.2.4 Cache Memory

The 68030 has two internal caches: 256 bytes each for instructions and data. Their function is to store frequently used values and make them available to the CPU without wait time on subsequent references.

This solves the problem of designing a main memory that is both large and fast. Because of caching, sufficient speed is attained with less expensive dynamic RAM, so you don't have to skimp on memory size.

RAM access requires wait cycles, which inevitably results in wasting some of the maximum processor speed. But wait time can be significantly reduced by reading a large part of the data and instructions from main memory only the first time they are referenced and then reading them from cache memory. Depending on the program, efficiency increases of up to 100% are possible.

The caching concept exploits the fact that, during program processing, the CPU spends most of its time in loops of no more than 10 to 30 instructions. Once such a loop has executed the first time, it is fully contained in the instruction cache and need not be read again from main memory.

This principle also applies to data. The most frequently referenced addresses tend to be localized. For example, if you look at a typical C program, you'll see that within a function, most data accesses refer to the local variables, which are placed on the stack in a single block.

Instruction Cache Design

Each cache in the 68030 consists of 16 rows, each of which contains four long words. Each row is assigned a tag-entry, in which the address (bits 8 - 31) and the FC2 bit of the function code (for distinguishing between supervisor and user mode) is placed. The tag-entry also contains four Valid bits, for the four long words.

Address bits 4 - 7 select the cache row, and bits 2 and 3 select one of the long words. When the CPU accesses main memory, a row is selected by the appropriate address bits and the row's tag-entry is compared with address bits 8 - 31 and the FC2 bit. If they agree, and if the Valid bit for the desired long word is set, the condition is called a cache-hit. In this case, the value can be read from the cache and RAM is not accessed.

A cache-miss occurs when the tag and address do not agree or the Valid bit is zero.

If just the Valid bit is missing, the CPU reads the value from main memory, transfers it to the cache and sets the Valid bit. If there was no tag-entry agreement, the CPU tries not only to retrieve the desired long word from RAM, but also to fill the entire row. A burst-fill is started for this purpose. The burst-fill takes advantage of the fact that the four long words occupy consecutive addresses in memory. A special access method of dynamic RAM enables extremely fast referencing of sequential data. A burst-fill for four long words barely takes longer than two normal accesses.

After a successful burst, the address is entered to the tag-field and all four Valid bits are set. Accessing this data again results in a cache-hit.

Function of Address bits in Cache:

Address	Function
A0 & A1	Byte within long word
A2 & A3	Long word within row
A4 - A7	Row number
A8 - A31 & FC2	Address of data in corresponding row (stored in tag-entry of row)

Cache Row Layout:

Tag (FC2,A31 - A8)	Longword 0	Longword 1	Longword 2	Lw 3

The data Cache

The data cache differs from the instruction cache in its additional requirement for write access. Write access is not relevant to instruction caching, since instructions are not altered. Also, data cache tag-entries contain all three function code bits because the data and program regions must be distinguished.

Basically the data cache is designed as a write-through cache. This means that all data is written to main memory, regardless of whether or not it is present in the cache. Thus values in RAM are always valid.

Two modes are available for transferring written data to the cache: with or without Write Allocation (WA). Without Write Allocation, a value is transferred to the cache only when it was already found there (i.e., when a cache-hit occurred). The new value replaces the old, without any change to the tag-entry or Valid bits. In this mode, values with new tag-addresses are written to the cache only on read accesses.

With Write Allocation, all data is transferred, regardless of whether a cache-hit or a cache-miss occurred. A miss clears the Valid bits of the remaining three long words. WA mode is necessary if both supervisor and user access are permitted at the same address, as is the case in the Amiga. The following example should help illustrate the problem without Write Allocation:

- The supervisor reads Value A at Location X, and A is loaded to the cache.
- The user program writes Value B to Location X. Since the cache distinguishes user from supervisor access, no hit occurs despite the same address. B is not transferred to the cache, but remains at Location X.
- The supervisor reads Location X again, but getting a hit, it reads it from the cache. It reads the wrong value (A instead of B).

The Cache Control Registers (CACR and CAAR)

Two registers control the functioning of both caches. The Cache Control Register (CACR) contains several control bits, while the Cache Address Register (CAAR) specifies the address of a long word in the cache. This address is necessary for clearing individual entries in the cache. The CAAR layout corresponds to the addressing of the individual cache entries in normal operation:

Cache Address Register (CAAR)

BitNr.:	31 ... 8	7 ... 4	3 and 2	1 and 0
Function:	Unused	Cache row	Long word	Unused

Cache Control Register (CACR)

BitNr.:	Name:	Function:
31 - 14	--	Unused
13	WA	Write Allocate
12	DBE	Data Burst Enable
11	CD	Clear Data Cache
10	CED	Clear Entry in Data Cache
9	FD	Freeze Data Cache
8	ED	Enable Data Cache
7 - 5	--	Unused
4	IBE	Instruction Burst Enable
3	CI	Clear Instruction Cache
2	CEI	Clear Entry in Instruction Cache
1	FI	Freeze Instruction Cache
0	EI	Enable Instruction Cache

WA Write Allocation mode is turned on if this bit is set.

DBE Enables burst on cache-miss with wrong tag-entry.

CD All entries in the data cache are cleared when a 1 is written to this bit. This only happens once; the CD bit is not stored.

CED Clears only the cache-entry whose address is in the CAAR.

FD Freezes the data cache. Subsequent read accesses without cache-hits do not overwrite old values. Cache contents are modified by write accesses only.

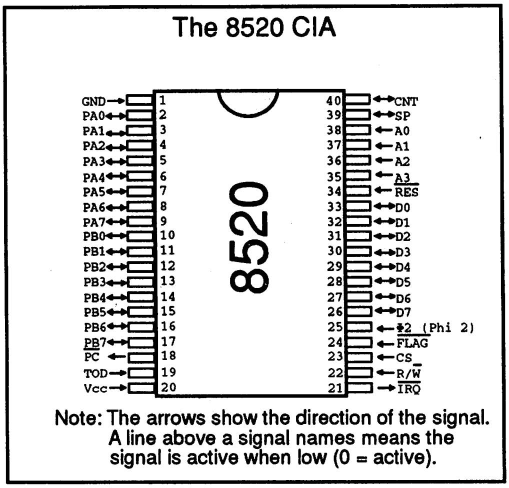
The FD bit is useful for optimizing program speed. It prevents one-time accesses (for which the cache provides no speed advantage) from destroying useful values in the cache.

ED The processor ignores a cache-hit if the Enable bit is cleared. All data is retrieved from main memory. The cache contents continue to be updated and can be used as soon as caching is enabled.

IBE Enables burst access for instruction cache.

CI	Clears all instruction cache entries.
CEC	Clears the entry whose address is in the CAAR.
FI	If the FI bit is set, the contents of the instruction cache cease to be updated.
EI	If the EI bit is zero, a cache-hit is ignored and all instructions are fetched from main memory.

11.3 The CIA 8520



The CIA 8520

The 8520 is a peripheral component of the Complex Interface Adapter (CIA) class, which basically means that its developers tried to support as many functions as possible on a single chip. A close inspection of the 8520 reveals great similarity to its old counterpart in the C64, namely the 6526. Only the functioning of registers 8 through 11 has changed slightly. This is certainly good news for anyone familiar with programming the 6526.

The 8520 has the following features: two freely programmable 8-bit parallel ports (PA and PB), two 16-bit timers (A and B), a bi-directional serial port (SP) and a 24-bit counter (event counter) with an alarm function upon reaching a programmed value. All functions can generate interrupts.

The functions of the 8520 are organized into 16 registers. To the processor they look like ordinary memory locations, since all peripheral components in a 68 x 0 system are memory mapped and can be read and written with the usual MOVEs and other processor instructions.

The 16 internal registers are selected with the four address-inputs, A0-A3. Details about the integration of the CIA into the Amiga system are found at the end of this section.

The following are the functions of the 16 registers (actually 15, since register 11 ($B) is unused):

The 8520 registers

Register		Name	Function
0	0	PRA	Port A data register
1	1	PRB	Port B data register
2	2	DDRA	Port A data direction register
3	3	DDRB	Port B data direction register
4	4	TALO	Timer A lower 8 bits
5	5	TAHI	Timer A upper 8 bits
6	6	TBLO	Timer B lower 8 bits
7	7	TBHI	Timer B upper 8 bits
8	8	Event low	Counter bits 0-7
9	9	Event mid	Counter bits 8-15
10	A	Event high	Counter bits 16-23
11	B	---	Unused
12	C	SP	Serial port data register
13	D	ICR	Interrupt control register
14	E	CRA	Control register A
15	F	CRB	Control register B

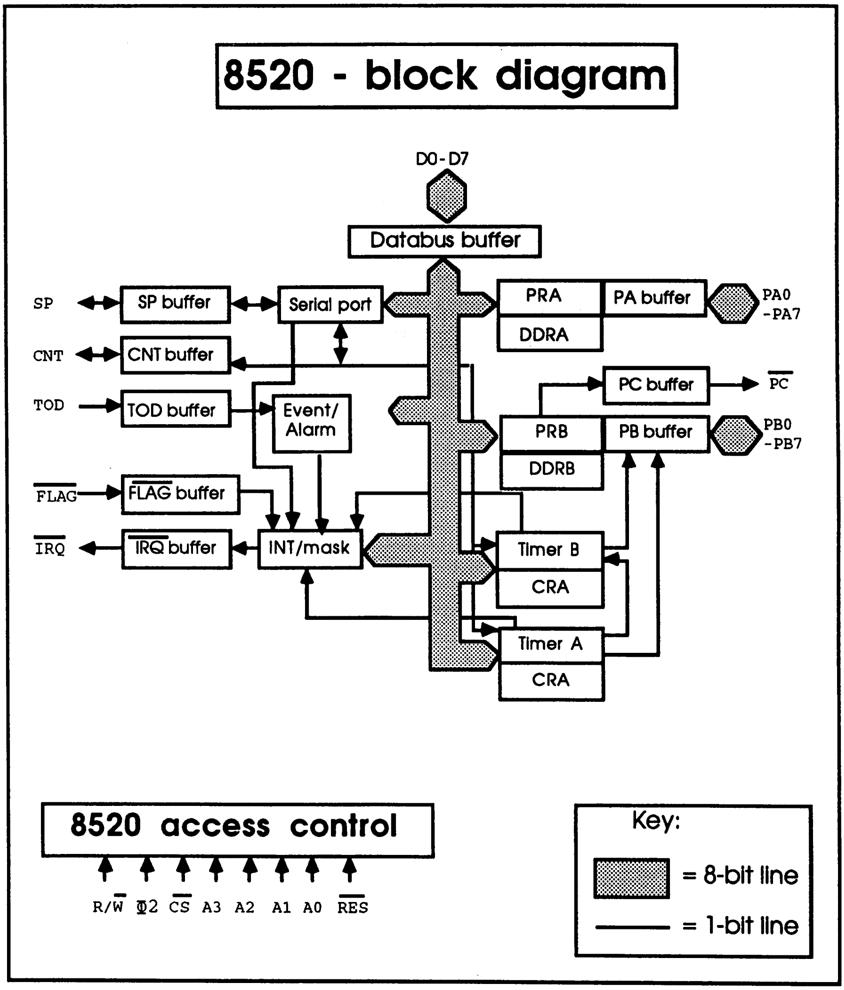
The parallel ports

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	B4	PB3	PB2	PB1	PB0
2	DDRA	DPA7	DPA6	DPA5	DPA4	DPA3	DPA2	DPA1	DPA0
3	DDRB	DPB7	DPB6	DPB5	DPB4	DPB3	DPB2	DPB1	DPB0

The 8520 has two 8-bit parallel ports, PA and PB, each of which is assigned a data register, PRA (Port Register A) and PRB (Port Register B). Associated with these registers are the chip's 16 port lines, PA0-PA7 and PB0-PB7. The 8520 allows the data direction of each port line to be individually controlled. This means that each port line can be used as input as well as output. For this purpose, each port has a data direction register, DDRA and DDRB. If a bit in a data direction register is 0, its corresponding line behaves as input, so that the level of the signal on this line can be interrogated by reading the appropriate bit of the data register.

If the bit is set to 1, the line becomes an output. Now the signal on the line is actually determined by the value of the corresponding bit in the data register.

In general, writing to a data register always stores the value in it, while reading always returns the states of the port lines. The bits in the data direction register determine whether the value of the data register is placed on the port lines. Therefore when reading a port that is configured as an output, the contents of the data register are returned, while when writing to an input port, the value is stored in the data register, but does not appear on the port lines until the port is configured as output.



Block Diagram of the 8520 CIA

To simplify data transfer through the parallel ports, the 8520 has two handshake lines, PC and FLAG.

The PC output goes low for one clock cycle on each access to data register B (PRB, reg. 1). The FLAG input responds to such downward transitions. Every time the state of the FLAG line changes from 1 to 0, the FLAG bit is set in the Interrupt Control Register (ICR, reg. $D). These two lines allow a simple form of handshaking in which the FLAG and PC lines of two CIAs are cross-connected.

The sender need only write its data to the port register and then wait for a FLAG signal before sending each additional byte. Since FLAG can generate an interrupt, the sender can even perform other tasks while it is waiting. The same applies to the receiver, except that it reads the data from the port instead of writing it.

The timers:

Read access:

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
4	TALO	TAL7	TAL6	TAL5	TAL4	TAL3	TAL2	TAL1	TAL0
5	TAHI	TAH7	TAH6	TAH5	TAH4	TAH3	TAH2	TAH1	TAH0
6	TBLO	TBL7	TBL6	TBL5	TBL4	TBL3	TBL2	TBL1	TBL0
7	TBHI	TBH7	TBH6	TBH5	TBH4	TBH3	TBH2	TBH1	TBH0

Write access:

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
4	PALO	PAL7	PAL6	PAL5	PAL4	PAL3	PAL2	PAL1	PAL0
5	PAHI	PAH7	PAH6	PAH5	PAH4	PAH3	PAH2	PAH1	PAH0
6	PBLO	PBL7	PBL6	PBL5	PBL4	PBL3	PBL2	PBL1	PBL0
7	PBHI	PBH7	PBH6	PBH5	PBH4	PBH3	PBH2	PBH1	PBH0

The 8520 has two 16-bit timers. These timers can count from a preset value down to zero. A number of modes are possible and can be selected through a control register, one for each timer (CRA and CRB).

Each timer consists internally of four registers (timer A: TALO+TAHI and PALO+PAHI), or two register pairs, since each low and high register pair forms the 16-bit timer value. Both register pairs have the same address, but one can only be read and the other only written. On a write access to one of the timer registers the value is first saved in a latch, then loaded into the timer register and decremented until the timer reaches zero. When this happens, the value is loaded from the latch into the timer register again.

Reading a timer register returns the current state of the timer. To get a correct value, though, the timer must be stopped.

The following example shows why:

- Timer state: $0100.
- A read access to register 5 returns the high byte of the current state: $01.
- Before the low byte (reg. 4) can be read, the timer is decremented again and the timer state is now $00FF.
- The low byte is read: $FF.
- Resulting timer state: $01FF.
- Instead of stopping the timer, which also causes problems since now timer pulses are ignored, a better method can be used: Read the high byte, then the low byte and then the high byte again. If the two high byte values match, then the value read is correct. If not, the process must be repeated.
- Which signals decrement the timers is determined for timer A by bit 5 and for timer B by bits 5 and 6 of the respective control registers.

Only two sources are possible for timer A:

1. Timer A is decremented with each clock cycle. The cycle frequency of the CIAs in the Amiga is 716 KHz (INMODE = 0).
2. Timer A is decremented with each high impulse on the CNT line (INMODE = 1).

Timer B has four input modes:

1. Clock cycles (INMODE bits = 00) (binary - the first digit stands for bit 6, the second for bit 5).
2. CNT impulse (INMODE bits = 01).
3. Timer A timeouts (allows two timers to form a 32-bit timer) (INMODE bits = 10).
4. Timer A timeouts when the CNT line is high (allows the length of a pulse on the CNT line to be measured) (INMODE bits = 11).

The timeouts of a timer are registered in the Interrupt Control Register (ICR). When timer A times out, the TA bit (no. 0) is set, while when timer B times out, the TB bit (no. 1) is set. These bits, like all of the bits in the ICR, remain set until the ICR is read.

In addition, it is also possible to output the timeouts to parallel port B. If the PBon bit is set in the control register for the given timer (CRA or CRB), then each timeout appears on the appropriate port line (PB6 for timer A and PB7 for timer B).

Two output modes can be selected with the OUTMODE bit:

OUTMODE = 0 Pulse mode

Each timeout appears as a positive pulse one clock period long on the corresponding port line.

OUTMODE = 1 Toggle mode

Each timeout causes the corresponding port line to change value from high to low or low to high. Each time the timer is started the output starts at high.

The timers are started and stopped with the START bit in the control registers. START = 0 stops the timer, START = 1 starts it.

The RUNMODE bit selects between one-shot mode and continuous mode. In one-shot mode the timer stops after each timeout and sets the START bit back to 0. In continuous mode the timer restarts after each timeout.

As mentioned before, writing to a timer register doesn't write the value directly to the register but to a latch (also called a prescaler, since the number of timeouts per second is equal to the clock frequency divided by the value in the prescaler). There are several ways to transfer the value from the latch to the timer:

1. Set the LOAD bit in the control register. This causes a forced load, that is, the value in the latch is transferred to the timer registers regardless of the timer state. The LOAD bit is called a strobe bit, which means that the bit is not stored but simply triggers a one-time operation. To cause another forced load, a 1 must be written to the LOAD bit again.
2. Each time the timer runs out, it is automatically reloaded with the value in the latch.

3. After a write access to the high register of a timer that is stopped (START = 0), the timer is automatically loaded with the value in the latch. Therefore the low byte of the timer should always be initialized first.

Assignment of the bits in control register A:

Register No. 14/$E Name: CRA							
D7	D6	D5	D4	D3	D2	D1	D0
not used	SPMODE 0=input 1=output	INMODE 0=clock 1=CNT	LOAD 1=force load (strobe)	RUNMODE 0=cont. 1=one-shot	OUTMODE 0=pulse 1=toggle	PBon 0=PB6off 1=PB6on	START 0=off 1=on

Assignment of the bits in control register B:

Register No. 15/$F Name: CRB						
D7	D6+D5	D4	D3	D2	D1	D0
ALARM 0=TOD 1=alarm	INMODE 00=clock 01=CNT 10=timer A 11=timer A+ CNT	LOAD 1=force load (strobe)	RUNMODE 0=cont. 1=one-shot	OUTMODE 0=pulse 1=toggle	PBon 0=PB7off 1=PB7on	START 0=off 1=on

The event counter:

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
8 $8	LSB event	E7	E6	E5	E4	E3	E2	E1	E0
9 $9	Event 8-15	E15	E14	E13	E12	E11	E10	E9	E8
10 $A	MSB event	E23	E22	E21	E20	E19	E18	E17	E16

As we mentioned earlier, there are only minor differences between the 8520 and the 6526. All of these differences concern the function of registers 8-11. The 6526 has a real-time clock which returns the time of day in hours, minutes and seconds in the individual registers. On the 8520 this clock is replaced by a simple 24-bit counter, called an event counter. This can lead to some confusion, because Commodore often uses the old designation TOD (Time-Of-Day) when referring to the 8520.

The operation of the event counter is simple. It is a 24-bit counter, meaning that it can count from 0 to 16777215 ($FFFFFF). With each rising edge (transition from low to high) on the TOD line, the counter value is incremented by one. When the counter has reached $FFFFFF, it starts over at 0 on the next count pulse. The counter can be set to a defined state by writing the desired value into the counter registers.

Register 8 contains bits 0-7 of the counter, the least significant byte (LSB), in register 9 are bits 8-15, and in register 10 are bits 16-23, the most significant byte (MSB) of the counter.

The counter stops on each write access so that no errors result from a sudden carry from one register to another (as described in the timer discussion). The counter starts running again when a value is written into the LSB (reg. 8). Normally the counter is written in the order: register 10 (MSB), then register 9, and finally register 8 (LSB).

To prevent carry errors when the counter is read, the counter value is written into a latch when the MSB (reg. 10) is read. Each additional access to a count register now returns the value of the latch, which can be read in peace while the counter continues to run internally. The latch is turned off again when the LSB is read. The counter should be read in the same order as it is written (see previous paragraph).

An alarm function is also built into the event counter. If the alarm bit (bit 7) is set to 1 in control register B, an alarm value can be set by writing registers 8-10. As soon as the value of the counter matches this alarm value, the alarm bit in the interrupt control register is set. The alarm value can only be set -- a read access to registers 8-10 always returns the current counter state, regardless of whether or not the alarm bit is set in control register B.

The serial port

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
12 $C	SDR	S7	S6	S5	S4	S3	S2	S1	S0

The serial port consists of the serial data register and an 8-bit shift register that cannot be accessed directly. The port can be configured as input (SPMODE=0) or output (SPMODE=1) with the SPMODE bit in control register A. In the input mode the serial data on the SP line are shifted into the shift register on each rising edge on the CNT line. After eight CNT pulses the shift register is full and its contents are transferred to the serial data register. At the same time, the SP bit in the interrupt control register is set. If more CNT pulses occur, the data continues to shift into the shift register until it is full again. If the user has read the serial data register (SDR) in the meantime, the new value is copied into the SDR and the transfer continues in this manner.

To use the serial port as output, set SPMODE to 1. The timeout rate of timer A, which must be operated in continuous mode, determines the baud rate (number of bits per second). The data are always shifted out of the shift register at half the timeout rate of timer A, so the maximum output rate is one quarter of the clock frequency of the 8520.

Transfer begins after the first data byte is written to the SDR. The CIA transfers the data byte into the shift register. The individual data bits now appear on the SP line at half the timeout rate of timer A, as the clock signal from timer A is reflected in the CNT line. (CNT changes value on each timeout; on every falling edge, that is, high to low transition on the CNT line, the next bit appears on the SP line.)

The transfer begins with the MSB of the data byte. Once all eight bits have been output, CNT remains high and the SP line retains the value of the last bit sent. In addition, the SP bit in the interrupt control register is set to show that the shift register can be supplied with new data. If the next data byte was loaded into the data register before the output of the last bit, the data output continues without interruption.

To keep the transfer continuous, the serial data register must be supplied with new data at the proper time. The SP and CNT lines are open-collector outputs so that CNT and SP lines of multiple 8520s can be connected together.

The Interrupt Control Register (ICR):

Read access = data register

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
13 $D	ICR	IR	0	0	FLAG	SP	Alarm	TB	TA

Write access = mask register

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
13 $D	ICR	S/C	x	x	FLAG	SP	Alarm	TB	TA

The ICR consists of a data register and a mask register. Each of the five interrupt sources can set its corresponding bit in the data register. Here again are all five possible interrupt sources:

1. Timeout of timer A (TA, bit 0).
2. Timeout of timer B (TB, bit 1).

3. Match of event counter value and alarm value (Alarm, bit 2).
4. The shift register of the serial port is full (input) or empty (output) (SP, bit 3).
5. Negative transition on the FLAG input (FLAG, bit 4).

If the ICR register is read, what is returned is always the value of the data register, which is subsequently cleared (all set bits, including the IR bit are cleared). If the value of the data register is still needed, it must be stored in RAM after the read.

The mask register can only be written. Its value determines whether a set bit in the data register can generate an interrupt. To make an interrupt possible, the corresponding bit in the mask register must be set to 1. The 8520 pulls the IRQ line low (it is active low) whenever a bit is set in both the data register and the mask register and sets the IR bit (bit 7) in the data register so that an interrupt is also signaled in software. The IRQ line does not return to 1 until the ICR is read and thus cleared.

The mask register cannot be written like an ordinary memory location. To set a bit in the mask register, the desired bit must be set and the S/C bit (Set/Clear, bit 7) must also be set. All other bits remain unchanged. To clear a bit, the desired bit must again be set, but this time the S/C bit is cleared. The S/C bit determines whether the set bits will set (S/C=1) or clear (S/C=0) the corresponding bits in the mask register. All cleared bits in the byte written to the mask register have no effect on it.

Here is an example: We want to allow an interrupt through the FLAG line. The current value of the mask register is 00000011 binary, meaning that both timer interrupts are allowed.

The following value must be written into the mask register: 10010000 binary (S/C = 1). The mask register then has the following contents: 00010011.

If you now want to turn the two timer interrupts off, write the following value: 00000011 (S/C=0). Now the mask register contains 00010000, and only the FLAG interrupt is allowed.

Integration of the CIAs into the Amiga system

As previously mentioned, the Amiga has two CIAs of the type 8520. The base address of the first 8520, which we call 8520-A, is $BFE001. The registers are not at contiguous memory addresses, however. Instead they are at 256 byte intervals.

This means that all of the 8520-A registers are at odd addresses because the 8520-A is connected to the lower 8 lines of the processor data bus (D0-7). Between the 68030 and the CIAs is a bus adapter. This forms the 68000-style synchronous bus interface. Originally this transfer protocol from the 8-bit (6800) era was implemented in the 68000 for compatibility with the peripheral ICs that existed at that time. As the dramatic success of the 68000 became apparent, special ICs for its asynchronous bus were made available, and the synchronous bus was dropped from the later processors (68020, 30 and 40). The special logic of the bus adapter allows the CIAs from older Amigas, which were also retained for compatibility, to be connected to the 68030 bus.

The following table lists the addresses of the individual registers with their uses in the Amiga (refer to the section on interfaces for more information on the individual port bits):

CIA-A: Register addresses

Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
$BFE001	PRA	/FIR1	/FIR0	/RDY	/TK0	/WPRO	/CHNG	/LED	OVL
$BFE101	PRB	Centronics parallel port							
$BFE201	DDRA	0	0	0	0	0	0	1	1
$BFE301	DDRB	Input or output depending on the application							
$BFE401	TALO	Timer A is used by the operating system for communication							
$BFE501	TAHI	with the keyboard							
$BFE601	TBLO	Timer B is used by the OS for various tasks							
$BFE701	TBHI								
$BFE801	E. LSB	The event counter in CIA-A counts 50 Hz							
$BFE901	E. 8-15	pulses from the power supply (called ticks), which							
$BFEA01	E. MSB	are taken from the power-line frequency							
$BFEC01	SP	Input for key codes from the keyboard							
$BFED01	ICR	Interrupt control register							
$BFEE01	CRA	Control register A							
$BFEF01	CRB	Control register B							

The second CIA, CIA-B, is referenced at address $BFD000. Its registers lie at even addresses because the data bus of CIA-B is connected to the upper half of the processor data bus.

CIA-B: Register addresses

Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
$BFD000	PRA	/DTR	/RTS	/CD	/CTS	/DSR	SEL	POUT	BUSY
$BFD100	PRB	/MTR	/SEL3	/SEL2	/SEL1	/SEL0	/SIDE	DIR	/STEP
$BFD200	DDRA	1	1	0	0	0	0	0	0
$BFD300	DDRB	1	1	1	1	1	1	1	1
$BFD400	TALO	Timer A is used only for serial data transfer							
$BFD500	TAHI	otherwise it is free							
$BFD600	TBLO	Timer B is used to synchronize the blitter with the screen							
$BFD700	TBHI	otherwise it is free							
$BFD800	E. LSB	The event counter in CIA-B counts the							
$BFD900	E. 8-15	horizontal sync pulses							
$BFDA00	E. MSB	15625 per second (PAL standard)							
$BFDC00	SP	Unused							
$BFDD00	ICR	Interrupt control register							
$BFDE00	CRA	Control register A							
$BFDF00	CRB	Control register B							

The following list shows the various signal lines of the Amiga's CIAs:

CIA-A

/IRQ	/INT2 input from Paula	
/RES	System reset line	
D0-D7	Processor data bus bits 0-7	
A0-A3	Processor address bus bits 8-11	
Phi2	CIA clock input (716 kHz)	
R/W	Processor R/W	
PA7	Game port 1 pin 6 (fire button)	
PA6	Game port 0 pin 6 (fire button)	
PA5	/RDY	"disk ready" signal from disk drive
PA4	/TK0	"disk track 00" signal from disk drive
PA3	/WPRO	"write protect" signal from disk drive
PA2	/CHNG	"disk change" signal from disk drive
PA1	LED	Control over the power LED (0 = on, 1 = off)
PA0	OVL	Memory overlay bit (do not change!)
SP	KDAT	Serial keyboard data
CNT	KCLK	Clock for keyboard data
PB0-PB7		Centronics port data lines
PC	/DRDY	Centronics handshake signal: data ready
FLAG	/ACK	Centronics handshake signal: data acknowledge

CIA-B

/IRQ	/INT6 input from Paula	
/RES	System reset line	
D0-D7	Processor data bus bits 8-15	
A0-A3	Processor address bus bits 8-11	
Phi2	CIA clock input (716 kHz)	
R/W	Processor R/W	
PA7	/DTR	Serial interface, /DTR signal
PA6	/RTS	Serial interface, /RTS signal
PA5	/CD	Serial interface, /CD signal
PA4	/CTS	Serial interface, /CTS signal
PA3	/DSR	Serial interface, /DSR signal
PA2	SEL	"select" signal for Centronics interface
PA1	POUT	"paper out" signal from Centronics interface
PA0	BUSY	"busy" signal from Centronics interface
SP	BUSY	connected directly to PA0
CNT	POUT	connected directly to PA1
PB7	/MTR	"motor" signal to disk drive
PB6	/SEL3	"drive select" for drive 3
PB5	/SEL2	"drive select" for drive 2
PB4	/SEL1	"drive select" for drive 1
PB3	/SEL0	"drive select" for drive 0 (internal)
PB2	/SIDE	"side select" signal to disk drive
PB1	DIR	"direction" signal to disk drive
PB0	/STEP	"step" signal to disk drive
FLAG	/INDEX	"index" signal from disk drive
PC	Not used	

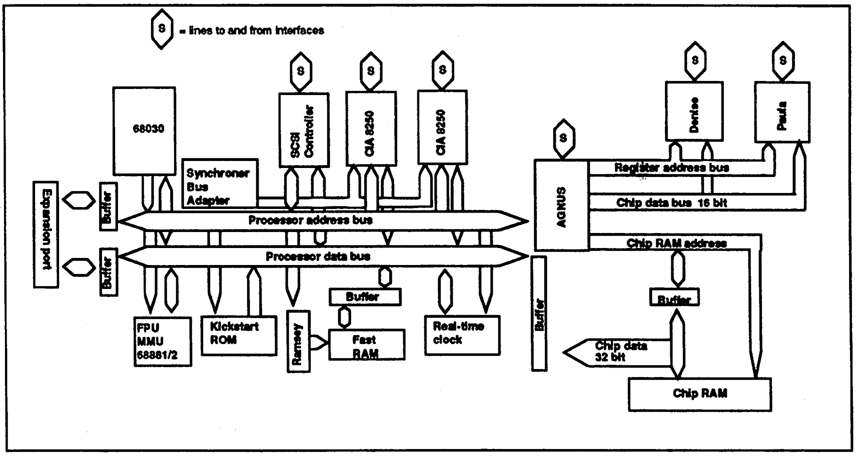
11.4 Custom Chips and the Amiga

The key component of every Amiga system, besides the processor, is the unit formed by Commodore's own specially developed custom chips, Agnus, Denise and Paula. In the course of the Amiga's development, Agnus and Denise have undergone several revisions. The versions used in the Amiga 3000 are called 8372B, 8373 and 8364. These custom chips handle sound generation, screen display, processor-independent diskette access and much more. These tasks are not strictly divided up among the chips so that one is in charge of sound generation, one of graphics and another of diskette operation, which is usually the case with such devices. Instead most tasks are shared among the chips, so that graphics display, for example, is accomplished by two chips working together.

Although the three chips could have been combined into one, it would be more expensive to produce such a complex circuit than the three separate chips.

Other special components are also needed to control a system as complex as the Amiga 3000. These include a revised Buster chip for controlling the expansion slots, a revised Gary as system controller, a newly developed 32-bit DMA controller for the SCSI interface (needed by the hard disk), a Western Digital SCSI interface chip, a controller chip for the FastRAM and a component named Amber as core of the built-in flicker fixer. Before we explain how Agnus, Denise, Paula, and the other special components work, first we'll discuss the structure of the Amiga 3000.

11.4.1 Basic Structure of the Amiga



The structure of the 68030

A simple computer system normally consists of a processor, the ROM with the operating system, a certain amount of RAM, and at least one peripheral component for data input and output. All components are connected to the address and data bus. The processor controls the system and only it can place addresses on the bus and thus read or write data to and from various system components, such as RAM. It also controls bus control signals like the R/W line. Every system also contains control circuits like an address decoder, which activates certain components based on values on the address bus.

Now let's discuss the Amiga. As you can see from the above diagram, the structure of the Amiga deviates somewhat from what we described. On the left side you see the 68030 microprocessor, whose data and address lines are connected directly to the two 8520 CIAs, the Kickstart ROM, the real-time clock and the DMA controller. The Gary chip manages the control lines so that the 68030 can access all these different components. It informs the 68030, for example, whether it must perform a 16- or a 32-bit access to a particular address.

A new chip called Ramsey takes over the management of RAM (configured with full 32-bit addresses, of course), with the capability of handling up to 16 Megabytes of fast RAM. This RAM can be accessed by the DMA controller or DMA-capable expansion cards as well as by the processor.

The four expansion slots are connected to the processor address and data lines with a set of drivers. These drivers are controlled by the A3000's Fat Buster chip, an enhanced version of the A2000's Buster. This chip also controls bus allocation on DMA access. When the SCSI controller or an expansion card wants to access RAM directly, first it must get permission to use the processor's bus lines. Buster manages these requests and communicates with the CPU, granting the appropriate DMA controller permission to use the bus as soon as the CPU frees it.

On the right side of the diagram we find the three custom chips Agnus, Denise and Paula, and the chip RAM, which are all connected to a common data bus. However, this data bus is separated from the processor data bus by a buffer, which can either connect the processor data bus to the chip data bus or can separate the two. The three custom chips are connected to each other through the address register bus, which is directed by Agnus. Since the chip RAM has a much larger address range than the custom chips and also requires multiplexed addresses, there is a separate chip RAM address bus. Multiplexed addresses implies that the RAM chips used in the Amiga have an address range of 2^{18} addresses (256K) and in order to access all the addresses of a chip, 18 address lines are needed. But the actual chips are very small, and such a large number of address lines would require a very large enclosure. To get around this problem, something called multiplexed addressing was introduced. The package has only nine address lines; first the upper nine bits of the address and then the lower nine are applied to these lines. The chip stores the upper nine and then, when the lower nine are applied to the address lines, it has the 18 address bits that it needs.

Why are these two buses separated? The reason is that the various input/output devices need a constant supply of data. For example, the data for individual dots on the screen must be read from the RAM fifty times per second, since a television picture according to the PAL standard is refreshed at the rate of fifty times per second.

A high-resolution graphic on the Amiga can require more than 64K of screen memory. This means that per second 50 x 64K access must be applied to memory. This is nearly 2 million memory accesses per second. If the processor had to perform this task, it would be hopelessly overloaded. Such a high data rate would leave even the 68030 little time for anything else. Furthermore, the Amiga can perform digital sound output and diskette accesses in addition to the graphics, all without using the CPU. The solution lies in the use of another processor that performs all these memory accesses itself. Such a processor is also called a DMA (Direct Memory Access) controller, and the A3000 has two of them, the SCSI-DMA chip and Agnus.

While the SCSI chip is needed only to speed up data transfer between RAM and the SCSI interface, Agnus has more numerous and varied capabilities. However, Agnus can access only the chip RAM. Agnus contains not only the DMA controller, but also the RAM controller for the chip RAM. This is why Agnus is also connected to the chip RAM address bus. The processor can access chip RAM only by using Agnus.

The other chips, Denise and Paula, and also the remainder of Agnus, are constructed like standard peripheral chips. They have a certain number of registers which can be read or written by the processor (or the DMA controller). The individual registers are selected through the register address bus. It has eight lines, so 256 different states are possible. There is no special chip selection. If the address bus has the value 255 or $FF, so that all lines are high, no register is selected. If a valid register number is on these lines, then the chip containing the selected register recognizes this and activates it. This task is performed in the individual chips by the register address decoders. The fact that the selection of a register depends only on its register address and not on the chip in which it is located means that two registers in two different chips can be written with the same value if they have the same register address. This capability is used for some of the registers that contain data needed by more than one chip.

Each chip register can be either a read register or a write register. Switching between read and write by means of a special R/W line, like in the 8520, is not possible. The register address alone determines whether a read or write address is taking place. Registers that can be both read and written are realized by having write access go to one register address and read access to another. This property is more clearly shown in the list of chip registers.

Since Agnus contains the DMA controller, it can also access the custom chip registers itself by outputting an address on the register address bus.

One obvious problem is still unresolved. There is only one data bus and one address bus, which both the processor and the DMA controller want to access. A bus can be controlled by only one bus controller at a time. If two chips tried to place an address on the bus simultaneously, there would be a problem known as bus contention, leading to a system crash. Therefore the chips must share access to the bus by taking turns. Naturally each would like to have the bus for itself as often as possible. This problem is solved by the Amiga on three levels:

First, both normally continuous buses are divided on the Amiga into two parts. One (on the left in the diagram) connects all the components that are usually accessed only by the processor. (Although the SCSI controller and expansion cards can also access RAM on the processor's behalf, these DMA accesses usually take place only when needed, for example, when accessing a SCSI hard disk, which reduces processor speed.) When the 68030 accesses one of these components, Gary uses the buffers to break the connections of the processor address and data buses to the chip address and data buses. This way both the processor and Agnus, each on its own side, can access the bus undisturbed. This gives the processor quick access to the operating system and to its RAM. This RAM connected directly to the processor data and address bus is called fast RAM, since the processor can always access it without slowing down, if it has the bus at that moment.

Secondly, bus accesses from Agnus and from the processor are nested, so that normally even on accesses to chip RAM or chip registers, a 68000 does not have to be delayed. For such an access the buffers connect the two systems again.

As a third and final solution, the processor can wait until Agnus has finished its DMA accesses and the bus is free again. This occurs only when very high graphics resolutions have been selected or the Blitter is being used. Agnus, Denise and Paula were originally drafted for an Amiga with a 68000 processor. Despite certain revisions for the A3000, they have some problems working with the 68030. Nesting the accesses to chip RAM on an Amiga with the 68000 enables alternating access; so the processor does not have to wait. The A3000's 68030, however, accesses memory with substantially higher speed, while Agnus's clock frequency remains unchanged. The result is that the A3000's CPU must insert wait cycles when it wants to access chip RAM.

Another disadvantage of the custom chips is their limitation to a 16-bit wide data bus. While the A3000 manages chip RAM as true 32-bit RAM, special buffers are required for RAM access by Agnus to ensure that access proceeds to the correct half of the chip RAM data bus.

11.4.2 The Structure of Agnus

The 2 Meg version of Agnus in the A3000 is also called Super Agnus.

All clock generation for the custom chips is integrated in Fat Agnus. Only the 28 MHz base clock must be supplied. Agnus also assumes the management of chip RAM, generating the necessary RAS and CAS signals together with the multiplexed RAM addresses. Agnus can manage chip RAM on its own. But since the A3000's developers wanted to endow it with true 32-bit chip RAM access, a conversion process was necessary to utilize the 16-bit wide data bus. Since Agnus is still used in the older Amiga models, other chips were connected to Agnus for this purpose, instead of being integrated into it.

Agnus's main responsibility is all of the DMA control. Each of the six possible DMA sources has its own control logic. They are all connected to the chip RAM address generator as well as the register address generator. These address generators create the RAM address of the desired chip RAM location and the register address of the destination register. In this manner the DMA logic units supply the appropriate chip registers with data from the RAM or write the contents of a given register into RAM.

Also connected to the chip RAM address generator is the refresh counter, which creates the refresh signals necessary for the operation of the dynamic RAM chips.

Agnus controls the synchronization of the individual DMA accesses. The fundamental reference for this is a screen line. In each screen line, 255 memory accesses take place, which Agnus allocates among the individual DMA channels and the 68030. Since it always needs the current row and column positions for this, Agnus also contains the raster and column counters. These counters for the beam position also create the horizontal and vertical synchronization signals, which signal to the monitor the start of a new line (H-sync) and a new picture (V-sync). The horizontal and vertical synchronization signals can also be fed in from outside Agnus and then control the internal raster line and column counters. This allows the video picture of the Amiga to be synchronized to that of another source, such as a video recorder. Called a genlock, this is easily accomplished on the Amiga. (Simply stated, synchronizing two video pictures means that the individual raster lines and the individual pictures of the two signals start at the same time.)

Two other important elements in Agnus are the Blitter and the Copper coprocessor. The Blitter is a special circuit that can manipulate or move areas of memory. It can be used to relieve the 68030 of some work, since it can perform these operations faster than the processor can. In addition, the Blitter is capable of drawing lines and filling surfaces. The Copper is a simple coprocessor. Its programs, called Copper lists, contain only three different commands. The Copper can change various chip registers at predetermined points in time.

The following are the functions of the individual pins:

Data bus: D0-D15

The 16 data lines are connected directly to the chip RAM data bus. Internally all of the chip registers are connected through a buffer to the bus.

Processor address bus: A1-A20

These inputs are connected with the address lines of the 68030 and are used by Agnus when the CPU accesses chip RAM or one of the chip registers.

CPU bus signals: _LDS,_UDS,_R/W and _AS

These signals inform Agnus about, among other things, the validity of processor addresses.

Register address bus: RGA1-RGA8 (ReGisterAddress)

On a DMA access Agnus selects the appropriate chip register over the register address bus. If the _REGEN line is low, meaning the processor is accessing a chip register, Agnus transfers the CPU-referenced register address to the register address bus. With a value of $FF on the register address bus (all lines high), this is inactive.

The address lines for the dynamic RAM: DRA0-DRA9 (Dynamic RAM Address)

Agnus always generates the multiplexed addresses for the chip RAM. On a DMA access these originate from one of the internal address counters, the processor signals access to RAM (_RAMEN low), and Agnus simply switches the addresses through to chip RAM. Agnus can address 2 Meg of chip RAM (2 x 10 address lines for 20 address bits, 2^{20} gives an area of roughly 1 million addresses, but since the chip RAM for Agnus has a width of 16 bits, the memory available to Agnus is 2 Meg).

The chip RAM control lines: _RAS,_CASU,_CASL,_WE

The _RAS and _CAS signals activate the dynamic RAM chips. The _WE line determines whether Agnus is writing data to chip RAM or reading from it.

The bus control signals: _RAMEN, _REGEN, _BLITS, _BLIT

These four lines are connected to Gary. With the _BLIT line Agnus tells Gary that it will take over the bus on the next bus cycle. This line always takes precedence over a processor bus request. If Agnus requires the bus for several consecutive bus cycles, the 68030 must wait.

The _RAMEN (RAM ENable) and _REGEN (REGister ENable) inform Agnus that the processor wants to access chip RAM or a chip register.

The BLITS signal (BLITter Slow down) signals Agnus that the processor is waiting for access. Depending on the internal status, Agnus gives up the bus to the processor for a cycle.

The control signals: RES, INT3, DMAL

The RES signal (RESet) is connected directly to the system reset line and returns Agnus to a predefined start-up state.

The INT3 line (INTerrupt at level 3) is an output and is connected directly to the Paula line with the same name. Agnus uses this line to inform the interrupt logic in Paula that a component in Agnus has generated an interrupt.

The DMAL line (DMA Request Line) also connects Agnus to Paula; only this time the connection occurs in the opposite direction. Paula uses this line to tell Agnus to perform a DMA transfer.

The synchronization signals: HSY, VSY, CSY and LP

Normally the synchronization signals for the monitor appear on the HSY (Horizontal SYnc) and VSY (Vertical SYnc) lines. The signal on the CSY (Composite SYnc) line is the sum of HSY and VSY and is used to connect to monitors that need a combined signal, as well as the circuit that creates the video signal, the video mixer.

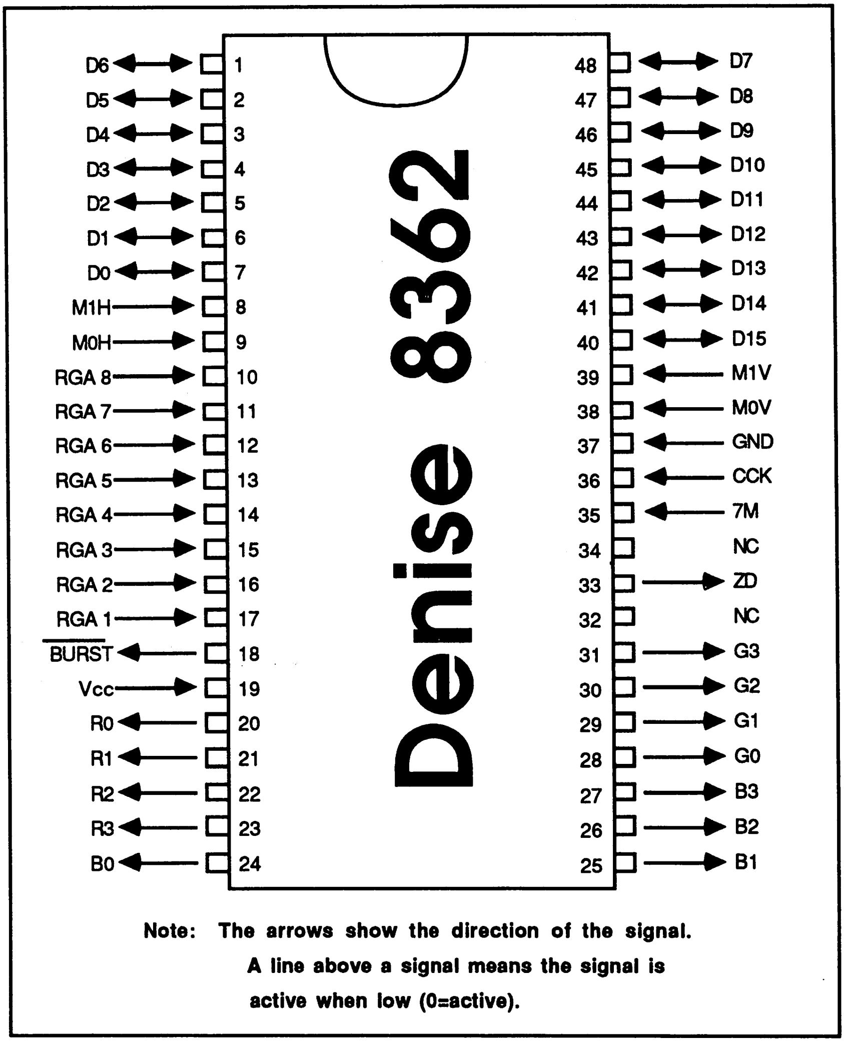
The LP line (Light Pen) is an input line for connecting a light pen. The content of the raster counter register is stored when a negative transition occurs on this pin.

The HSY and VSY lines can also be used as inputs and thus allow Agnus to be externally synchronized (genlock).

The clock lines: 28 MHz, 7 MHz, CCK, CCKQ, _CDAC

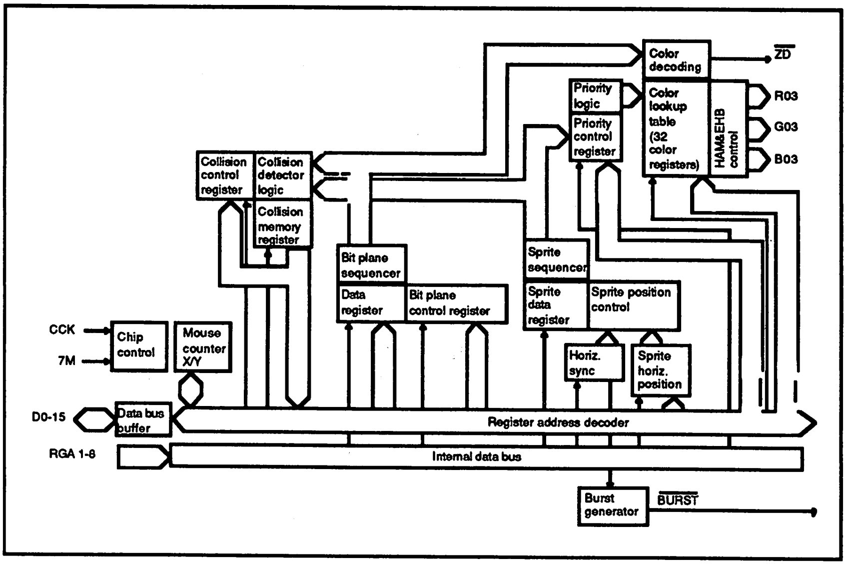
The 28 MHz signal forms the base clock for Agnus. The two 7-MHz signals, 7 MHz and _CDAC, and the two 3.5-MHz signals, CCK and CCKQ, are produced from it. These four serve as clock signals for Denise, Paula and a few other chips.

11.4.3 The Structure of Denise



Denise

In general, the function of Denise can be described as graph generation. The first part of this task is already accomplished by Agnus. Agnus fetches the current graphic data from the chip RAM and writes them to the registers responsible for the bit level manipulations in Denise. It does the same for the sprite data. Denise always contains all graphic and sprite data for 16 pixels, since a bit always corresponds to one pixel on the screen and the data registers all have a width of one word, or 16 bits. These data must be converted into the appropriate RGB representation by Denise. First, the graphic data are converted from a parallel 16-bit representation to a serial data stream by means of the bit-level sequencer. Since a maximum of six bit levels are possible, this function block is repeated six times. The serial data streams from the individual bit-level sequencers are now combined into a maximum 6-bit wide data stream.



Block circuit diagram of Denise

The priority control logic selects the valid data for the current pixel based on its priority from among the graphic data from the bit-level sequencers and the sprite data from the sprite sequencers. According to this data the color decoder selects one of the 32 color registers. The value of this register is then output as a digital RGB signal. If the Hold-And-Modify (HAM) or the Extra-Half-Bright (EHB) mode is selected, the data from the color register is modified accordingly before it leaves the chip.

The data from the sequencers is also fed into the collision-control logic. As its name implies, this checks the data for a collision between the bit levels and the sprites and places the results of this test into the collision register.

The last function of Denise has nothing to do with the screen display. Denise also contains the mouse counter, which contains the current X and Y positions of the mice.

Here is a functional description of Denise's pins:

The data bus: D0-D15

The 16 data bus lines are, like those of Agnus, connected to the chip data bus.

Register address bus: RGA1-RGA8

The register address bus is a pure input on Denise. The register address decoder selects the appropriate internal register with the help of the value on the register address bus.

The clock inputs: CCK and 7M

Denise's timing is regulated by the CCK signal. The CCK pin is connected to the CCK pin on Agnus. The clock signal on the 7M line (7 Megahertz) has a frequency of 7.15909 MHz. The Denise chip needs this additional frequency to output the individual pixels because the pixel frequency is greater than the 3.58 MHz of the CCK signal. A pixel at the lowest resolution (320 pixels/line) has exactly the duration of a 7M clock signal. In high-resolution mode (640 pixels/line) two pixels are output per 7M cycle, one on each edge of its signal. The output signals: R0-3, G0-3, B0-3, ZD and BURST.

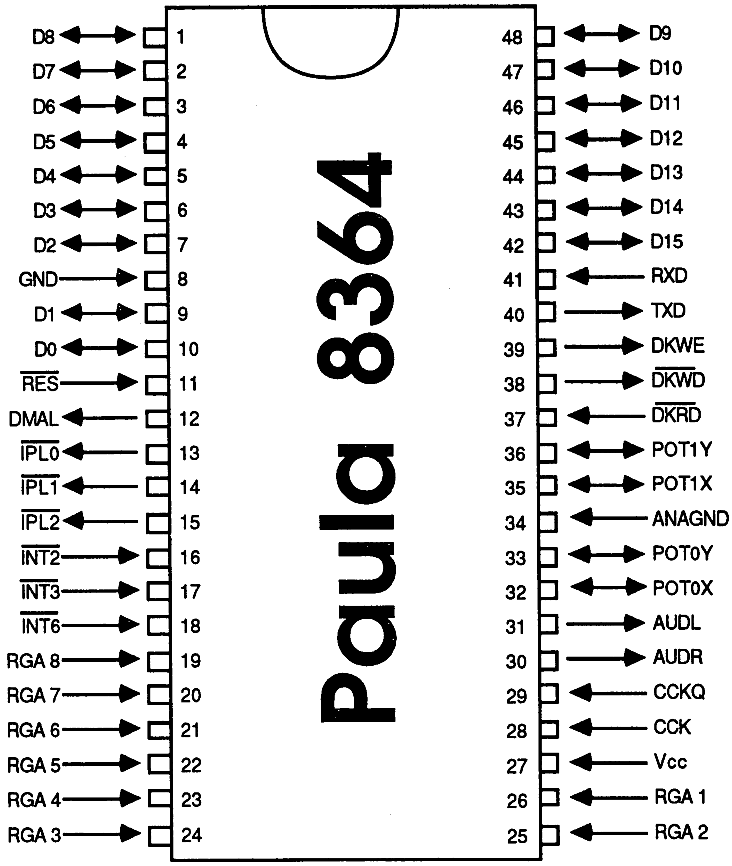
The lines R0-3, G0-3 and B0-3 represent the RGB outputs of Denise. Denise outputs the corresponding values digitally. Each of the three color components is represented by four bits. This allows 16 values per component and 16x16x16 (4096) total colors. After they leave Denise, the three color signals run through a buffer and then through three digital-to-analog converters to transform them into an analog RGB signal, which is then fed to the RGB port.

The last output signal of Denise is the ZD signal (Zero Detect or background indicator). It is always low when a pixel is being displayed in the background color (i.e., when its color comes from color register number 0). This signal is used in the genlock adapter for switching between the external video signal (ZD=0) and the Amiga's video signal (ZD=1). The ZD signal is also available on the RGB port.

The mouse/joystick inputs: M0H, M1H, M0V, M1V

These four inputs correspond directly to the mouse inputs of the two game ports (or joystick connectors). Since the Amiga has two game ports, it must actually have eight inputs. Apparently only four pins were free on Denise so Commodore used the following method to read all the inputs: The eight input lines of the two game ports go to a switch, which connects the four lines of either the front or the back port to the four inputs on Denise. This switching is performed in synchronization with Denise's clock, so that Denise can internally distribute the four lines to two registers, one for each game port. The section on interfaces contains more information about the game ports.

11.4.4 The Structure of Paula



**NOTE: The arrows show the direction of the signal.
A line above a signal means the signal is active when low (0=active).**

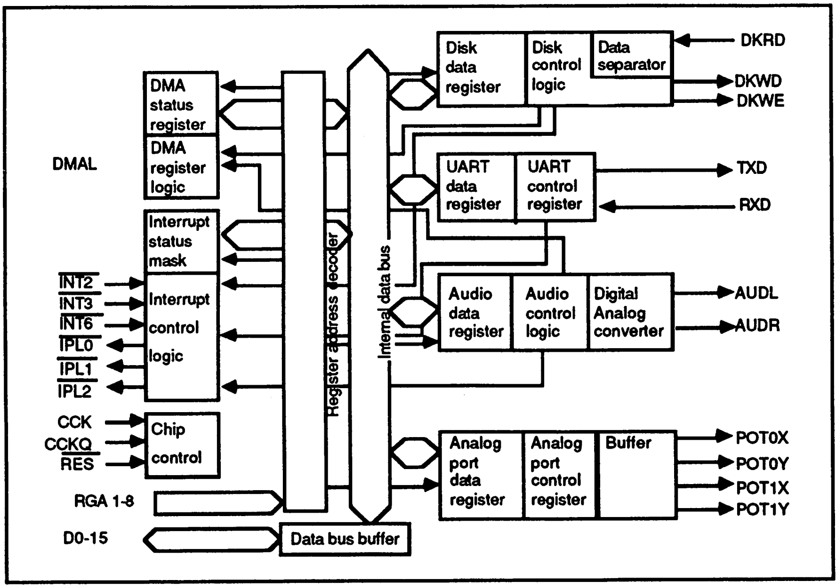
Paula

Paula's tasks fall mainly in the I/O area, namely the diskette I/O, the serial I/O, the sound output and reading the analog inputs. In addition, Paula handles all interrupt control. All the interrupts that occur in the system run through this chip. From the fourteen possible interrupt sources, Paula creates the interrupt signals for the 68030. Interrupts on levels 1-6 are generated on the 68030's IPL lines. Paula gives the programmer the possibility to allow or prohibit each of the fourteen interrupt sources.

The disk data transfer and the sound output are performed using DMA. Since, in these two functions, Agnus does not know when the next data word is ready for a DMA transfer, Paula has a DMAL line, which it can use to tell Agnus when a DMA access is needed.

The serial communication is handled by a UART (Universal Asynchronous Receive Transmit) component inside Paula.

The function of the UART, the four audio channels and the analog ports are described later in the section on programming the custom chips. The following is a description of the pin functions:



Block circuit diagram of Paula

Data bus: D0-15

As with the other chips, connected to the chip data bus.

Register address bus: RGA 1-8

As with Denise.

The clock signals and reset: CCK, CCKQ and RES

Paula contains the same clock signals as Agnus. The reset line RES returns the chip to a defined start-up state.

DMA request: DMAL

With this line Paula signals Agnus that a DMA transfer is needed.

Audio outputs: AUDL and AUDR

The outputs AUDL and AUDR (AUDio Left and AUDio Right) are analog outputs on which Paula places the sound signals it generates. AUDL carries the internal sound channels 0 and 3, and AUDR the channels 1 and 2.

The serial interface lines: TXD and RXD

RXD (Receive Data) is the serial input to the UART, and TXD (Transmit Data) is the serial output. These lines have TTL levels, which means that their input/output voltages range from 0 to 5 volts. An additional level converter subsequently creates the +12/-5 volts for the Amiga's serial RS232 interface.

The analog inputs: POT0X, POT0Y, POT1X, POT1Y

The inputs POT0X and POT0Y are connected to the corresponding lines from game port 0, and POT1X and POT1Y are connected to port 1. Paddles or analog joysticks can be connected to these inputs. These input devices contain variable resistors, called potentiometers, which lie between +5 volts and the POT inputs. Paula can read the values of these resistors and place them in internal registers. The POT inputs can also be configured as outputs through software. Unfortunately the sampling rate is only 50 Hz (the screen repeat frequency). Therefore it is not possible, for example, to use a VCR (Voltage Controlled Resistor) to digitize music and speech.

The disk lines: DKRD, DRWD, DKWE

Through the DKRD lines (DisK ReaD) Paula receives the read data from the diskette. The DKWD line (DisK Write) is the output for data to the disk drive. The DKWE line (DisK Write Enable) serves to switch the drive from read to write.

The interrupt lines: INT2, INT3, INT6 and IPL0, IPL1, IPL2

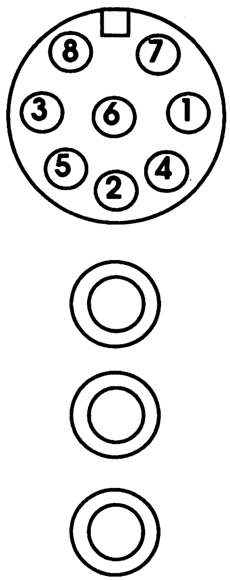
Paula receives instructions through the three INT lines to create an interrupt on the appropriate level. The INT2 line is normally the one connected to the CIA-A 8520. This line is also connected to the expansion port and the serial interface. If it is low, Paula creates an interrupt on level 2, provided that an interrupt at this level is allowed. The INT3 line is connected to the corresponding output from Agnus and the INT6 line to CIA-B and the expansion port. All other interrupts occur within the I/O components in Paula.

The IPL0-IPL2 lines (Interrupt Pending Level) are connected directly to the corresponding processor lines. Paula uses these to create a processor interrupt at a given level.

11.5 The Amiga Interfaces

Every computer needs contact with the outside world. Because of various connections and interfaces, it's possible to connect the Amiga to virtually any external device.

11.5.1 The Audio Outputs

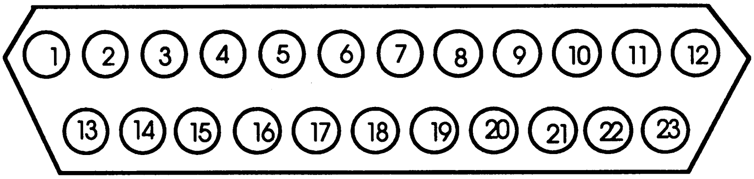


The audio outputs

The audio signal is available through two phono connectors on the rear of the case. The right stereo channel is the red connector and the left is the white. A standard stereo phono cable can be used to connect these jacks to a stereo (AUX, TAPE or CD input). The output resistance of each channel is 1 KOhm (1000 Ohms).

The outputs are protected against short circuit and have 360 Ohms impedance.

11.5.2 The RGB Connector



INPUT	1	XCLK	External clock frequency
INPUT	2	XCLKEN	Switch for external clock
OUTPUT	3	R	Analog red signal
OUTPUT	4	G	Analog green signal
OUTPUT	5	B	Analog blue signal
OUTPUT	6	DI	Digital brightness signal
OUTPUT	7	DB	Digital blue signal
OUTPUT	8	DG	Digital green signal
OUTPUT	9	DR	Digital red signal
OUTPUT	10	CSY	Buffered composite sync signal
IN/OUT	11	HSY	Horizontal synchronization signal
IN/OUT	12	VSY	Vertical synchronization signal
	13	GND	
OUTPUT	14	ZD	Background indicator signal
OUTPUT	15	C1U	Amiga C1U timer (3.58 MHz)
	16	GND	
	17	GND	
	18	GND	
	19	GND	
	20	GND	
	21	-5 volts	
	22	+12 volts	
	23	+5 volts	

The RGB connector

The RGB connector allows various RGB monitors as well as special expansions, such as a genlock adapter, to be connected to the Amiga. To connect an analog RGB monitor like the standard Amiga monitor, the three analog RGB outputs and the CompositeSync output are used. The RGB signal on these three lines comes from the conversion of the buffered RGB digital signals from Denise into suitable analog signals by means of three 4-bit digital-to-analog converters. The Composite Sync signal comes from Agnus and is formed by mixing the horizontal and vertical sync signals. All of these four lines are provided with transistor buffers and 75 Ohm series resistances.

The lines DI, DB, DG and DR are provided for connecting a digital RGB monitor. The source of the digital RGB signals is the digital RGB output from Denise. Each of the three color lines is connected to the most significant (highest) respective color line from Denise (e.g., DB to B3 from Denise). Interestingly, the intensity or brightness line DI is connected to the B0 line. The four lines have 47 Ohm output resistances and TTL levels.

The HSY and VSY connections on the RGB connector are provided for monitors that require separate synchronization signals. Use these lines carefully, since they are connected through 47 Ohm resistors directly to the HSY and VSY pins of Agnus. They also have TTL levels.

If the genlock bit in Agnus is set (see the section on programming the hardware), these two lines become inputs. The Amiga then synchronizes its own video signal to the synchronization signals on the HSY and VSY lines. These lines also require TTL levels as input. As usual, the synchronization signals are active low, which means that the lines are normally at 5 volts. Only during the active synchronization pulse is the line at 0 volts.

Using certain control bits from Agnus, it is also possible to reverse the polarity of the synchronization signals (refer to Section 11.7).

Kickstart Versions 1.2 and later automatically recognize on reset whether signals are present on the two Sync lines. If so, the Amiga switches to external synchronization.

Another signal, related to genlock, is the ZD signal (Zero Detect). The Amiga places this signal low whenever the pixel currently being displayed comes from a specified color register or bit-plane.

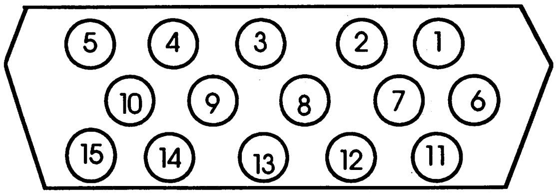
During the vertical blanking gaps, when VSY=0, the function of the ZD line changes. Then it reflects the state of the GAUD (Genlock AUDio enable) bit from Agnus register $100 (BPLCON0). This signal is used by the genlock interface to switch the sound signal.

The ZD line is usually of no interest to the normal user, since it is required only by the genlock interface. The ZD signal from Denise pin 33 is buffered with a 74HC244 driver, so that the signal has TTL levels.

The remaining lines of the RGB connector have nothing to do with the RGB signal. The C1U line is a 3.58 MHz clock line and corresponds to the inverted CLK signal of the custom chips.

The XCLK (eXternal CLocK) and XCLKEN (eXternal CLocKENable) lines are used to feed an external clock frequency into the Amiga. All clock signals in the Amiga are derived from a single 28MHz clock. This 28MHz master clock can be replaced by another clock frequency on the XCLK input by pulling the XCLKEN low. The ground pin 13 should be used when using the XCLK and XCLKEN lines. It is connected directly with the ground line of the clock generation circuit. The fed-in clock should not differ greatly from the master clock (28 MHz).

11.5.3 The VGA Connector




```
1  Red
2  Green
3  Blue
4-8   Ground
10-12 Ground
13 H-Sync
14 V-Sync
```

The VGA connector

The VGA connector is used for connecting IBM-VGA-compatible or multisync monitors. It carries an analog RGB signal with separate H and V Sync lines. Internally it is connected to the flicker fixer output. A switch on the rear panel of the A3000 can be set so that the flicker fixer passes the RGB signal unchanged to the VGA connector.

Since the synchronization signal on the VGA connector is output only, it is not possible to connect a genlock adapter.

11.5.4 The Video Slot

The A3000 video slot is closely tied to the signals on the RGB port. It consists of two 36-pin slot connectors, identical to those of the expanded IBM bus, which are arranged in line with a Zorro expansion slot. The slots have the following pin configurations:

Rear slot (relative to rear of chassis):

Pin	Function	Pin	Function
1	Reserved	2	Reserved
3	Left audio output	4	Right audio output
5	Reserved	6	+5 volts
7	Analog red	8	+5 volts
9	Ground	10	+12 volts
11	Analog green	12	Ground
13	Ground	14	Composite sync, direct
15	Analog blue	16	/XCLKEN
17	Ground	18	BURST
19	/C4 sync signal	20	Ground
21	Ground	22	Horizontal sync
23	B0	24	Ground
25	B3	26	Vertical sync
27	G3	28	Composite sync, buffered
29	R3	30	/ZD (also called /PIXELSW)
31	-5 volts	32	Ground
33	XCLK	34	/C1 sync signal
35	+5 volts	36	Pstrobe

Front slot:

Pin	Function	Pin	Function
1	Ground	2	R0
3	R1	4	R2
5	Ground	6	G0
7	G1	8	G2
9	Ground	10	B1
11	B2	12	Ground
13	Composite video	14	TBASE
15	CDAC sync signal	16	POUT (Paper out)
17	/C3 sync signal	18	BUSY
19	/LPEN	20	/ACK (Acknowledge)
21	SEL (Select)	22	Ground
23	PD0	24	PD1
25	PD2	26	PD3
27	PD4	28	PD5
29	PD6	30	PD7
31	/LED	32	Ground
33	Left audio unfiltered	34	Audio ground
35	Right audio unfiltered	36	Audio ground

Almost all these signals are carried either by the RGB port or the Centronics port. The rest of the signals have the following meanings:

Left and right audio outputs:

These two pins are connected directly to the audio sockets.

Audio left/right unfiltered:

The audio signals on these lines have not yet gone through the low pass filter.

R0 to R3, B0 to B3 and G0 to G3:

These are the digital RGB signals from Denise, buffered through 74HCT244.

/LPEN:

This is the Agnus lightpen input.

/LED:

This indicates the status of the power LED control line. It tells the genlock card whether the audio filter is on or off.

Composite video:

In the A3000, this signal occurs at this slot only (unlike the older Amiga models, which also had a audio jack for it). It is a video-compatible black-and-white signal that can be used, for example, to connect the Amiga to a TV with video input.

TBASE:

TBASE is the time base for the CIA-A event counter, which Kickstart uses as a system clock. A jumper can be used to determine the source of TBASE. This jumper, called J350, connects the TBASE line either with the ticks from the AC electrical source (50 Hz) or with the VSync line from Agnus. Since its frequency is also 50 Hz, the jumper position is normally the same. Preferably the jumper is on the source frequency (pins 1 and 2), which is generally more constant, causing the clock to run more accurately.

11.5.5 The Centronics Interface

The Centronics interface of the Amiga is a computer enthusiast's dream. Any one of a tremendous array of IBM-compatible printers can be connected directly to it.

Output	1	/Strobe - data ready
I/O	2	PD0, Data bit 0
I/O	3	PD1, Data bit 1
I/O	4	PD2, Data bit 2
I/O	5	PD3, Data bit 3
I/O	6	PD4, Data bit 4
I/O	7	PD5, Data bit 5
I/O	8	PD6, Data bit 6
I/O	9	PD7, Data bit 7
Input	10	/Acknowledge - Data received
I/O	11	BUSY - printer busy
I/O	12	Paper out
I/O	13	Select - printer ON-LINE
	14	+5 volts
	15	Unused
Output	16	Reset/buffered reset line from Amiga
	17-25	GND

Internally all of the Centronics port lines (except 5 volts and Reset) are connected directly to the port lines of the individual CIAs. The exact assignment is as follows:

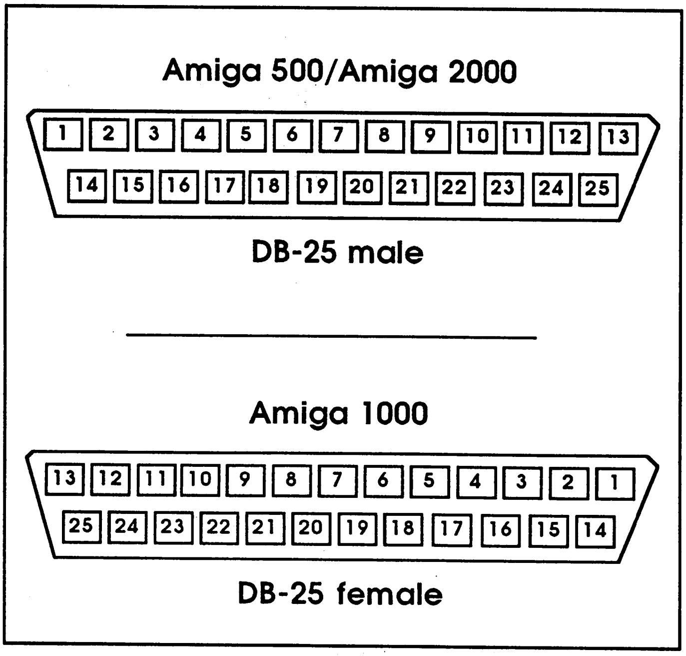
Pin no.	Function	CIA	Pin	Designation
1	Strobe	A	18	PC
2	Data bit 0	A	10	PB0
3	Data bit 1	A	11	PB1
4	Data bit 2	A	12	PB2
5	Data bit 3	A	13	PB3
6	Data bit 4	A	14	PB4
7	Data bit 5	A	15	PB5
8	Data bit 6	A	16	PB6
9	Data bit 7	A	17	PB7
10	Acknowledge	A	24	FLAG
11	Busy	B	2	PA0
	and		39	SP
12	Paper out	B	3	PA1
	and		40	CNT
13	Select	B	4	PA2

The Centronics interface is a parallel interface. The eight data lines carry one data byte. When the computer has placed a valid data byte on the data lines, it clears the STROBE line to 0 for 1.4 microseconds, signaling the printer that a valid byte is ready for it. The printer must then acknowledge this by pulling the Acknowledge line low for at least one microsecond. Once the printer has indicated receipt of the data byte, the computer can place the next one on the bus.

The printer uses the BUSY line to indicate that it is occupied and cannot accept any more data at the moment.

This occurs when the printer buffer is full, for example. The computer then waits until BUSY goes high again before it continues sending data. With the Paper Out line the printer tells the computer that it is out of paper. The Select line is also controlled by the printer and indicates whether it is ONLINE (selected, SEL high) or OFFLINE (unselected, SEL low). The Centronics port is well suited as a universal interface for connecting home-built expansions like an audio digitizer or an EPROM burner, since almost all of its lines can be programmed to be either inputs or outputs.

11.5.6 The Serial Interface



The serial interface

The serial interface has all of the usual RS-232 signal lines. In addition, there are many signals on this connector that have nothing to do with serial communications. The lines TXD, RXD, DSR, CTS, DTR, RTS and CD belong to the RS-232 interface. The TXD and RXD lines are the actual serial data lines.

The TXD line is the serial output from the Amiga and RXD is the input. They are connected to the corresponding lines of Paula. The DTR line tells the peripheral device that the Amiga's serial interface is in operation.

Conversely, with the DSR line the peripheral device signals the Amiga that its interface is ready for operation.

The RTS line tells the peripheral that the Amiga wants to send serial data over the RS-232. The peripheral uses the CTS line to tell the Amiga that it is ready to receive it. The CD signal is usually used only with a modem and indicates that a carrier frequency is being received. These five RS-232 control lines are connected to CIA-B, PA3-PA7 as follows: DSR-PA3; CTS-PA4; CD-PA5; RTS-PA6; DTR-PA7. The RI line is connected through a transistor to the SEL line of the Centronics interface.

	1	GND	Frame Ground
Output	2	TXD	Transmit Data
Input	3	RXD	Receive Data
Output	4	RTS	Request To Send
Input	5	CTS	Clear To Send
Input	6	DSR	Data Set Ready
	7	GND	Signal Ground
Input	8	CD	Carrier Detect
	9	+12 volts	
	10	-12 volts	
Output	11	AUDOUT	left sound channel output
	12		Unused
	13		Unused
	14		Unused
	15		Unused
	16		Unused
	17		Unused
Input	18	AUDIN	right sound channel input
	19		Unused
Output	20	DTR	Data Terminal Ready
	21		Unused
Input	22	RI	Ring Indicator
	23		Unused
	24		Unused
	25		Unused

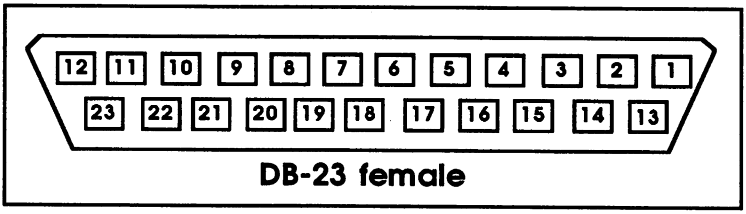
Fortunately, the RS-232 lines are routed through RS-232 drivers instead of being connected directly to the chips. Thus with the appropriate cable, this interface can be connected to almost all the common terminals and modems. Type 1488 inverting RS-232 signal-converters are used as the output drivers. They operate on current supply in the range of +12 to -12 volts. This is the range of the output signals as well. As input buffers, type 1489A chips are used. These accept an input range of -12 to +0.5 volts as low and a range of +3 to +25 volts as high.

According to RS-232 interface conventions, the control lines must be active high, whereas the data lines RXD and TXD are active low (logic 1 is represented by a low signal). Since the drivers invert, the CIA-B port bits that correspond to the control lines are also active low. This means that a bit with the value 0 in CIA-B sets the corresponding RS-232 control line to high. This also applies to the inputs.

The remaining lines on the RS-232 connector have nothing to do with RS-232. The AUDOUT line is connected to the left audio channel and has its own 1 KOhm output resistance. The AUDIN line is connected directly to the AUDR pin of Paula through a 47 Ohm resistor. Audio signals fed into the Amiga on the AUDIN line are sent along with the right sound channel from Paula over the low-pass filter to the right audio output. Nothing else is done to the signal.

The INT2 line is connected directly to the INT2 input of Paula and can generate a level 2 processor interrupt if the corresponding mask bit is set in Paula (see the section on interrupts). The E line is connected via a buffer to the processor E clock (see 11.2.1). A frequency of 3.58 MHz is available on the MCLK line, but this is neither in phase with the RGB interface clock nor with the two 3.58 MHz clocks of the custom chips. Finally, the reset signal is also available on this connector. As you might expect, it too is buffered.

11.5.7 The External Drive Connector



The external drive connector

Input	1	/RDY	Disk ready signal
Input	2	/DKRD	Read data from disk
	3	GND	
	4	GND	
	5	GND	
	6	GND	
	7	GND	
Output	8	/MTRX	Motor on/off
Output	9	/SEL3	Select drive 3
Output	10	/DRES	Disk reset (turn motors off)
Input	11	/CHNG	Disk change
	12	+5 volts	
Output	13	/SIDE	Side selection
Input	14	/WPRO	Write protect
Input	15	/TK0	Track 0 indicator
Output	16	/DKWE	Switch to write
Output	17	/DKWD	Write data to disk
Output	18	/STEP	Move read/write head
Output	19	/DIR	Direction of head movement
	20		unused
Output	21	/SEL2	Select drive 2
Input	22	/INDEX	Index signal from drive
	23	+12 volts	

2	/CHNG
4	/INUSE1
6	/INUSE0
8	/INDEX
10	/SEL0
12	/SEL1
14	Unused
16	/MTR0
18	DIR
20	/STEP
22	/DKWD
24	/DKWE
26	/TK0
28	/WPRO
30	/DKRD
32	/SIDE
34	/RDY

All odd pins are grounded.

Power connector for the internal drive:

1	+5 volts
2	GND
3	GND
4	+12 volts

The disk drive connection on the Amiga is compatible with the Shugart bus. It allows up to four Shugart-compatible disk drives to be connected. The four drives are selected with the four drive selection SELx signals, where x is the number of the drive to be selected. Two of the drives are intended for internal installation in the A3000, so only the lines SEL2 and SEL3 are available on the external drive connector. The SEL0 and SEL1 lines are connected to the internal drives using the internal connector. The following is a description of the Shugart bus signals on the Amiga:

SELX

The Amiga uses the SELX line to select one of the four drives. Except for the MTRX and DRES lines, all other signals are active only after a drive has been activated with the corresponding SELX line.

MTRX

Normally this line turns on all the drive motors. Since this is not practical in a system that can have up to four drives, each drive has its own flip-flop to allow the motors to be controlled separately. A flip-flop is an electronic component that can store a data bit. When a given drive's SEL line goes low, the flip-flop for this drive takes on the value of the MTRX line. The output of the flip-flop is connected to the drive's MTR line. So, for example, if the SEL0 line is pulled low while the MTRX line is at 0, the motor of the first internal disk drive turns on.

For the internal drives these flip-flops are located right on the motherboard. Their outputs are routed via an OR gate to the MTR line of the internal floppy connector. There are separate lines for the LEDs: INUSE0 and INUSE1. An additional flip-flop is required for each external drive.

RDY

When a drive's MTR line is at 0, the RDY (ReaDY) line is used to signal the Amiga that the drive motor has reached its optimum speed and the drive is now ready for read or write accesses. If the MTR line is at 1, meaning the drive motor is turned off, the RDY line is used for a special identification mode (see the following).

DRES

The DRES (Drive RESet) line is connected to the standard Amiga reset and is used only to reset the motor flip-flops so that all drive motors are turned off.

DKRD

The data from the drive selected by SELX travels over the DKRD (DisK Read Data) line to the DKRD pin on Paula.

DKWD

The DKWD (DisK Write Data) line carries data from Paula's DKWD pin to the current drive, where it is then written to the diskette.

DKWE

The DKWE (DisK Write Enable) line switches the drive from read to write. If the line is high, data is read from the diskette. If it is low, data can be written.

SIDE

The SIDE line determines which side of a diskette will be selected for reading or writing. If it is high, side 0 (the lower read/write head) is active. If it is low, side 1 is active.

WPRO

The WPRO (Write PROtect) line tells the Amiga whether the inserted diskette is write-protected. If a write-protected diskette is in the drive, the WPRO line is 0.

STEP

A rising edge on the STEP line (transition from low to high) moves the read/write head of the drive one track in or out, depending on the state of the DIR line. The STEP signal should be at 1 when the SEL line of the activated drive is set back to high or there may be problems with the diskette-change detection.

DIR

The DIR (DIRection) line sets the direction in which the head moves when a pulse is sent on the STEP line. Low means that the head moves in toward the center of the disk and high indicates movement out toward the edge of the disk. Track 0 is the outermost track on the disk.

TK0

The TK0 (TracK 0) line is low whenever the read/write head of the selected drive is on track 0. This allows the head to be brought to a defined position.

INDEX

The INDEX signal is a short pulse which the drive delivers once per revolution of the diskette, between the start and end of a track.

CHNG

With the CHNG (CHaNGe) line, the drive notifies the Amiga of a diskette change. As soon as the diskette is removed from the drive, the CHNG line goes to 0. It remains at 0 until the computer issues a STEP pulse. If there is a diskette in the drive again by this time, CHNG jumps back to 1. Otherwise it remains at 0, and the computer must issue STEP pulses at regular intervals to detect when a diskette has again been inserted in the drive. These regular STEP pulses are the cause of the clicking noise the Amiga drive makes when no diskette is inserted.

INUSE0, INUSE1

The INUSE lines exist only on the internal floppy connector. If INUSE0 is pulled low, drive DF0 turns its LED on. INUSE1 serves the same purpose for drive DF1.

To recognize whether a drive is connected to the bus, there is a special drive identification mode. This involves reading a serial 32-bit data word from the drive. To start the identification, the MTR line of the drive in question must be turned on briefly and then off again (the description of the MTRX line explains how this is done). This resets the serial shift register in the drive. The individual data bits can then be read by 32 iterations of the following procedure: pull the SELX line low, read the value of the RDY line as a data bit, then return the SELX line to high. The first bit received is the MSB (Most Significant Bit) of the data word. Since the RDY line is active low, the data bits must be inverted.

The following are the standard definitions for external drives:

```
$0000 0000    No drive connected                 (00)
$FFFF FFFF    Standard Amiga 3 1/2" drive        (11)
$5555 5555    Amiga 5 1/4" drive,  2x40 tracks   (01)
```

As you can see, there are currently so few different identifications that only the first two bits must be read. The values in parentheses are the combinations of these two bits.

As mentioned before, all the lines except DRES affect only the drive selected by SELX. Originally the MTRX line was also independent of SELX, but the Amiga developers changed this by adding the motor flip-flops.

All lines on the Shugart bus are active low, since the outputs in the Amiga as well as in the drives themselves are provided with open-collector drivers. In the Amiga these are type 7407 drivers.

The four inputs CHNG, WPRO, TK0 and RDY are connected in this order directly to PA4-PA7 of CIA-A. The eight outputs STEP, DIR, SIDE, SEL0, SEL1, SEL2, SEL3 and MTR are connected through the previously mentioned drivers to the internal and external drive connectors. Since these drivers are non-inverting, the bits from the CIAs are inverted. The DKRD, DKW and DKWE lines come from Paula.

Except for the MTRX line and the SEL signals, the connections to the internal and external floppies are the same.

Installing a second internal drive

As mentioned earlier, a second internal drive can be installed in the A3000. If you examine the connecting cable for the built-in floppy, you will discover a second plug as well as an additional power supply connector.

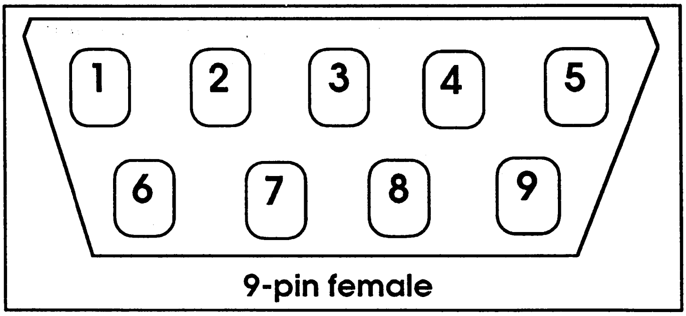
On the built-in drive, usually next to the plug for the connecting cable, there is a switch or jumper for selecting the SELX signal to which the drive should react. This switch should be set to SEL1, since SEL0 is already assigned to the first built-in floppy. The drive can then be installed and the cable connected.

As a final step, the Amiga must be told that an additional drive is present. This happens using the RDY line in the special identification procedure described earlier. The circuit in the A3000 that initiates this recognition procedure for the two internal drives is connected to the RDY line by means of the J351 jumper. This jumper is located on the left, next to the rear slot for the perpendicular bus board, directly behind Denise. Simply switch it over from pins 2-3 to 1-2.

11.5.8 The Game Ports

Use as:

		Mouse port	Joystick	Paddle	Lightpen
Input	1	V-pulse	Up	Unused	Unused
Input	2	H-pulse	Down	Unused	Unused
Input	3	VQ-pulse	Left	Left button	Unused
Input	4	HQ-pulse	Right	Right button	Unused
I/O	5	(Button 3)	Unused	Right port	Button
I/O	6	Button 1	Firebutton	Unused	LP signal
	7	+5 volts	+5 volts	+5 volts	+5 volts
	8	GND	GND	GND	GND
I/O	9	Button 2	Unused	Left port	Unused



Game port pin configuration

The game ports Game-Ports are inputs for input devices other than the keyboard, such as a mouse, joystick, trackball, paddle or lightpen. There are two game ports, the left one being designated as game port 0 and the right as game port 1. The pin assignment of both ports is identical, except that the LP line is present only on game port 0. Internally the game ports are connected to CIA-A, Agnus, Denise and Paula. The individual pins are wired as follows:

Game port 1:

No.	Chip	Pin
1	Denise	M0V (via multiplexer)
2	Denise	M0H (via multiplexer)
3	Denise	M1V (via multiplexer)
4	Denise	M1H (via multiplexer)
5	Paula	P0Y
6	CIA-A	PA6
and	Agnus	
9	Paula	P0X

Game port 2:

No.	Chip	Pin
1	Denise	M0V (via multiplexer)
2	Denise	M0H (via multiplexer)
3	Denise	M1V (via multiplexer)
4	Denise	M1H (via multiplexer)
5	Paula	P1Y
6	CIA-A	PA7
and	Agnus	LP
9	Paula	P1X

The function of the multiplexers was explained previously. The pin assignments for the various input devices were chosen so that almost all standard joysticks, mice, paddles and lightpens can be used. The button line is usually connected to a switch that is pressed when the lightpen touches the screen.

The LP line is the actual lightpen signal, which is generated by the electronics in the pen when the electron beam passes its tip. On the A1000, the lightpen line was connected to game port 0. This meant that you had to use a different mouse connection or disconnect the mouse entirely in order to use a lightpen. Because of this disadvantage, the lightpen on the A3000 is normally connected to port 1. Jumper J352 is used to select where the lightpen will go. Use the jumper to connect pins 2 and 3, and the lightpen will go to game port 1.

All the lines labeled button and the four directions for the joystick are active low. The various input devices contain switches that connect their inputs to ground (GND). A high signal on the input means an open switch, while a closed switch generates a low.

Variable resistors (potentiometers) can be connected to the P0X, P0Y, P1X and P1Y analog inputs. Their value should be 470 KOhms and they should be connected between the corresponding inputs and +5 volts.

The two fire-button lines connected to CIA-A can naturally also be programmed as outputs. Don't overwrite the lowest bit of the port register; otherwise the system crashes (PA0:OVL). The section on programming the custom chips explains how the game port lines are read.

The +5 volt line on the two game ports is not connected directly to the Amiga power supply. A current-protection circuit is inserted in these lines which limits the short-term peak current to 700 mA and the operating current to 400 mA.

11.5.9 The Zorro Bus

Configuration of the 100-pin expansion slot: Zorro II

1	GND	2	GND
3	GND	4	GND
5	+5 volts	6	+5 volts
7	/OWN	8	-5 volts
9	/SLAVEn	10	+12 volts
11	/CFGOUTn	12	CFGINn
13	GND	14	/C3
15	CDAC	16	/C1
17	/OVR	18	XRDY
19	/INT2	20	-12 volts
21	A5	22	/INT6
23	A6	24	A4
25	GND	26	A3
27	A2	28	A7
29	A1	30	A8
31	FC0	32	A9
33	FC1	34	A10
35	FC2	36	A11
37	GND	38	A12
39	A13	40	/EINT7
41	A14	42	/EINT5
43	A15	44	/EINT4
45	A16	46	/BERR
47	A17	48	/VPA
49	GND	50	E
51	/VMA	52	A18
53	/RES	54	A19
55	/HLT	56	A20
57	A22	58	A21
59	A23	60	/BRn
61	GND	62	/BGACK
63	PD15	64	/BGn
65	PD14	66	/DTACK
67	PD13	68	/R/W
69	PD12	70	/LDS
71	PD11	72	/UDS
73	GND	74	/AS
75	PD0	76	PD10
77	PD1	78	PD9
79	PD2	80	PD8
81	PD3	82	PD7
83	PD4	84	PD6
85	GND	86	PD5
87	GND	88	GND
89	GND	90	GND
91	GND	92	7MHz
93	DOE	94	/BUSRST
95	/GBG	96	/EINT1
97	reserved	98	reserved
99	GND	100	GND

In the previous chart "n" equals the number of expansion slots (1-4).

The set of four 100-pin expansion slots, called the Zorro bus, is located on the upright-mounted board in the center of the A3000. The slots can accept all types of expansion cards. On the A2000 this usually refers to hard disk controllers and RAM expansions, but on the A3000 is more likely to mean networks, high-resolution graphic cards, etc.

For this reason the bus specifications have been thoroughly revised. Expansion slots in the A3000 can now operate in two distinct modes: Zorro II compatible (as in the A2000) or the new Zorro III standard.

The Zorro II mode is based mainly on the signals of the 68000:

A0 - A23	Address bus (24 bits, i.e., 16 Megabyte address space)
PD0 - PD15	Processor data bus
IPL0 - IPL2	Processor interrupt lines
FC0 - FC2	Function code lines from the 68000
/AS, /UDS, /LDS, /R/W, /DTACK, /VMA, /VPA	Bus control lines
/RES, /HLT, /BERR, /BG, /BGACK, /BR, E	Miscellaneous control lines from the 68000

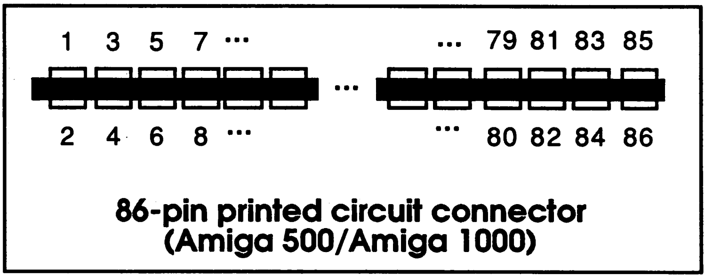
The remaining signals have the following functions:

INT2 and INT6:

These two lines are connected to the Paula pins with the same names. They are used to generate a level 2 or level 6 interrupt.

CDAC, /C3, /C1, 7M and 28M:

These are the various Amiga clock signals. The clock signals CCK and CCKQ are generated by Agnus and serve as base clock signals for Agnus, Denise, Paula and the chip RAM.



Pin configuration of the expansion port

/OVR

With /OVR low the DTACK signal produced by Gary can be disabled for the memory range from $200000 to $9FFFFF.

/XRDY

This signal serves a similar purpose to /OVR. If /XRDY is pulled low less than 60 nanoseconds after /AS, Gary delays the /DTACK signal until /XRDY again goes high. This allows the use of slow expansion cards that cannot respond without wait states.

/BUSRES

This is a buffered reset signal. While the Amiga can also be reset from an expansion card with the RES circuit, the /BUSRES line is intended only for resetting the card itself. Normally you would not want to reset the Amiga from a card, and should therefore only use the /BUSRES line.

/SLAVEn

Every slot has its own /SLAVE line. An expansion card must set /SLAVE to low as soon as it recognizes an address that is valid for it, so that the data and address buffers can be correctly switched. If more than one card sets /SLAVE to low, Gary generates a bus error. The same holds true if a card outside the ranges $200000-$B7FFFF and $E80000-$EFFFFF has /SLAVE set to low.

/CFGIN and /CFGOUT

The /CFGOUT (ConFiG-OUT) line of one slot is always connected to the /CFGIN (ConFiG-IN) line of the next. Each card is configured as soon as the /CFGIN line of its slot is low. When the autoconfiguration of a card is complete, it sets its /CFGOUT output to low to allow configuration of the card in the next slot to proceed.

DOE

This signal comes from Buster and tells the active expansion card that it may activate its data drivers. This prevents data collisions.

/BRn, /BGn and /BGACK

With these lines a card can take over the bus, in effect becoming the DMA controller. The ability to do this is required, for example, by a fast network card. Bus control is assumed as follows: the card pulls /BR low ->, the processor responds with /BG ->, the card pulls /BGACK low and returns /BR to high. It now owns the bus until it returns /BGACK to high.

What happens, though, if two cards pull /BR low at the same instant? To avoid problems, each slot has its own /BR and /BG signals. If more than one slot activates its /BR signal, Buster sees the slot with the lowest number (the one nearest the coprocessor slot) first, and passes the /BR on to the 68030. The /BG response is returned by Buster to this same slot, which becomes bus master and pulls /BGACK low. The other slots that have their /BR lines low must wait until the one currently active has finished its DMA.

/OWN

A card must pull this line low when it has assumed the role of bus master as previously described. This is necessary to reverse the direction of the data or address buffer, since the bus master generating the addresses is now on the other side of the buffer.

/GBG

This is the original /BG from the processor.

The interrupt lines /EINT1, /EINT4, /EINT5 and /EINT7

These lines generate the corresponding level interrupts. Unlike the /INT2 and /INT6 lines of the expansion port, they cannot be disabled using Paula.

Also on the Zorro bus are the different operating voltages of the A3000: +/- 5 and 12 volts.

The previous description of the pin configuration and functions refers to operation of the expansion bus in Zorro II mode. The A3000 also recognizes a different type of bus protocol, the new Zorro III standard.

When you insert an expansion card into the A3000, the expansion library contained in Kickstart, using the autoconfiguration information on the expansion, checks to see whether it is a Zorro II or a Zorro III card. If the library recognizes a Zorro II card, for example, one developed also for the A2000 circuitry, the bus behaves as previously described, that is, compatible to the A2000 expansion slots.

If the card is determined to be a Zorro III card, the Amiga operating system assigns it an address outside the Amiga's former 16-Megabyte address space. The bus controller treats processor accesses to addresses within the first 16 Meg as Zorro II, and beyond that as Zorro III.

How does the Zorro III standard differ from Zorro II?

Zorro III is a completely new concept. Although the same 100-pin slots are used, the assignments of many of the lines have changed. The essential features of Zorro III are as follows:

- *A full 32 bits for data and address bus*
  The Zorro III bus is a 32-bit bus system, fully supporting the capabilities of the 68030. Since a slot's 100 pins are not sufficient for both data and address lines (2 x 32), these lines are multiplexed. To initiate an access, the Amiga places the 32 address bits on the bus. Then, using a specific control signal, it tells the Zorro III expansion card to store this address. After that the address lines serve as data lines. Address bits 0-7 are not multiplexed, but rather remain stable throughout the access process.

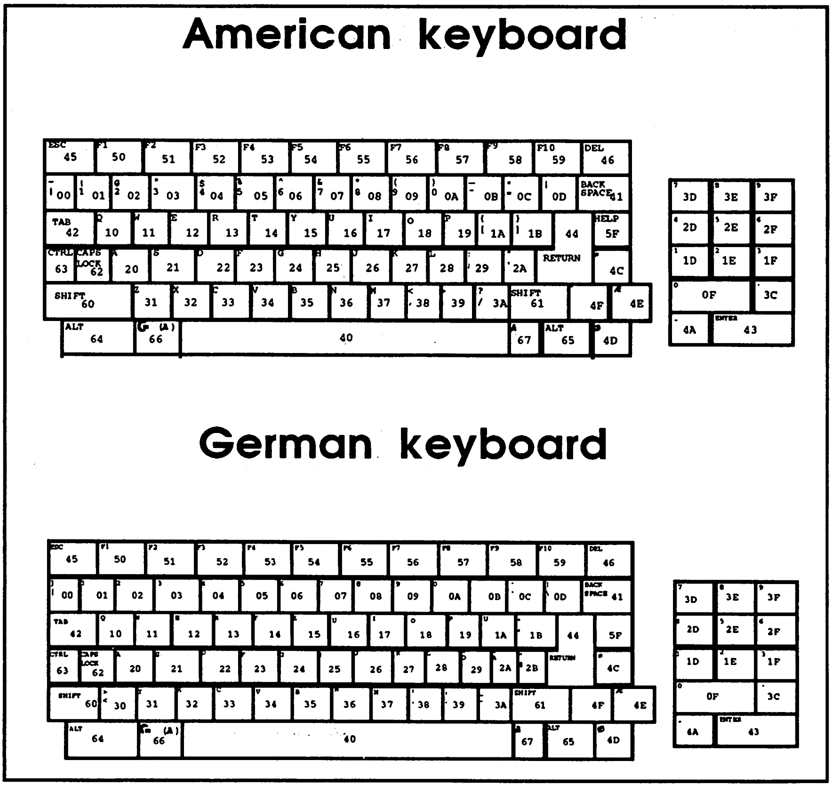
- *Asynchronous bus control*
  The speed of the old Zorro II bus was limited by the timing of the 68000. Bus transfer could not execute faster than a 68000 memory access at a clock frequency of 7.14 MHz. The Zorro III timing is asynchronous, so the speed does not depend on how fast the Amiga or card hardware may be (of course there are models with far higher clock frequencies than previously thought possible).

- *Enhanced interrupt capabilities*
  In the Zorro III bus, expansions can finally fully utilize the interrupt capabilities of the 68030, meaning that a card can use its own interrupt vectors (see the section on 68030).

11.6 The Keyboard

The Amiga keyboard is an intelligent keyboard. It has its own microprocessor, which handles the time-consuming job of reading the keys and returning complete key codes to the Amiga. The following figure shows the layout of the keys and their codes for the German and American versions of the keyboard. As you can see, the codes do not correspond to the ASCII standard. The keyboard only returns raw key codes, which the operating system converts to ASCII using a translation table called the key map. There is, however, a system to the raw key code assignments:

\$00-\$3F	Codes for the letters, digits and punctuation characters. Their assignments correspond to the arrangement on the keyboard.
\$40-\$4F	Codes for the standard special keys like [Spacebar], [Enter], [Tab] etc.
\$50-\$5F	Function keys and HELP.
\$60-\$67	Keys for selecting keyboard control levels ([Shift], Amiga, [Alt] and [Ctrl]).



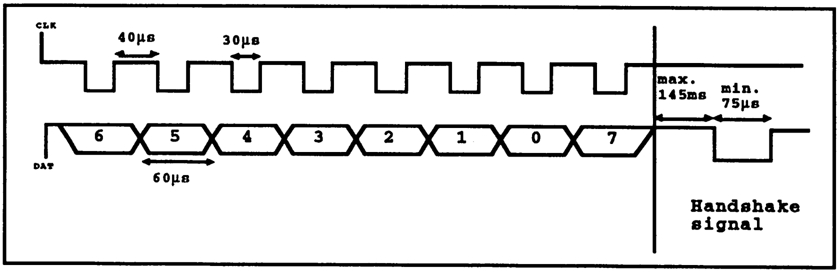
The keyboard

The keyboard processor can do more than read the keys. It can distinguish between when a key is pressed and when it is released. As you can see, all keyboard codes are only 7 bits (values range from \$00-\$7f). The eighth bit is the KEYup/down flag. It is used by the keyboard to tell the computer whether the key was just pressed or released. If the eighth bit is 0, this means that the key was just pressed (KEYdown). If it is 1, then the key was just released (KEYup). This way the Amiga always knows which keys are currently pressed.

The keyboard can thus be used for other purposes that require various keys to be held simultaneously. This includes music programs, for example, which use the keyboard for playing polyphonically.

One exception is the [Caps Lock] key. The keyboard simulates a toggle switch with this key. The first time it is pressed, it engages and the LED goes on. It does not disengage until it is pressed again. The LED then turns off. This behavior is also reflected in the KEYup/down flag. If [Caps Lock] is pressed, the LED turns on, and the key code for [Caps Lock] is sent to the computer along with a cleared eighth bit to show that a key was just pressed. When the key is released, no KEYup code is sent, and the LED stays on. Not until [Caps Lock] is pressed again is a KEYup code sent (with a set eighth bit), and the LED turns off.

11.6.1 Data Transfer from the Keyboard



Data transfer from the keyboard

The keyboard is connected to the Amiga by a four-line coiled cable. Two of the lines are used to supply power to the keyboard electronics (5 volts). The entire data transfer takes place over the remaining two lines. One of these lines is used for data (KDAT), and the other is the clock line (KCLK). Inside the Amiga, KDAT is connected to the serial input SP, and KCLK is connected to the CNT pin of CIA-A. The data transfer is unidirectional. It always runs from the keyboard to the computer. The processor in the keyboard places the individual data bits on the data line (KDAT), accompanied by 20 microsecond-long low pulses on the clock line (KCLK). Between the individual clock pulses are 40 microsecond-long pauses. This amounts to a transfer time of 60 microseconds for each bit, or 480 microseconds per 8-bit byte. The resulting data transfer rate is 16666 baud (bits/second).

After the last bit has been sent, the keyboard waits for a handshake pulse from the computer. The Amiga sends this signal by pulling the KDAT line low for at least 75 microseconds. The exact process can be seen in above the diagram. The bits are not sent in the usual order 7-6-5-4-3-2-1-0, but rotated one bit position to the left: 6-5-4-3-2-1-0-7. For example, the key code for "J" with the eighth bit set is 10100110, and after rotation it is 01001101. The KEYup/down flag is always the last bit sent.

The data line is active low. This means that a 0 is represented by a high signal and a 1 by a low.

The CIA shift register in the Amiga reads the current bit on the SP line at each clock pulse. After eight clock pulses the CIA has received a complete data byte. The CIA then normally generates a level 2 interrupt, which causes the operating system to do the following:

- Read the serial data register in the CIA.
- Invert and right-rotate the byte to get the original key code back.
- Output the handshake pulse.
- Process the received code.

Synchronization

In order to have an error-free data transfer, the timing of the sender and receiver must match. The bit position for the serial transfer must be identical for both. Otherwise the keyboard may have sent all eight bits, while the serial port of the CIA is still somewhere in the middle of the byte. Such a loss of synchronization occurs whenever the Amiga is turned on or the keyboard is plugged into a running Amiga. The computer has no way of recognizing improper synchronization. This task is handled by the keyboard.

After each byte is sent, the keyboard waits a maximum of 145 milliseconds for the handshake signal. If it does not occur in this time, the keyboard processor assumes that a transfer error has occurred and enters a special mode in which it tries to restore the lost synchronization. It sends a 1 on the KDAT line together with a clock pulse and waits another 145ms for the synchronization signal. It repeats this until it receives a handshake signal from the Amiga. Synchronization is now restored.

The data byte received by the Amiga is incorrect, however. The state of the first seven bits is uncertain. Only the last bit received is definitely a 1, because the keyboard processor only outputs 1's during the procedure described above. Since this last bit is the KEYup/down flag, the incorrect code is always a KEYup code, or a released key. This causes less program disturbances than if an incorrect KEYdown code had been sent. This is why each byte is rotated one bit to the left before it is sent, so that the KEYup/down flag is always the last bit sent.

Special codes

There are some other special cases in transmission, which the keyboard tells the Amiga through special key codes.

The following table contains all possible special codes:

Code	Meaning
$FF9	Last key code was incorrect
$FA	Keyboard buffer is full
$FC	Error in keyboard self-test
$FD	Start of keys held on power up
$FE	End of keys held on power up

$F9

The $F9 code is always sent by the keyboard after a loss of synchronization and subsequent resynchronization. This is how the Amiga knows that the last key code was incorrect. After this code the keyboard retransmits the lost key code.

$FA

The keyboard has an internal buffer of 10 characters. When this buffer is full, it sends a $FA to the computer to signal that it must empty the buffer or lose characters.

$FC

After it is turned on, the keyboard processor performs a self-test. This is indicated by the brief lighting of the [Caps Lock] LED. If it discovers an error, it sends a $FC to the Amiga and then goes into an endless loop in which it flashes the LED.

$FD & $FE

If the self-test was successful, the keyboard transmits all the keys that were held when the computer was powered up. To tell the computer this, it starts the transmission with the $FD code.

Then follow the codes of the keys pressed on power up, terminated by the code $FE. After that normal transmission begins.

If no keys were pressed, $FD and $FE are sent in immediate succession.

Reset through the keyboard

The keyboard can also generate a reset on the Amiga. If the two Amiga keys and the Ctrl key are pressed simultaneously, the keyboard processor pulls the KCLK line low for about 0.5 seconds. This causes the reset circuit in the Amiga to generate a processor reset. After at least one of these keys has been released, the keyboard also resets itself. This can be seen by the flashing of the Caps Lock LED.

11.7 Programming the Hardware

The previous sections involved closer looks at the hardware structure of the Amiga. The following pages show how the three custom chips are programmed. Now we'll begin an introduction to software, especially concerning the creation of graphics and sound.

To successfully program the Amiga at the machine level, you must know the memory layout and the addresses of the individual hardware registers.

11.7.1 The Memory Layout

The first figure shows the normal memory configuration of the Amiga as it appears after booting. The entire address range of the 68030 comprises 4 gigabytes (addresses from 0 to $FFFFFFFF). However, this huge address space is not uniformly allocated. In the lower 16 megabytes, space is at a premium because this is where all the system components that existed in the A2000 are located. The remainder is used for the internal fast RAM, the new chips, including the SCSI DMA controller, and the Zorro III expansions area.

RAM

In the Amiga there is a distinction between chip RAM and fast RAM. The chip RAM is used by the custom chips, Agnus, Denise and Paula, to store graphics and sound data. One megabyte of this RAM comes factory-installed, and sockets for adding a second megabyte are included.

The internal fast RAM is available only to the processor, the SCSI DMA chip and any expansion cards. Agnus cannot access it.

Using the new 4-megabit RAM chips, the Amiga can be upgraded to a maximum of 16 megabytes of internal RAM.

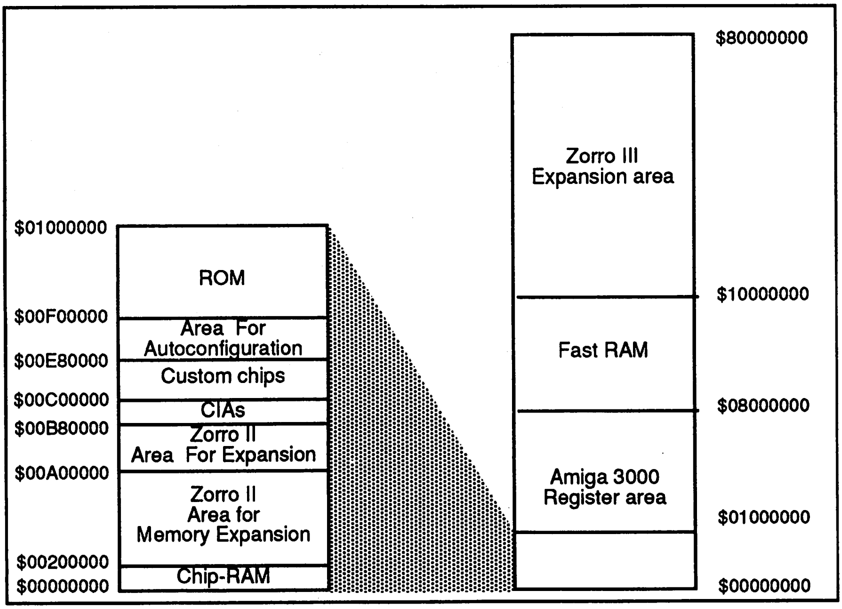
When equipping the A3000 with RAM, first you must decide whether to use 1-Mbit or 4-Mbit RAM chips. The two types cannot be mixed. The jumper J852 selects the chip capacity:

J852 connecting pins 2 and 3: 1 Mbit, organized as 256 K x 4
J852 connecting pins 1 and 2: 4 Mbit, organized as 1 M x 4

If you are upgrading with 4-Mbit RAM chips, the eight factory-installed chips must be removed. However, they can be used to expand the chip RAM.

The RAM is organized in four banks of eight chips each. The socket assignments of the banks are as follows:

Bank 0:	U850 - U857 or U850D - U857D
Bank 1:	U858 - U865
Bank 2:	U866 - U873
Bank 3:	U874 - U881



RAM/ROM allocation in the Amiga

The following types of RAM chips can be used:

xx4256-80 (1 Mbit) or xx4400-80 (4 Mbit) ZIP-mounted (these are the upright as opposed to the flush-mounted DIL chips).

For greater speed, static-column RAM chips can also be used: xx4258-80 or xx4402-80.

These enable the 68030 CPU to operate in burst mode, supplying the internal cache with new data at a rate faster than that of normal data access (this mode is turned on by the SETCPU command in the CLI).

The base address of fast RAM with a 1 megabyte configuration is $07F00000. As memory is added, the base address shifts down accordingly (e.g., $07D00000 with 4 megabytes).

The CIAs

Details about the CIAs can be found in Section 11.3. The following are the addresses of the individual registers:

CIA-A	CIA-B	Name	Function
$BFE001	$BFD000	PA	Port register A
$BFE101	$BFD100	PB	Port register B
$BFE201	$BFD200	DDRA	Data direction register A
$BFE301	$BFD300	DDRB	Data direction register B
$BFE401	$BFD400	TALO	Timer A low byte
$BFE501	$BFD500	TAHI	Timer A high byte
$BFE601	$BFD600	TBLO	Timer B low byte
$BFE701	$BFD700	TBHI	Timer B high byte
$BFE801	$BFD800	E. LSB	Event counter bits 0-7
$BFE901	$BFD900	E. MID	Event counter bits 8-15
$BFEA01	$BFDA00	E. MSB	Event counter bits 16-24
$BFEB01	$BFDB00	---	Unused
$BFEC01	$BFDC00	SP	Serial port register
$BFED01	$BFDD00	ICR	Interrupt control register
$BFEE01	$BFDE00	CRA	Control register A
$BFEF01	$BFDF00	CRB	Control register B

The custom chips

The various custom chip registers occupy a 510-byte area. Each register is 2 bytes (one word) wide. All registers are on even addresses.

The base address of the register area is at $DFF000. The effective address of a register is then $DFF000 + register address. The following list shows the names and functions of the individual chip registers. Most of the register descriptions are unfamiliar now since we haven't discussed the functions of the registers, but this list will give you an overview and will later serve as a reference.

There are four types of registers:

r (Read)

This register can only be read.

w (Write)

This register can only be written.

s (Strobe)

An access to a register of this type causes a one-time action to occur in the chip. The value of the data bus (i.e., the word to be written to the register) is irrelevant. These registers are usually accessed only by Agnus.

er (Early Read)

A register designated as early read is a DMA output register. It contains the data to be written into the chip RAM through DMA. There are two such registers (DSKDATR and BLTDDAT - output registers for the disk and the Blitter). They are accessed only by the DMA controller in Agnus, when their contents are written into the chip RAM. The processor cannot access these registers.

A, D, P

These three letters represent the three chips Agnus, Denise and Paula. They indicate in which chip the given register is found. It is also possible for a register to be located in more than one chip. On such a write access, the value is then written into two or even all three chips. This is the case when the contents of a given register are needed by more than one chip.

For the programmer it is unimportant where the registers are located. The entire area can be treated as one custom chip. The programmer needs to know only the address and function of the desired register.

p, d

A lowercase "d" means that this register is accessible only by the DMA controller. Registers preceded by a lowercase "p" are used only by the processor or the Copper. If both letters precede a register, it means that it is usually accessed by the DMA, but also by the processor from time to time.

Number of registers: 227

Registers are normally accessed only by the DMA controller: 54

Base address of the register area: $DFF000

Name	Reg.addr.	Chip	R/W	p/d	Function
BLTDDAT	000	A	er	d	Blitter output data (from Blitter to RAM)
DMACONR	002	AP	r	p	Read DMA control register
VPOSR	004	A	r	p	MSB of vertical position
VHPOSR	006	A	r	p	Vertical and horizontal beam position
DSKDATR	008	P	er	d	Disk read data (from disk to RAM)
JOY0DAT	00A	D	r	p	Joystick/mouse position game port 0
JOY1DAT	00C	D	r	p	Joystick/mouse position game port 1
CLXDAT	00E	D	r	p	Collision register
ADKCONR	010	P	r	p	Read audio/disk control register
POT0DAT	012	P	r	p	Read potentiometer game port 0
POT1DAT	014	P	r	p	Read potentiometer game port 1
POTGOR	016	P	r	p	Read pot. port data
SERDATR	018	P	r	p	Read serial port and status
DSKBYTR	01A	P	r	p	Read disk data byte and status
INTENAR	01C	P	r	p	Read interrupt enable
INTREQR	01E	P	r	p	Read interrupt request
DSKPTH	020	A	w	p	Disk DMA address bits 16-20
DSKPTL	022	A	w	p	Disk DMA address bits 1-15
DSKLEN	024	P	w	p	Disk DMA block length
DSKDAT	026	P	w	d	Disk write data (from RAM to disk)
REFPTR	028	A	w	d	Refresh counter
VPOSW	02A	A	w	p	Write MSB of vertical beam position
VHPOSW	02C	A	w	p	Write vertical and horizontal beam position
COPCON	02E	A	w	p	Copper control register
SERDAT	030	P	w	p	Write serial data and stop bits
SERPER	032	P	w	p	Serial port control register and baud rate
POTGO	034	P	w	p	Write pot. port data and start bit
JOYTEST	036	D	w	p	Write in both mouse counters
STREQU	038	D	s	d	Horizontal sync with VB and equal frame
STRVBL	03A	D	s	d	Horizontal sync with vertical blank
STRHOR	03C	DP	s	d	Horizontal synchronization signal
STRLONG	03E	D	s	d	Long horizontal line marker

The following registers can be accessed by Copper when COPCON = 1.

Name	Reg.addr.	Chip	R/W	p/d	Function
BLTCON0	040	A	w	p	Blitter control register 0
BLTCON1	042	A	w	p	Blitter control register 1
BLTAFWM	044	A	w	p	Mask for first data word from A
BLTALWM	046	A	w	p	Mask for last data word from A
BLTCPTH	048	A	w	p	Address of source data C bits 16-20
BLTCPTL	04A	A	w	p	Address of source data C bits 1-15
BLTBPTH	04C	A	w	p	Address of source data B bits 16-20
BLTBPTL	04E	A	w	p	Address of source data B bits 1-15
BLTAPTH	050	A	w	p	Address of source data A bits 16-20
BLTAPTL	052	A	w	p	Address of source data A bits 1-15
BLTDPTH	054	A	w	p	Address of destination data D bits 16-20
BLTDPTL	056	A	w	p	Address of destination data D bits 1-15
BLTSIZE	058	A	w	p	Start bit and size of Blitter window
BLTCON0L	05A	A	w	p	Like BLTCON0, bits 0-7
BLTSIZEV	05C	A	w	p	Width of Blitter window
BLTSIZEH	05E	A	w	p	Height of Blitter window
BLTCMOD	060	A	w	p	Blitter modulo for source data C
BLTBMOD	062	A	w	p	Blitter modulo for source data B
BLTAMOD	064	A	w	p	Blitter modulo for source data A
BLTDMOD	066	A	w	p	Blitter modulo for destination data D
---	068				Unused
---	06A				Unused
---	06C				Unused
---	06E				Unused
BLTCDAT	070	A	w	d	Blitter source data register C
BLTBDAT	072	A	w	d	Blitter source data register B
BLTADAT	074	A	w	d	Blitter source data register A
---	076				Unused
---	078				Unused
---	07A				Unused
DENISEID	07C	D	r	p	Chip identification from Denise
DSKSYNC	07E	P	w	p	Disk sync pattern

The following registers can always be written by the Copper.

Name	Reg.addr.	Chip	R/W	p/d	Function
COP1LCH	080	A	w	p	Address of 1st Copper list bits 16-20
COP1LCL	082	A	w	p	Address of 1st Copper list bits 1-15
COP2LCH	084	A	w	p	Address of 2nd Copper list bits 16-20
COP2LCL	086	A	w	p	Address of 2nd Copper list bits 1-15
COPJMP1	088	A	s	p	Jump to start of 1st Copper list
COPJMP2	08A	A	s	p	Jump to start of 2nd Copper list
COPINS	08C	A	w	d	Copper command register
DIWSTRT	08E	A	w	p	Upper left corner of display window
DIWSTOP	090	A	w	p	Lower right corner of display window
DDFSTRT	092	A	w	p	Start of bit-plane DMA (horiz. pos.)
DDFSTOP	094	A	w	p	End of bit-plane DMA (horiz. pos.)

Name	Reg.addr.	Chip	R/W	p/d	Function
DMACON	096	ADP	w	p	Write DMA control register
CLXCON	098	D	w	p	Write collision control register
INTENA	09A	P	w	p	Write interrupt enable
INTREQ	09C	P	w	p	Write interrupt request
ADKCON	09E	P	w	p	Audio, disk and UART control register
AUD0LCH	0A0	A	w	p	Address of audio data bits 16-20
AUD0LCL	0A2	A	w	p	On sound channel 0, bits 1-15
AUD0LEN	0A4	P	w	p	Channel 0 length of audio data
AUD0PER	0A6	P	w	p	Channel 0 period duration
AUD0VOL	0A8	P	w	p	Channel 0 volume
AUD0DAT	0AA	P	w	d	Channel 0 audio data (to D/A converter)
---	0AC				Unused
---	0AE				Unused
AUD1LCH	0B0	A	w	p	Address of audio data bits 16-20
AUD1LCL	0B2	A	w	p	On sound channel 1, bits 1-15
AUD1LEN	0B4	P	w	p	Channel 1 length of audio data
AUD1PER	0B6	P	w	p	Channel 1 period duration
AUD1VOL	0B8	P	w	p	Channel 1 volume
AUD1DAT	0BA	P	w	d	Channel 1 audio data (to D/A converter)
---	0BC				Unused
---	0BE				Unused
AUD2LCH	0C0	A	w	p	Address of audio data bits 16-20
AUD2LCL	0C2	A	w	p	On sound channel 2, bits 1-15
AUD2LEN	0C4	P	w	p	Channel 2 length of audio data
AUD2PER	0C6	P	w	p	Channel 2 period duration
AUD2VOL	0C8	P	w	p	Channel 2 volume
AUD2DAT	0CA	P	w	d	Channel 2 audio data (to D/A converter)
---	0CC				Unused
---	0CE				Unused
AUD3LCH	0D0	A	w	p	Address of audio data bits 16-20
AUD3LCL	0D2	A	w	p	On sound channel 3, bits 1-15
AUD3LEN	0D4	P	w	p	Channel 3 length of audio data
AUD3PER	0D6	P	w	p	Channel 3 period duration
AUD3VOL	0D8	P	w	p	Channel 3 volume
AUD3DAT	0DA	P	w	d	Channel 3 audio data (to D/A converter)
---	0DC				Unused
---	0DE				Unused
BPL1PTH	0E0	A	w	p	Address of bit-plane 1, bits 16-20
BPL1PTL	0E2	A	w	p	Address of bit-plane 1, bits 1-15
BPL2PTH	0E4	A	w	p	Address of bit-plane 2, bits 16-20
BPL2PTL	0E6	A	w	p	Address of bit-plane 2, bits 1-15
BPL3PTH	0E8	A	w	p	Address of bit-plane 3, bits 16-20
BPL3PTL	0EA	A	w	p	Address of bit-plane 3, bits 1-15
BPL4PTH	0EC	A	w	p	Address of bit-plane 4, bits 16-20
BPL4PTL	0EE	A	w	p	Address of bit-plane 4, bits 1-15
BPL5PTH	0F0	A	w	p	Address of bit-plane 5, bits 16-20
BPL5PTL	0F2	A	w	p	Address of bit-plane 5, bits 1-15
BPL6PTH	0F4	A	w	p	Address of bit-plane 6, bits 16-20
BPL6PTL	0F6	A	w	p	Address of bit-plane 6, bits 1-15
---	0F8				Unused

Name	Reg.addr.	Chip	R/W	p/d	Function
---	0FA				Unused
---	0FC				Unused
---	0FE				Unused
BPLCON0	100	AD	w	p	Bit-plane control register 0
BPLCON1	102	D	w	p	Control register 1 (scroll values)
BPLCON2	104	D	w	p	Control register 2 (priority control)
BPLCON3	106	D	w	p	Control register 3
BPL1MOD	108	A	w	p	Bit-plane modulo for uneven planes
BPL2MOD	10A	A	w	p	Bit-plane modulo for even planes
---	10C				Unused
---	10E				Unused
BPL1DAT	110	D	w	d	Bit-plane 1 data (to RGB output)
BPL2DAT	112	D	w	d	Bit-plane 2 data (to RGB output)
BPL3DAT	114	D	w	d	Bit-plane 3 data (to RGB output)
BPL4DAT	116	D	w	d	Bit-plane 4 data (to RGB output)
BPL5DAT	118	D	w	d	Bit-plane 5 data (to RGB output)
BPL6DAT	11A	D	w	d	Bit-plane 6 data (to RGB output)
---	11C				Unused
---	11E				Unused
SPR0PTH	120	A	w	p	Sprite data 0, bits 16-18
SPR0PTL	122	A	w	p	Sprite data 0, bits 1-15
SPR1PTH	124	A	w	p	Sprite data 1, bits 16-18
SPR1PTL	126	A	w	p	Sprite data 1, bits 1-15
SPR2PTH	128	A	w	p	Sprite data 2, bits 16-18
SPR2PTL	12A	A	w	p	Sprite data 2, bits 1-15
SPR3PTH	12C	A	w	p	Sprite data 3, bits 16-18
SPR3PTL	12E	A	w	p	Sprite data 3, bits 1-15
SPR4PTH	130	A	w	p	Sprite data 4, bits 16-18
SPR4PTL	132	A	w	p	Sprite data 4, bits 1-15
SPR5PTH	134	A	w	p	Sprite data 5, bits 16-18
SPR5PTL	136	A	w	p	Sprite data 5, bits 1-15
SPR6PTH	138	A	w	p	Sprite data 6, bits 16-18
SPR6PTL	13A	A	w	p	Sprite data 6, bits 1-15
SPR7PTH	13C	A	w	p	Sprite data 7, bits 16-18
SPR7PTL	13E	A	w	p	Sprite data 7, bits 1-15
SPR0POS	140	AD	w	dp	Sprite 0 start position (vert. and horiz.)
SPR0CTL	142	AD	w	dp	Sprite 0 control reg. and vertical stop
SPR0DATA	144	D	w	dp	Sprite 0 data register A (to RGB output)
SPR0DATB	146	D	w	dp	Sprite 0 data register B (to RGB output)
SPR1POS	148	AD	w	dp	Sprite 1 start position (vert. and horiz.)
SPR1CTL	14A	AD	w	dp	Sprite 1 control reg. and vertical stop
SPR1DATA	14C	D	w	dp	Sprite 1 data register A (to RGB output)
SPR1DATB	14E	D	w	dp	Sprite 1 data register B (to RGB output)
SPR2POS	150	AD	w	dp	Sprite 2 start position (vert. and horiz.)
SPR2CTL	152	AD	w	dp	Sprite 2 control reg. and vertical stop
SPR2DATA	154	D	w	dp	Sprite 2 data register A (to RGB output)
SPR2DATB	156	D	w	dp	Sprite 2 data register B (to RGB output)
SPR3POS	158	AD	w	dp	Sprite 3 start position (vert. and horiz.)
SPR3CTL	15A	AD	w	dp	Sprite 3 control reg. and vertical stop
SPR3DATA	15C	D	w	dp	Sprite 3 data register A (to RGB output)

Name	Reg.addr.	Chip	R/W	p/d	Function
SPR3DATB	15E	D	w	dp	Sprite 3 data register B (to RGB output)
SPR4POS	160	AD	w	dp	Sprite 4 start position (vert. and horiz.)
SPR4CTL	162	AD	w	dp	Sprite 4 control reg. and vertical stop
SPR4DATA	164	D	w	dp	Sprite 4 data register A (to RGB output)
SPR4DATB	166	D	w	dp	Sprite 4 data register B (to RGB output)
SPR5POS	168	AD	w	dp	Sprite 5 start position (vert. and horiz.)
SPR5CTL	16A	AD	w	dp	Sprite 5 control reg. and vertical stop
SPR5DATA	16C	D	w	dp	Sprite 5 data register A (to RGB output)
SPR5DATB	16E	D	w	dp	Sprite 5 data register B (to RGB output)
SPR6POS	170	AD	w	dp	Sprite 6 start position (vert. and horiz.)
SPR6CTL	172	AD	w	dp	Sprite 6 control reg. and vertical stop
SPR6DATA	174	D	w	dp	Sprite 6 data register A (z. RGB output.)
SPR6DATB	176	D	w	dp	Sprite 6 data register B (to RGB output)
SPR7POS	178	AD	w	dp	Sprite 7 start position (vert. and horiz.)
SPR7CTL	17A	AD	w	dp	Sprite 7 control reg. and vertical stop
SPR7DATA	17C	D	w	dp	Sprite 7 data register A (to RGB output)
SPR7DATB	17E	D	w	dp	Sprite 7 data register B (to RGB output)
COLOR00	180	D	w	p	Color palette register 0 (color table)
COLOR01	182	D	w	p	Color palette register 1 (color table)
COLOR02	184	D	w	p	Color palette register 2 (color table)
COLOR03	186	D	w	p	Color palette register 3 (color table)
COLOR04	188	D	w	p	Color palette register 4 (color table)
COLOR05	18A	D	w	p	Color palette register 5 (color table)
COLOR06	18C	D	w	p	Color palette register 6 (color table)
COLOR07	18E	D	w	p	Color palette register 7 (color table)
COLOR08	190	D	w	p	Color palette register 8 (color table)
COLOR09	192	D	w	p	Color palette register 9 (color table)
COLOR10	194	D	w	p	Color palette register 10 (color table)
COLOR11	196	D	w	p	Color palette register 11 (color table)
COLOR12	198	D	w	p	Color palette register 12 (color table)
COLOR13	19A	D	w	p	Color palette register 13 (color table)
COLOR14	19C	D	w	p	Color palette register 14 (color table)
COLOR15	19E	D	w	p	Color palette register 15 (color table)
COLOR16	1A0	D	w	p	Color palette register 16 (color table)
COLOR17	1A2	D	w	p	Color palette register 17 (color table)
COLOR18	1A4	D	w	p	Color palette register 18 (color table)
COLOR19	1A6	D	w	p	Color palette register 19 (color table)
COLOR20	1A8	D	w	p	Color palette register 20 (color table)
COLOR21	1AA	D	w	p	Color palette register 21 (color table)
COLOR22	1AC	D	w	p	Color palette register 22 (color table)
COLOR23	1AE	D	w	p	Color palette register 23 (color table)
COLOR24	1B0	D	w	p	Color palette register 24 (color table)
COLOR25	1B2	D	w	p	Color palette register 25 (color table)
COLOR26	1B4	D	w	p	Color palette register 26 (color table)
COLOR27	1B6	D	w	p	Color palette register 27 (color table)
COLOR28	1B8	D	w	p	Color palette register 28 (color table)
COLOR29	1BA	D	w	p	Color palette register 29 (color table)
COLOR30	1BC	D	w	p	Color palette register 30 (color table)
COLOR31	1BE	D	w	p	Color palette register 31 (color table)
HTOTAL	1C0	A	w	p	Clock count per line (VARBEAM=1)

Name	Reg.addr.	Chip	R/W	p/d	Function
HSSTOP	1C2	A	w	p	H-sync stop position
HBSTRT	1C4	A	w	p	H-blank start position
HBSTOP	1C6	A	w	p	H-blank stop position
VTOTAL	1C8	A	w	p	Number of lines per picture
VSSTOP	1CA	A	w	p	V-sync stop line
VBSTRT	1CC	A	w	p	V-blank start line
VBSTOP	1CE	A	w	p	V-blank stop line
SPRHSTRT	1D0	A	w	p	UHRES sprite start line
SPRHSTOP	1D2	A	w	p	UHRES sprite stop line
BPLHSTRT	1D4	A	w	p	UHRES bit-plane start line
BPLHSTOP	1D6	A	w	p	UHRES bit-plane stop line
HHPOSW	1D8	A	w	p	Write DUAL-mode column counter
HHPOSR	1DA	A	r	p	Read DUAL-mode column counter
BEAMCON0	1DC	A	w	p	Raster beam control register
HSSTRT	1DE	A	w	p	H-sync start position
VSSTRT	1E0	A	w	p	V-sync start position
HCENTER	1E2	A	w	p	H-pos. of V-sync in interlace mode
DIWHIGH	1E4	A,D	w	p	Screen window, upper bits for start/stop
BPLHMOD	1E6	A	w	p	UHRES bit-plane modulo
SPRHPTH	1E8	A	w	p	UHRES sprite pointer (bits 16-20)
SPRHPTL	1EA	A	w	p	UHRES sprite pointer (bits 0-15)
BPLHPTH	1EC	A	w	p	UHRES bit-plane pointer (bits 16-20)
BPLHPTL	1EE	A	w	p	UHRES bit-plane pointer (bits 0-15)
					The registers 1F0 to 1FC are unoccupied

ROM

The figure on page 771 shows the ROM area as it appears after booting. The 512K of ROM at $00F80000 contains the Amiga Kickstart. This configuration can change. After a reset, the 68030 fetches the address of the first instruction from memory location 4, called the reset vector. If the memory configuration could not be changed, the 68000 would fetch the reset vector from chip RAM, which is at address 4. Since the contents of this location are undefined at start-up, the processor would jump to some random address and the system would crash. The solution to this is as follows: The chip that is responsible for the memory configuration has an input that is connected to the lowest port line of CIA-A (PA0). This OVL (Memory Overlay) line is normally at 0, and the memory configuration corresponds to the figure. After a reset, the port line automatically goes high, causing the ROM area at $00F80000 to $00FFFFFF to be mapped into the range from 0 to $7FFFF. This means that address 4 (the reset vector) then corresponds to address $F80004. Here the 68030 finds a valid reset address, which tells it to jump to the Kickstart program. In the course of the reset routine the OVL line is set to 0 and the normal memory configuration returns.

You must be very careful when experimenting with this line. If the program that tries to set the OVL line is running in chip RAM, the result can be catastrophic, because the program more or less switches itself out of the memory range and the processor lands somewhere in the Kickstart, which takes the place of the chip RAM after the switch.

Since the final version of the operating system was not yet ready when the first A3000's were manufactured, the Kickstart had to be booted from the hard disk. In place of the Kickstart, a boot program was placed in ROM. After the Kickstart was loaded to fast RAM, it would be transferred to $00F80000 with the 68030's PMMU.

11.7.2 Fundamentals

As mentioned in the previous section, there are some registers that are accessed by the processor and some that are read and written via DMA. We'll begin by discussing the former.

Programming the chip registers

The chip registers can be addressed directly (e.g., changing the value of the background color register). The register has the name COLOR00. Looking in the register table, you see that it has a register address of $180.

So, we must add the base address of the register area (i.e., the address of the first register in the address range that the 68030 accesses). This is $DFF000. Also, the register address of COLOR00 yields $DFF180. A simple MOVE.W command can be used to initialize the register:

```
MOVE.W #value,$DFF180      ;value in COLOR00
```

If more than one register is accessed, it is a good idea to store the base address in an address register and use indirect addressing with an offset. Here is an example:

```
LEA    $DFF000,A5          ;store base address in A5
MOVE.W #value1,$180(A5)    ;value1 in COLOR00
MOVE.W #value2,$182(A5)    ;value2 in COLOR01
MOVE.W ... etc.
```

Normally the chip registers are accessed as previously shown. However, the registers can also be accessed as a long word. In this case two registers are always written at once. This makes sense for the address registers, which consist of a pair of registers holding a single 21-bit address, with which the entire 2048K chip RAM area can be accessed. All data for the custom chips must be in the chip RAM. Since the chips always address the memory word-wise, the lowest bit (bit 0) is irrelevant. The address register points only to even addresses. Since a chip register is only one word (16 bits) wide, two successive registers are used to store the 21-bit memory address. The first register contains the upper 5 bits (bits 16-20) and the second contains the lower 16 (bits 0-15). This makes it possible to initialize both registers with a single long-word access. Example: Setting the pointer for the first bit-plane to address $40000. BPL1PTH is the name of the first register (bits 16-20) and BPL1PTL (bits 0-15) is the name of the second. Register address of BPL1PTH: $0E0, BPL1PTL = $0E2.

A5 contains the base address $DFF000.

```
MOVE.L #$40000,$0E0(A5)    ;initializes BPL1PTH and BPL1PTL with the correct values.
```

Any given register address can never be both read from and written to. Most registers are write-only registers and cannot be read. This also includes the registers previously mentioned. Others can only be read. Only a few can be read and written, but these have two different register addresses, one for reading and one for writing. The DMA control register, which will be discussed in detail later, is such a register. It can be written through the register address $096 (DMACON), while address $002 is used for reading (DMACONR).

DMA access

DMA involves the direct access of a special component, called the DMA controller, to the system memory. In the case of the Amiga, the DMA controller is housed in Agnus. It represents the connection between the various input/output components of the custom chips and the chip RAM. The DMA process follows the same pattern regardless of whether diskette, screen or audio data is involved. A given I/O component, such as the disk controller, needs new data or has data that it wants to store in memory. The DMA controller waits until the memory for this channel is free (not being accessed by another DMA channel or the processor) and then transfers the data to or from RAM itself. For the sake of simplicity there is no special transfer of the data from the I/O device to the DMA controller. It always takes place through registers. Each of these I/O components has two different types of registers. One type is the normal registers which are accessed by the processor and in which the various operating parameters are stored. The second is the data registers that contain the data for the DMA controller. For a DMA transfer this involves simply the corresponding data register and a RAM location. Depending on the direction of the transfer, either a read register is selected and the chip RAM is set for write, or a write register is used and the chip RAM is set for read. Since the two can be connected through the data bus, the data are automatically routed to their destination. Data are not stored in any temporary registers.

The DMA transfer adds a third type of register: the DMA address register which holds the address or addresses of the data in RAM, depending on the needs of the I/O device.

There are many central control registers that are not assigned to a special I/O device, but have higher-level control functions. The DMACON register belongs in this category.

The data registers can also be written by the processor, since they are realized in the form of normal registers. However, this is not generally useful, since the DMA controller can accomplish this more quickly and efficiently.

Some I/O components do not have DMA channels. The 68030 must read and write their data itself. This group includes only those devices which by their nature do not deal with large quantities of data, so that DMA is not needed, such as the joystick and mouse inputs.

The following DMA channels are present:

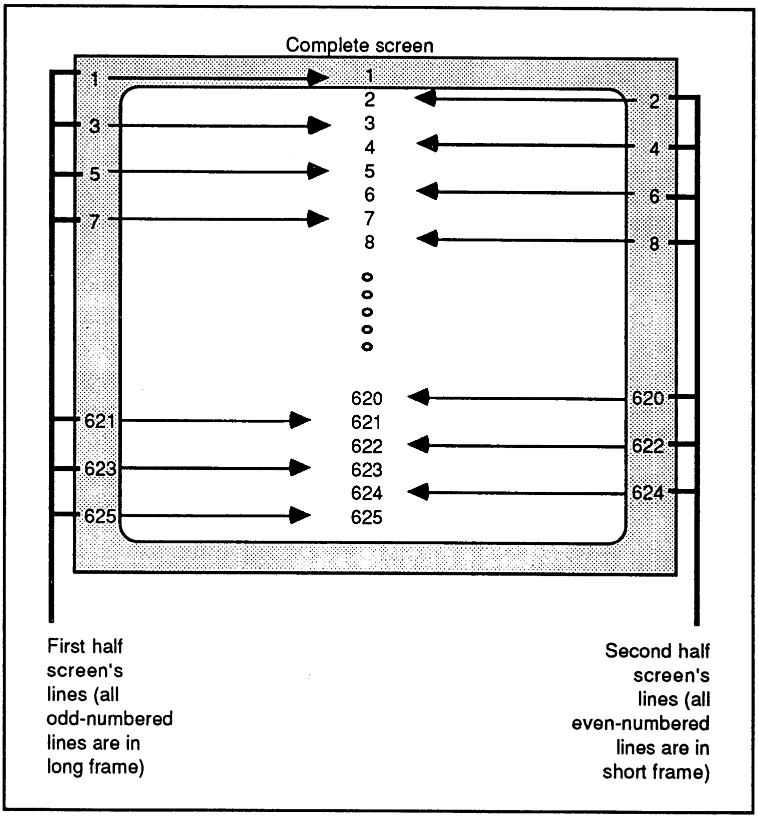
Bit-plane DMA — Through this DMA channel the screen data are read from memory and written into the data registers of the individual bit-planes. From there they go to the bit-plane sequencers, which convert the data for output to the screen.

Sprite DMA — Transfers the sprite data from the RAM to the sprite data registers.

Disk DMA	Transfers data from disk to RAM or from RAM to disk.
Audio DMA	Reads digital sound data from RAM and writes it to the appropriate audio data registers.
Copper DMA	The coprocessor (Copper) receives its command words through this channel.
Blitter DMA	Transfers data from and to the Blitter.

There are a total of six DMA channels which all want to access the memory, and the processor, which naturally also wants to have the chip RAM for itself as often as possible. To solve the problems that result from this, a complex system of time multiplexing was devised in which the individual channels have defined positions. Since this is oriented to the video picture, first we must briefly discuss its construction. This section has been kept as simple as possible, since we are discussing the programming of the custom chips and not the hardware.

Construction of the video picture



Construction of the picture

The timing of the Amiga screen output corresponds exactly to the standard of the country where the Amiga is sold, PAL for Europe and NTSC for the US. The 8361 Agnus chip is available in an NTSC US version and a PAL version for Europe. A PAL video picture consists of 625 horizontal lines, an NTSC picture of 525 horizontal lines. Each of these lines is constructed from left to right. A pause follows every line, called the horizontal blanking gap, in which the electron beam that draws the picture has time to go back from right to left. During this blanking gap the electron beam is dark so that it cannot be seen tracing back to the left side. Then the process starts over again and the next line appears.

To keep the picture free of flickering, it must be continually redrawn. Since our eyes cannot discern changes above a certain frequency, the number of pictures per second is placed above this limit. With the PAL standard, the number of individual pictures is set to 50 per second (30 per second for NTSC). But now we encounter a problem. If all 625 lines were drawn 50 times per second, the result would be 31250 lines per second. If monitors and televisions were built to these specifications, they would not be affordably priced, so a trick is used. On one hand, the number of pictures should not be less than 50 per second or the screen begins to flicker, while on the other hand there must be enough lines per picture. The solution is as follows: 50 pictures are displayed per second, but the 625 lines are divided into two pictures. The first picture contains all the odd lines (1,3,5...625), while the second contains all the even lines (2,4,6...624). Two of these half-pictures (called frames) are combined to form the entire picture, which contains 625 lines. Naturally, the number of complete pictures per second is only half as large as the number of half-pictures, or 25 per second. The line frequency for this technique is only 15625 Hz (25x625 or 50x312.5).

In spite of the high resolution of 625 lines, flickering occurs when a contour is restricted to only one line. Then it is displayed only every 25th of a second, which is perceived by the eye as a visible flickering. This effect can be seen on televisions (especially on the horizontal edges of surfaces), since these consist of only a single horizontal line.

The term for this technique of alternating display of even and odd lines is interlacing. Two additional terms are used to distinguish the difference between the two types of half-pictures. A long frame is the one in which the odd lines are displayed, and the other is called a short frame. They are called long and short frames because there is one more odd line than even and it takes slightly longer to display the frame containing the extra line (from 1 to 625 there are 313 odd and 312 even numbers).

After each frame there is a pause before the next frame begins. This blank space between frames is called the vertical blanking gap. The picture created by the Amiga also follows this scheme, although with some deviations.

Normally the second half-picture (short frame) is somewhat delayed so that the even lines appear exactly between the odd lines.

On the Amiga both frames are identical so that the frequency is actually 50 Hz. As a result, the number of lines is limited to 313. This can be clearly seen by the vertical distance between two lines on the screen, since the frames are no longer displaced, but drawn on top of each other.

To increase the number of lines, the Amiga can also create its picture in interlace mode. Then a full 625 lines are possible on PAL systems, but the disadvantages of interlace operation must be considered. More about this later.

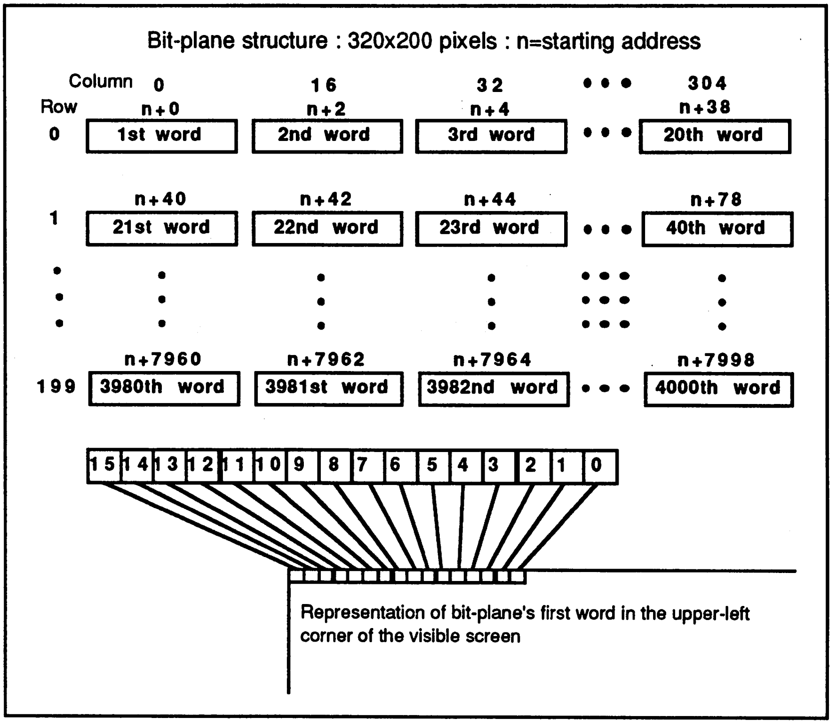
Construction of the Amiga screen output

Bit-planes

The Amiga always displays its picture in a type of graphic mode (i.e., each point on the screen has a corresponding representation in memory).

The simplest way to build a screen image in memory is to define a contiguous block of RAM in which a set bit corresponds to a point (pixel) displayed on the monitor. This basic construction is called a bit-plane and is the fundamental element of all screen display in the Amiga. A single line on the screen will consist of a certain number of words determined by the width of the picture. Since each bit represents one pixel, a word comprises 16 pixels. For a screen display of 320 pixels per line, 20 (320/16) words per line are needed.

In a single bit-plane, only one of two possible conditions can exist for a given bit position. The bit is either set or cleared. However, by combining several bit-planes, the possibilities are greatly expanded. The planes can be logically superimposed so that those bits having the same position within their respective planes are considered as a unit. The first pixel on the screen is the result of combining the first bit of the first word of all the bit-planes. The value of the bit combination determines the color of the pixel on the screen. There are various ways of deriving colors from bit combinations, and we'll discuss these in more detail later.



Bit-plane construction

Different graphic resolutions

The Amiga recognizes three different horizontal resolutions. The high resolution mode normally has 640 pixels per line, the low resolution has 320. The A3000's new Denise even permits a 1280 pixel-per-line display called super hi-res mode. The word "normally" means that this value can change. It is better to define the different resolutions in terms of time per pixel. A pixel in super hi-res mode is displayed for 35 nanoseconds, in normal hi-res mode for 70 nanoseconds and in low-resolution mode for 140 nanoseconds. Comparing lo-res to hi-res, the electron beam traces across the screen for twice the time to produce a single pixel. In this time it covers twice the distance, producing a pixel that appears twice as wide in low as in high resolution.

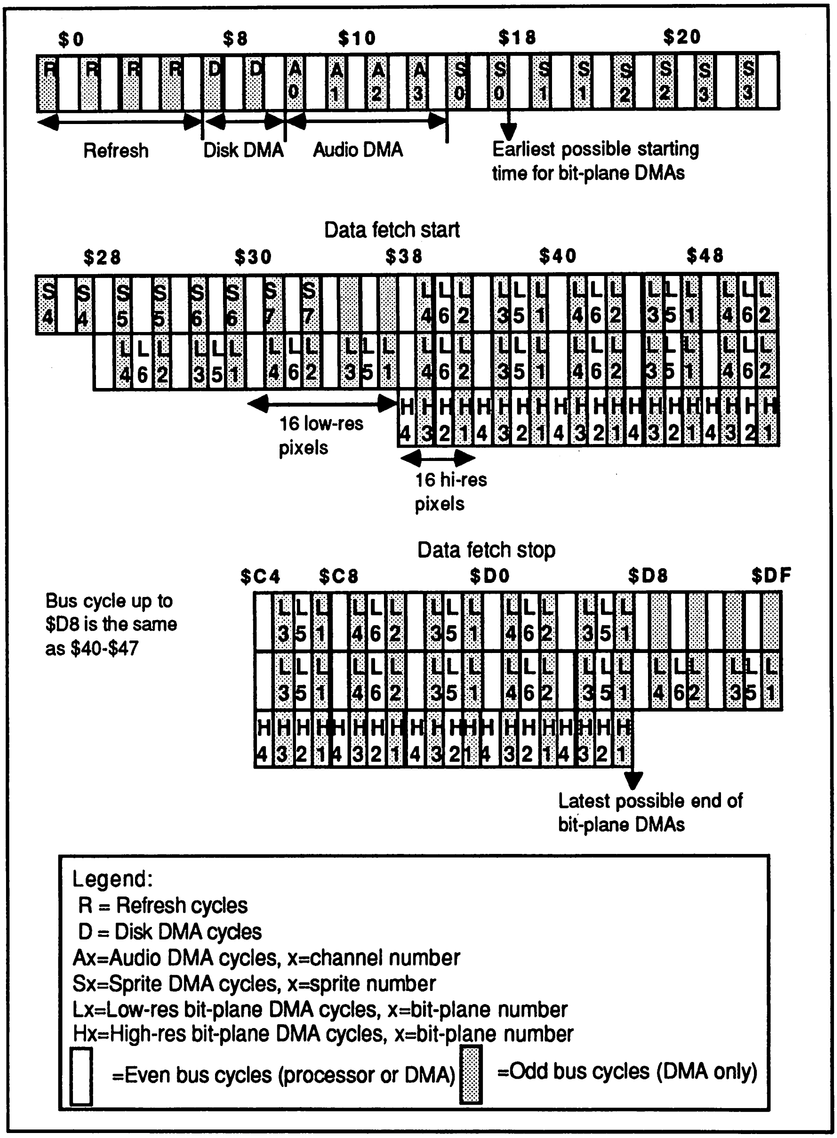
What is more important for the programmer to know, however, is that in high-resolution mode only four bit-planes can be active at a time, while in low-resolution mode up to six planes are allowed. In super hi-res mode only two bit-planes may be used. Furthermore, a limitation of 64 colors is imposed on the color palette. This is a consequence not of higher pixel frequency, but of certain limitations in the chip design.

Construction of a horizontal raster line

A raster line is a complete horizontal line, including both the horizontal blanking gap and the visible region. This raster line serves as a timing measure for all DMA processes, particularly for screen-associated DMAs. To understand the division of the raster line, you must know how memory access to chip RAM and the custom chip registers is distributed between the DMA controller and the processor. Accesses to these two storage areas must conform to what are called bus cycles. The bus cycles determine the timing of the chip RAM. One memory access can take place in each bus cycle. It doesn't matter whether the data is read or written.

For example, if the processor wants to access the bus it gets control of the bus for one bus cycle. The DMA controller cannot access RAM again until the following cycle. A bus cycle lasts 280 nanoseconds. Almost four memory accesses are possible in one microsecond.

For compatibility reasons, processor accesses to chip memory are executed according to the same scheme as in 68000-based Amiga models. This requires the 68030 to constantly insert wait states, so that the result is a maximum of one access every 560 nanoseconds. During this time two bus cycles elapse. The 68030 can use every other bus cycle. These cycles are called even cycles. The remaining cycles, the odd cycles, are reserved exclusively for the DMA controller.



Raster timing

The figure shows the development of a raster line over time. It takes 63.5 microseconds. This yields 227.5 bus cycles per line. Of these the first 225 can be taken by the DMA controller. The figure shows how this is done: The letters within the individual cycles represent the corresponding

DMA channels. While the DMA controller has exclusive use of the odd cycles, it must share the even ones with the processor. Still, DMA access always takes priority. Blitter DMA and Copper DMA always take place during even cycles. There is no defined time for these two, but once Copper DMA access begins, it takes all the even cycles until it has finished its task. It has precedence over the Blitter. When the Blitter gains access, it also takes all the even cycles until it is finished. Some cycles can still be left free for the 68030.

As you can see, disk, audio and sprite DMA accesses take only odd bus cycles and do not affect the speed of the processor. The four cycles designated "R" are refresh cycles. They are used to refresh the contents of the chip RAM (see the end of this section).

The distribution of the bit-plane DMA is more complicated. For the first 16 pixels to be displayed on the screen, all the bit-planes must be read. While these 16 pixels are appearing on the screen, the bit-planes for the next 16 pixels must be read. If the lowest resolution is enabled, two pixels are output during each bus cycle. This means that the bit-planes must be read every eight bus cycles. As long as no more than four bit-planes are active, the odd cycles suffice. If five or six planes are used, two even cycles must also be used so that all the data can be read within the eight bus cycles. It's even tighter in high-resolution mode. Here four pixels are displayed per bus cycle. If only odd cycles are to be taken, no more than two hi-res planes can be active. With the maximum allowable number of four hi-res planes, all bus cycles are taken. As a result, the processor loses more than half of its free bus cycles. Its speed also decreases by the same amount, assuming that the current program is in the chip RAM, since the processor still has full-speed access to any fast RAM and to the Kickstart ROM.

The times labeled as data fetch start and data fetch stop designate the start and stop of the DMA accesses for the bit-planes. They determine the width and horizontal position of the visible picture. If the bit-plane DMA starts early and ends late, more data words are read and more pixels are displayed. The normal resolution of 320 or 640 pixels per line can be varied by changing these values. If the data fetch start is set below $30, the bit-plane DMA channel uses cycles normally reserved for sprite DMA. Depending on the exact value of data fetch start, up to seven sprites may be lost this way. Only sprite 0, which is generally used for the mouse pointer, cannot be turned off in this manner.

The top line in the figure represents the division of the DMA cycles for a low-resolution screen with the normal width of 320 pixels. The start of the bit-plane DMA, data fetch start, is at $38, and the end, data fetch stop, is at $D0. The data from bit-plane number 1 is read in the cycles designated L1, the bit-plane 2 data in L2, and so on. If the corresponding bit-planes are not enabled, their DMA cycles are also omitted.

The second line represents the course of a raster line in which the data fetch points are moved outward. Up to the data fetch start everything is the same as the top line, but here the DMA starts at $28. As a result, sprites 5 to 7 are lost. The data fetch stop position is moved to the right to the maximum value of $D8.

The third line shows the distribution of the DMA cycles in a high-resolution screen where the data fetch values match those of the first line.

No bit-plane DMA accesses occur during the vertical blanking gap.

The DMA control register

The individual DMA channels are enabled and disabled through a central DMA control register, DMACON.

The DMACON register addresses are $096 (write) and $02 (read)

Bit	Name	Function (when set)
15	SET/CLR	Set/clear bits
14	BBUSY	Blitter busy (read only)
13	BZERO	Result of all Blitter operations is 0 (read only)
12 and 11		Unused
10	BLTPRI	Blitter DMA has priority over processor
9	DMAEN	Enable all DMA (for bits 0 to 8)
8	BPLEN	Enable bit-plane DMA
7	COPEN	Enable Copper DMA
6	BLTEN	Enable Blitter DMA
5	SPREN	Enable sprite DMA
4	DSKEN	Enable disk DMA
3-0	AUDxEN	Enable audio DMA for sound channel x (bit number corresponds to number of sound channel)

The DMACON register is not written like a normal register. You can only set or clear bits. This is determined by bit 15 of the data word written to the DMACON register. If this bit is 1, all the bits that are set in the data word are also set in the DMACON register. If bit 15 is 0, all the bits that are set in the data word are cleared in the DMACON register. The remaining bits in DMACON remain unaffected.

Bit 9, designated DMAEN, is something of a main switch. If it is 0, all DMA channels are inactive, regardless of bits 0 to 8. To enable DMA you must set both the appropriate DMA channel bit and the DMAEN bit. Here is an example:

Only the bit-plane DMA is enabled (BPLEN = 1), but without the DMAEN bit. The value of the DMACON register is $0100. Now you want to enable the disk DMA. DSKEN and DMAEN must be set and BPLEN cleared.

```
MOVE.W #$0100,$DFF096       ;Clears the BPLEN bit (SET/CLR = 0)
MOVE.W #$8210,$DFF096       ;Sets DSKEN and DMAEN (SET/CLR = 1)
```

The DMACON register now contains the desired value of $0210. Bits 13 and 14 can only be read. They supply information about the status of the Blitter, which is discussed in more detail in the Blitter section.

Bit 10 controls the priority of the Blitter over the processor. If it is set, the Blitter has absolute priority over the 68030. This may go so far as to deny the processor all access to the chip registers and chip RAM throughout the entire Blitter operation. When it is cleared, the processor gets every fourth even bus cycle from the Blitter. This prevents the processor from being held up when an operating system routine or a program in fast RAM, both of which execute at full speed, must access chip RAM, for example, to get an operating system data structure or a 68030 exception vector.

Reading the current electron beam position

Since all DMA timing is oriented according to the position within a raster line, it is sometimes useful to know where on the line the electron beam is currently located. Agnus has an internal counter for this, which contains both the horizontal and vertical screen position. Two registers allow the processor access to this counter:

VHPOS $006 (read, VHPOSR) and $02C (write, VHPOSW)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	V7	V6	V5	V4	V3	V2	V1	V0	H8	H7	H6	H5	H4	H3	H2	H1

VPOS $004 (read, VPOSR) and $02A (write, VPOSW)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	LOF	--	--	--	--	--	--	--	--	--	--	--	--	V10	V9	V8

The bits designated H1 to H8 represent the horizontal beam position and correspond directly to the numbers for the individual bus cycles in the figure. They have a precision of two low or four high-resolution pixels. The value for the horizontal position can vary between $0 and $E3 (0 to 227). The horizontal blanking gap falls in the range from $F to $35.

The bits for the vertical position, the current screen line, span both registers. The lower bits, V0-V7, are in VHPOS, and the upper bits, V8-V10, are in VPOS. Together they yield the number of the current screen line.

Lines from 0 to 312 are possible. The vertical blanking gap (the screen is always dark in this range) runs from line 0 to line 25. The LOF (LOng Frame) bit indicates whether the image currently being displayed is a long (odd lines) or short (even lines) frame. This bit is needed only in interlace mode. Normally it is 1.

The beam position can also be set, but this capability is rarely needed. The POS registers have another function in connection with a light pen. When the lightpen input of Agnus is activated and the lightpen is held against the screen, they store its position. This means that their contents are frozen as soon as the lightpen detects the electron beam moving past its tip. The counters are released again at the end of the vertical blanking gap, line 26. To read the lightpen position, you must proceed as follows:

- Wait for line 0 (start of the vertical blanking gap). This is most easily done by means of the vertical blanking interrupt (see next section).
- Read the two counter registers.

If the vertical position is between 0 and 25 (within the vertical blanking gap), no lightpen signal was received. If the value is outside this range, it represents the position of the lightpen.

To conclude this section, here are some more details about the refresh cycles:

Agnus possesses an integrated 9-bit refresh counter. It can be written through register address $28 (be careful because the memory contents can be lost this way). At the start of each raster line, Agnus places four refresh addresses on the chip RAM address bus. This means the contents of each memory row are refreshed every four milliseconds.

While the row address is being output on the chip RAM address bus, Agnus places the addresses of certain strobe registers on the register address bus. These strobe signals serve to inform the other chips, Denise and Paula, of the start of a raster line or a picture. This is necessary because the counter for the screen position is inside Agnus and there are no lines for transmitting the synchronization signals to the other chips. There are four strobe addresses:

Addr.	Chip	Function
$38	D	Vertical blanking gap of a short frame
$3A	D	Vertical blanking gap
$3C	D P	This strobe address is created in every raster line outside the vertical blanking gap
$3E	D	Marker for a long raster line (228 cycles)

During the first refresh cycle, one of the first three strobe addresses is always referenced. Normally this is $3C, and within the vertical blanking gap $38 or $3A, depending on whether it is a short or long frame.

With the fourth address the situation is as follows: A raster line has a purely computational length of 227.5 bus cycles. But since there are no half-cycles, lines alternate between 227 and 228 bus cycles. The strobe address $3E signals the 228-cycle lines and is created during the second refresh cycle.

11.7.3 Interrupts

Almost all the I/O components of the custom chips and the two CIAs can generate an interrupt. A special circuit inside Paula manages the interrupt sources and creates the interrupt signals for the 68030. The processor's autovector interrupts, levels 0 to 6, are used for this. No provision is made for the non-maskable interrupt (NMI), level 7. The two registers involved are the interrupt request register (INTREQ) and the interrupt mask (enable) register (INTENA). The assignment of the bits in the two registers is identical.

Interrupt enable and interrupt request register layout

INTREQ	= $09C (write)
INTREQR	= $01E (read)
INTENA	= $09A (write)
INTENAR	= $01C (read)

Bit	Name	IE	Function
15	SET/CLR		Write/read (see DMACON register)
14	INTEN	(6)	Enable interrupts
13	EXTER	6	Interrupt from CIA-B or expansion port
12	DSKSYN	5	Disk sync value recognized
11	RBF	5	Serial port receive buffer full
10	AUD3	4	Output audio data channel 3
9	AUD2	4	Output audio data channel 2
8	AUD1	4	Output audio data channel 1
7	AUD0	4	Output audio data channel 0
6	BLIT	3	Blitter ready
5	VERTB	3	Start of vertical blanking gap reached
4	COPER	3	Reserved for Copper interrupts
3	PORTS	2	Interrupt from CIA-A or expansion port
2	SOFT	1	Reserved for software interrupts
1	DSKBLK	1	Disk DMA transfer ended
0	TBE	1	Serial transmit buffer empty

The lower 13 bits represent the individual interrupt sources. The CIA interrupts are combined into a single interrupt. The bits in the DMAREQ register indicate which interrupts have occurred. A bit is set if the corresponding interrupt has occurred. In order to generate a processor interrupt, the corresponding bit must be set in the DMAENA register and the INTEN bit must also be set. The INTEN bit acts as the main switch for the remaining 14 interrupt sources, which can be turned on and off with the individual bits of the INTENA register. Only when INTEN is 1 can any interrupts be generated.

If both the INTEN bit and the two corresponding bits in the INTENA and INTREQ registers are set, a processor interrupt is generated.

The corresponding autovector numbers are listed in the IL (Interrupt Level) column in the table. Here are the addresses of the seven interrupt autovectors:

Vector no.	Address	Autovector level
25	100/$64+(VBR)	Autovector level 1
26	104/$68+(VBR)	Autovector level 2
27	108/$6C+(VBR)	Autovector level 3
28	112/$70+(VBR)	Autovector level 4
29	116/$74+(VBR)	Autovector level 5
30	120/$78+(VBR)	Autovector level 6
(31	124/$7C+(VBR)	Autovector level 7)

VBR = Vector Base Register (see section on 68030)

As you can see, the interrupts that require faster processing are given higher interrupt levels. To change the bits in these two registers, you must work with a SET/CLR bit using the same procedure described for the DMACON register.

After processing an interrupt, the processor must reset the INTREQ register bit that generated it. In contrast to the interrupt control registers of the CIAs, the bits in the INTREQ register are not automatically cleared on reading.

Setting a bit in the INTREQ register with a MOVE command has the same effect as if the corresponding interrupt had occurred. This is how a software interrupt is created, for example (SOFT, bit 2). The Copper can also create its own interrupt by writing into INTREQ.

One peculiarity is bit 14 in the INTREQ register, which has no specific function there as it does in INTENA. But when it is set by writing to INTREQ, and INTEN in the INTENA register is high, a level 6 interrupt is generated.

On each interrupt from CIA-A, bit 3 in the DMAREQ register is set. For CIA-B this is bit 13. The interrupt source in the corresponding CIA must be determined by reading the interrupt control register of the CIA. Interrupts 3 and 13 can also be generated by expansion cards on the expansion port.

Interrupt bit 5 indicates the vertical blanking interrupt. This occurs at the start of each video frame at the start of the vertical blanking gap (line 0), and 50 times per second. The remaining interrupts are discussed in the appropriate sections.

11.7.4 The Copper Coprocessor

The Copper is a simple coprocessor. It writes certain values into the various registers of the custom chips automatically at defined points in time. More accurately, the Copper can change the contents of some registers at any screen position. By doing so, it can divide the screen into different regions, which can then have different colors and resolutions. For example, this capability is used to implement multiple screens.

The Copper is designated a coprocessor because, like a real processor, it has a program stored in memory that executes command by command. The Copper recognizes only three different commands, but they are quite versatile:

MOVE

The MOVE command writes an immediate value into a custom chip register.

WAIT

The WAIT command waits until the electron beam reaches a certain screen position.

SKIP

The SKIP command skips the next command if the electron beam has already reached a certain screen position. This allows conditional branches to be built into the program.

A Copper program is called a Copper list. It is a series of consecutive instructions, each consisting of two words. For example:

```
Wait (X1,Y1)     ;waits until screen position X1,Y1 is reached
Move #0,$180     ;writes the value 0 to the background color register
Move #9,$181     ;writes the value 1 to color register 1
Wait (X2,Y2)     ;waits until screen position X2,Y2 is reached
  ...             etc.
```

The Copper list alone is not sufficient to operate the Copper. Other registers are required which contain parameters needed by the Copper.

The Copper register

Reg.	Name	Function
$080	COP1LCH	These two registers together contain the
$082	COP1LCL	20-bit address of the first Copper list.
$084	COP2LCH	These two registers together contain the
$086	COP2LCL	20-bit address of the second Copper list.
$088	COPJMP1	Loads the address of the first Copper list into the Copper program counter.
$08A	COPJMP2	Loads the address of the second Copper list into the Copper program counter.
$02E	COPCON	This register contains only a single bit (bit 0). If it is set, the Copper can also access the registers from $040 to $7E. (These belong to the Blitter.)

All the Copper registers are write-only registers.

The COPxLC registers contain addresses of Copper lists. Since such an address is 19 bits long, two registers are needed per address. Both registers of a pair can be written with one MOVE.L command to the first register. The Copper lists, like all other data for the custom chips, must lie within the 2 Meg chip RAM.

The Copper uses an internal counter as a pointer to the current command. It is incremented by two each time a command is processed. To start the Copper at a given address, the start address of the Copper list must be transferred to the program counter. The COPJMPx registers are used for this. They are strobe registers, meaning that a write access to one of them simply activates a particular action -- they are not used to store actual values. The values written to them are completely irrelevant.

In the Copper these two registers cause the contents of the corresponding COPxLC registers to be copied into the program counter. If a write access is made to COPJMP1, the address in COP1LC is copied into the program counter, which causes the Copper to execute the program at that address. This also applies to COPJMP2 and COP2LC.

At the start of the vertical blanking gap, line 0, the program counter is automatically loaded with the value from COP1LC. This causes the Copper to execute the same program for every picture.

The command structure

	MOVE		WAIT		SKIP	
Bit	BW1	BW2	BW1	BW2	BW1	BW2
15	x	DW15	VP7	BFD	VP7	BFD
14	x	DW14	VP6	VM6	VP6	VM6
13	x	DW13	VP5	VM5	VP5	VM5
12	x	DW12	VP4	VM4	VP4	VM4
11	x	DW11	VP3	VM3	VP3	VM3
10	x	DW10	VP2	VM2	VP2	VM2
9	x	DW9	VP1	VM1	VP1	VM1
8	RA8	DW8	VP0	VM0	VP0	VM0
7	RA7	DW7	HP8	HM8	HP8	HM8
6	RA6	DW6	HP7	HM7	HP7	HM7
5	RA5	DW5	HP6	HM6	HP6	HM6
4	RA4	DW4	HP5	HM5	HP5	HM5
3	RA3	DW3	HP4	HM4	HP4	HM4
2	RA2	DW2	HP3	HM3	HP3	HM3
1	RA1	DW1	HP2	HM2	HP2	HM2
0	0	DW0	1	0	1	1

Legend:

x	This bit is unused. It should be initialized to 0.
RA	Register address
DW	Data word
VP	Vertical beam position
VM	Vertical mask bits
HP	Horizontal beam position
HM	Horizontal mask bits
BFD	Blitter finish disable

The MOVE command

The MOVE command is indicated by a 0 in bit 0 of the first command word. With this command it is possible to write an immediate value to a custom chip register. The register address of the desired register comes from the lower 9 bits of the first data word. Bit 0 must always remain 0 (it is already 0 for the register addresses because the registers lie only on even addresses). The second command word contains the data byte to be written to the register.

There are some limitations regarding the register address. Normally the Blitter cannot affect the registers in the range from $000 to $07F. If the lowest (and only) bit in the COPCON register is set, then the Copper can write to the registers in the range from $040 to $07F. This allows the

Copper to use the Blitter. Access to the lowest registers ($000 to $03F) is never allowed.

The WAIT command

The WAIT command is indicated by a 1 in bit 0 of the first word and a 0 in bit 0 of the second word. It instructs the Copper to hold further execution until the desired beam position is reached. If the position is already greater than that specified by the WAIT command at the time the command is executed (the beam is already past the specified position), the Copper continues with the next instruction immediately.

This position can be set separately for the vertical lines and horizontal rows. Vertically the resolution is one raster line. But since there are only eight bits for the vertical position and there are 313 lines, the WAIT command cannot distinguish between the first 256 and the remaining 57 lines.

For example, the lowest eight bits are the same for both line 0 and line 256. To wait for a line in the lower range, two wait commands must be used.

1. WAIT for line 255 column 224 ($FFE1).
2. WAIT for the desired line, ignoring the ninth bit.

Horizontally there are 112 possible positions, since the two lower bits of the horizontal position, HP0 and HP1, cannot be specified. The command word of the WAIT command holds only bits HP2 to HP8. This means that the horizontal coordinate of a WAIT command can only be specified in steps of four low-resolution pixels.

The second command word contains mask bits. These can be used to determine which bits of the horizontal and vertical position are actually considered in the comparison with the current beam position. Only the position bits whose mask bits are set are considered. This opens up many possibilities. For example:

```
Wait for vertical position $0F and vertical mask $0F
```

causes the WAIT condition to be fulfilled every 16 lines (whenever the lower four bits are all 1), since bits 4 to 6 are not considered (mask bits 4 to 6 are at 0). The seventh bit of the vertical position cannot be masked. The previous example works only in the range of lines 0 to 127 and 256 to 313.

The BFD (Blitter Finish Disable) bit has the following function: If the Copper is used to start a Blitter operation, it must know when the Blitter is finished with the previous operation. If the BFD bit is cleared, the Copper waits at every WAIT command until the Blitter has finished its operation. Only then is the WAIT condition checked. This can be prevented by setting the BFD bit, causing the Copper to ignore the current Blitter status. If the Copper shouldn't affect any of the Blitter registers, this bit is set to 1.

The SKIP command

The SKIP command is identical to the WAIT command, except that bit 0 in the second command word is set to distinguish it from the WAIT command. The SKIP command checks to see if the actual beam position is greater than or equal to that given in the command word. If this comparison is positive, the Copper skips the next command. Otherwise it continues program processing by executing the next command. The SKIP command allows conditional branches to be constructed. The command following SKIP can be a MOVE into one of the COPJMP registers, causing a jump to be made based on the beam position.

Construction of a Copper list

A simple Copper list consists of a sequence of WAIT and MOVE commands, and a few SKIP commands. Its start address is found in COPLC1. A trick must be used to end the Copper list. After the last instruction comes a WAIT command with an impossible beam position. This effectively ends the processing of the Copper list until, at the start of a new picture, the COPLC1 address is loaded into the program counter again to restart processing. WAIT ($0,$FE) fulfills this condition, because a horizontal position greater than $E4 is not possible.

The Copper interrupt

As you know, there is a special bit in the interrupt registers for the Copper interrupt. This interrupt can be generated with a MOVE command to the INTREQ register:

```
MOVE #$8010,INTREQ         ;set SET/CLR and COPER
```

Any other bit in this register can be affected the same way, but bit 4 is provided especially for the Copper.

A Copper interrupt can be used to tell the processor that a certain screen position has been reached. This allows what are called raster interrupts to be programmed (i.e., the interruption of the processor in a certain screen line (and column)).

The Copper DMA

The Copper fetches its commands from memory through its own DMA channel. It uses the even bus cycles and has precedence over the Blitter and the 68030. Each command requires two cycles, since two command words must be read. The WAIT command requires an additional cycle when the desired beam position is reached. The Copper leaves the bus free during the wait phase of a WAIT command.

The COPEN bit in the DMACON register is used to turn the Copper DMA on and off. If this bit is cleared, the Copper releases the bus and does not execute any more commands. If it is set, it starts its program execution at the address in its program counter. Therefore, it is absolutely necessary to supply this with a valid address before starting the Copper DMA. A Copper running in an unknown area of memory can crash the system. The usual initialization sequence for the Copper looks like this:

```
LEA     $DFF000,A5                ;Base address of registers to A5
MOVE.W #$0080,DMACON(A5)          ;Copper DMA off
MOVE.L #Copperlist,COP1LCH(A5)    ;Set address of Copper list
COPJMP1(A5)                       ;Transfer this address to Copper program
                                  ;counter
MOVE.W #$8080,DMACON(A5)          ;enable Copper DMA
```

Sample program

Finally, here is a sample program. It uses two WAIT commands and three MOVE commands to display black, red and yellow bars on the screen. It can be created with a simple Copper list and offers a good example. Enter the program with a standard assembler for the Amiga (such as AssemPro):

```
;*** Example for a simple Copper list ***

;CustomChip-Register

INTENA  = $9A            ;Interrupt-Enable register (write)
DMACON  = $96            ;DMA-Control register (write)
COLOR00 = $180           ;Color palette register 0

;Copper-Register

COP1LC  = $80            ;Address of 1st Copper list
COP2LC  = $84            ;Address of 2nd Copper list
COPJMP1 = $88            ;Jump to Copper list 1
COPJMP2 = $8a            ;Jump to Copper list 2

;CIA-A Port register A (Mouse key)

CIAAPRA = $BFE001

;Exec Library Base Offsets

OpenLibrary = -30-522    ;LibName,Version/a1,d0
Forbid    = -30-102
Permit    = -30-108
AllocMem = -30-168       ;ByteSize,Requirements/d0,d1
FreeMem  = -30-180       ;MemoryBlock,ByteSize/a1,d0

;graphics base

StartList = 38

;Other Labels

Execbase = 4
Chip = 2 ;Request Chip-RAM

;*** Initialize program ***

;Request memory for Copper list

Start:
move.l Execbase,a6
moveq  #Clsize,d0              ;Set parameters for AllocMem
moveq  #chip,d1                ;ask for Chip-RAM
jsr    AllocMem(a6)            ;request memory
```

```
move.l d0,CLadr                    ;Store address of RAM area
beq.s  Ende                        ;Error! -> End

;Copy Copper list to CLadr

lea    CLstart,a0
move.l CLadr,a1
moveq  #CLsize-1,d0                ;Set loop counter
CLcopy:
move.b (a0)+,(a1)+                 ;Copy Copper list byte by byte
dbf    d0,CLcopy

;*** Main program ***

jsr    forbid(a6)                  ;Task-switching off
lea    $dff000,a5                  ;Base address of registers to A5
move.w #$03a0,dmacon(a5)           ;DMA off
move.l CLadr,cop1lc(a5)            ;Address of Copper list to COP1LC
clr.w  copjmp1(a5)                 ;Load address to Copper program counter

;Turn on Copper-DMA

move.w #$8280,dmacon(a5)

;Wait for left mouse key

Wait: btst #6,ciaapra              ;Test bit
bne.s Wait                         ;Set? Then wait.

;*** End program ***

;Reactivate old Copper list

move.l #GRname,a1                  ;Set parameters for OpenLibrary
clr.l  d0
jsr    OpenLibrary(a6)             ;Open Graphics Library
move.l d0,a4                       ;Address of GraphicsBase to a4
move.l StartList(a4),cop1lc(a5)    ;Load address of Startlist
clr.w  copjmp1(a5)
move.w #$83e0,dmacon(a5)           ;All necessary DMA channels on
jsr    permit(a6)                  ;Task-switching on

;Free memory of Copper list

move.l CLadr,a1                    ;Set parameters for FreeMem
moveq  #CLsize,d0
```

```
jsr     FreeMem(a6)                     ;Free memory
Ende:
clr.l   d0                              ;Clear error-flag
rts                                     ;End program

;Variables

CLadr: dc.l 0

;Constants

GRname: dc.b "graphics.library",0
even                                    ;align

;Copper-List

CLstart:
dc.w color00,$0000                      ;Background color black
dc.w $780f,$fffe                        ;Wait for line 120
dc.w color00,$0f00                      ;Switch to red
dc.w $d70f,$fffe                        ;Line 215
dc.w color00,$0fb0                      ;Gold
dc.w $ffff,$fffe                        ;Impossible position: End Copper-List
CLend:

CLsize = CLend - CLstart

;End of program
```

This program installs the Copper list and then waits until the left mouse button is pressed. Unfortunately, this isn't as easy as it sounds.

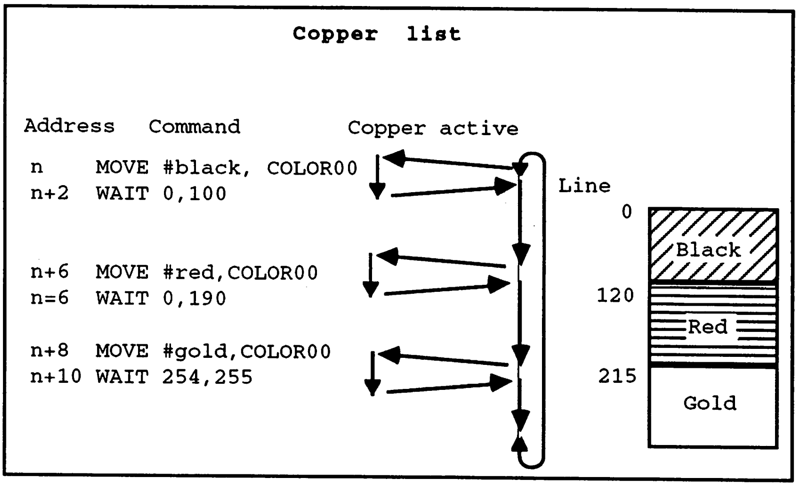
First, you need memory in which to store the Copper list. Like all data for the custom chips, it must be in the chip RAM. Since you can't be sure whether the program is actually in the chip RAM, it is necessary to copy the Copper list into the chip RAM. In a multitasking operating system like that of the Amiga, you can't just write something into memory; you must request the memory. This is done in the program with the AllocMem routine. This returns the address of the requested chip RAM in D0. The Copper list is then copied into memory at this address.

Next, the task switching is disabled by a call to Forbid so that the Amiga processes only your program. This prevents your program from being disturbed by another.

Finally, the Copper is initialized and started. After this, the program tests for the left mouse button by reading the appropriate port bit of CIA-A. If the mouse button is pressed, the processor exits the wait loop.

To get back to the old display, a special Copper list is loaded into the Copper and started. This Copper list is called the startup Copper list and it initializes the screen. Its address is found in the variable area for the part of the operating system responsible for the graphics functions. At the end, task switching is re-enabled with Permit and the occupied memory is released again with FreeMem.

This program contains a number of operating system functions, which you are probably not familiar with yet. Unfortunately, this cannot be avoided if you want to make the program work correctly. But it doesn't matter if you don't understand everything yet. We are discussing the Copper in this section, and this part of the program should be understandable. In the later sections of this book you'll discover the secrets of the operating system and its routines. Enter this example and experiment with the Copper list. Change the WAIT command or add new ones. You can also experiment with a SKIP command.



The Copper list

One more thing about the Copper list: The two WAIT commands contain $E as the horizontal position. This is the start of the horizontal blanking gap. This way the Copper performs the color switch outside the visible area. If 0 is used as the horizontal position, the color switching can be seen at the extreme right edge of the screen.

11.7.5 Playfields

The screen output of the Amiga consists of two basic elements: sprites and playfields. In this section we'll discuss the structure and programming of all types of playfields. The playfield is the basis of the normal screen display. It consists of one to six bit-planes. A playfield is a graphic screen that is built up from a variable number of individual memory areas (the bit-planes). The Amiga provides various ways to display playfields:

- Between 2 and 4096 colors simultaneously in one picture.
- Resolutions of 16 by 1 to 736 by 568 pixels.
- Two completely independent playfields are possible.
- Smooth scrolling in both directions.

All these capabilities can be divided into two groups.

1. Combining the bit-planes to achieve the colors of the individual pixels (displaying the bit pattern from the bit-planes).
2. Determining the form, size and position of the playfields(s).

The various display options

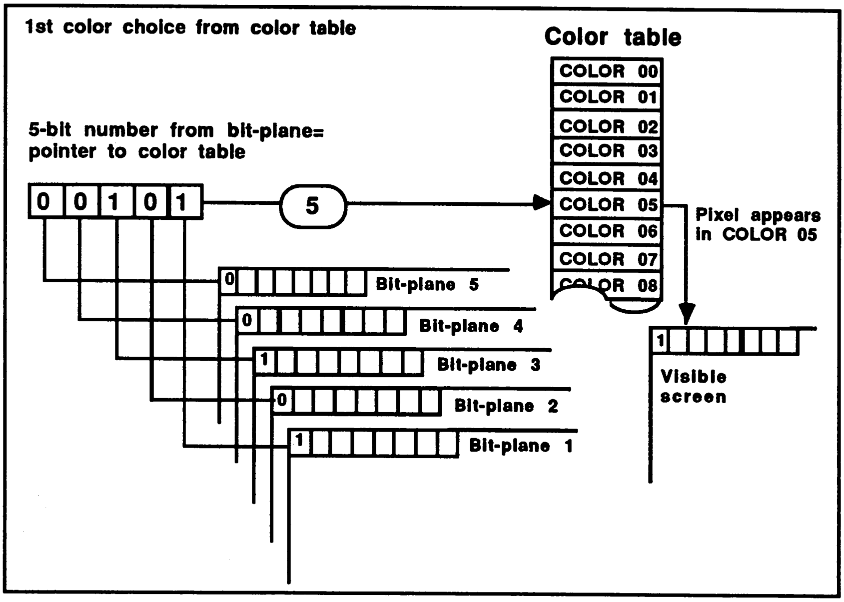
By using from 1 to 6 bit-planes, you are using the corresponding number of bits to represent each pixel. This value must then be converted into one of 4096 colors, since each pixel can naturally have only one color.

The Amiga creates its colors by mixing the component colors, red, green and blue. Each of these three components can have 16 different intensity levels. This results in 4096 color shades (16*16*16 = 4096). Storing a color value requires four bits per component, or 12 bits per color.

If you wanted to allow one of 4096 colors for each pixel, you would need 12 bits per pixel. But a maximum of six bits is possible. Therefore, the six bits must be converted into one of the 4096 possible colors for the visible point.

The color palette

A color palette or color table is used to do this. On the Amiga this contains 32 entries, each of which can hold a 12-bit color value. The value of the first color register COLOR00 is used for the background color and the border color.



Color selection

Color palette registers 0-31 (COLOR00 to COLOR31) are write-only:

Register addr.	Color palette register
$180	COLOR00
$182	COLOR01
...	etc.
$1BE	COLOR31

Structure of a table element:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COLORxx:	x	x	x	x	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
R0-R3	4-bit value for the red component															
G0-G3	4-bit value for the green component															
B0-B3	4-bit value for the blue component															

The four bits labeled "x" are not used.

The value obtained from the bit-planes is used as a pointer to a table element. Since there are only 32 of these color table registers, a maximum of five bit-planes can be combined in this mode. The bit from the lowest-numbered bit-plane supplies the LSB of the table entry. The bit from the highest-numbered bit-plane supplies the MSB.

This method of obtaining the color from a table allows a maximum of 32 colors in a picture, but these colors can be selected from a total of 4096. In high-resolution mode only four planes can be active at one time (16 colors is the limit). In this display mode it doesn't matter how many planes are combined. Some registers may simply remain unused:

Number of bit-planes	Colors	Color registers used
1	2	COLOR00 - COLOR01
2	4	COLOR00 - COLOR03
3	8	COLOR00 - COLOR07
4	16	COLOR00 - COLOR15
5	32	COLOR00 - COLOR31

The extra half-bright mode

In low-resolution mode a maximum of six bit-planes can be used. This yields a range of values of 2^6 or 0 to 63. However, there are only 32 color registers available. The extra half-bright mode uses a special technique to get around this. The lower five bits (bits 0 to 4 from planes 1 to 5) are used as the pointer to a color register. The contents of this color register is output directly to the screen if bit 5 (from bit-plane 6) is 0. If this bit is 1, each component of the color value is divided by 2 before being sent to the screen.

Dividing by 2 means that the bits of the three color components are shifted one bit to the right, which amounts to a binary halving. The intensity of each component is then only half as great, but the proportions of the three components remains constant. The same color will be displayed on the screen, but only half as bright.

Example:

Bit no.:	5 4 3 2 1 0
Value from bit-plane:	1 0 0 1 0 0

Yields table entry no. 8 (binary 01000 is 8), COLOR08 contains the following value (color: orange):

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
1	1	1	0	0	1	1	0	0	0	0	1

Since bit 5 = 1, the values are shifted by 1 bit:

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
0	1	1	1	0	0	1	1	0	0	0	0

This value still corresponds to orange, but now it's only half as bright. By selecting appropriate color values for the 32 registers, it is possible for each pixel to take on one of 64 possible colors in the extra half-bright mode. The color registers store the bright colors, which can then be dimmed by setting bit 5.

The hold-and-modify mode

This mode allows the display of all 4096 colors in one picture. Like the extra half-bright mode, it is possible only at low-resolution, since all six bit-planes are required. In this mode the colors in a normal picture seldom make extreme changes from pixel to pixel. Usually smooth transitions from bright to dark or dark to bright are needed.

In the hold-and-modify mode, called HAM for short, the color of the previous pixel is modified by the one that follows it. This is responsible for the fine gradations of shading that can be achieved (e.g., by incrementing the blue component by one step with each successive pixel). The limitation is that only one component can change at a time (i.e., only the red, blue or green intensity can be affected from one pixel to the next). To get a smooth transition from dark to light, all three color components must change for many color mixes. In the HAM mode this can be accomplished only by setting one of the components to the desired value at each pixel. This requires three pixels. By comparison, the color of a pixel can also be changed directly by fetching one of 16 colors from the color table. How is the value from the bit-planes interpreted in HAM mode?

The upper two bits (bits 4 and 5 from bit-planes 5 and 6) determine the use of the lower four bits (bit-planes 1 to 4). If bits 4 and 5 are 0, the remaining four bits are used as a pointer to one of the color palette registers as usual. This allows 16 colors to be selected directly. With a non-zero combination of bits 4 and 5, the color of the last pixel (to the left of the current one) is taken, two of the three color components are held constant, while the third is replaced by the lower four bits of the current pixel. The top two bits select the component to be changed. This sounds more complicated than it is. The following table explains the use of the various bit combinations:

Bit no.: 5	4	3	2	1	0	Function
0	0	C3	C2	C1	C0	Bits C0 to C3 are used as a pointer to one of the color registers in the range of COLOR00 to COLOR15. This is identical to normal color selection.
0	1	B3	B2	B1	B0	The red and green values of the last (left) pixel are used for the current pixel. The old blue value is replaced by the value in B0 to B3.
1	0	R3	R2	R1	R0	The blue and green values of the last pixel are used for the current pixel. The old red value is replaced by the value in R0 to R3.
1	1	G3	G2	G1	G0	The blue and red values of the last pixel are used for the current pixel The old green value is replaced by the value in G0 to G3.

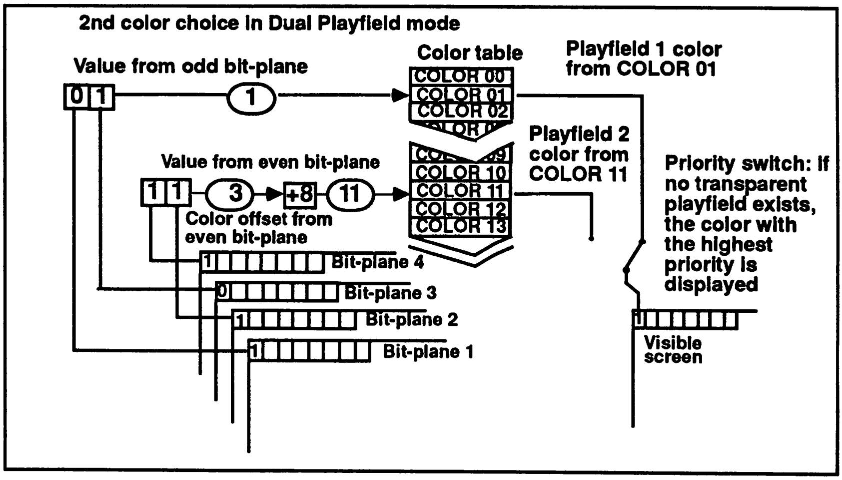
The border color (COLOR00) is used as the color of the previous pixel for the first pixel on a line.

The dual playfield mode

The previously described modes use only one playfield. The dual playfield mode allows two completely independent playfields to be displayed simultaneously. It's as though there are two screens superimposed on each other on the same monitor. They can (almost) be used completely independently of each other.

This is especially interesting for games. For example, a telescope effect can be produced very easily. The front playfield is filled with black pixels; all except for a hole in the middle through a section of the second playfield can be seen.

Each of the two playfields gets half the active bit-planes for its display. Playfield 1 is formed from the odd planes, playfield 2 from the even ones. If an odd number of bit-planes is being used, playfield 1 has one more available to it than playfield 2.



The dual playfield principle

The color selection in dual playfield mode is performed as usual: The value belonging to a pixel from all the odd bit-planes (playfield 1) or all the even planes (playfield 2) is used as a pointer to an entry in the color table. Since each playfield can consist of a maximum of three planes, a maximum of eight colors are possible. For playfield 1, the lower eight entries of the color table are used (COLOR00 to COLOR07). For playfield 2, an offset of 8 is added to the value from the bit-planes, which puts its colors in positions 8 to 15 (COLOR08 to COLOR15).

If a pixel has a value of 0, it is made transparent. This means that screen elements lying behind it can be seen. This can be the other playfield, sprites or simply the background (COLOR00).

The dual playfield mode can also be used in the high-resolution mode. Each playfield has only four colors in this mode. The division of the color registers doesn't change, but the upper four registers of each playfield are unused (playfield 1: COLOR04 to 07, playfield 2: COLOR12 to 15).

Division of the bit-planes in dual playfield mode:

Bit-planes	Planes in playfield 1	Planes in playfield 2
1	Plane 1	none
2	Plane 1	Plane 2
3	Planes 1 and 3	Plane 2
4	Planes 1 and 3	Planes 2 and 4
5	Planes 1, 3 and 5	Planes 2 and 4
6	Planes 1, 3 and 5	Planes 2, 4 and 6

Color selection in dual playfield mode:

Playfield 1		Playfield 2	
Planes 5, 3, 1	Color reg.	Planes 6, 4, 2	Color reg.
0 0 0	Transparent	0 0 0	Transparent
0 0 1	COLOR01	0 0 1	COLOR09
0 1 0	COLOR02	0 1 0	COLOR10
0 1 1	COLOR03	0 1 1	COLOR11
1 0 0	COLOR04	1 0 0	COLOR12
1 0 1	COLOR05	1 0 1	COLOR13
1 1 0	COLOR06	1 1 0	COLOR14
1 1 1	COLOR07	1 1 1	COLOR15

Construction of the playfields

As previously mentioned, a playfield consists of a given number of bit-planes. What do these bit-planes look like? We said earlier that they were conceived as continuous areas of memory, in which a screen line was represented by a number of words depending on the screen width. In the normal case this is 20 words for low resolution (320 pixels divided by 16 pixels per word) and 40 (640/16) for high resolution.

The following steps are needed to determine the exact construction of the playfield:

- Define the desired screen size
- Set the bit-plane size
- Select the number of bit-planes
- Initialize the color table
- Set the desired mode (hi-res, lo-res, HAM, etc.)

- Construct the Copper list
- Initialize the Copper
- Activate the Copper and bit-plane DMA

Setting the screen size

The Amiga allows the upper left corner and the lower right corner of the visible area of the playfields to be set anywhere. This allows the picture position and size to be varied. The resolution is one raster line vertically and one low-resolution pixel horizontally. Two registers contain the values. DIWSTRT (DIsplay Window STaRT) sets the horizontal and vertical start positions of the screen window (i.e., the line and column where the display of the playfield begins).

DIWSTOP (DIsplay Window STOP) contains the end position + 1. This refers to the first line/column after the playfield. If the playfield extends up to 250 lines, 251 must be given as the DIWSTOP value.

The border color is displayed outside the visible area (this corresponds to the background color and comes from the COLOR00 register).

DIWSTRT $08E (write-only)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

DIWSTOP $90 (write-only)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

The start position stored in DIWSTRT is limited to the upper left quadrant of the screen, lines and columns 0 to 255, since the missing MSB's, V8 and H8, are assumed to be 0. The same applies to the horizontal end position, only here H8 is assumed to be 1, which places it in the range of column 256 to 458. A different method is used for the vertical end position. Positions less and greater than row 256 should be possible. The MSB of the vertical end position, V8, is created by inverting the V7 bit. This makes an end position in the range of lines 128 to 312 possible. For end positions from 256 to 312, you set V7 to 0 and V8 to 1. If V7 is 1, V8 will be 0, indicating a position between 128 and 255.

The normal screen window has an upper left corner position of horizontal 129 and vertical 41 (129,41) and a lower right corner position of 448,296. DIWSTOP must be set at 449,297. The corresponding hexadecimal values for DIWSTRT and DIWSTOP are $2981 and $29C1. With these values the normal Amiga screen of 640 by 256 pixels (or 320 by 256) is centered in the middle of the monitor.

Why isn't the whole screen area used? There are several reasons for this. First, a normal monitor does not display the entire picture. Its visible range normally begins a few columns or lines after the blanking gap. In addition, a picture tube is not rectangular. If the screen window was set as high and wide as the monitor tube, the corners of the tube would hide part of the picture.

Another limitation on the DIWSTRT and DIWSTOP values is imposed by the blanking gaps. The vertical gap is in the range of lines 0 to 25. This limits the visible vertical area to lines 26 to 312 ($1A to $138). The horizontal blanking gap lies between columns 30 to 106 ($1E to $6A). Visible horizontal positions begin at column 107 ($6B).

After the position of the screen window has been set, the start and end of the bit-plane DMA must be set as well. Proper timing of the reading of data from the bit-planes is required to ensure that the pixels will appear at the desired time on the screen. Vertically, this isn't a problem. Screen DMA begins and ends in the lines established by DIWSTRT and DIWSTOP as the boundaries of the screen window.

Horizontally, it is somewhat more complicated. In order for a pixel to be displayed on the screen, the current words of all bit-planes are required by the electronics. For six bit-planes in low resolution, this takes eight bus cycles. For high resolution, four cycles are required. (Remember: In one bus cycle, two low-resolution or four high-resolution pixels are displayed.)

In addition, the hardware needs a half bus cycle before the data can appear on the screen. Therefore, the bit-plane DMA must begin exactly 8.5 cycles (17 pixels) before the start of the screen window (4.5 cycles or 9 pixels in high resolution).

The bus cycle of the first bit-plane DMA in the line is stored in the DDFSTRT (Display Data Fetch STaRT) register, and that of the last in DFFSTOP (Display Data Fetch STOP):

DDFSTRT $092 (write-only), DDFSTOP $094 (write-only)

Bit no.:	15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	x	x	x	x	x	x	x	H8	H7	H6	H5	H4	H3	x	x

The resolution is eight bus cycles in lo-res mode (with H3 always 0) and four in hi-res mode. H3 serves as the lowest bit. The reason for the limited resolution lies in the division of the bit-plane DMA. In lo-res mode, each bit-plane is read once every eight bus cycles, so the DDFSTRT value must be an integral multiple of eight (H1 to H3 = 0). The same applies to hi-res mode, except that the bit-planes are read every four bus cycles (H1 and H2 = 0). Regardless of the resolution, the difference between DIWSTRT and DIWSTOP must always be divisible by eight, since the hardware always divides the lines into sections of eight bus cycles. Even in hi-res mode the bit-plane DMA is performed for eight bus cycles beyond DIWSTOP, so that 32 points are always read.

The correct values for DIWSTRT and DIWSTOP are calculated as follows:

Calculation of DDFSTRT and DDFSTOP in lo-res mode:

```
HStart   = Horizontal start of screen window
DDFSTRT = (HStart/2 - 8.5) AND $FFF8
DDFSTOP = DDFSTRT + (pixels per line/2 - 8)
```

For HStart = $81 and 320 pixels per line this gives:

```
DDFSTRT = ($81/2 - 8.5) AND $FFF8 = $38
DDFSTOP = $38 + (320/2 - 8) = $D0
```

Calculation of DDFSTRT and DDFSTOP in hi-res mode:

```
DDFSTRT = (HStart/2 - 4.5) AND $FFFC
DDFSTOP = DDFSTRT + (pixels per line/4 - 8)
```

For HStart = $81 and 640 pixels per line this gives:

```
DDFSTRT = ($81/2 - 4.5) AND $FFFC = $3C
DDFSTOP = $3C + (640/4 - 8) = $D4
```

DDFSTRT may not be less than $18 and DDFSTOP may not be greater than $D8. A DDFSTRT value less than $28 does not make sense, since pixels would then have to be displayed during the horizontal blanking gap, which is not possible (except in scrolling). Since the DMA cycles of bit-planes and sprites overlap with DDFSTRT positions less than $34, some sprites may not be visible, depending on the value of DDFSTRT.

Moving the screen window

If you want to move the screen window horizontally by using DIWSTRT and STOP, it may occur that the difference between DIWSTRT and DDFSTRT is not exactly 8.5 bus cycles (17 pixels), since DFFSTRT can only be set in steps of eight bus cycles. In such a case, a part of the first data word would disappear into the invisible area to the left of the screen window. To prevent this, it is possible to shift the data to the right before sending it to the screen, so that it matches the start of the screen window. The section on scrolling explains how this is done.

Setting bit-map addresses

The values in DDFSTRT and DDFSTOP determine how many data words are displayed per line. The start address must now be set for each bit-map so that the DMA controller can find pixel data. Twelve registers contain these addresses. A pair of registers, BPLxPTH and BPLxPTL, is used for each bit-plane. Together they are referred to as simply BPLxPT (Bit-plane x Pointer).

Addr.	Name	Function	
$0E0	BPL1PTH	Start address of	Bits 16-20
$0E2	BPL1PTL	bit-plane 1	Bits 0-15
$0E4	BPL2PTH	Start address of	Bits 16-20
$0E6	BPL2PTL	bit-plane 2	Bits 0-15
$0E8	BPL3PTH	Start address of	Bits 16-20
$0EA	BPL3PTL	bit-plane 3	Bits 0-15
$0EC	BPL4PTH	Start address of	Bits 16-20
$0EE	BPL4PTL	bit-plane 4	Bits 0-15
$0F0	BPL5PTH	Start address of	Bits 16-20
$0F2	BPL5PTL	bit-plane 5	Bits 0-15
$0F4	BPL6PTH	Start address of	Bits 16-20
$0F6	BPL6PTL	bit-plane 6	Bits 0-15

The DMA controller does the following when displaying a bit-plane: The bit-plane DMA remains inactive until the first line of the screen window is reached (DIWSTRT). Now it gets the data words from the various bit-planes at the column stored in DFFSTRT. It uses BPLxPT as a pointer to the data in the chip RAM. After each data word is read, BPLxPT is incremented by one word. The words read go to the BPLxDAT registers. These registers are used only by the DMA channel. When all six BPLxDAT registers have been provided with the corresponding data words from the bit-planes, the data goes bit by bit to the video logic in Denise, which selects one of the 4096 colors, depending on the color mode chosen, and then outputs this to the screen.

When DFFSTOP is reached, the bit-plane DMA pauses until DFFSTRT for the next line, then the process is repeated until the last line of the screen window (DIWSTOP) is displayed.

The BPLxPT now points to the first word after the bit-plane. But since the BPLxPT should point to the first word in the bit-plane by the next picture, it must be set back to this value. The Copper takes care of this quickly and easily. A simple Copper list for a playfield with four bit-planes looks like this:

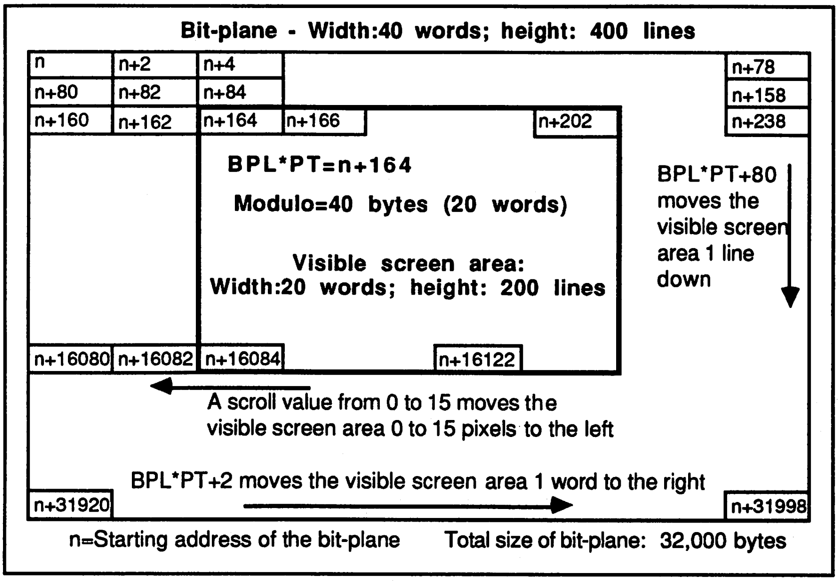
```
AddrPlanexH = Address of bitplane x, bits 16-20
AddrPlanexL = Address of bitplane x, bits 0-15
MOVE #AddrPlane1H,BPL1PTH     ;initialize pointer to bitplane 1
MOVE #AddrPlane1L,BPL1PTL     ;
MOVE #AddrPlane2H,BPL2PTH     ;initialize pointer to bitplane 2
MOVE #AddrPlane2L,BPL2PTL     ;
MOVE #AddrPlane3H,BPL3PTH     ;initialize pointer to bitplane 3
MOVE #AddrPlane3L,BPL3PTL     ;
```

```
MOVE #AddrPlane4H,BPL4PTH      ;initialize pointer to bitplane 4
MOVE #AddrPlane4L,BPL4PTL      ;
WAIT ($FF,$FE)                 ;End of Copper list (wait for
                               ;impossible screen position)
```

Resetting the BPLxPT is absolutely necessary. If you don't use a Copper list, this must be done by the processor in the vertical blanking interrupt.

Scrolling and extra-large playfields

The previous playfields were always the same size as the screen. However, it would often be useful to have a large playfield in memory, not all of which is visible on the screen at one time, but which can be smoothly scrolled in all directions. This is easily done on the Amiga. The following sections illustrate this in both the X and Y directions.



Scrolling

Extra-tall playfields and vertical scrolling

Vertical scrolling is very easy to do. The necessary bit-planes are placed in memory as usual, but this time they contain more lines than the screen. With a standard window height of 256 lines, for example, a double-height playfield is simply 512 lines in memory. In order to move the screen window smoothly over this extra-tall playfield, you change the values of BPLxPT. If you want the screen window to show lines 100 to 356, the BPLxPT pointer must be set to the first word of the 100th line. With a screen width of 320 pixels, each line occupies 20 words (40 bytes) in memory. Multiplying by 100 lines gives an address of 4000. Add this to the starting address of the playfield, and you have the desired value for BPLxPT. To scroll the playfield in the screen window, simply change this value by one or more lines with each picture, depending on the scroll speed desired. Since the BPLxPT can only be changed outside the visible area, a Copper list is used. You can change the values in the Copper list, and the Copper automatically writes them to the BPLxPT registers at the right time. You just have to be careful not to change the Copper list while the Copper is accessing its commands. Otherwise the processor might change one word of the address while the Copper is reading it and the Copper gets the wrong address.

Extra-wide playfields and horizontal scrolling

Special registers exist for horizontal scrolling and extra-wide playfields (all write-only):

$108	BPL1MOD	Modulo value for the odd bit-planes
$10A	BPL2MOD	Modulo value for the even bit-planes

BPLCON1 $102

Bit no:	15-8	7	6	5	4	3	2	1	0
Function:	Unused	P2H3	P2H2	P2H1	P2H0	P1H3	P1H2	P1H1	P1H0
P1H0 - P1H3	Position of the even planes (four bits)								
P2H0 - P2H3	Position of the odd planes (four bits)								

The modulo values from the BPLxMOD registers allow (so to speak) rectangular memory areas. This principle is used often in the Amiga hardware. Inside a large memory area, which is divided into rows and columns, a smaller area of specified height and width can be defined. Let's say that the large memory area, in this case our playfield, is 640 pixels wide and 256 high. This gives us 256 lines of 40 words (80 bytes) each. The smaller area corresponds to the screen window and has the normal size of 320 x 200 pixels, or only 20 words per line. The problem is that when a line is output, BPLxPT is incremented by 20 words. But in order to get the next line of your playfield, it must be incremented by 40 words. After each line, another 20 words must be added to BPLxPT. The

Amiga can take care of this automatically. The difference between the two different line lengths is written into the modulo-register. After a line is output, this value is automatically added to the BPLxPT.

- Width of playfield: 80 bytes (40 words).
- Width of screen window: 40 bytes (20 words).
- Modulo value needed: 40 bytes (The modulo value must always be an even number of bytes).
- Start = start address of the first line of the playfield.

Output of the 1st line:

Word:	0	1	2	3	...	19
BPLxPT:	Start	Start+2	Start+4	Start+6	...	Start+38

After the last word is output, BPLxPT is incremented by 1 word:

```
BPLxPT = Start+40
```

After the end of the line, the modulo value is added to BPLxPT:

```
BPLxPT = BPLxPT + modulo BPLxPT = Start+40 + 40 = Start+80
```

Output of the 2nd line:

Word:	0	1	2	3	...	19
BPLxPT:	Start+80	Start+82	Start+84	Start+86	...	Start+118

This example shows the left half of the large bit-map being displayed in the screen window. To start at a different horizontal position, simply add the desired number of words to the start value of BPLxPT and keep the modulo value the same.

The initial values are as previously shown. The only difference is that BPLxPT is not at Start, but at Start+40, so the right half of the large playfield is displayed.

Output of the 1st line:

Word:	0	1	2	3	...	19
BPLxPT:	Start+40	Start+42	Start+44	Start+46	...	Start+78

After the last word is output:

```
BPLxPT = Start+80
```

Now the modulo value is added to BPLxPT:

```
BPLxPT = BPLxPT+modulo BPLxPT = Start+80 + 40 = Start+120
```

Output of the 2nd line:

Word:	0	1	2	3	...	19
BPLxPT:	Start+120	Start+122	Start+124	Start+126	...	Start+158

Separate modulo values can be set for the odd and even bit-planes. This allows two different sized playfields in the dual-playfield mode.

If this mode is not being used, set both BPLxMOD registers to the same modulo value.

The screen can be moved horizontally in steps of 16 pixels with the BPLxPT and BPLxMOD registers. Fine scrolling in single pixel steps is possible with the BPLCON1 register. The lower four bits contain the scroll value for the even planes, bits 4 to 7 for the odd planes. This scroll value delays the output of the pixel data read for the corresponding planes. If it is zero, the data is output exactly 8.5 (in hi-res, 4.5) bus cycles after the DDFSTRT position; otherwise it appears up to 15 pixels later, depending on the scroll value. So, the picture is shifted to the right within the screen window by the value in BPLCON1.

Smooth scrolling of the screen contents to the right can be accomplished by incrementing the value of BPLCON1 from 0 to 15 and then setting it back to 0 while decrementing the BPLxPT by one word as previously described.

Left scrolling can be accomplished by decrementing the scroll value from 15 to 0 and then incrementing BPLxPT by one word. BPLCON1 should be changed only outside the visible area. This can be done during the vertical blanking interrupt or the Copper can be used. The value in the Copper list can be changed at any time and will be written to the BPLCON1 register during the vertical blanking gap.

When the BPLCON1 value is used to shift the picture to the right, excess pixels on the left are correctly eliminated from view, but no new ones can appear on the right until new pixel data has been read. To do this, the DDFSTRT value must be set ahead of its normal start by 8 bus cycles (4 cycles in hi-res). The DDFSTRT value is calculated as usual from the desired screen window and then decremented by 8 (or 4). From the normal DDFSTRT value of $38 we get $30 (sprite 7 is lost). The extra word read is normally not visible. But when the scroll value is other than zero, its pixels appear in the free positions at the left edge of the screen window. If the window is 320 pixels wide, 21, instead of the usual 20, data words are now read per line. This must be considered when calculating the bit-planes and modulo values.

The screen window can also be placed at any desired horizontal position by using the scroll value. If the difference between DIWSTRT and DFFSTRT is more than 17 pixels, you simply shift the read data to the right by the amount over 17.

The interlace mode

Although the interlace mode doubles the number of displayable lines, its programming technique differs from that of the normal display mode only by a different modulo value and a new Copper list. As explained earlier, in interlace mode the odd and even lines are displayed in alternate pictures. To allow an interlace playfield to be represented normally in memory, the modulo value is set equal to the number of words per line. After a line is output, the length of a line is added again to BPLxPT, which amounts to skipping over the next line. In each picture only every other line is displayed. Now the BPLxPT only needs to be set to the first or second line of the playfield, depending on the frame type, so that only the even or the odd lines will be shown. In a long frame BPLxPT is set to line 1 (odd lines only); in a short frame it is set to line 2 (even lines only). The Copper list for an interlace playfield is somewhat more complicated, because two lists are needed. There is one for each frame type, so for each picture, we alternate between them:

Copper list for an interlaced playfield:

```
Line1 = address of first line of bitplane.
Line2 = address of second line of bitplane.
```

Copper1:

```
MOVE #Line1Hi,BPLxPTH          ;set pointer for BPLxPT to the
MOVE #Line1Lo,BPLxPTL          ;address of the first line
...                            ;other Copper commands
MOVE #Copper2Hi,COP1LCH        ;set address of Copper list
MOVE #Copper2Lo,COP1LCL        ;to Copper2
WAIT ($FF,$FE)                 ;end of 1st Copper list
```

Copper2:

```
MOVE #Line2Hi,BPLxPTH          ;set pointer for BPLxPT to the
MOVE #Line2Lo,BPLxPTL          ;address of the second line
...                            ;other Copper commands
MOVE #Copper1Hi,COP1LCH        ;set address of Copper list
MOVE #Copper1Lo,COP1LCL        ;to Copper1
WAIT ($FF,$FE)                 ;end of 2nd Copper list
```

The Copper alternates its Copper list after each frame by loading the address of the other list into COP1LC at the end of a command list. This address is automatically loaded into the program counter of the Copper at the start of the next frame. The interlace mode should be initialized carefully so that the Copper list for odd lines is actually processed within a long frame:

- Set COP1LC to Copper1.
- Set the LOF-bit (bit 15) in the VPOS register ($2A) to 0. This ensures that the first frame, after interlace mode is enabled, is a long frame and therefore suited to Copper1. The LOF bit is inverted with each frame in interlace mode. If it is set to 0, it changes to 1 at the start of the next frame. The first frame is sure to be a long frame.
- Interlace mode on.
- Wait for first line of next picture (line 0).
- Copper DMA on.

All other register functions remain unchanged in interlace mode. All line specifications (such as in DIWSTRT) always refer to the line number within the current frame (0 - 311 for a short frame and 0 - 312 for a long frame). If the interlace mode is enabled without changing other registers, a faint flickering is noticeable because the lines of the frames are now displaced from each other, even though both frames contain the same graphics data. When doubly-large bit-planes and the appropriate modulo values are set up with suitable Copper lists so that different data is displayed in each frame, then the desired doubling in the number of lines is noticed.

The interlace mode now results in a stronger flickering since each line is displayed only once every two frames, thus being refreshed 25 times per second. This flickering can be reduced to a minimum by selecting the lowest possible contrast (intensity difference) between colors displayed. It is more difficult for the human eye to distinguish changes at low contrast.

The control registers

There are three control registers for activating the various modes: BPLCON0 to BPLCON2. BPLCON1 contains the scroll value. The other two are constructed as follows:

BPLCON0 $100

Bit no.	Name	Function
15	HIRES	High-resolution mode on (HIRES = 1)
14	BPU2	The three BPUx bits comprise a 3-bit number
13	BPU1	which contains the number
12	BPU0	of bit-planes used (0 to 6).
11	HOMOD	Hold-and-modify on (HOMOD = 1)
10	DBPLF	Dual playfield on (DBPLF = 1)
9	COLOR	Video output color (COLOR = 1)
8	GAUD	Genlock audio on (GAUD = 1)
7	---	Unused
6	SHRES	Super hi-res on
5-4	---	Unused
3	LPEN	Activate lightpen input (LPEN = 1)
2	LACE	Interlace mode on (LACE = 1)
1	ERSY	External synchronization on (ERSY = 1)
0	BPLCON3ON	Bit-plane control register 3 on

HIRES

The HIRES bit enables the high-resolution display mode (hi-res, 640 pixels/line).

BPL0 - BPL2

These three bits form a 3-bit number which selects the number of active bit-planes. Only values between 0 and 6 are allowed.

HOMOD and DBPLF

These two bits select the corresponding modes. They cannot both be active at the same time. The extra-half-bright mode is automatically activated when all six bit-planes are enabled and neither HOMOD nor DBPLF is selected.

LACE

When the LACE bit is set, the LOF-frame bit in the VPOS register is inverted at the start of each new frame, causing the desired alternation between long and short frames.

SHRES

Super-hi-res mode is enabled with this bit. This mode was introduced with the new ECS custom chips of the A3000.

COLOR

The COLOR bit turns the color burst output of Agnus on. Only when Agnus delivers this color burst signal can the video mixer create a color video signal. Otherwise it is black and white. The RGB output is not affected by this.

ERSY

The ERSY bit switches the connections for the vertical and horizontal synchronization signals from output to input. This allows the Amiga picture to be synchronized by external signals. The genlock interface uses this bit to be able to mix the Amiga's picture with another video image. The GAUD bit is also provided for the genlock interface.

BPLCON3ON

This activates the new ECS chip's BPLCON3 register.

BPLCON2 $104

Bit no.:	15-7	6	5	4	3	2	1	0
Function:	Gen.	PF2PRI	PF2P2	PF2P1	PF2P0	PF1P2	PF1P1	PF1P0

PF2P0-PF2P2 and PF1P0-PF1P2 determine the priority of the sprites in relation to the playfields (see the next section).

PF2PRI: If this bit is set, the even planes have priority over (appear before) the odd planes. It has a visible effect only in dual playfield mode.

Activating the screen display

The upper bits of the BPLCON2 register contain more control bits for genlock use.

After all the registers have been loaded with the desired values, the DMA channel for the bit-planes must be enabled and, if the Copper is used (which is normally the case), its DMA channel must also be enabled. The following MOVE command accomplishes this by setting the DMAEN, BPLEN and COPEN bits in the DMA control register DMACON:

```
MOVE.W #$8310,$DFF096
```

Sample programs

Program 1: Extra-half-bright demo

This program creates a playfield with the standard dimensions 320 by 256 pixels in lo-res mode. Six bit-planes are used so the extra-half-bright mode is activated automatically. At the beginning the program allocates the memory needed.

Since the addresses of the individual bit-planes are not known until this time, the Copper list is not copied from the program, but created directly in the chip RAM. It contains only the commands for setting the BPLxPT registers.

To show you something of the 64 possible colors, the program draws 16x16-pixel blocks in all colors at random positions. The VHPOS register is used as a random-number generator.

```
;*** Demo for the Extra-Halfbright-Mode ***

;CustomChip-Register

INTENA  = $9A  ;Interrupt-Enable-Register (write)
DMACON  = $96  ;DMA-Control register (write)
COLOR00 = $180 ;Color palette register 0
VHPOSR  = $6   ;Ray position (read)

;Copper Register

COP1LC  = $80 ;Address of 1. Copperlist
COP2LC  = $84 ;Address of 2. Copperlist
COPJMP1 = $88 ;Jump to Copperlist 1
COPJMP2 = $8a ;Jump to Copperlist 2

;Bitplane Register

BPLCON0 = $100 ;Bitplane Control register 0
BPLCON1 = $102 ;1 (Scrollw value)
BPLCON2 = $104 ;2 (Sprite<>Playfield Priority)
BPL1PTH = $0E0 ;Number of  1. Bitplane
BPL1PTL = $0E2 ;
BPL1MOD = $108 ;Modulo-Value for odd Bit-Planes
BPL2MOD = $10A ;Modul0-Value for even Bit-Planes
DIWSTRT = $08E ;Start of the screen windows
DIWSTOP = $090 ;End of the screen windows
DDFSTRT = $092 ;Bit-Plane DMA Start
DDFSTOP = $094 ;Bit-Plane DMA Stop

;CIA-A Port register A (Mouse key)

CIAAPRA = $bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522        ;LibName,Version/a1,d0
Forbid      = -30-102
Permit      = -30-108
AllocMem    = -30-168        ;ByteSize,Requirements/d0,d1
FreeMem     = -30-180        ;MemoryBlock,ByteSize/a1,d0

;graphics base

StartList   = 38

;other Labels
```

```
Execbase    = 4
Planesize   = 40*200          ;Size of Bitplane: 40 Bytes by 200 lines
CLsize      = 13*4            ;The Copperlist with 13 commands
Chip        = 2               ;Chip-RAM request
Clear       = Chip+$10000     ;clear previous Chip-RAM

;*** Initialize programm ***

Start:

;Request memory for the  Bitplanes

 move.l Execbase,a6
 move.l #Planesize*6,d0       ;Memory size of all Planes
 move.l #clear,d1             ;Memory to be with filled with nulls
 jsr    AllocMem(a6)          ;Request memory
 move.l d0,Planeadr           ;Address of the first memory Plane
 beq    Ende                  ;Error! -> Ende

;Request memory for  Copperlist

 moveq  #Clsize,d0            ;Size of the Copperlist
 moveq  #chip,d1
 jsr    AllocMem(a6)
 move.l d0,CLadr
 beq    FreePlane             ;Error! -> Free RAM for Bitplanes

;Build Copperlist

 moveq  #5,d4                 ;6 Planes = 6 loops to run through
 move.l d0,a0                 ;Address of the Copperlist to a0
 move.l Planeadr,d1
 move.w #bpl1pth,d3           ;first Register to d3

MakeCL: move.w d3,(a0)+       ;BPLxPTH ins RAM
 addq.w #2,d3                 ;next Register
 swap   d1
 move.w d1,(a0)+              ;Hi-word of the Plane address in RAM
 move.w d3,(a0)+              ;BPLxPTL ins RAM
 addq.w #2,d3                 ;next Register
 swap   d1
 move.w d1,(a0)+              ;Lo-word of the  Plane address in RAM
 add.l  #planesize,d1         ;Address of the next Plane calculated

 dbf    d4,MakeCL

 move.l #$fffffffe,(a0)       ;End of Copperlist
```

```
;*** Main programm ***

;DMA and Task switching off

 jsr    forbid(a6)
 lea    $dff000,a5
 move.w #$03e0,dmacon(a5)

;Copper initialization

 move.l CLadr,cop1lc(a5)
 clr.w  copjmp1(a5)

;Color table with different color fills

 moveq  #31,d0                 ;Value for color register
 lea    color00(a5),a1
 moveq  #1,d1                  ;first color
SetTab:
 move.w d1,(a1)+               ;Color in color register
 mulu   #3,d1                  ;calculate next color
 dbf    d0,SetTab

;Playfield initialization

 move.w #$3081,diwstrt(a5)     ;Standard value for
 move.w #$30c1,diwstop(a5)     ;screen window
 move.w #$0038,ddfstrt(a5)     ;and BitplaneDMA
 move.w #$00d0,ddfstop(a5)
 move.w #%0110001000000000,bplcon0(a5) ;6 Bitplanes
 clr.w  bplcon1(a5)            ;no Scrolling
 clr.w  bplcon2(a5)            ;Priority makes no difference
 clr.w  bpl1mod(a5)            ;Modulo for all Planes equals Null
 clr.w  bpl2mod(a5)

;DMA on
 move.w #$8380,dmacon(a5)

;Bitplane modification

 moveq  #40,d5                 ;Bytes per line
 clr.l  d2                     ;Begin with color 0

Loop:   clr.l d0
 move.w vhposr(a5),d0          ;Random value to  d0
 and.w  #$3ffe,d0              ;Unnecessary Bits masked out
 cmp.w  #$2580,d0              ;Large as Plane?
 bcs    Continue               ;When not , then continue
```

```
 and.w  #$1ffe,d0            ;else erase upper bit
Continue: move.l Planeadr,a4 ;Address of the  1.Bitplane to a4
 add.l  d0,a4                ;Calculate address of the Blocks
 moveq  #5,d4                ;Number for Bitplanes
 move.l d2,d3                ;Color in work register

Block:
 clr.l  d1
 lsr    #1,d3                ;one Bit of color number in X-Flag
 negx.w d1                   ;use d1 to adjust X-Flag
 moveq  #15,d0               ;16 lines per Block
 move.l a4,a3                ;Block address in working register

Fill:
 move.w d1,(a3)              ;Word in Bitplane
 add.l  d5,a3                ;compute next line
 dbf    d0,Fill

 add.l  #Planesize,a4        ;next Bitplane
 dbf    d4,Block

 addq.b #1,d2                ;next color
 btst   #6,ciaapra           ;mouse key pressed?
 bne    Loop                 ;no  -> then continue

;*** End programm ***

;Activate old  Copperlist

 move.l #GRname,a1           ;Set parameter for OpenLibrary
 clr.l  d0
 jsr    OpenLibrary(a6)      ;Graphics Library open
 move.l d0,a4
 move.l StartList(a4),cop1lc(a5) ;Address of Startlist
 clr.w  copjmp1(a5)
 move.w #$8060,dmacon(a5)    ;reenable  DMA
 jsr    permit(a6)           ;Task-Switching on

;Free memory for Copperlist

 move.l CLadr,a1             ;Set parameter for FreeMem
 moveq  #CLsize,d0
 jsr    FreeMem(a6)          ;Free memory

;Free memory for  Bitplanes
FreePlane:
 move.l Planeadr,a1
 move.l #Planesize*6,d0
```

```
 jsr     FreeMem(a6)

Ende:
 clr.l  d0
 rts                            ;Program end

;Variables

CLadr:     dc.l 0
Planeadr: dc.l 0

;Constants

GRname: dc.b "graphics.library",0

;Program end
 end
```

Program 2: Dual playfield & smooth scrolling

This program uses several effects at once. First it creates a dual-playfield screen with one low-resolution bit-plane per playfield. Then it enlarges the screen window so that no borders can be seen. Finally, it scrolls playfield 1 horizontally and playfield 2 vertically.

The usual routines for memory allocation and release, etc. are used at the start and end, as in the previous example.

Both playfields are filled with a checkerboard pattern of 16x16 pixel blocks.

The main loop of the program, which performs the scrolling, first waits for a line within the vertical blanking gap, in which the operating system has already processed any interrupt routines and the Copper has set the BPLxPT's. Then it increments the vertical scroll counter, calculates the new BPLxPT for playfield 2, and writes it to the Copper list.

The horizontal scroll position results from separating the lower four bits of the scroll counter from the rest. The lower four bits are written into the BPLCON1 register as the scroll value for playfield 1, and the 5th bit is used to calculate the new BPLxPT, which is copied to the Copper list.

Both the horizontal and vertical scroll counters are incremented from 0 to 31 and then reset to 0. This is sufficient for a smooth scrolling effect, since the pattern used for the playfields repeats every 32 pixels.

```
*** Dual-Playfield & Scroll Demo ***

;CustomChip-Register

INTENA   = $9A ;Interrupt-Enable-Register (write)
INTREQR  = $1e ;Interrupt-Request-Register (read)
DMACON  = $96  ;DMA-Control register (write)
COLOR00 = $180 ;Color palette register 0
VPOSR   = $4   ;half line position (read)

;Copper Register

COP1LC  = $80 ;Address of 1. Copperlist
COP2LC  = $84 ;Address of 2. Copperlist
COPJMP1 = $88 ;Jump to Copperlist 1
COPJMP2 = $8a ;Jump to Copperlist 2

;Bitplane Register

BPLCON0 = $100 ;Bitplane control register 0
BPLCON1 = $102 ;1 (Scroll value)
BPLCON2 = $104 ;2 (Sprite<>Playfield Priority)
BPL1PTH = $0E0 ;Pointer to 1. Bitplane
BPL1PTL = $0E2 ;
BPL1MOD = $108 ;Modulo-Value for odd Bit-Planes
BPL2MOD = $10A ;Module-value for even Bit-Planes
DIWSTRT = $08E ;Start of screen windows
DIWSTOP = $090 ;End of screen windows
DDFSTRT = $092 ;Bit-Plane DMA Start
DDFSTOP = $094 ;Bit-Plane DMA Stop

;CIA-A Port register A (Mouse key)

CIAAPRA = $bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0
Forbid      = -30-102
Permit      = -30-108
AllocMem    = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem     = -30-180 ;MemoryBlock,ByteSize/a1,d0
```

```
;graphics base

StartList   = 38

;Misc Labels

Execbase    = 4
Planesize   = 52*345           ;Size of the Bitplane
Planewidth  = 52
CLsize      = 5*4              ;The Copperlist contains 5 commands
Chip        = 2                ;request Chip-RAM
Clear       = Chip+$10000      ;clear previous Chip-RAM

;*** Pre-programm ***

Start:

;Request memory for Bitplanes

 move.l Execbase,a6
 move.l #Planesize*2,d0        ;memory size of the Planes
 move.l #clear,d1
 jsr    AllocMem(a6)           ;Request memory
 move.l d0,Planeadr
 beq    Ende                   ;Error! -> End

;Request memory for the Copperlist

 moveq  #Clsize,d0
 moveq  #chip,d1
 jsr    AllocMem(a6)
 move.l d0,CLadr
 beq    FreePlane              ;Error! -> Free memory for the Planes

;Build Copperlist

 moveq  #1,d4                  ;two Bitplanes
 move.l d0,a0
 move.l Planeadr,d1
 move.w #bpl1pth,d3

MakeCL: move.w d3,(a0)+
 addq.w #2,d3
 swap   d1
 move.w d1,(a0)+
 move.w d3,(a0)+
 addq.w #2,d3
 swap   d1
```

```
 move.w d1,(a0)+
 add.l  #planesize,d1        ;Address of the next Plane

 dbf    d4,MakeCL

 move.l #$fffffffe,(a0)      ;End of the Copperlist

;*** Main programm ***

;DMA and Task switching off

 jsr    forbid(a6)
 lea    $dff000,a5
 move.w #$01e0,dmacon(a5)

;Copper initialization

 move.l CLadr,cop1lc(a5)
 clr.w  copjmp1(a5)

;Playfield initialization

 move.w #0,color00(a5)
 move.w #$0f00,color00+2(a5)
 move.w #$000f,color00+18(a5)
 move.w #$1a64,diwstrt(a5)   ;26,100
 move.w #$39d1,diwstop(a5)   ;313,465
 move.w #$0020,ddfstrt(a5)   ;read one extra word
 move.w #$00d8,ddfstop(a5)
 move.w #%0010011000000000,bplcon0(a5) ;Dual-Playfield and
 clr.w  bplcon1(a5)          ;Scroll to start on 0
 clr.w  bplcon2(a5)          ;Playfield 1 or Playfield 2
 move.w #4,bpl1mod(a5)       ;Modulo on 2 Words
 move.w #4,bpl2mod(a5)

;DMA on
 move.w #$8180,dmacon(a5)

;Bitplanes filled with checker pattern

 move.l planeadr,a0
 move.w #planesize/2-1,d0    ;loop value
 move.w #13*16,d1            ;Height = 16 Lines
 move.l #$ffff0000,d2        ;checker pattern
 move.w d1,d3

fill:   move.l d2,(a0)+
 subq.w #1,d3
```

```
 bne.s   continue
 swap    d2                    ;pattern change
 move.w d1,d3
continue: dbf d0,fill


;Playfields scroll

 clr.l   d0                    ;vertical Scroll position
 clr.l   d1                    ;horizontal Scroll position
 move.l CLadr,a1               ;Address of the Copperlist
 move.l Planeadr,a0            ;Address of the first Bitplane

;Wait on Raster line 16  (for the  Exec-Interrupts)

wait:  move.l vposr(a5),d2     ;read Position
 and.l #$0001FF00,d2           ;horizontal Bits masked
 cmp.l #$00001000,d2           ;wait on line 16
 bne.s wait

;Playfield 1 vertical scroll

 addq.b #2,d0                  ;raise vertical Scroll value
 cmp.w   #$80,d0               ;already  128 (4*32)?
 bne.S   novover
 clr.l   d0                    ;Then back to  0
novover:
 move.l d0,d2                  ;copy scroll value
 lsr.w   #2,d2                 ;copy divided by 4 s
 mulu    #52,d2                ;Number Bytes per line * Scroll position
 add.l   a0,d2                 ;plus Address of first Plane
 add.l   #Planesize,d2         ;plus Plane size
 move.w d2,14(a1)              ;give End address for Copperlist
 swap    d2
 move.w d2,10(a1)

;Playfield 2 horizontal scroll

 addq.b #1,d1                  ;raise horizontal Scroll value
 cmp.w   #$80,d1               ;already 128 (4*32)
 bne.S   nohover
 clr.l   d1                    ;then back to 0
nohover:
 move.l d1,d2                  ;copy scroll value
 lsr.w   #2,d2                 ;copy divided by 4
 move.l d2,d3                  ;copy Scroll position
 and.w   #$FFF0,d2             ;lower  4 Bit masked
 sub.w   d2,d3                 ;lower 4 Bit in d3 isolated
```

```
 move.w d4,bplcon1(a5)        ;last Value in BPLCON1
 move.w d3,d4                 ;new scroll value to  d4
 lsr.w  #3,d2                 ;new Address for Copperlist
 add.l  a0,d2                 ;calculate
 move.w d2,6(a1)              ;and write in Copperlist
 swap   d2
 move.w d2,2(a1)

 btst   #6,ciaapra            ;Mouse key pressed?
 bne.s  wait                  ;NO -> continue

;*** Check programm ***

;Activate old Copperlist

 move.l #GRname,a1            ;Set parameter for OpenLibrary
 clr.l  d0
 jsr    OpenLibrary(a6)       ;Graphics Library open
 move.l d0,a4
 move.l StartList(a4),cop1lc(a5)
 clr.w  copjmp1(a5)
 move.w #$83e0,dmacon(a5)
 jsr    permit(a6)            ;Task-Switching permitted

;Free memory used by Copperlist

 move.l CLadr,a1              ;Set parameter for FreeMem
 moveq  #CLsize,d0
 jsr    FreeMem(a6)           ;Free memory

;Free memory used by Bitplanes
FreePlane:
 move.l Planeadr,a1
 move.l #Planesize*2,d0
 jsr    FreeMem(a6)

Ende:
 clr.l  d0
 rts                          ;Program ends

;Variables

CLadr:    dc.l 0
Planeadr: dc.l 0
test:     dc.l 0

;Constants
```

```
GRname:   dc.b "graphics.library",0

 end
;Program end
```

11.7.6 Sprites

Sprites are small graphic elements that can be used completely independently of the playfields. Each sprite is 16 pixels wide and can have a maximum height of the entire screen window. It can be displayed anywhere on the screen.

Normally a sprite is in front of the playfield(s) and its pixels hide the graphic behind it. The mouse pointer, for example, is implemented as a sprite. Up to eight sprites are possible on the Amiga. A sprite normally has three colors, but two sprites can be combined into one to give a fifteen-color sprite.

Construction of the sprites

Color selection

The color selection for sprites is very similar to that of a dual-playfield screen. A sprite has a width of 16 pixels, which are represented by two data words. The words act like "mini bit-planes," since the color of a pixel is formed by combining corresponding bits of both the words.

The color of the first (leftmost) pixel of the sprite is selected by the high-value bits (bit 15) of the two words. The two low-value bits (bit 0) determine the color of the last pixel. Each pixel is represented by two bits, which means it can have one of four different colors. The color table is used to determine the actual color from this value.

There are no special color registers for the sprites. The sprite colors are obtained from the latter half of the table, color registers 16-31. This means that sprite and playfield colors do not come into conflict unless playfields with more than 16 colors are created.

The following table shows the assignment of color registers and sprites:

Sprite no.	Sprite data	Color register
0 & 1	0 0	transparent
	0 1	COLOR17
	1 0	COLOR18
	1 1	COLOR19
2 & 3	0 0	transparent
	0 1	COLOR21
	1 0	COLOR22
	1 1	COLOR23
4 & 5	0 0	transparent
	0 1	COLOR25
	1 0	COLOR26
	1 1	COLOR27
6 & 7	0 0	transparent
	0 1	COLOR29
	1 0	COLOR30
	1 0	COLOR31

Each two successive sprites have the same color registers.

As in dual-playfield mode, the bit combination of two zeros does not represent a color, but causes the pixel to be transparent. The color of anything behind this pixel is visible in its place, whether this is another sprite, a playfield, or just the background.

If three colors are not enough, two sprites can be combined with each other. The two-bit combinations of the sprites then make up a four-bit number. Sprites can only be combined in successive even-odd pairs (i.e., no. 0 with no. 1, no. 2 with no. 3, etc.). The two data words of the higher-numbered sprite contribute the two high-order bits of the total 4-bit value. This value then serves as a pointer to one of fifteen color registers, with the value zero meaning transparent. The color registers are the same for all four sprite pairs: COLOR16 to COLOR31.

Sprite data	Color register	Sprite data	Color register
0 0 0 0	transparent	1 0 0 0	COLOR24
0 0 0 1	COLOR17	1 0 0 1	COLOR25
0 0 1 0	COLOR18	1 0 1 0	COLOR26
0 0 1 1	COLOR19	1 0 1 1	COLOR27
0 1 0 0	COLOR20	1 1 0 0	COLOR28
0 1 0 1	COLOR21	1 1 0 1	COLOR29
0 1 1 0	COLOR22	1 1 1 0	COLOR30
0 1 1 1	COLOR23	1 1 1 1	COLOR31

The sprite DMA

The Amiga sprites can be programmed very easily. Almost all the work is handled by the sprite DMA channels. The only thing needed to display a sprite on the screen is a special sprite data list in memory. It contains almost all the data needed for the sprite. The DMA controller only needs to be told the address of this list in order for the sprite to appear.

The DMA controller has a DMA channel for each sprite. This can read only two data words in each raster line. This is why a normal sprite is limited to a 16-pixel width and four colors. Since these two data words can be read in every line, the height of a sprite is limited only by that of the screen window.

Construction of a sprite data list

A sprite data list consists of individual lines, each containing two data words. For each raster line, one of these list lines is read via DMA. It can contain either two control words to initialize the sprite, or two data words with the pixel data.

The control words determine the horizontal column and the first and last line of the sprite.

After the DMA controller has read these words and placed them in the appropriate registers, it waits until the electron beam reaches the starting line of the sprite. Then two words are read for each raster line and are output by Denise at the appropriate horizontal position on the screen until the last line of the sprite has been processed. The next two words in the sprite data list are again treated as control words.

If both words are zero, the DMA channel ends its activity. However, it's also possible to specify a new sprite position. The DMA controller then waits for the start line, and the process is repeated until two control words with the value zero are found as the end marker of the list.

Construction of a sprite data list (Start = starting address of the list in chip RAM):

Address	Contents
Start+4	1st and 2nd data words of the 1st line of the sprite
Start+8	1st and 2nd data words of the 2nd line of the sprite
Start+12	1st and 2nd data words of the 3rd line of the sprite
Start+4*n	1st and 2nd data words of the nth line of the sprite
Start+4*(n+1)	0,0 End of the sprite data list

Construction of the first control word:

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	E7	E6	E5	E4	E3	E2	E1	E0	H8	H7	H6	H5	H4	H3	H2	H1

Construction of the second control word:

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	L7	L6	L5	L4	L3	L2	L1	L0	AT	0	0	SHSH1	0	E8	L8	H0

H0 to H8	Horizontal position of the sprite (HSTART)
E0 to E8	First line of the sprite (VSTART)
L0 to L8	Last line of the sprite+1 (VSTOP)
SHSH1	Extra bit for the horizontal position
AT	Attach control bit

The (starting) horizontal position and the starting and ending vertical positions of the sprite are expressed with nine bits each. These bits are divided somewhat impractically between the two control registers.

The resolution in the horizontal direction is one low-resolution pixel, while in the vertical direction it is one raster line. These values are independent of the mode of the playfield(s) and cannot be changed.

The sprites are limited to the screen window (set by DIWSTRT and DIWSTOP). If the coordinates set by the control words are outside this area, the sprites are only partially visible, if at all, since all points that are not within the screen window are cut off.

The horizontal and vertical start positions refer to the upper left corner of the sprite. The vertical stop position defines the first line after the sprite (i.e., the last line of the sprite + 1). The number of lines in the sprite is VSTOP - VSTART.

The following example list displays a sprite at the coordinates 180,160, approximately in the center of the screen. It has a height of eight lines. The last line (VSTOP) is 168. If you combine both data words within a line, you get numbers between 0 and 3, which represent one of the three sprite colors or the transparent pixels. This makes the sprite easier to visualize:

```
0000002222000000
0000220000220000
0002200330022000
0022003113002200
0022003113002200
0002200330022000
0000220000220000
0000002222000000
```

In the data list the two words must be given separately:

```
Start:
dc.w $A05A,$A800     ;HSTART = $B4, VSTART = $A0, VSTOP = $A8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w 0,0             ;End of the sprite data list
```

The AT bit in the 2nd control word determines whether two sprites are combined. It effects only those sprites with odd numbers (sprites 1, 3, 5, 7). For example, if it is set in sprite 1, its data bits are combined with those of sprite 0 to make four-bit pointers to the color table. The order of the bits is then as follows:

Sprite 1 (odd number), second data word:	Bit 3 (MSB)
Sprite 1, first data word:	Bit 2
Sprite 0 (even number), second data word:	Bit 1
Sprite 0, first data word:	Bit 0 (LSB)

If two sprites are to be combined in this manner, their positions must also match. If this is not the case, the old three-color representation is automatically re-enabled. The simplest thing to do is to write the same control words in the two sprite data lists. We'll now give an example of a sprite data list for a fifteen-color sprite. For the sake of simplicity our sprite consists of only four lines. Again, we first visualize the sprite by superimposing the data words. The digits represent the colors of the corresponding pixels. In order to display all fifteen colors and transparent, the hexadecimal digits "A" to "F" are used.

```
0011111111111100
1123456789ABCD11
11EFEFEFEFEFEF11
0011111111111100
```

The structure of the data words can be seen from line 2:

Colors of the sprite:	1123456789ABCD11
Sprite 1, data word 2:	0000000011111100
Sprite 1, data word 1:	0000111100001100
Sprite 0, data word 2:	0011001100110000
Sprite 0, data word 1:	1101010101010111

Horizontal position (HSTART) is 180. The first line of the sprite (VSTART) is 160, and the last line (VSTOP) is 164.

The data list for the entire sprite looks as follows:

```
StartSprite0:
dc.w $A05A,$A400        ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=0
dc.w %0011 1111 1111 1100,%0000 0000 0000 0000
dc.w %1101 0101 0101 0111,%0011 0011 0011 0000
dc.w %1101 0101 0101 0111,%0011 1111 1111 1100
dc.w %0011 1111 1111 1100,%0000 0000 0000 0000
dc.w 0,0

StartSprite1:
dc.w $A05A,$A480        ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=1
dc.w %0000 0000 0000 0000,%0000 0000 0000 0000
dc.w %0000 1111 0000 1100,%0000 0000 1111 1100
dc.w %0011 1111 1111 1100,%0011 1111 1111 1100
dc.w %0000 0000 0000 0000,%0000 0000 0000 0000
dc.w 0,0
```

Multiple sprites through one DMA channel

After a sprite has been displayed, the DMA channel is free. In the previous example, the last sprite data was read in line 163. After that the sprite DMA channel is turned off with the two zeros at the end of the data list. But as we mentioned before, it is also possible to continue using the DMA channel. To do this, simply put two new control words in place of the two zeros in the data list. The only condition is that there must be at least one line free between the first line of the new sprite and the last line of the previous one. For example, if the previous sprite extends through line 163, then the next cannot start before line 165. The reason for this is that the two control words must be read in the line in between (164). The sprite DMA then proceeds as follows:

Line	Data through the DMA channel
162	Second-last line of the 1st sprite through this channel
163	Last line of the 1st sprite
164	Control words of the 2nd sprite
165	First line of the 2nd sprite
166	Second line of the 2nd sprite

The following example displays the three-color sprite from our first example in two different positions on the screen:

```
Start:
                ;First sprite through this DMA channel at line 160 ($A0)
                ;Horizontal position: 180 ($B4)
dc.w $A05A,$A800 ;HSTART = $B4, VSTART = $A0, VSTOP = $A8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
                  ;Now comes the second sprite over this DMA channel
                  ;at line 176 ($B0), horizontal position 300 ($12C)
dc.w $B096,$B800    ;HSTART = $12C, VSTART = $B0, VSTOP = $B8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w 0,0            ;End of the sprite data list
```

Activating the sprites

After a correct data list has been constructed in the chip RAM and the desired colors have been written into the color table, the DMA controller must be told at what address the list is stored before the sprite DMA can be enabled. Each DMA channel has a register pair in which the starting address of the data list must be written:

SPRxPT register (SPRite x PoinTer, points to data list for sprite DMA channel x):

Reg.	Name	Function	
$120	SPR0PTH	Pointer to the sprite data list	Bits 16-20
$122	SPR0PTL	for sprite DMA channel 0	Bits 0-15
$124	SPR1PTH	Pointer to the sprite data list	Bits 16-20
$126	SPR1PTL	for sprite DMA channel 1	Bits 0-15
$128	SPR2PTH	Pointer to the sprite data list	Bits 16-20
$12A	SPR2PTL	for sprite DMA channel 2	Bits 0-15
$12C	SPR3PTH	Pointer to the sprite data list	Bits 16-20
$12E	SPR3PTL	for sprite DMA channel 3	Bits 0-15
$130	SPR4PTH	Pointer to the sprite data list	Bits 16-20
$132	SPR4PTL	for sprite DMA channel 4	Bits 0-15
$134	SPR5PTH	Pointer to the sprite data list	Bits 16-20
$136	SPR5PTL	for sprite DMA channel 5	Bits 0-15
$138	SPR6PTH	Pointer to the sprite data list	Bits 16-20
$13A	SPR6PTL	for sprite DMA channel 6	Bits 0-15
$13C	SPR7PTH	Pointer to the sprite data list	Bits 16-20
$13E	SPR7PTL	for sprite DMA channel 7	Bits 0-15

All SPRxPT registers are write-only

The DMA controller uses these registers as pointers to the current address in the sprite data lists. At the start of each picture they contain the address of the first control word. With each data word read they are incremented by one word so that at the end of the picture they point to the first word after the data list. For the same sprites to be displayed in each frame, these pointers must be set back to the start of the sprite data list before each frame. As with the bit-plane pointers BPLxPT, this is most easily done by the Copper in the vertical blanking gap. The pertinent section of the Copper list might look like this:

```
StartSpritexH = starting address of sprite data list for sprite x, bits 16-19
StartSpritexL = bits 0-15
CopperlistStart
MOVE #StartSprite0H,SPR0PTH      ;Initialize sprite DMA
MOVE #StartSprite0L,SPR0PTL      ;channel 0
```

```
MOVE #StartSprite1H,SPR1PTH      ;Initialize sprite DMA
MOVE #StartSprite1L,SPR1PTL      ;channel 1
MOVE #StartSprite2H,SPR2PTH      ;Initialize sprite DMA
MOVE #StartSprite2L,SPR2PTL      ;channel 2
... ... ...                      ;Same for channels 3 to 6
MOVE #StartSprite7H,SPR4PTH      ;Initialize sprite DMA
MOVE #StartSprite7L,SPR4PTL      ;channel 7
... ... ...                      ;Other Copper tasks
WAIT $FFFE                       ;End of Copper list
```

There is no way to turn the sprite DMA channels on and off individually. The SPREN bit (bit 5) in the DMACON register turns the sprite DMA on for all eight sprite channels. If you don't want to use all of them, the unused channels must process empty data lists. To do this, their SPRxPT's are set to two memory words with contents of zeros. The two zeros at the end of an existing data list can be used for this.

All eight SPRxPT's must always be initialized within the vertical blanking gap. Even if the data list is nothing but the two zeros, the DMA channel's SPRxPT points to the first word after them at the end of a frame.

Naturally, the SPRxPT can also be initialized by the processor in the vertical blanking interrupt.

As the last step, the sprite DMA must be enabled. As previously mentioned, this is done for all eight sprite DMA channels by using the SPREN bit in the DMACON register. The following MOVE command accomplishes this:

```
MOVE.W #$8220,$DFF096       ;Set SPREN and DMAEN in DMACON register
```

Moving sprites

The values of the two control words in the sprite data list determine the position of a sprite. To move a sprite, these values must be changed step by step.

This can be done directly by the processor when using the appropriate MOVE commands. The control words must be modified at the right time. Otherwise, the following problem can occur:

The processor modifies the first control word. Before it can change the second control word, the DMA controller reads both words. Since they no longer belong together, what appears on the screen may not make any sense.

The easiest way to avoid this is to change the control words only during the vertical blanking interrupt, after the Copper has initialized the SPRxPT).

The sprite/playfield priority

The priority of a playfield or sprite determines whether it appears in front of, behind, or between the other screen elements. The sprite with the highest priority appears in front of all other elements. Nothing can cover it. The priority of a sprite is determined by its number. The lower the number, the higher the priority. Also, sprite 0 has priority over all other sprites.

For the playfields, a control bit determines whether number 1 or 2 appears in front. But what is the priority of the sprites in reference to the playfields?

On the Amiga it is possible to position the playfields almost anywhere between the sprites. The sprites are always handled in pairs when it comes to setting the priority of playfield vs. sprites. The pair combinations are the same as those used for fifteen-color sprites, always an even-numbered sprite with its odd successor:

```
sprites 0 & 1, sprites 2 & 3, sprites 4 & 5, sprites 6 & 7
```

The four sprite pairs can be viewed as a stack of four elements. If you look at the stack from above, the underlying elements can only be seen through holes in the overlying ones. The holes correspond to the transparent points in the bit-planes or sprites and the parts of the screen that a sprite cannot cover because of its size. The order of elements in the stack cannot be changed.

But two of the elements, namely the playfields, can be placed anywhere between the four sprite pairs. Five positions are possible for each playfield:

Position	Order from front to back				
0	PLF	SPR0&1	SPR2&3	SPR4&5	SPR6&7
1	SPR0&1	PLF	SPR2&3	SPR4&5	SPR6&7
2	SPR0&1	SPR2&3	PLF	SPR4&5	SPR6&7
3	SPR0&1	SPR2&3	SPR4&5	PLF	SPR6&7
4	SPR0&1	SPR2&3	SPR4&5	SPR6&7	PLF

The BPLCON2 register contains the priority of the playfields with respect to the sprites:

BLPCON2 $104 (write-only)

Bit no.:	15-7	6	5	4	3	2	1	0
Function:	Gen.	PF2PRI	PF2P2	PF2P1	PF2P0	PF1P2	PF1P1	PF1P0

PF2PRI

If this bit is set, playfield 2 appears in front of playfield 1.

PF1P0 to PF1P2

These three bits form a 3-bit number that determines the position of playfield 1 (all odd bit-planes) between the four sprite pairs. Values from 0 to 4 are allowed (see previous table).

PF2P0 to PF2P2

These three bits have the same function as bits PF1P0 to PF1P2, but for playfield 2 (all even bit-planes).

Example:

```
BPLCON2 = $0003
```

This means that playfield 1 appears before playfield 2, PF2P0-2 = 0, PF1P0-2 = 3. This yields the following order, from front to back:

```
PLF2 SPR0&1 SPR2&3 SPR4&5 PLF1 SPR6&7
```

If we look closely, we see a paradox. The PF2PRI bit is 0, so playfield 1 should appear in front of playfield 2. The order previously shown contradicts this. The possible consequences of such a situation depend on which of the various elements are present at a given pixel location.

When one of the sprites 0 to 5 is present between playfields 1 and 2, its priority causes it to appear in front of playfield 1.

Since playfield 1 is in front of playfield 2, the sprite is visible at this point, even though it is actually behind playfield 2. In contrast, if only playfield 2 and the sprite are at a given position, playfield 2 covers the sprite.

This is because the playfield/playfield priority has precedence over the sprite/playfield priority.

If the dual-playfield mode is not used, there is only one playfield, which is formed from both the even and odd bit-planes. The PF2PRI and PL2P0-PL2P2 bits then have no function.

Collisions between graphic elements

It is often very useful to know whether two sprites have collided with each other or with the background. For example, in a game program this might indicate that a player had scored a hit.

When the pixels of two sprites overlap at a certain screen position (i.e., both have a non-transparent pixel at the same coordinates), this is treated as a collision between the two sprites. A collision of the playfields with each other or with a sprite is also possible.

Each recognized collision is noted in the collision data register, CLXDAT:

CLXDAT $00E (read-only)

Bit no.	Collision between
15	Unused
14	Sprite 4 (or 5) and sprite 6 (or 7)
13	Sprite 2 (or 3) and sprite 6 (or 7)
12	Sprite 2 (or 3) and sprite 4 (or 5)
11	Sprite 0 (or 1) and sprite 6 (or 7)
10	Sprite 0 (or 1) and sprite 4 (or 5)
9	Sprite 0 (or 1) and sprite 2 (or 3)
8	Playfield 2 (even bit-planes) and sprite 6 (or 7)
7	Playfield 2 (even bit-planes) and sprite 4 (or 5)
6	Playfield 2 (even bit-planes) and sprite 2 (or 3)
5	Playfield 2 (even bit-planes) and sprite 0 (or 1)
4	Playfield 1 (odd bit-planes) and sprite 6 (or 7)
3	Playfield 1 (odd bit-planes) and sprite 4 (or 5)
2	Playfield 1 (odd bit-planes) and sprite 2 (or 3)
1	Playfield 1 (odd bit-planes) and sprite 0 (or 1)
0	Playfield 1 and playfield 2

While on a sprite, any non-transparent pixel can cause a collision; we can specify which colors of the playfields are to be considered in collision detection. Moreover, it is possible to include or exclude any odd-numbered sprite from collision detection. All this can be set with the bits in the collision control register, CLXCON.

CLXCON $098 (write-only)

Bit no.	Name	Function
15	ENSP7	Enable collision detection for sprite 7
14	ENSP5	Enable collision detection for sprite 5
13	ENSP3	Enable collision detection for sprite 3
12	ENSP1	Enable collision detection for sprite 1
11	ENBP6	Compare bit-plane 6 with MVBP6
10	ENBP5	Compare bit-plane 5 with MVBP5
9	ENBP4	Compare bit-plane 4 with MVBP4
8	ENBP3	Compare bit-plane 3 with MVBP3
7	ENBP2	Compare bit-plane 2 with MVBP2
6	ENBP1	Compare bit-plane 1 with MVBP1
5	MVBP6	Value for collision with bit-plane 6
4	MVBP5	Value for collision with bit-plane 5
3	MVBP4	Value for collision with bit-plane 4
2	MVBP3	Value for collision with bit-plane 3
1	MVBP2	Value for collision with bit-plane 2
0	MVBP1	Value for collision with bit-plane 1

The ENSPx bits (ENable SPrite x) determine whether the corresponding odd-numbered sprite is regarded in collision detection. For example, if the ENSP1 bit is set, a collision between sprite 1 and another sprite or a playfield is registered. Such a collision sets the same bit in the collision data register as for sprite 0. Therefore, it is not possible to tell by looking at the register contents whether sprite 0 or sprite 1 caused the collision. Furthermore, collisions between sprites 0 and 1 are not detected. These facts should be kept in mind when selecting and using sprites.

If two sprites have been combined into one fifteen-color sprite, the appropriate ENSPx bit must be set in order to have correct collision detection.

For the playfields, the programmer can set which combinations of the bit-planes generate a collision and which do not. The ENBPx bits (ENable Bitplane x) determine which bit-planes are considered in collision detection. If all ENBPx bits of a playfield are set, a collision is possible at every pixel whose bit combination matches that of the MVBPx bits (Match Value Bitplane x).

The ENBPx bits determine whether the bits from plane x are compared with the value of MVBPx. If the bits of all planes for which ENBPx is set match the corresponding MVBPx bits for a given pixel, then this pixel can generate a collision.

Complicated? An example makes it clearer:

The ENBPx bits are set, as are all of the MVBPx bits. Now only those playfield pixels whose bit combinations are binary 111111 can generate a collision. If only the lower three MVBPx bits are set, then a collision is possible only if the pixel in the playfield has the combination 000111.

If a collision is to be allowed for all pixels with the bit combinations 000111, 000110, 000100 or 000101, the MVBP bits must be 000100. The lower two bits should always satisfy the collision condition, so the corresponding ENBPx bits are cleared. The ENBP value is 111100.

Examples for possible bit combinations:

ENBPx	MVBPx	Collision possible with bit pattern
111111	111111	111111
111111	111000	111000
111100	1111xx	111100, 111101, 111110, 111111
011111	x00000	000000, 100000
000000	xxxxxx	Collision possible with any bit pattern

The values of bits marked with an x are irrelevant. If not all six bit-planes are active, the ENBPx bits of the unused planes must be set to 0.

The various combinations of the ENBPx and MVBPx bits allow a variety of different collision detection strategies. For example, the CLXCON register can be set so that sprites can collide only with the red and green pixels of the playfield, but not with other colors. Or a collision may be possible only at the transparent pixels of playfield 1 if the underlying pixels of playfield 2 are black, etc.

Other sprite registers

Besides the SPRxPT registers, each sprite has four additional registers. They are normally supplied with data automatically by the DMA controller. However, it is also possible to access them through the processor.

SPRxPOS	First control word
SPRxCTL	Second control word
SPRxDATA	First data word of a line (low word)
SPRxDATB	Second data word of a line (high word)

Again, x stands for a sprite number from 0 to 7. The addresses of these registers can be found in the register overview.

The DMA controller writes the two control words of a sprite directly into the two registers SPRxPOS and SPRxCTL. When a value is written into the SPRxCTL register, whether by DMA or the 68030, Denise turns the sprite output off. The sprite will no longer be output to the screen.

The DMA controller now waits for the line specified in VSTART. Then it writes the first two data words into the SPRxDATA and SPRxDATB registers.

Now the sprite will be displayed, because writing to the SPRxDATA register causes Denise to enable the sprite output again. The desired horizontal position from the SPRxCTL and SPRxPOS registers is compared with the actual screen column, and the sprite is displayed at the correct location on the monitor.

The DMA controller writes two new data words in SPRxDATA/B in each line until the last line of the sprite (VSTOP) is past. Then it fetches the next control words and places them in SPRxPOS and SPRxCTL. This turns the sprite off again until the next VSTART position is reached. If both control words were zero, the DMA controller ends the sprite DMA for the corresponding channel until the start of the next frame. At the end of the vertical blanking gap, it starts again at the current address in SPRxPT.

Displaying sprites without DMA

A sprite can also be easily displayed without the DMA channel. You simply write the desired control words directly into the SPRxPOS and SPRxCTL registers.

Only the HSTART position and the AT bit have to contain valid values. VSTART and VSTOP are used only by the DMA channel.

You can begin the sprite output in any line by writing the two data words into the SPRxDATA and SPRxDATB registers. Since writing to SPRxDATA enables the sprite output, it is better to write to SPRxDATB first. If the contents of the two registers are not changed, they are displayed again in each line. The result is a vertical column.

To turn the sprite off again, simply write some value to SPRxPOS.

```
;*** Sprite Demo ***

;Customchip registers

INTENA  = $9A
INTREQR = $1e ;Interrupt request register (read)
DMACON  = $96 ;DMA control register (write)
COLOR00 = $180;Color palette register 0
VPOSR   = $4  ;Beam position (read)
JOY0DAT = $A  ;Mouse position for port 0

;Copper registers

COP1LC = $80  ;Address of 1st Copperlist
COP2LC = $84  ;Address of 2nd Copperlist
```

```
COPJMP1 = $88 ;Jump to Copperlist 1
COPJMP2 = $8a ;Jump to Copperlist 2

;Bitplane registers

BPLCON0 = $100 ;Bitplane control register 0
BPLCON1 = $102 ;1 (Scroll values)
BPLCON2 = $104 ;2 (Sprite<>playfield priority)
BPL1PTH = $0E0 ;Pointer to 1st bitplane
BPL1PTL = $0E2 ;
BPL1MOD = $108 ;Modulo value for odd bitplanes
BPL2MOD = $10A ;Module value for even bitplanes
DIWSTRT = $08E ;Start of screen window
DIWSTOP = $090 ;End of screen window
DDFSTRT = $092 ;Bitplane DMA start
DDFSTOP = $094 ;Bitplane DMA stop

;Sprite registers

SPR0PTH = $120 ;Pointer to sprite data list for sprite 1
SPR0PTL = $122
SPR1PTH = $124
SPR1PTL = $126


;CIA-A port register A (Mouse button)

CIAAPRA = $bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0
Forbid = -30-102
Permit = -30-108
AllocMem = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem = -30-180  ;MemoryBlock,ByteSize/a1,d0

;graphics base

StartList = 38

;Other labels

Execbase = 4
Planesize = 52*345 ;Size of bitplane
Planewidth = 52
CLsize = 19*4 ;Size of Copperlist in bytes
Chip = 2 ;Request chip RAM
```

```
Clear = Chip+$10000 ;Clear previous chip RAM

;*** Start program ***

Start:

;Request memory for bitplanes

        move.l Execbase,a6
        move.l #Planesize,d0   ;Memory requirement of planes
        move.l #clear,d1
        jsr    AllocMem(a6)       ;Request memory
        move.l d0,Planeadr
        beq        Ende              ;Error! -> End

;Request memory for Copperlist

        moveq  #Clsize,d0
        moveq  #chip,d1
        jsr    AllocMem(a6)
        move.l d0,CLadr
        beq    FreePlane          ;Error! -> FreePlane
;Request memory for sprite data list

        moveq  #Sprsize,d0
        moveq  #chip,d1
        jsr    AllocMem(a6)
        move.l d0,Spradr
        beq    FreeCL
;Set up Copperlist in chip RAM
;Bitplanepointer
        move.l CLadr,a0
        move.w #bpl1ptl,d2
        move.l Planeadr,d1
        bsr    setadr

;Pointer to 1st sprite
        move.w #spr0ptl,d2
        move.l Spradr,d1
        bsr    setadr
;Remaining (unused) sprite pointers
        moveq  #6,d0
        move.w #spr1ptl,d3
spr_set:
        move.l Spradr+Sprsize-4,d1
        move.w d3,d2
        bsr    setadr
        addq.w #4,d3
```

```
        dbf     d0,spr_set

        move.l #$fffffffe,(a0)

;Copy sprite data list

        move.w #Sprsize/4-1,d0
        lea    Sprstart,a0
        move.l Spradr,a1
spr_copy:
        move.l (a0)+,(a1)+
        dbf    d0,spr_copy

;*** Main program ***

;Disable DMA and task-switching

        jsr    forbid(a6)
        lea    $dff000,a5
        move.w #$0300,dmacon(a5)

;Initialize Copper

        move.l CLadr,cop1lc(a5)
        clr.w  copjmp1(a5)

;Initialize playfield

        move.w #0,color00(a5)           ;Playfield colors
        move.w #$0f00,color00+2(a5)
        move.w #$000f,color00+34(a5)    ;Sprite colors
        move.w #$00ff,color00+36(a5)
        move.w #$00f0,color00+38(a5)
        move.w #$1a64,diwstrt(a5) ;26,100
        move.w #$39d1,diwstop(a5) ;313,465
        move.w #$0028,ddfstrt(a5)
        move.w #$00d8,ddfstop(a5)
        move.w #%0001001000000000,bplcon0(a5)
        clr.w  bplcon1(a5)
        move.w #8,bplcon2(a5)
        move.w #2,bpl1mod(a5)

;DMA on
        move.w #$83a0,dmacon(a5)        ;Bitplane & sprite DMA on

;Fill bitplanes with checkerboard pattern

        move.l planeadr,a0
```

```
        move.w #planesize/4-1,d0  ;Loop counter
        move.w #13*16,d1
        move.l #$ffff0000,d2      ;Checkerboard pattern
        move.w d1,d3

fill:   move.l d2,(a0)+
        subq.w #1,d3
        bne.s  continue
        swap   d2                 ;Change pattern
        move.w d1,d3
continue: dbf  d0,fill

;Wait for raster line 16 (after Exec-interrupts)

wait:   btst   #6,ciaapra    ;Mouse button pressed?
        beq.s  endit
        move.l vposr(a5),d2
        and.l  #$0001FF00,d2
        cmp.l  #$00001000,d2
        bne.S  wait

;Move sprite

        move.w joy0dat(a5),d0 ;Mouse position
        move.w d0,d1
        and.w  #$ff,d0
        lsr.w  #8,d1
        add.w  #150,d0 ;Add offset for null position,  so
        add.w  #30,d1 ;sprite always remains visible
        jsr    setcor ;Display sprite at position in d0,d1

        bra.S  wait          ;No -> continue

;*** End program ***

;Reactivate old Copperlist

endit:  move.l #GRname,a1    ;Set parameters for OpenLibrary
        clr.l  d0
        jsr    OpenLibrary(a6) ;Open Graphics library
        move.l d0,a4
        move.l StartList(a4),cop1lc(a5)
        clr.w  copjmp1(a5)
        move.w #$83a0,dmacon(a5)
        jsr    permit(a6)

;Release memory for sprite data
```

```
        move.l Spradr,a1
        moveq  #Sprsize,d0
        jsr    FreeMem(a6)

;Release memory for Copperlist
FreeCL:
        move.l CLadr,a1    ;Set parameters for FreeMem
        moveq  #CLsize,d0
        jsr    FreeMem(a6)

;Release memory for bitplanes
FreePlane:
        move.l Planeadr,a1
        move.l #Planesize,d0
        jsr    FreeMem(a6)

Ende:
        clr.l  d0
        rts                  ;End program

;Subprograms

;setadr writes the Copper commands for initializing a DMA address counter
;in the Copperlist
;a0 - pointer to Copperlist (incremented by setadr)
;d1 - to written address (e.g. bitplane)
;d2 - address of pointer register, low  (e.g. bpl1ptl)

setadr:
        move.w d2,(a0)+ ;move ptl
        move.w d1,(a0)+ ;addr bits 1-15
        swap   d1
        subq.w #2,d2    ;switch to pth
        move.w d2,(a0)+ ;move pth
        move.w d1,(a0)+ ;addr bits 16-18
        rts

;setcor writes the X,Y coordinates of the sprite to the sprite data list
;in the chip RAM
;d0,d1 - X,Y coordinates
;Address of sprite data list: Spradr
;Height of sprite in lines: Sprhigh
;a0,d2,d3 are used internally

setcor:
        movem.l d0-d3/a0,-(sp)
        move.w  d0,d3  ;H0 bit to second control word
        and.w   #1,d3  ;Clear rest
```

```
        lsr.w   #1,d0   ;H1-H8 to position
        move.w  d0,d2   ;in first controlword
        and.w   #$ff,d2  ;Clear E0-E7

        move.w  d1,d0
        add.l   #Sprhigh,d0 ;Last line of sprite to d0
        asl.w   #8,d1
        bcc     noE8
        bset    #2,d3   ;Set E8 in second word
noE8:   or.w    d1,d2   ;E0-E7 to first word
        asl.w   #8,d0
        bcc     noL8  ;Set L8
        bset    #1,d3
noL8:   or.w    d0,d3   ;L0-L7 to second word
        move.l  Spradr,a0 ;Transfer new value to memory
        move.w  d2,(a0)+
        move.w  d3,(a0)

        movem.l (sp)+,d0-d3/a0
        rts

;Variables

CLadr:     dc.l 0
Planeadr: dc.l 0
Spradr:   dc.l 0
test:     dc.l 0

;Constants

GRname: dc.b "graphics.library",0

;Sprite data list
        align ;even
Sprstart:
        dc.w $a05a,$a800
        dc.w %0000000000000000,%0000000111100000
        dc.w %0000000000000000,%0000011000011000
        dc.w %0000000110000000,%0000100011000100
        dc.w %0000001111000000,%0001100100100110
        dc.w %0000001111000000,%0001100100100110
        dc.w %0000000110000000,%0000100011000100
        dc.w %0000000000000000,%0000011000011000
        dc.w %0000000000000000,%0000000111100000
Sproff: dc.w 0,0
Sprend:

Sprsize = Sprend-Sprstart
```

```
Sprhigh = 9
        end
```

11.7.7 ECS Capabilities

The features previously described (with the exception of the 2 Meg chip RAM) were already present in the custom chips of the A1000. But in the course of the A3000's development, an improved chip set, called the Enhanced Chip Set (ECS), was also developed. To ensure software-compatibility, the developers did not change anything in the programming of previously existing modes. However, some new registers have been added for utilizing the additional capabilities of the new chips:

Super-HiRes mode

In this mode the horizontal resolution is doubled from 640 to 1280 pixels/line.

Freely programmable screen display

The geometry of the generated video image can be freely programmed by selecting, not only the number of pixels per line, but also the number of lines and the video frequency. With Super-HiRes mode, a flicker-free picture can be produced even at a resolution of 640 x 512 pixels.

Larger bit-planes

The Blitter now supports bit-planes up to 32768 x 32768 pixels in size.

Expanded genlock capabilities

Through a modified "chromakey" scheme, every video register can control the video overlay.

Super-HiRes mode

Previously there were two possibilities for horizontal resolution: HiRes (high resolution) and LoRes (low resolution). In LoRes mode a pixel had a duration of 140 nanoseconds (ns), in HiRes mode 70 ns. The new Super-HiRes mode gives double the resolution of HiRes mode, with a duration of 35 ns per pixel. To achieve this, Agnus must read twice as much data per bit-plane from the chip RAM. The maximum possible number of bit-planes has been halved to only two, representing a maximum of four colors in the Super-HiRes mode.

Unfortunately, the entire palette of 4096 colors is not accessible in the Super-HiRes mode. Only 64 colors are possible, with two bit each for the red, green and blue components. The programming of the color registers for this mode follows a rather complicated scheme, which is illustrated below.

	R	G	B
Bit-plane (color 0):	ab--	cd--	ef--
Bit-plane (color 1):	gh--	ij--	kl--
Bit-plane (color 2):	mn--	op--	qr--
Bit-plane (color 3):	st--	uv--	wx--

	BIT	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
C	00	.	.	.	.	A	B	A	B	C	D	C	D	E	F	E	F
O	01	.	.	.	.	G	H	A	B	I	J	C	D	K	L	E	F
L	02	.	.	.	.	M	N	A	B	O	P	C	D	Q	R	E	F
O	03	.	.	.	.	S	T	A	B	U	V	C	D	W	X	E	F
R	04	.	.	.	.	A	B	G	H	C	D	I	J	E	F	K	L
	05	.	.	.	.	G	H	G	H	I	J	I	J	K	L	K	L
R	06	.	.	.	.	M	N	G	H	O	P	I	J	Q	R	K	L
E	07	.	.	.	.	S	T	G	H	U	V	I	J	W	X	K	L
G	08	.	.	.	.	A	B	M	N	C	D	O	P	E	F	Q	R
I	09	.	.	.	.	G	H	M	N	I	J	O	P	K	L	Q	R
S	0A	.	.	.	.	M	N	M	N	O	P	O	P	Q	R	Q	R
T	0B	.	.	.	.	S	T	M	N	U	V	O	P	W	X	Q	R
E	0C	.	.	.	.	A	B	S	T	C	D	U	V	E	F	W	X
R	0D	.	.	.	.	G	H	S	T	I	J	U	V	K	L	W	X
	0E	.	.	.	.	M	N	S	T	O	P	U	V	Q	R	W	X
	0F	.	.	.	.	S	T	S	T	U	V	U	V	W	X	W	X

The following shows the color selection scheme for sprites, which are subject to the same limitations as the playfields in Super-HiRes mode.

Color selection for Super-HiRes sprites

	R	G	B
Sprite (color 16) :	AB--	CD--	EF--
Sprite (color 17) :	GH--	IJ--	KL--
Sprite (color 18) :	MN--	OP--	QR--
Sprite (color 19) :	ST--	UV--	WX--

	BIT	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
C	10	.	.	.	.	A	B	A	B	C	D	C	D	E	F	E	F
O	11	.	.	.	.	G	H	A	B	I	J	C	D	K	L	E	F
L	12	.	.	.	.	M	N	A	B	O	P	C	D	Q	R	E	F
O	13	.	.	.	.	S	T	A	B	U	V	C	D	W	X	E	F
R	14	.	.	.	.	A	B	G	H	C	D	I	J	E	F	K	L
	15	.	.	.	.	G	H	G	H	I	J	I	J	K	L	K	L
R	16	.	.	.	.	M	N	G	H	O	P	I	J	Q	R	K	L
E	17	.	.	.	.	S	T	G	H	U	V	I	J	W	X	K	L
G	18	.	.	.	.	A	B	M	N	C	D	O	P	E	F	Q	R
I	19	.	.	.	.	G	H	M	N	I	J	O	P	K	L	Q	R
S	1A	.	.	.	.	M	N	M	N	O	P	O	P	Q	R	Q	R
T	1B	.	.	.	.	S	T	M	N	U	V	O	P	W	X	Q	R
E	1C	.	.	.	.	A	B	S	T	C	D	U	V	E	F	W	X
R	1D	.	.	.	.	G	H	S	T	I	J	U	V	K	L	W	X
	1E	.	.	.	.	M	N	S	T	O	P	U	V	Q	R	W	X
	1F	.	.	.	.	S	T	S	T	U	V	U	V	W	X	W	X

Although the Super-HiRes mode allows fewer colors than lower resolutions, it does enable a more precise positioning of sprites. An additional bit for the horizontal position in the second control word of the sprite data list (bit 4) allows positioning of sprites at a resolution of 70 nanoseconds (i.e., two Super-HiRes pixels).

The Super-HiRes mode is enabled with bit 6 of the first bit-plane control register, BPLCON0. The bit for normal HiRes mode (bit 15) must also be cleared.

Programmable geometry of the video image

Previously, a computer conformed to either the PAL or the NTSC video standard, and the geometry of the computer's video image (i.e., the number of lines on a screen and the number of pixels in a line) was fixed accordingly. With the Agnus chip of the A3000 the geometry of the image can be freely programmed. You can switch between the two standards or, with the new Super-HiRes mode, even create new formats. For example, under Kickstart 2.0, there is a productivity mode capable of producing a non-interlaced display of 640 x 480 pixels (computed without overscan).

Some new registers were introduced to achieve this flexibility:

HTOTAL $1C0 (write-only) Number of cycles per line

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function	-----	-----	-----	h8	h7	h6	h5	h4	h3	h2	h1					

The duration of each line is the number of clock cycles in HTOTAL + 1, the clock being the color clock CCK (3.54 MHz, with a 280 ns period). For a normal Pal image, this number would be 227.5, for the productivity mode 114.

The number of lines is placed in the VTOTAL register.

VTOTAL $1C8 (write-only) Highest line displayed

There are a total of VTOTAL + 1 lines displayed per image.

The previous two registers are used to establish the geometry of the video image. The following new registers determine the exact timing of the creation of the horizontal and vertical synchronization signals and of the blanking signals:

$1E0	VSSTART	Starting line of vertical sync signal
$1CA	VSSTOP	Ending line of vertical sync signal
$1DE	HSSTART	Starting column of horizontal sync signal
$1C2	HSSTOP	Ending column of horizontal sync signal
$1E2	HCENTER	Starting column of vertical sync in interlace mode
$1C4	HBSTART	Starting column of horizontal blanking signal (HBLANK)
$1C6	HBSTOP	Ending column of horizontal blanking signal (HBLANK)
$1CC	VBSTART	Starting line of vertical blanking signal (VBLANK)
$1CE	VBSTOP	Ending line of vertical blanking signal (VBLANK)

In connection with all the new registers, there is an additional register, BEAMCON0, which indicates how they are to be used:

BEAMCON0 $1DC (write-only)

Bit	Name	Function
15	---	
14	HARDDIS	Disable normal blanking signal
13	LPENDIS	Disable lightpen
12	VARVBEN	Activate VBSTART/STOP registers
11	LOLDIS	Disable 227/228 cycle/line switching
10	CSCBEN	Enable composite-sync bypass
9	VARVSYEN	Enable variable V-sync
8	VARHSYEN	Enable variable H-sync
7	VARBEAM	Activate HTOTAL/VTOTAL registers
6	DUAL	Special Ultra-HiRes mode (not implemented)
5	PAL	Switch Agnus to PAL
4	VARCSYNC	Enable variable composite-sync
3	BLANKEN	Output blanking signal <!to/on?!> composite-sync pin
2	CSYTRUE	Composite sync active-high
1	VSYTRUE	Vertical sync active-high
0	HSYTRUE	Horizontal sync active-high

A further change in the ECS chip set affects the definition of the screen window. With the new DIWHIGH register, the window can now be changed. The DIWHIGH register is activated by setting it after writing the desired values to the old DIWSTRT and DIWSTOP registers:

DIWHIGH $1E4 (write only)

Bit	Name	Function
15	---	
14	---	
13	H8	Horizontal stop, high-value bit
12	---	
11	---	
10	V10	
9	V9	Vertical stop, three high-value bits
8	V8	
7	---	
6	---	
5	H8	Horizontal start, high-value bit
4	---	
3	---	
2	V10	
19	V9	Vertical start, three high-value bits
0	V8	

11.7.8 The Blitter

What is a Blitter? The name Blitter stands for "block image transferor." This is the main task of the Blitter: moving and copying data blocks in memory; this usually involves graphics data. The Blitter can also perform logical operations on multiple memory areas and write the result back into memory. It accomplishes these tasks very quickly. Simple data moves proceed at speeds of up to 16 million pixels per second.

In addition, the Blitter can fill surfaces and draw lines. The combination of these two capabilities enables the drawing of any type of filled polygon.

The operating system uses the Blitter for almost all graphic operations. It handles the text output, draws gadgets, moves windows, etc. In addition, it is used to decode data from the diskette, which shows that the many-faceted capabilities of the Blitter are not limited to graphics.

Using the Blitter to copy data

The Blitter always follows the same procedure when copying data: One to three memory areas and the data sources are combined together using the selected logical operation and the result is written back into memory. The spectrum ranges from simple copying to complex combinations of multiple data areas. The addresses of the source data areas, named A, B and C, and the destination area D can be anywhere in the chip RAM (from 0 to $1FFFFF).

The Blitter supports "rectangular memory areas." The memory, like a bit-map, is divided into columns and rows. It is also possible to process small areas inside a large bit-map by using what are called modulo values. You may recall that such modulo values are also used in playfields, to define bit-planes that are wider than the screen window.

The following steps are necessary to start a Blitter operation:

- Select the Blitter mode: Copy data.
- Select the source data areas (not all three sources have to be used) and the destination area.
- Select the logical operation.

- Define other operating parameters (scrolling, masking, address direction).
- Define the window in which the Blitter operation is to take place and start the Blitter.

Defining the Blitter window

You may wonder why we're starting with a discussion of the last step. Actually, the definition of the desired window is the basis of all the other settings. But when the Blitter is programmed, this value is not written to the appropriate register until the end, because that is what starts the Blitter. For that reason, this point also appears last in the previous list. However, you must understand the Blitter window concept in order to understand the other values.

The Blitter window is the area of memory that is to be processed by the Blitter operation. It is constructed like a bit-plane (i.e., divided into rows (lines) and columns) where a column corresponds to one word (two bytes). The number of words in the window is equal to the product of the rows and columns: R*C.

Since the desired memory area is divided into rows and columns, the Blitter is very well suited for processing bit-planes.

However, linear memory areas can also be accessed. The division into rows and columns simply makes the programming easier. Actually, the individual lines reside at contiguous addresses in memory. For small data fields that are not divided into rows and columns, it is also possible to set the window width or height to 1.

The Blitter processes the Blitter window line by line. The Blitter operation begins with the first word of the first line and ends with the last word of the last line. The BLTSIZE register contains the window size:

BLTSIZE $058 (write-only)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	H9	H8	H7	H6	H5	H4	H3	H2	H1	H0	W5	W4	W3	W2	W1	W0

H0-H9 These ten bits represent the height of the Blitter window in lines. The window can have a height between 1 and 1024 lines (2^{10} = 1024). A height of 1024 lines is selected by setting the height value to 0. For all other values the height corresponds directly to the number of lines. A height of 0 lines is not possible.

W0 These six bits represent the width of the window. The width can vary between 1 and

-W5 64 words (2^6 = 64). In terms of graphic pixels, this can be up to 1024 pixels. As with the height, the maximum width is set by making the width value = 0.

The following formula is applied to the height and width to derive the necessary BLTSIZE value: BLTSIZE = Height*64 + Width.

It must be modified somewhat when using the two extremes (Height = 1024 and Width = 64):

```
BLTSIZE = (Height AND $3FF)*64 + (Width AND $3F)
```

The BLTSIZE register should always be the last register initialized. The Blitter is automatically started when a value is written to BLTSIZE.

The Blitter can also process larger windows with the built-in ECS chips of the A3000. For this, the two new registers BLITSIZV and BLITSIZH are used:

BLITSIZV	$5C	Number of lines in Blitter window (15 bits)
BLITSIZH	$5E	Number of words per line (11 bits), start Blitter

Since BLITSIZH starts the Blitter, BLITSIZV must be written first.

Source and destination data areas

During a Blitter operation, data are combined together from completely different areas of memory. Even though the Blitter window defines the number and organization of data words to be processed, the positioning of this window within the three source areas and the destination area must still be specified.

For example, suppose that you want the Blitter to copy a small rectangular graphic, stored somewhere in chip RAM, into the screen memory. For this simple task there is only one source area. The selection of the Blitter window is easy. The entire graphic is to be copied, so the width and height of the Blitter window correspond to that of the graphic in memory.

So that the Blitter also knows where this graphic can be found, you write the address of the first word of the top line into the appropriate register.

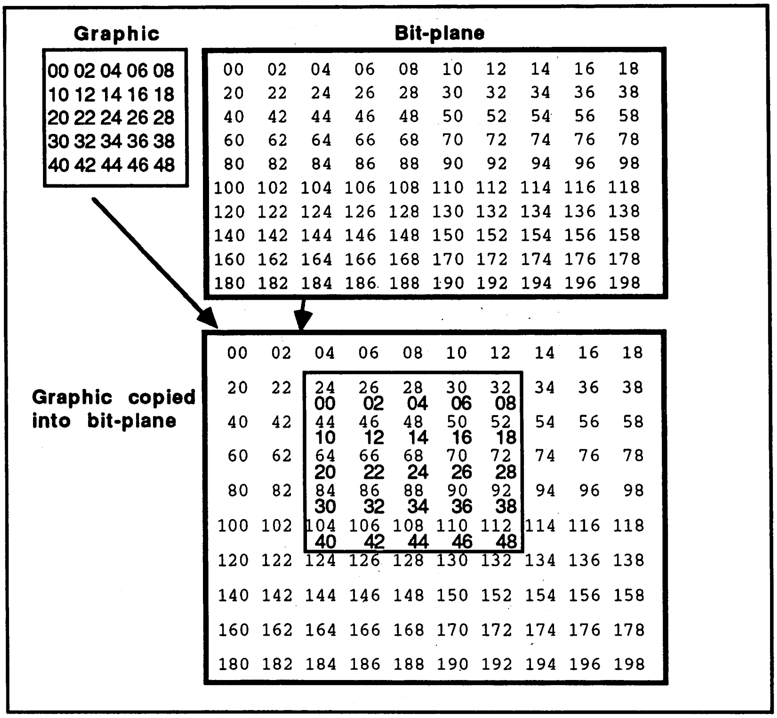
But how is the destination area defined? The graphic is to be copied into the screen memory, which means that it must be transferred into the current bit-plane. (For the sake of simplicity, the graphic and the screen memory are each assumed to consist of a single bit-plane.) But the bit-plane is wider than the small graphic. If the Blitter were to copy the graphic directly into the bit-plane, the result would not appear as desired.

In addition to the address of the destination area, the Blitter must also know its width. This information is communicated by a modulo value. The modulo value is added to the address pointer after each line of the Blitter window is processed. The words that are not affected are skipped and the pointer indicates the start of the next line. The source and destination areas have independent modulo registers so that they can have different widths.

The following figure illustrates our example. The graphic consists of five lines, each ten words wide. The numbers represent the corresponding word addresses relative to the initial address of the graphic. The bit-plane has dimensions of ten lines by twenty words. How do we choose the Blitter window, starting addresses and modulo values?

The Blitter window must correspond to the graphic, since the latter is to be copied completely. The height of the window is five lines and the width is ten words. The value that must be written to the BLTSIZE register is 330 (5*64 + 10) or hexadecimal $014A.

The starting address of the source data is equal to the address of the first word of the graphic. Since the line width of the graphic is equal to the line width of the Blitter window, the modulo value for the source is 0.



Plane copy principle

The modulo value must now be calculated for the destination area. To do this, simply take the difference between the actual line width and that of the Blitter window.

In our example, this is 20 words minus 10 words: The modulo value for the destination area is 10 words. Modulo values must be specified in bytes in the Blitter modulo registers. Modulo value = modulo in words * 2.

Finally, the Blitter needs the starting address of the destination data. This determines the bit-plane position to which the graphic is copied, and is equal to the starting address of the bit-plane, and the address of the word at which the upper left corner of the graphic is to be placed. In our figure this is the address of the bit-plane and 24.

How does the Blitter operation proceed?

After the addresses and modulo values have been defined and the BLTSIZE initialized, the Blitter begins copying the data. It fetches the word at the starting address of the source data and stores it at the destination address. Then it adds one word to both addresses and copies the next word.

This is repeated until the number of words per line set in BLTSIZE have been processed. Before the Blitter continues with the next line, it adds the modulo values to the address pointers so that the next line starts at the right address.

After all lines have been copied, the Blitter turns off and waits for its next job. After a Blitter operation, the address registers contain the address of the last word, 2, and the modulo value.

The address registers are called BLTxPT, where x represents one of the three sources A, B, C or the destination area D. Like other address registers, they occur in pairs, with one for bits 0-15 and one for bits 16-20:

Reg.	Name	Function
048	BLTCPTH	Starting address of Bits 16-20
04A	BLTCPTL	source data area C Bits 0-15
04C	BLTBPTH	Starting address of Bits 16-20
04E	BLTBPTL	source data area B Bits 0-15
050	BLTAPTH	Starting address of Bits 16-20
052	BLTAPTL	source data area A Bits 0-15
054	BLTDPTH	Starting address of Bits 16-20
056	BLTDPTL	destination data area D Bits 0-15

Each of the four areas has its own modulo register:

060	BLTCMOD	Modulo value for source C
062	BLTBMOD	Modulo value for source B
064	BLTAMOD	Modulo value for source A
066	BLTDMOD	Modulo value for destination D

Copying with ascending or descending addresses

In our example the Blitter worked with ascending addresses (i.e., it started at the starting address and incremented until reaching the ending address). The ending address is logically higher than the starting address.

However, there is a case in which such addressing leads to errors: the copying of a memory area to a higher address, where the source and destination areas partially overlap. Here is an example:

Result: Address	Source data	Destination data	Desired	Actual
0	Source1			
2	Source2			
4	Source3			
6	Source4	Dest1	Source1	Source1
8	Source5	Dest2	Source2	Source2
10		Dest3	Source3	Source3
12		Dest4	Source4	!Source1!
14		Dest5	Source5	!Source2!

The five source data words are to be written to the address of the destination data. If the Blitter begins by copying Source1 to the desired destination address (Dest1), it overwrites Source4 before the data there can be copied. This is because Source4 and Dest1 have the same address (the two areas overlap). The same thing happens with Source5 and Dest2.

When the Blitter reaches the address of Source4, it finds Source1 instead. Source1 (not Source4) ends up in Dest4, and Source2 (not Source5) ends up in Dest5. Source4 and Source5 are lost.

To solve this problem, the Blitter has a descending address mode and the ascending mode.

In this mode it starts at the addresses in BLTxPT and decrements these values by 2 bytes after each word is copied. Also, the modulo value is subtracted instead of added. The ending address lies before the starting address.

This must naturally be considered when initializing the BLTxPT's. Normally these are set to the upper left corner of the Blitter window in the given data area (A, B, C or D). In descending mode the addressing is backwards. Correspondingly, BPLxPT must point to the lower right corner.

The modulo and BLTSIZE values are identical to those for the ascending mode.

In general, the following statements can be made regarding mode selection:

1. No overlap between source and destination areas:

   Either ascending or descending mode; both work correctly in this case.

2. Source and destination areas overlap partially, and the destination is before the source:

   Only ascending mode works correctly.

3. Source and destination areas overlap partially, and the destination is after the source (see example):

   Only descending mode works correctly.

Selecting the logical operations

As previously mentioned, there are three source data areas associated with the destination area. The logical operations are always performed on a bit basis so that the destination bit D must be obtained from the data bits A, B and C.

The Blitter recognizes 256 different operations. These take place in two steps:

1. Eight different boolean equations are applied to the three source data bits. Each of these yields a 1 from a different combination of A, B and C.
2. The eight results of the previous equations are selectively combined with a logical OR. The result is the destination bit D.

The term "boolean equation" refers to a mathematical expression representing a combination of logical operations. This type of computation is called boolean algebra, after the English mathematician George Boole (1815 to 1864). The explanations of the logical functions of the Blitter can be understood without a knowledge of boolean algebra, but the boolean equations are nevertheless included.

There are eight possible combinations of three bits. Each of the eight equations is true for one of them (its result is 1). By using the eight control bits LF0 to LF7 you can select whether the result of the equation has any effect on the formation of D. All result bits whose corresponding

LFx bit is 1 are combined with a logical OR function. An OR function means that the result will be 1 if at least one of the input bits is 1. In other words, a logical OR returns a 0 only if all inputs are 0.

With the eight LFx bits you can choose which combinations of the three input bits A, B and C will cause the output bit D to be 1. The term for the eight boolean input equations is "minterm." The following table gives an overview of the input combinations for each LFx bit.

In the Minterm row, a lowercase letter represents a logical inversion of the corresponding input bit. Normally this is indicated with a bar over the letter.

The Input bits row contains the bit combination for which the corresponding equation is true. The order of the bits is A B C.

	LF7	LF6	LF5	LF4	LF3	LF2	LF1	LF0
Minterm:	ABC	ABc	AbC	Abc	aBC	aBc	abC	abc
Input bits:	111	110	101	100	011	010	001	000

Selecting the individual minterms is easy. For each input combination for which the output bit D should be 1, set the corresponding LFx bit.

In our first example we simply copy the source data from A directly to D. The B and C sources are not used. Which minterms must be selected for this?

D can be 1 only when A = 1. Only the upper four terms LF4 to LF7 come into play, since A = 1 only for these terms. Since B does not play a role, we choose a term in which B is 1 and a term in which B is 0, but which are otherwise identical.

Now B has no effect on D because the remainder of the equation is unchanged for both values of B and its result depends only on this remainder. The same holds true for C. If we look at the table of input combinations, we see that LF4 to LF7 must be activated. Then the result depends only on A, since for any combination of B and C, one of these four equations is always true for A = 1, and D is 1. If A = 0, all four are false and D = 0.

If you're familiar with boolean algebra, you can obtain the appropriate minterms yourself. The required expression is A = D. Since B and C are always present in the Blitter, they must be integrated into the equation as well:

```
A*(b+B)*(c+C)=D
```

The term x+X is always true (equal to 1) and is used when the result D is independent of the value of X. To get the required minterms simply multiply it out:

```
1.  A*(b+B)*(c+C)=D
2.  (A*b+A*B)*(c+C)=D
3.  A*b*c+A*B*c+A*b*C+A*B*C=D
```

or without the AND operators:

```
Abc+ABc+AbC+ABC=D
```

Now we only need to set the LFx bits of the corresponding minterms. Boolean algebra has helped us to arrive at our goal. Here are some examples of common Blitter operations and the corresponding LFx bit settings:

- *Invert a data area: a = D.*

  Required LFx combination: 00001111.

  Boolean algebra:
  a = D
  a*(b+B)*(c+C) = D
  (ab+aB)*(c+C) = D
  abc+aBc+abC+aBC = D

- *Copy a graphic to a bit-plane without changing the bit-plane's contents. This corresponds to logically ORing the graphic A and the bit-plane B: A + B = D.*

  Required LFx combination: 11111100.

  Boolean algebra:
  A + B = D
  A(b+B)(c+C)+B(a+A)(c+C) = D
  (Ab+AB)(c+C)+(Ba+BA)(c+C) = D
  Abc+ABc+AbC+ABC+Bac+BAc+BaC+BAC = D

Abc+ABc+AbC+ABC+aBc+aBC = D

Here are the rules for determining the LFx bits needed:

1. Determine which of the eight ABC combinations should cause D to be 1.
2. Set the LFx bits for these combinations.
3. If all three sources aren't needed, you must select all combinations in which the unused bits occur and in which the desired bits have the proper value.

Shifting the input values

For some tasks the Blitter's limitation to word boundaries can cause trouble. For example, you may want to shift a certain area within a bitmap by a few bits (i.e., by only a portion of a word). Or perhaps you want the Blitter to write a graphic at specific screen coordinates that don't match a word boundary.

In order to handle this problem, the Blitter has the capability to shift the data words from sources A and B to the right by up to 15 bits. This allows it to move the data to any desired bit position. All bits that are pushed out to the right by the shift operation move into the free bits in the next word. The entire line is shifted bit by bit. A device called a barrel shifter is used inside the Blitter to shift the words.

It requires no additional time for the shift operation, regardless of how many bits are moved. Adding a shift of the data does not limit the Blitter's speed in any way.

Example for shifting data by three bits:

Before:

Data word 1	Data word 2	Data word 3
00011111 10011100	00010101 01111111	11100001 11100101

After:

Data word 1	Data word 2	Data word 3
xxx00011 11110011	10000010 10101111	11111100 00111100

The three xxx bits depend on the previous data word, from which they are shifted out.

Masking

It is possible to use the Blitter to copy a graphic whose borders are not on word boundaries from screen memory. Data that is to the left of the graphic but within the first data word should not be copied along with the graphic itself. To make this possible, the Blitter can apply a mask to the first and last data words of a line. This means that you can choose which bits of these words should be copied. Undesired data can be erased from the edges of the line.

Only source A can be masked in this manner. Two registers contain the masks for the two edges. A bit is copied in the Blitter operation only if it is set in the mask register. All others are cleared.

$044 BLTAFWM BLiTter source A First Word Mask

Mask for the first data word in the line.

$046 BLTALWM BLiTter source A Last Word Mask

Mask for the last data word in the line.

Bits 0-15 contain the corresponding mask bits. For example, "1" represents a set bit, "." for a cleared bit:

Graphic data in the bit-plane:

```
Column 1            Column 2            Column 3
........11111111    1111111111111111    1.............11
111111......1111    11..........1111    1111........1111
....11........11    1111.........111    11111....1111111
....11.........1    11111.........11    1111111111111111
....11.........1    11111.........11    1111111111111111
....11........11    1111.........111    11111....1111111
111111......1111    11..........1111    1111........1111
........11111111    1111111111111111    1.............11

FirstWordMask:      LastWordMask:
0000000011111111    1111110000000000
```

Result:

```
Column 1            Column 2            Column 3
........11111111    1111111111111111    1...............
............1111    11..........1111    1111............
..............11    1111.........111    11111...........
...............1    11111.........11    111111..........
...............1    11111.........11    111111..........
..............11    1111.........111    11111...........
............1111    11..........1111    1111............
........11111111    1111111111111111    1...............
```

By masking out the unwanted picture elements at the edges, you get the desired graphic.

Note: When the width of the Blitter window is only one word (BLTSIZE width = 1) both masks come together. They both operate on the same input word. Only the input bits whose mask bits are set in both masks are allowed through.

The Blitter control registers

The Blitter has two control registers, BLTCON0 and BLITCON1. The following Blitter control bits are found in these two registers:

BLTCON0 $040

Bit no.	Name	Function
15	ASH3	ASH0-3 contain the shift distance
14	ASH2	for the input data from source A
13	ASH1	ASH0-3 = 0 means no shift
12	ASH0	
11	USEA	Enables the DMA channel for source A
10	USEB	Enables the DMA channel for source B
9	USEC	Enables the DMA channel for source C
8	USED	Enables the DMA channel for destination D
7	LF7	Selects minterm ABC (bit comb. of ABC: 111)
6	LF6	Selects minterm ABc (bit comb. of ABC: 110)
5	LF5	Selects minterm AbC (bit comb. of ABC: 101)
4	LF4	Selects minterm Abc (bit comb. of ABC: 100)
3	LF3	Selects minterm aBC (bit comb. of ABC: 011)
2	LF2	Selects minterm aBc (bit comb. of ABC: 010)
1	LF1	Selects minterm abC (bit comb. of ABC: 001)
0	LF0	Selects minterm abc (bit comb. of ABC: 000)

BLTCON1 $042

Bit no.	Name	Function
15	BSH3	BSH0-3 contain the shift distance
14	BSH2	for the input data from source B
13	BSH1	BSH0-3 = 0 means no shift
12	BSH0	
1-5		Unused
4	EFE	Exclusive Fill Enable
3	IFE	Inclusive Fill Enable
2	FCI	Fill Carry In
1	DESC	DESC = 1 switches to descending mode
0	LINE	LINE = 1 activates the line mode

The LINE bit switches the Blitter into its line-drawing mode. If you want to copy data with the Blitter, LINE must be 0.

Ascending or descending addresses can be selected with the DESC bit. If DESC = 0, the Blitter works in ascending mode; if DESC = 1, it works in descending mode.

The EFE and IFE bits activate the surface-filling mode of the Blitter. They must both be 0 for the Blitter to operate in the normal mode. The FCI bit is used only in the fill mode.

The Blitter DMA

The data from the source areas A, B and C and the output data D are read from or written to memory through four DMA channels. This Blitter DMA can be enabled for all channels with the BLTEN bit (bit 6) of the DMACON register. The Blitter has four data registers for its DMA transfers:

Addr.	Name	Function
000	BLTDDAT	Output data D
070	BLTCDAT	Data register for source C
072	BLTBDAT	Data register for source B
074	BLTADAT	Data register for source A

The DMA controller reads the needed input values from memory and writes them to the data registers. When the Blitter has processed the input data, BLTDDAT contains the result. The DMA controller then transfers the contents of BLTDDAT to the chip RAM.

The DMA transfer through these four registers can be enabled and disabled by using the four USEx bits. For example, USEA = 0 disables the DMA channel for data register A. The Blitter continues to access the value in BLTADAT, so with each new word from the active sources the same word is fetched from source A. For this reason unused sources must have USEx set to 0 and must be prevented from affecting the result by the appropriate selection of minterms. However, the same word is always read when the DMA channel is disabled. For example, you can fill the memory with a repeating pattern that has been written directly into BLTxDAT.

In addition to BLTEN, three other bits in the DMACON register pertain to the Blitter:

Bit 10 BLTPRI

This bit was already explained in the Fundamentals section (11.7.2). If it is 1, the Blitter has absolute priority over the processor.

Bit 14 BBUSY (read-only)

BBUSY signals the status of the Blitter. If it is 1, it is currently performing an operation.

After the Blitter window is set in BLTSIZE the Blitter begins its DMA and sets BBUSY until the last word of the Blitter window has been processed and written back into memory. It then ends its DMA and clears BBUSY.

At the same time BBUSY is cleared, the Blitter-finished bit in the interrupt request register is also set.

Bit 13 BZERO

The BZERO bit indicates whether all the result bits of a Blitter operation were 0. In other words, BZERO is set when none of the operations performed on any of the data words resulted in a 1. One use of this bit is to perform collision detection.

The minterms are set so that D is 1 only if the two sources are also 1. If the graphics in both sources intersect at least one point, the result is 1 and BZERO is cleared. At the end of the Blitter operation you can determine whether or not a collision occurred. USED is set to 0 in this application so that the output data aren't written to memory.

Using the Blitter to fill surfaces

What does it mean to "fill a surface"? The Blitter understands a surface to be a two-dimensional area of memory to be filled with points. Normally this surface belongs to a graphic or a bit-plane.

In order to fill a surface, the Blitter must recognize its boundaries. You need a definition of a boundary line that the Blitter can understand. Many fill functions exist in most drawing programs and also in Amiga-BASIC with the PAINT command.

These functions cause an area of the screen to be filled, starting with some initial point, until the program encounters a boundary line. This allows the painting of completely arbitrary surfaces, assuming that they are enclosed by a continuous line. The Blitter is not able to perform such a complex fill operation. It only works line by line and fills the free space between two set bits which mark the boundaries of the desired surface. The following examples show how the Blitter fill operation works:

Correct fill operation:

```
         Before:                         After:
...........1.1...........     ...........111...........
........1.......1........     ........111111111........
......1...........1......     ......1111111111111......
.....1....1...1....1.....     .....111111...111111.....
.....1....1...1....1.....     .....111111...111111.....
......1...........1......     ......1111111111111......
........1.......1........     ........111111111........
......1...........1......     ......1111111111111......
```

Incorrect operation due to improper border bits:

```
         Before:                        After:
...........111...........     11111111111111...........
........111...111........     ........111111111........
......11...111...11......     11111111111111...11......
.....1....1...1....1.....     .....111111...111111.....
.....1....1...1....1.....     .....111111...111111.....
......11...111...11......     11111111111111...11......
........1.......1........     ........111111111........
......1111111111111......     1111111111111111111......
```

In the first example, the surface is bounded properly for the Blitter and filled correctly. However, in example 2, a closed boundary line is drawn around the figure. If you try to fill such a graphic with the Blitter, chaos results.

The reason for this is the algorithm that the Blitter uses. It is extremely simple. The Blitter starts at the right side of the line. As it proceeds to the left, it uses the Fill Carry bit (FC) to determine whether an output bit must be set. The output bit corresponds to the value of the FC bit, which normally (as in our example) starts out as 0.

When the Blitter encounters an input bit that is set, the value of the FC bit changes (from 0 to 1 in our example). This causes subsequent output bits to have the new value (now 1), until another set bit is encountered in the input. Then the FC bit will be switched again (back to 0).

In this way the area between two set bits is always filled. As you can see from the second example, the fill logic gets confused by an odd number of set bits.

The FCI bit (Fill Carry In) in BLTCON1 determines the initial value of the FC bit. If FCI is cleared, everything proceeds as previously described. But if FCI = 1, the Blitter starts to fill from the edge until it encounters the first set input bit. The fill procedure is then reversed.

Example of the effect of the FCI bit:

```
  Output graphic          FCI=0              FCI=1
......1.....1.....   ......1111111.....   1111111.....111111
....1.........1...   ....11111111111...   11111.........1111
...1....1.1....1..   ...111111.111111..   1111....111....111
...1....1.1....1..   ...111111.111111..   1111....111....111
....1.........1...   ....11111111111...   11111.........1111
......1.....1.....   ......1111111.....   1111111.....111111
```

In the examples up to now, the input bits (the boundaries of the surface) have been retained in the filled graphic. This is always the case when the fill mode is activated by setting the ICE bit (InClusive fill Enable) in the BLTCON1 register.

In contrast to this is the ECE mode (ExClusive fill Enable), which is enabled by setting the bit with this name in BLTCON1. In this mode the boundary bits at the left edge of a filled surface (whenever the fill carry bit changes from 1 to 0) are not retained in the output picture.

This causes all surfaces to be one pixel narrower. Only in the ECE mode is it possible to get surfaces with a width of only one bit. It is impossible in the ICE mode because the definition of a surface, however narrow, requires at least two boundary bits, both of which will appear in the output.

Difference between ICE and ECE modes:

```
  Output graphic           ICE                 ECE
........11......11   ........11......11   .........1.......1
......1...1....1.1   ......11111....111   .......1111.....11
...1...11..1..1..1   ...111111111..1111   ....1111.111...111
1......11...11...1   111111111111111111   .1111111.1111.1111
..1..............1   ..1111111111111111   ...111111111111111
....1...........1.   ....1111111111111.   .....111111111111.
......1...1...11..   ......11111...11..   .......1111....1..
```

Bit wise operation of the different fill operations:

Input pattern: 11010010

Bit no.	Input bit	FCI = 0 FC	ICE	ECE	FCI = 1 FC	ICE	ECE
-	11010010	FC=FCI			FC=FCI		
0	0	0	0	0	1	1	1
1	1	1	10	10	0	11	01
2	0	1	110	110	0	011	001
3	0	1	1110	1110	0	0011	0001
4	1	0	11110	01110	1	10011	10001
5	0	0	011110	001110	1	110011	110001
6	1	1	1011110	1001110	0	1110011	0110001
7	1	0	11011110	01001110	1	11110011	10110001

FC=FCI means that the FC bit assumes the value of the FCI bit from BLTCON1 at the start of the fill operation.

How is a Blitter fill operation started? The Blitter can perform this fill operation in addition to an ordinary copy procedure. It is enabled by setting either the ICE bit or the ECE bit in BLTCON1 depending on the desired mode. The Blitter forms the output data D from the three sources A, B and C and the selected minterms as usual. If neither of the two fill modes is active, the Blitter writes this data directly to its output data register (BLTDDAT, $000). From there the data is written to memory via DMA if USED = 1.

In the fill mode, the output data D is used as input data for the fill circuit. The result of the fill operation is then written into the output data register BLTDDAT.

The following steps are needed to perform a fill operation:

- Select the BLTxPT, BLTxMOD and minterms so that the output data D contains the correct boundary bits for the surface to be filled.
- Select descending mode (the Blitter fills from right to left and this works only when the words are referenced with descending addresses).
- Select the desired fill mode: set ICE or ECE; set or clear FCI as desired.
- LINE = 0 (Line mode off).

- Set BLTSIZE to the size of the graphic to be filled.

The Blitter now begins the fill procedure. When it is done, it sets BLTBUSY to 0 as usual. The speed of the Blitter is not limited by activating the fill mode.

The Blitter can fill surfaces at a maximum speed of 16 million pixels per second. The major application of the fill mode is in drawing filled polygons. The desired polygon is drawn in an empty memory area using line mode and then filled very rapidly by the Blitter.

Using the Blitter to draw lines

The Blitter is an extremely versatile tool. In addition to its excellent capabilities for copying data and filling surfaces, it has a powerful mode for drawing lines. Like the other Blitter modes, the line mode is extremely fast: up to a million pixels per second.

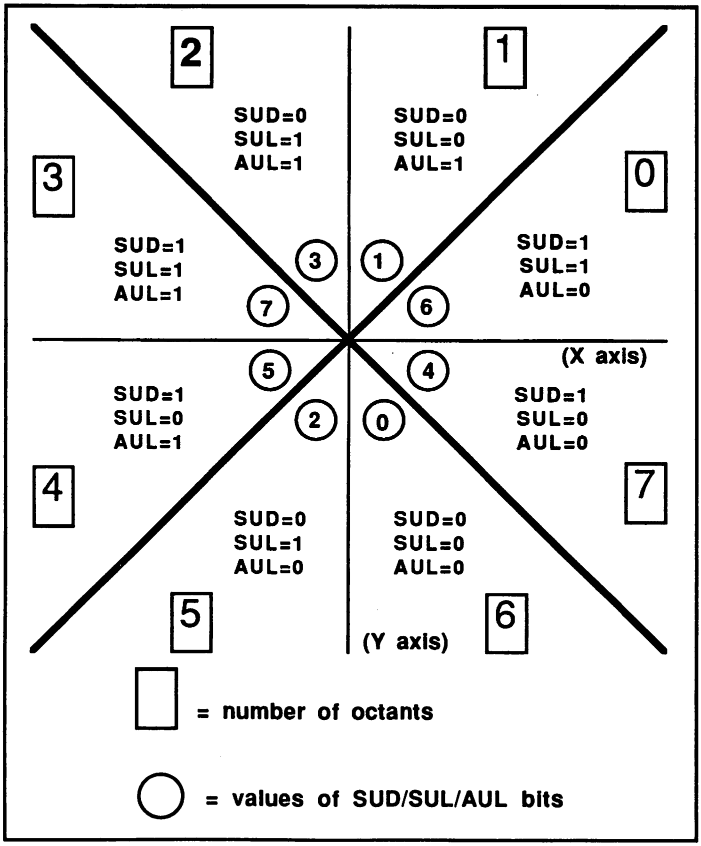
What exactly is "drawing lines"? When a line is drawn, two points are connected to each other by a continuous series of points. Since the resolution of a computer graphic is limited, the optimal points cannot always be chosen. The actual pixels may lie slightly above or below the intended ideal line. Such a line usually resembles a staircase. The higher the resolution, the smaller the steps, but they can never be completely eliminated.

Example of a line in a computer graphic:

```
The two points                 are connected by a line
......................            ......................
..................1...            ..................111...
......................            ...............111......
......................            ...........1111.........
......................            ........111.............
.....1................            .....111................
......................            ......................
```

The Blitter can draw lines up to a length of 1024 pixels. Unfortunately, you cannot specify the coordinates of the two endpoints. Like solid surfaces, lines must be defined in a style recognizable to the Blitter.

First, the Blitter needs to know the octants in which the line is located. The coordinate system is divided into eight parts; you'll find that the octants are found in many graphic processors.



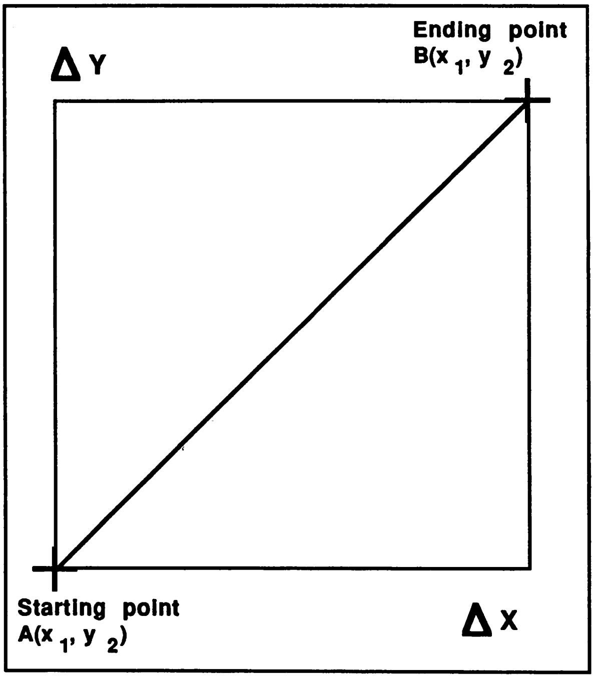
Blitter octants

The figure shows this division. The starting point of the line is located at the origin of the coordinate system (the intersection of the X and Y axes). The end point is located in one of the eight octants, according to its coordinates. The number of this octant can be determined with three logical comparisons. X1 and Y1 are the coordinates of the start point and X2 and Y2 are those of the end point:

If X1 is less than X2, the end point is in octant 0, 1, 6 or 7, while if X1 is greater than X2, it is in 2, 3, 4 or 5. If X1 and X2 are equal, it is on the Y axis. Then all eight octants are possible.

Similarly: If Y1 is less than Y2, possible octants of the end point are 0, 1, 2 and 3, and if Y1 is greater than Y2, the octants are 4, 5, 6 and 7. If Y1 = Y1, all are possible.

For the last comparison we need the X and Y differences: DeltaX = |X2-X1|, DeltaY = |Y2-Y1|. If DeltaX is greater than DeltaY, the end point can be located in octant 0, 3, 4 or 7. If DeltaX is less than DeltaY, it is in octant 1, 2, 5 or 6. For DeltaX = DeltaY, all octants.



Selection of start and end points

The end point is located in the octant that occurred in all three comparisons. If a point is on the border between two octants, it doesn't matter which is chosen.

The digits in the "Code" column correspond to the circled numbers in the figure. The Blitter needs a combination of three bits, depending on the octant in which the end point of the line is located. The bits are called SUD (Sometimes Up or Down), SUL (Sometimes Up or Left) and AUL (Always Up or Left).

"Code" is the 3-bit number formed by these bits (SUD = MSB and AUL = LSB).

When programming the line you must first determine the octant of the end point and then write the corresponding code value into the Blitter.

Selecting the correct octant:

Point coordinates	Octant	Code	Point coordinates	Octant	Code
Y1 <= Y2 X1 <= X2 DeltaX >= DeltaY	0	6	Y1 >= Y2 X1 >= X2 DeltaX >= DeltaY	4	5
Y1 <= Y2 X1 <= X2 DeltaX <= DeltaY	1	1	Y1 >= Y2 X1 >= X2 DeltaX <= DeltaY	5	2
Y1 <= Y2 X1 >= X2 DeltaX <= DeltaY		3	Y1 >= Y2 X1 <= X2 DeltaX <= DeltaY	6	0
Y1 <= Y2 X1 >= X2 DeltaX >= DeltaY	3	7	Y1 >= Y2 X1 <= X2 DeltaX >= DeltaY	7	4

Lines with patterns

When drawing a line, the Blitter uses a mask to determine whether the points of the line should be set, cleared, or given a pattern. The mask is 16 bits wide, so the pattern repeats every 16 points. The relationship between the pattern and the appearance of the line can best be understood with a couple of examples:

```
("." = 0, "1" = 1, A = start point and B = end point)
```

Output picture: Mask = "1111111111111111":

```
.........11111111......B.        .........11111111....11B.
.......111......111......        .......111......11111....
.....11............11....        .....11.........11.11....
....11..............11...        ....11.......111....11...
....11..............11...        ....11...111.......11...
.....11............11....        .....11111.........11....
.......111......111......        .....11111......111......
..A......11111111........        ..A11...11111111........
```

Zero bits in the mask cause line points to be cleared:

Output picture: Mask = "0000000000000000":

```
....11111111111111111..B.        ....11111111111111111..B.
....11111111111111111....        ....11111111111111.......
....11111111111111111....        ....111111111111..111....
....11111111111111111....        ....111111111...11111....
....11111111111111111....        ....111111...11111111....
....11111111111111111....        ....1111..11111111111....
....11111111111111111....        ....1...1111111111111....
..A.11111111111111111....        ..A..1111111111111111....
```

If we combine ones and zeros in the mask, the line takes on a pattern:

Mask = "1111111000111000

```
.A111111.........................................
...........111...1...............................
.................111111..........................
..........................111..11................
.................................11111...........
........................................111...111........
.............................................1111...B
```

Drawing boundary lines

In the section on filling surfaces with the Blitter, we explained that the boundary lines of these surfaces can only be one pixel wide.

If these lines are drawn with the Blitter, it's possible that several line points lie on the same horizontal line. To prevent this, the Blitter can be made to draw lines with only one point per raster line:

```
Normal line:                   Line with one point/raster line:
................1111             ................1...
............1111....             ............1.......
........1111........             ........1...........
....1111............             ....1...............
1111................             1...................
```

The definition of slope

So the Blitter knows where to draw the line, it needs a Blitter-style definition of the slope of the line. This slope is formed from the results of three terms, all based on the DeltaX and DeltaY values, as explained in the section on octants (DeltaY and DeltaX represent the width and height of the rectangle for which the line forms a diagonal).

First the two values must be compared with each other to find the larger/smaller of the two. The smaller delta is called Sdelta and the larger one is called Ldelta. Then the three expressions required by the Blitter are as follows:

1. 2*Sdelta
2. 2*Sdelta - Ldelta
3. 2*Sdelta - 2*Ldelta

Also, the Blitter has a SIGN flag which must be set to 1 if 2*Sdelta < Ldelta.

Register functions in line mode

The Blitter uses the same registers when drawing lines as it does when copying data (it doesn't have any more), but the functions change:

BLTAPTL
: The value of the expression "2*Sdelta-Ldelta" must be written into BLTAPTL.

BLTCPT & BLTDPT
: These two register pairs (BLTCPTH and BLTCPTL, BLTDPTH and BLTDPTL) must be initialized with the start address of the line. This is the address of the word in which the start point of the line is located.

BLTAMOD — The value of the expression "2*Sdelta-2*Ldelta" must be stored in BLTAMOD.

BLTBMOD — "2*Sdelta"

BLTCMOD & BLTDMOD — The width of the entire picture in which the line is to be drawn must be stored in these two modulo registers. As usual, this takes the form of an even number of bytes. With a normal bit-plane of 320 pixels (40 bytes) in the X direction, the value for BLTCMOD or BLTDMOD = 40.

BLTSIZE — The width (bits 0 to 5) must be set to 2. The height (bits 6 to 15) contains the length of the line in pixels. A height of 0 indicates a line length of 1024 pixels. The correct line length is always the value of Ldelta.

Drawing a line is started by writing to the BLTSIZE register. Therefore, it should be the last register initialized.

BLTADAT — This register must be initialized to $8000.

BLTBDAT — BLTBDAT contains the mask with which the line is drawn.

BLTAFWM — $FFFF is stored in this mask register.

BLTCON0

Bit no.	Name	Function
15	START3	The 4-bit value START0-3 contains the position of the start point
14	START2	of the line within the word
13	START1	at the start address of the line (BLTCPT/BLTDPT)
12	START0	The four lower bits of the X coordinate of the start point
11	USEA = 1	This combination of the USEx bits is necessary
10	USEB = 0	for the line mode
9	USEC = 1	
8	USED = 1	
7	LF7	The LFx bits must be initialized with $CA
to 0	LF0	(D = aC + AB)

BLTCON1

Bit no.	Name	Function
15	Texture3	This is the value for shifting the mask.
14	Texture2	Normally Texture0-3 is set to Start0-3.
13	Texture1	The pattern in the mask register BLTBDAT
12	Texture0	then starts with the first point of the line.
11-7 = 0		Unused, always set to 0.
6	SIGN	If 2*Sdelta<Ldelta, set SIGN to 1.
5	---	Unused, always set to 0.
4	SUL	These three bits must be initialized
3	SUD	with the SUL/SUD/AUL code
2	AUL	of the corresponding octant.
1	SING = 1	Draw lines with only one point per raster line.
0	LINE = 1	Put the Blitter in line drawing mode.

A numerical example:

You want to draw a line in a bit-plane. The bit-plane is 320 by 200 pixels large and lies at address $40000. The starting point of the line has the coordinates X=20 and Y=185. The end point lies at X=210 and Y=35. (The coordinates are in relation to the upper left corner of the bit-plane.) DeltaX = 190, DeltaY = 150.

1st step: Find the octant of the end point

To do this, the three comparisons discussed earlier are performed; the result: X1 < X2, Y1 > Y2 and DeltaX > DeltaY. This yields octant number 7 and a value for the SUD/SUL/AUL code of 4.

2nd step: Address of the starting point

This is calculated as follows:

```
starting address of bitplane + (number of lines - Y1 - 1) * bytes per line +
2*(X1/16)
```

The fractional portion of the division is ignored. After inserting the values:

```
$40000 + (200-185-1)*40 + 2 = $40232
```

this value is placed in BLTCPT and BLTDPT. The number of bytes per line is also written to the BLTCMOD and BLTDMOD registers.

3rd step: Starting point of the line in START0-3

Required calculation: X1 AND $F. Numerically:

```
START0-3 = 20 AND $F = 4
```

4th step: Values for BLTAPTL, BLTAMOD and BLTBMOD

```
DeltaY < DeltaX, meaning that Sdelta = DeltaY and Ldelta = DeltaX.
BLTAPTL = 2*Sdelta-Ldelta = 2*150-190 = 110
BLTAMOD = 2*Sdelta-2*Ldelta = 2*150-2*190 = -80
BLTBMOD = 2*Sdelta = 300
2*Sdelta>Ldelta
```

Therefore SIGN = 0.

5th step: Length of the line for BLTSIZE

```
Length = Ldelta = DeltaX = 190.
```

The value of the BLTSIZE register is calculated from the usual formula: Length*64 + Width. Width must always be set to 2 when drawing lines. BLTSIZE = DeltaX*64+2 = 12162 or $2F82.

6th step: Combining the values for the two BLTCONx registers

The START value must be stored in the correct position in BLTCON0, in addition to $CA for the LFx bits and 1011 for USEx. In our example, this results in $ABCA.

BLTCON1 contains the code for the octant and the control bits. We want to draw our line normally, so SING = 0. LINE must naturally be 1. SIGN was already calculated and is 0 in this example. Together this makes $0011.

In assembly language the initialization of the registers might look like this:

```
LEA $DFF000,A5                 ;Base address of the custom chips to A5
MOVE.L #$40232, BLTCPTH(A5)    ;Start address to BLTCPT
MOVE.L #$40232, BLTDPTH(A5)    ;and BLTDPT
MOVE.W #40, BLTCMOD(A5)        ;Width of bitplane to BLTCMOD
MOVE.W #40, BLTDMOD(A5)        ;and BLTDMOD
MOVE.W #110, BLTAPTL(A5)
```

```
MOVE.W #-80, BLTAMOD(A5)
MOVE.W #300, BLTBMOD(A5)
MOVE.W #$ABCA, BLTCON0(A5)
MOVE.W #$11, BLTCON1(A5)
MOVE.W #12162, BLTSIZE(A5)      ;Now the blitter starts
                                ;drawing the line
```

Other drawing modes

Up to now we always used $CA as the value for the LFx bits. This causes the points on the line to be set or cleared according to the mask, while the other points remain unchanged.

But other combinations of LFx are also useful. To understand this, you must know how the LFx bits are interpreted in the line mode:

The Blitter can only address memory by words. In line mode the input data enters the Blitter through source channel C. The mask is stored in the B register. The A register determines which point in the word read is the line point. It always contains exactly one set bit, which is shifted by the Blitter to the correct position. The normal LFx value of $CA causes all bits, for which the A bit is 0, to be taken directly from source C. However, if A is 1, the destination bit is taken from the corresponding mask bit.

If you know how the LFx bits are used, you can choose other drawing modes. For example, $4A causes all the line points to be inverted.

The Blitter DMA cycles

As we explained in the section on fundamentals, the Blitter uses only even bus cycles. Since it has priority over the 68030, it is interesting to know how many cycles are left for the processor. This depends on the number of active Blitter DMA channels (A, B, C and D). The following table shows the course of a Blitter operation for all fifteen possible combinations of active and inactive Blitter DMA channels. The letters A, B, C and D represent the corresponding DMA channels. Behind them, the digit 1 represents the first data word of the Blitter operation, the digit 3 for the last word, and the digit 2 for all words in between. A dashed line (--) indicates that this bus cycle is not used by the Blitter.

Usage of even bus cycles by the Blitter:

Comments:

The table is only valid if the following conditions are fulfilled:

1. The Blitter is not disturbed by Copper or bit-plane DMA accesses.
2. The Blitter is running in normal mode (neither drawing lines nor filling surfaces).
3. The BLITPRI bit in the DMACON register is set and the Blitter has absolute priority over the 68030.

Active DMA channels	Usage of even bus cycles												
None	--	--	--	--	--	--	--	--	--	--	--	--	--
D	D0	--	D1	--	D2	--	--	--	--	--	--	--	--
C	C0	--	C1	--	C2	--	--	--	--	--	--	--	--
C D	C0	--	--	C1	D0	--	C2	D1	--	D2	--	--	--
B	B0	--	--	B1	--	--	B2	--	--	--	--	--	--
B D	B0	--	--	B1	D0	--	B2	D1	--	D2	--	--	--
B C	B0	C0	--	B1	C1	--	B2	C2	--	--	--	--	--
B C D	B0	C0	--	--	B1	C1	D0	--	B2	C2	D1	--	D2
A	A0	--	A1	--	A2	--	--	--	--	--	--	--	--
A D	A0	--	A1	D0	A2	D1	--	D2	--	--	--	--	--
A C	A0	C0	A1	C1	A2	C2	--	--	--	--	--	--	--
A C D	A0	C0	--	A1	C1	D0	A2	C2	D1	--	D2	--	--
A B	A0	B0	--	A1	B1	--	A2	B2	--	--	--	--	--
A B D	A0	B0	--	A1	B1	D0	A2	B2	D1	--	D2	--	--
A B C	A0	B0	C0	A1	B1	C1	A2	B2	C3	--	--	--	--
A B C D	A0	B0	C0	--	A1	B1	C1	D0	A2	B2	B3	D1	D2

Explanations:

As you can see, the output data D0 doesn't get to RAM until the A1, B1 and C1 data have been read. This results from the pipelining in the Blitter. Pipelining means that the data is processed in multiple stages in the Blitter. Each stage is connected to the output of the preceding one and the input of the next. The first stage gets the input data (for example, A0, B0 and C0), processes it and passes it on to the second stage. While it is being further processed in that stage, the next input data is fed into the input stage (A1, B1 and C1). When the first data reaches the output stage, the Blitter has long since read the next data. Two data pairs are always at different processing stages in the Blitter at any given time during a Blitter operation.

The table also allows the processing time of a Blitter operation to be calculated. Every microsecond the Blitter has two bus cycles available. If it's moving a 64K block (32768 words) from A to D, it needs 2*32768 cycles. But if the same block is combined with source C, a total of 3*32768 cycles are needed, because two input words must be read for each output word produced.

The table also shows that the Blitter is not capable of using every bus cycle even if only one DMA channel is active.

Sample programs

Program 1: Drawing lines with the Blitter

This program can be used as a universal routine for drawing lines with the Blitter. It shows how the necessary values can be calculated. The program is quite simple:

At the start of the program the memory is requested and the Copper list is constructed. The only new part is the OwnBlitter routine. As its name indicates, it can be used to gain control of the Blitter. Correspondingly, there is a call to DisownBlitter at the end of the program so that the Blitter returns to the control of the operating system.

The program uses only one hi-res bit-plane, with standard dimensions of 640 x 256 pixels. In the main loop, the program draws lines that go from one side of the screen through the center of the screen to the other side. When a screen has been filled in this manner, the program shifts the mask used to draw the lines and starts over again.

Comments:

The coordinate specifications in the program start from point 0,0 in the upper left corner of the screen and are not mathematical coordinates, as were used in the previous discussions. This means that when comparing the Y values, the greater/less than sign is reversed.

```
;*** Lines with the Blitter

;Custom chip register

INTENA  = $9A   ;Interrupt enable register (write)
DMACON  = $96   ;DMA-Control register (write)
DMACONR = $2    ;DMA-Control register (read)
COLOR00 = $180  ;Color palette register
```

```
VHPOSR  = $6    ;Position (read)

;Copper Register

COP1LC  = $80   ;Address of 1st. Copper-List
COP2LC  = $84   ;Adderss of 2nd. Copper-List
COPJMP1 = $88   ;Jump to  Copper-List 1
COPJMP2 = $8a   ;Jump to Copper-List 2

;Bitplane Register

BPLCON0 = $100  ;Bitplane control register 0
BPLCON1 = $102  ;1 (Scroll value)
BPLCON2 = $104  ;2 (Sprite<>Playfield Priority)
BPL1PTH = $0E0  ;Pointer to 1st. bitplane
BPL1PTL = $0E2  ;
BPL1MOD = $108  ;Modulo value for odd Bit-Planes
BPL2MOD = $10A  ;Modulo value for even Bit-Planes
DIWSTRT = $08E  ;Start of screen window
DIWSTOP = $090  ;End of screen window
DDFSTRT = $092  ;Bit-Plane DMA Start
DDFSTOP = $094  ;Bit-Plane DMA Stop

;Blitter Register

BLTCON0 = $40   ;Blitter control register 0 (ShiftA,Usex,LFx)
BLTCON1 = $42   ;Blitter control register 1 (ShiftB,misc. Bits)
BLTCPTH = $48   ;Pointer to source C
BLTCPTL = $4a

BLTBPTH = $4c   ;Pointer to source B
BLTBPTL = $4e

BLTAPTH = $50   ;Pointer to source A
BLTAPTL = $52

BLTDPTH = $54   ;Pointer to target data D
BLTDPTL = $56

BLTCMOD = $60   ;Modulo value for source C
BLTBMOD = $62   ;Modulo value for source B
BLTAMOD = $64   ;Modulo value for source A
BLTDMOD = $66   ;Modulo value for target D
BLTSIZE = $58   ;HBlitter window width/height
BLTCDAT = $70   ;Source C data register
BLTBDAT = $72   ;Source B data register
BLTADAT = $74   ;Source A data register
BLTAFWM = $44   ;Mask for first data word from source A
```

```
BLTALWM = $46   ;Mask for first data word from source B


;CIA-A Port register A (Mouse key)

CIAAPRA = $bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0

Forbid      = -30-102
Permit      = -30-108
AllocMem    = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem     = -30-180 ;MemoryBlock,ByteSize/a1,d0

;Graphics Library Base Offsets

OwnBlitter    = -30-426
DisownBlitter = -30-432

;graphics base

StartList = 38

;other Labels

Execbase   = 4
Planesize  = 80*200      ;Bitplane size: 80 Bytes by 200 lines
Planewidth = 80

CLsize     = 3*4         ;The Copper-List contains 3 commands
Chip       = 2           ;allocate Chip-RAM
Clear      = Chip+$10000 ;Clear Chip-RAM first


;*** Initialization ***

Start:

;A;llocate memory for bit plane

 move.l Execbase,a6
 move.l #Planesize,d0    ;Memory requirement for bit plane
 move.l #clear,d1
 jsr    AllocMem(a6)     ;Allocate memory
 move.l d0,Planeadr
 beq    Ende             ;Error! -> End
```

```
;Allocate memory for Copper-List

 moveq  #Clsize,d0
 moveq  #chip,d1
 jsr    AllocMem(a6)
 move.l d0,CLadr
 beq    FreePlane          ;Error! -> FreePlane

;Create Copper-List

 move.l d0,a0              ;Address of Copper-List from a0
 move.l Planeadr,d0        ;Address of Bitplane
 move.w #bpl1pth,(a0)+     ;First Copper command in RAM
 swap   d0
 move.w d0,(a0)+           ;Hi-Word of Bit plane address in RAM
 move.w #bpl1ptl,(a0)+     ;second command in RAM
 swap   d0
 move.w d0,(a0)+           ;Lo-Word of  Bitplane address in RAM
 move.l #$fffffffe,(a0)    ;End of Copper-List

;Allocate Blitter

 move.l #GRname,a1
 clr.l  d0
 jsr    OpenLibrary(a6)
 move.l a6,-(sp)           ;ExecBase from the Stack
 move.l d0,a6              ;GraphicsBase from a6
 move.l a6,-(sp)           ;and from the Stack
 jsr    OwnBlitter(a6)     ;Take over Blitter

;*** Main program ***

;DMA and Task-Switching off

 move.l 4(sp),a6           ;ExecBase to a6
 jsr    forbid(a6)         ;Task-Switching off
 lea    $dff000,a5
 move.w #$03e0,dmacon(a5)

;Copper initialization

 move.l CLadr,cop1lc(a5)
 clr.w  copjmp1(a5)

;Set color

 move.w #$0000,color00(a5)   ;Black background
```

```
 move.w #$0fa0,color00+2(a5) ;Yellow line

;Playfield initialization

 move.w #$2081,diwstrt(a5)   ;20,129
 move.w #$20c1,diwstop(a5)   ;20,449
 move.w #$003c,ddfstrt(a5)   ;Normal  Hires Screen
 move.w #$00d4,ddfstop(a5)
 move.w #%1001001000000000,bplcon0(a5)
 clr.w  bplcon1(a5)
 clr.w  bplcon2(a5)
 clr.w  bpl1mod(a5)
 clr.w  bpl2mod(a5)

;DMA on
 move.w #$83C0,dmacon(a5)

;Draw lines

;Determine start values:

 move.l Planeadr,a0       ;Constant parameter for DrawLine
 move.w #Planewidth,a1    ;into correct register
 move.w #255,a3           ;Size of  Bitplane in Register
 move.w #639,a4
 move.w #$0303,d7         ;Start pattern

Loop:  rol.w #2,d7        ;Shift pattern
 move.w d7,a2             ;Pattern in register for DrawLine

 clr.w  d6                ;Clear loop variable
LoopX:

 clr.w  d1                ;Y1 = 0
 move.w a3,d3             ;Y2 = 255
 move.w d6,d0             ;X1 = Loop variable
 move.w a4,d2
 sub.w  d6,d2             ;X2 = 639-Loop variable

 bsr DrawLine

 addq.w #4,d6            ;Increment loop variable
 cmp.w  a4,d6            ;Test if greater than 639
 ble.s  LoopX            ;if not. continue loop

 clr.w  d6               ;Clear loop variable
LoopY:
```

```
 move.w a4,d0           ;X1 = 639
 clr.w  d2              ;X2 = 0
 move.w d6,d1           ;Y1 = loop variable
 move.w a3,d3
 sub.w  d6,d3           ;Y2 = 255-loop variable

 bsr DrawLine           ;Draw line

 addq.w #2,d6           ;Increment loop variable
 cmp.w  a3,d6           ;Is loop variable greater than 255?
 ble.s  LoopY           ;if not, continue loop

 btst   #6,ciaapra      ;Mouse key pressed?
 bne    Loop            ;No, continue

;*** End program ***

;Wait till blitter is ready

Wait:  btst #14,dmaconr(a5)
 bne    Wait

;Activate old Copper-List

 move.l (sp)+,a6       ;Get GraphicsBase from Stack
 move.l StartList(a6),cop1lc(a5)
 clr.w  copjmp1(a5)    ;Activate Startup-Copper-List
 move.w #$8020,dmacon(a5)
 jsr    DisownBlitter(a6)    ;Release blitter
 move.l (sp)+,a6        ;ExecBase from Stack
 jsr    Permit(a6)      ;Task Switching on

;Release memory for Copper-List

 move.l CLadr,a1        ;Set parameter for FreeMem
 moveq  #CLsize,d0
 jsr    FreeMem(a6)     ;Release memory

;Release Bitplane memory

FreePlane:

 move.l Planeadr,a1
 move.l #Planesize,d0
 jsr    FreeMem(a6)

Ende:
```

```
 clr.l  d0
 rts                       ;Program end

;Variables

CLadr:    dc.l 0           ;Address of Copper-List
Planeadr: dc.l 0           ;Address of Bitplane

;Constants

GRname: dc.b "graphics.library",0

 align                     ;even

;*** DrawLine Routine ***

;DrawLine draws a line with the Blitter.
;The following parameters are used:
;d0  =  X1   X-coordinate of Start points
;d1  =  Y1   Y-coordinate of Start points
;d2  =  X2   X-coordinate of End points
;d3  =  Y2   Y-coordinate of End points
;a0 must point to the first word of the bitplane
;a1 contains bitplane width in bytes
;a2 word written directly to mask register
;d4 to d6 are used as work registers


DrawLine:

;Compute the lines starting address

 move.l a1,d4              ;Width in work register
 mulu   d1,d4              ;Y1 * Bytes per line
 moveq  #-$10,d5           ;No leading characters: $f0
 and.w  d0,d5              ;Bottom four bits masked from X1
 lsr.w  #3,d5              ;Remainder divided by  8
 add.w  d5,d4              ;Y1 * Bytes per line + X1/8
 add.l  a0,d4              ;plus starting address of the Bitplane
                           ;d4 now contains the starting address
                           ;of the line
                           ;Compute octants and deltas

 clr.l  d5                 ;Clear work register
 sub.w  d1,d3              ;Y2-Y1  DeltaY from D3
 roxl.b #1,d5              ;shift leading char from DeltaY in d5
 tst.w  d3                 ;Restore N-Flag
 bge.s  y2gy1              ;When DeltaY positive, goto y2gy1
```

```
 neg.w  d3               ;DeltaY invert (if not positive)
y2gy1:
 sub.w  d0,d2            ;X2-X1  DeltaX to D2
 roxl.b #1,d5            ;Move leading char in DeltaX to d5
 tst.w  d2               ;Restore N-Flag
 bge.s  x2gx1            ;When DeltaX positive, goto x2gx1
 neg.w  d2               ;DeltaX invert (if not  positive)
x2gx1:

 move.w d3,d1            ;DeltaY to d1
 sub.w  d2,d1            ;DeltaY-DeltaX
 bge.s  dygdx            ;When DeltaY > DeltaX, goto dygdx
 exg    d2,d3            ;Smaller Delta goto d2
dygdx: roxl.b #1,d5      ;d5 contains results of 3 comparisons
 move.b Octant_table(pc,d5),d5 ;get matching octants
 add.w  d2,d2            ;Smaller Delta * 2

;Test, for end of last blitter operation

WBlit:   btst #14,dmaconr(a5);BBUSY-Bit test
 bne.s   WBlit           ;Wait until equal to 0

 move.w d2,bltbmod(a5)   ;2* smaller Delta to BLTBMOD
 sub.w  d3,d2            ;2* smaller Delta - larger Delta
 bge.s  signnl           ;When 2* small delta > large delta to signal
 or.b   #$40,d5          ;Sign flag set
signnl: move.w d2,bltaptl(a5) ;2*small delta-large delta in BLTAPTL
 sub.w  d3,d2            ;2* smaller Delta - 2* larger Delta
 move.w d2,bltamod(a5)   ;to BLTAMOD

;Initialization other info

 move.w #$8000,bltadat(a5)
 move.w a2,bltbdat(a5)   ;Mask from a2 in BLTBDAT
 move.w #$ffff,bltafwm(a5)
 and.w  #$000f,d0        ;bottom  4 Bits from  X1
 ror.w  #4,d0            ;to START0-3
 or.w   #$0bca,d0        ;USEx and LFx set
 move.w d0,bltcon0(a5)
 move.w d5,bltcon1(a5)   ;Octant in Blitter
 move.l d4,bltcpth(a5)   ;Start address of line to
 move.l d4,bltdpth(a5)   ;BLTCPT and BLTDPT
 move.w a1,bltcmod(a5)   ;Width of Bitplane in both
 move.w a1,bltdmod(a5)   ;Modulo Registers

;BLTSIZE initialization and Blitter start

 lsl.w  #6,d3            ;LENGTH * 64
```

```
 addq.w #2,d3           ;plus (Width = 2)
 move.w d3,bltsize(a5)

 rts

;Octant table with LINE =1:
;The octant table contains code values
;for each octant, shifted to the correct position

Octant_table:

 dc.b 0 *4+1 ;y1<y2, x1<x2, dx<dy = Okt6
 dc.b 4 *4+1 ;y1<y2, x1<x2, dx>dy = Okt7
 dc.b 2 *4+1 ;y1<y2, x1>x2, dx<dy = Okt5
 dc.b 5 *4+1 ;y1<y2, x1>x2, dx>dy = Okt4
 dc.b 1 *4+1 ;y1>y2, x1<x2, dx<dy = Okt1
 dc.b 6 *4+1 ;y1>y2, x1<x2, dx>dy = Okt0
 dc.b 3 *4+1 ;y1>y2, x1>x2, dx<dy = Okt2
 dc.b 7 *4+1 ;y1>y2, x1>x2, dx>dy = Okt3

 end
```

Program 2: Filling surfaces with the Blitter

This program is very similar to the first program. It shows how you can create colored polygons by drawing border lines and filling them with the Blitter. Since most of it is identical to the first program, we've only printed the parts that must be changed in program 1 to create program 2. The first part that must be changed starts at the comment "Draw lines ***" and ends at the comment "*** End program ***." This area must be replaced by the section in the following listing labeled "Part 1."

Also, the old octant table at the end of the program must be replaced with the new one following the heading "Part 2." The new octant table is required because, when filling surfaces, the Blitter needs boundary lines with only one pixel per line. In the new octant table, the LINE bit and the SING bit are set.

The program labeled "Part 1" draws two lines and then fills the area between them with the Blitter. Then it waits for the mouse button to be clicked.

```
;*** Filling surfaces with the blitter ***

;Part 1:

;Draw filled triangle

;Set starting value

move.l Planeadr,a0                  ;Set constant parameters for
move.w #Planewidth,a1               ;the LineDraw routine
move.w #$ffff,a2                    ;Mask to $FFFF -> no pattern

;* Draw border lines *

;Line from 320,10 to 600,200
move.w #320,d0
move.w #10,d1
move.w #600,d2
move.w #200,d3
bsr.L  drawline                     ;Draw line

;Line from 319,10 to 40,200
move.w #319,d0
move.w #10,d1
move.w #40,d2
move.w #200,d3
bsr.L  drawline                     ;Draw line

;* Fill surface *

;Wait until blitter has drawn last line

Wline: btst #14,dmaconr(a5)         ;Test BBUSY
bne.S  Wline

add.l  #Planesize-2,a0              ;Address of last word
move.w #$09f0,bltcon0(a5)           ;USEA and D, LFx: D = A
move.w #$000a,bltcon1(a5)           ;Inclusive Fill plus Descending
move.w #$ffff,bltafwm(a5)           ;Set first and last word mask
move.w #$ffff,bltalwm(a5)
move.l a0,bltapth(a5)               ;Address of last word of bit-
move.l a0,bltdpth(a5)               ;plane to address register
move.w #0,bltamod(a5)               ;No modulo
move.w #0,bltdmod(a5)
move.w #$ff*64+40,bltsize(a5)       ;Start blitter

;Wait for mouse button
```

```
end:  btst #6,ciaapra                  ;Mouse button pressed?
bne.S end                              ;No -> continue

;End of Part 1.

;Part 2:

;Octant table with SING =1 and LINE =1:

Octant_table:

 dc.b 0 *4+3 ;y1<y2, x1<x2, dx<dy = Oct6
 dc.b 4 *4+3 ;y1<y2, x1<x2, dx>dy = Oct7
 dc.b 2 *4+3 ;y1<y2, x1>x2, dx<dy = Oct5
 dc.b 5 *4+3 ;y1<y2, x1>x2, dx>dy = Oct4
 dc.b 1 *4+3 ;y1>y2, x1<x2, dx<dy = Oct1
 dc.b 6 *4+3 ;y1>y2, x1<x2, dx>dy = Oct0
 dc.b 3 *4+3 ;y1>y2, x1>x2, dx<dy = Oct2
 dc.b 7 *4+3 ;y1>y2, x1>x2, dx>dy = Oct3
```

11.7.9 Sound Output

Fundamentals of electronic music

All sounds, whether music, noise or speech, occur in the form of oscillations in the air; these are the sound waves that reach our ears. A normal musical instrument creates these oscillations either directly, in which the air blown through it is made to oscillate (e.g., a flute) or indirectly, where part of the instrument creates the tone (oscillation) and then the air picks it up (e.g., string instruments).

An electronic instrument creates oscillations in its circuits that correspond to the desired sound. These oscillations aren't audible until they are converted to sound waves by a loud-speaker. On the Amiga the speaker built into the monitor is normally used. Unfortunately, because of its size and quality, it is not capable of high-fidelity translation of the electrical oscillations into sound waves. Therefore, you should connect your Amiga to a good amplifier/speaker system to get the full effect of its musical capabilities. What parameters determine the sound that comes from the computer?

Frequency

The first is the frequency of a sound. It determines whether the pitch sounds high or low. The frequency is actually the number of oscillations per second, measured in Hertz (Hz). One oscillation per second is 1 Hz, and a kilohertz is 1000 Hz. The human ear can discern sounds between 16 and 16000 Hz. Those who know something about music know that the standard A has a frequency of 440 Hz. The connection between frequency and pitch is as follows: With each octave the frequency doubles. The next higher A has a frequency of 880 Hz, while the A on the octave below the standard has a frequency of 220 Hz.

The frequency of a tone does not have to be constant. For example, it can periodically vary around the actual pitch by a few Hz, creating an effect called vibrato.

Volume

The second parameter of a sound is its volume. By volume we mean the amplitude of the oscillation. The volume of a sound is measured in deciBels (dB). The range of human hearing is about 1dB to 120 dB. Each increase of about 10 dB doubles the audible volume. The volume of sound is also called sound pressure or intensity.

The volume can be influenced by many parameters. The simplest is naturally the volume control on the monitor or amplifier. This simply changes the amplitude of the electrical oscillations. But the distance between the listener and the speaker also has an effect on the volume. The farther you are from the speaker, the softer the sound becomes.

Also, the furnishings in the room, open or closed doors, etc. can also affect the amplitude of sound waves. Therefore, the absolute volume is not that important. More interesting is the relative volume of sounds between each other, such as whether a sound is louder or softer then its predecessor.

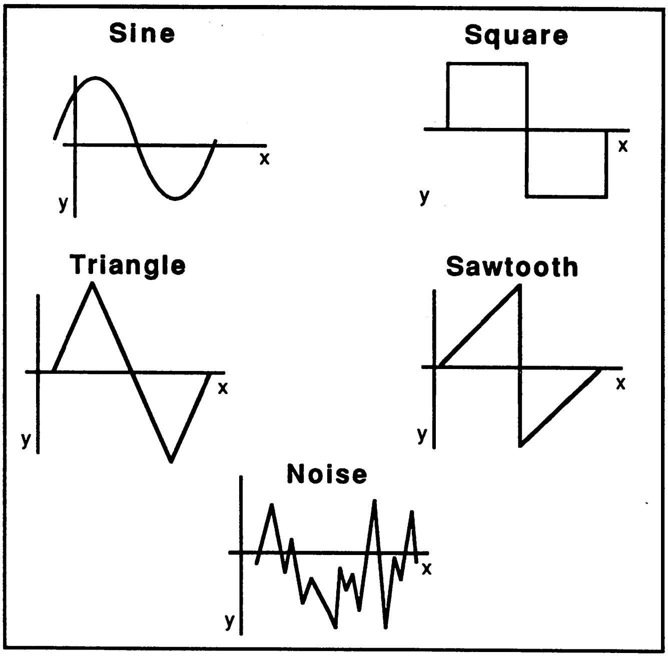
There is a relationship between the volume of a sound and its frequency. The cause of this is the sensitivity of the human ear. High and low sounds are perceived as being softer than those in the middle range, even if they physically have the same sound pressure in deciBels. This middle pitch range runs from about 1000 to 3000 Hz. The oscillations of human speech fall within this frequency range, which is probably the reason for the higher sensitivity.

The volume of a sound can also change periodically within a given range. This effect is called tremolo. However, there is the variation in volume from the start to the end of a sound. A sound can start out loud and then slowly die out. It can also start out loud, then drop a certain amount and stop abruptly. Or it starts softly and then slowly becomes louder. There are almost no limits to the possible combinations here.

Tone color or timbre

The third and last parameter of a sound is somewhat more complicated. This is the timbre or tone color, and it plays an important role. There are hundreds of different instruments which can all play a sound with the same frequency and volume, but still they sound different from one another. The reason for this lies in the shape of the oscillation. The following figure shows four common waveforms. Why do they sound different?

Each waveform, regardless of what it looks like, can be represented as a mix of sine waves of different frequencies having a fixed relationship to each other. For a square wave, the first wave (or harmonic) has the fundamental frequency of the sound, the second harmonic has three times the fundamental frequency but only a third of the amplitude. The third harmonic has five times the frequency but a fifth of the amplitude, and so on.

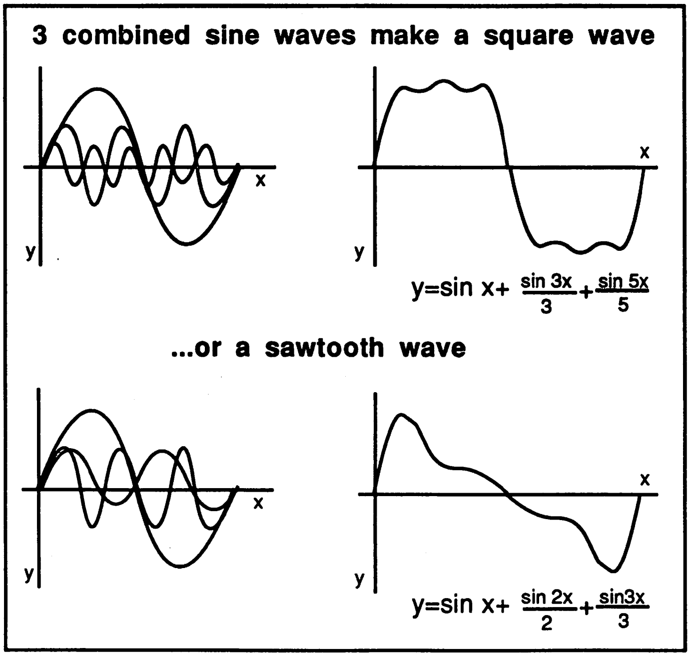


Waveforms

The next figure shows this for a square wave and a sawtooth wave. For the sake of simplicity only the first three harmonics of each waveform are shown.

As we said, all periodic waveforms can be represented as sums of sine waves. This is called the harmonic series of a sound. The pure sine wave consists only of the fundamental frequency. A square wave consists of an infinite number of harmonics. The number of harmonics and their frequency and amplitude relationship determine the timbre of a sound.

The harmonic series is important because the human ear reacts only to sine waves. A sound whose waveform deviates from a pure sine wave is divided into its harmonics by the ear. You should keep these facts in mind when reading the following discussion.

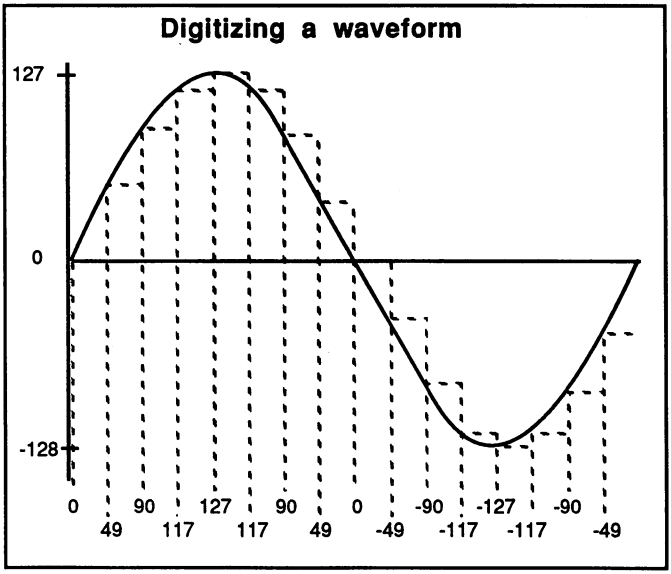


Waveforms

Noise

In addition to pitched tones there are also noises. While you can define a tone very precisely and also create it electronically, this is much more difficult for noises. They have neither a given frequency nor a defined amplitude variation and no actual waveform. They represent an arbitrary combination of sound events. The basis of many noises is called white noise, which is a mix of an infinite number of sounds whose frequencies and phases have no definite relationship to each other. The wind produces this sound, for example, because millions of air molecules are put into oscillation as they collide with one another or with objects on the earth's surface. These random oscillations make up an undefinable mixture of sounds we know as the rustling of wind.

Sound creation on the Amiga



Digital waveforms

The main criterion for judging the acoustical capabilities of a computer is its versatility. All three parameters of a sound (i.e., frequency, volume and timbre) should be able to be adjusted independently.

The Amiga's developers tried to achieve this goal as nearly as possible. Not to be limited to predefined waveforms, the digital equivalent of the desired waveform is stored in memory and then converted to the corresponding electrical oscillation by a digital-to-analog converter. In other words, the oscillation is digitized and stored in the computer. During output, the digitized data is converted back to analog form and sent to the amplifier.

In order to convert the waveforms to a form understandable to the computer, their patterns must be represented by numbers. To do this, divide one cycle of the desired waveform into an even number of equal-sized sections. Begin as close as possible to a point where the wave intersects the X axis. For each of the sections, put the corresponding Y value into memory. This produces a sequence of numbers whose elements represent snapshots of the wave at given points in time. These digitized values are called samples.

On output, the Amiga converts the number values from memory back into the corresponding output voltages. But since the wave is divided into a limited number of samples by the digitization, the output curve can be reconstructed only with this number of voltages. This results in the staircase form of the wave shown in the previous figure.

The quality of sounds reproduced in this way as opposed to their original waveforms depends essentially on two quantities:

One is the resolution of the digitized signals. This is the value range of the samples. On the Amiga this is eight bits, or from -128 to +127. Each input value can take one of 256 possible values in memory. Since the resolution of analog signals is theoretically unlimited, but that of the individual samples is limited, conversion errors result. These are called quantization or rounding errors. When the input value lies somewhere between two numbers (it doesn't correspond exactly to one of the 256 digital steps), it is rounded up or down. The maximum possible quantization error is 1/256 of the digitized value (also called an error of 1 LSB).

A factor called the quantization noise is tied to the quantization error. As the name indicates, this reveals itself as noise matching the magnitude of the quantization error.

A value range of eight bits allows moderately good reproduction of the original wave. However, higher resolution is needed for high-fidelity reproduction. For example, a CD player works with 16 bits.

The second parameter for the quality of digitized sound is the sampling rate. This is the number of samples per second. Naturally, a higher number of samples results in better reproduction. The sampling rate can be set within certain bounds on the Amiga. First you must consider how many samples are used per digitized cycle of the waveform. In our example this is 16 values. There is little audible difference between the resulting staircase waveform and a normal sine signal.

The output of the digitized sound

Once the desired waveform has been converted to the corresponding numbers and written into memory, you naturally want to hear it. The Amiga has four sound channels, which all work according to the following principle:

A digitized wave is read from memory through DMA and output through a digital/analog converter. This process is repeated continually so that the single cycle of the waveform creates a continuous tone. Channels 0 and 3 are sent to the left stereo channel, while 1 and 2 are sent to the right.

Each audio channel has its own DMA channel. Since the DMA on the Amiga is performed on words, two samples are combined into one data word. For this reason you always need an even number of samples. The upper half of the word (bits 8-15) is always output before the lower half (bits 0-7).

The data list for our digitized sound wave (where "Start" is the starting address of the list in chip RAM) looks as follows in memory:

```
Start:
dc.b 0,49               ;1st data word, samples 1 and 2
dc.b 90,117             ;2nd data word, samples 3 and 4
dc.b 127,117            ;3rd data word, samples 5 and 6
dc.b 90,49              ;4th data word, samples 7 and 8
dc.b 0,-49              ;5th data word, samples 9 and 10
dc.b -90,-117           ;6th data word, samples 11 and 12
dc.b -127,-117          ;7th data word, samples 13 and 14
dc.b -90,-49            ;8th data word, samples 15 and 16
End:
```

The digital/analog converter requires the samples to be stored as signed 8-bit numbers. In digital technology, they must appear in two's complement form. The assembler accomplishes this conversion for us, so the negative values can be written directly in the data list.

Now you must select one of the four channels over which to output the tone. The corresponding DMA channel must then be initialized. Five registers per channel set the operating parameters. The first two form an address register pair, which you should recognize from the other DMA channels. They are called AUDxLCH and AUDxLCL, or together AUDxLC, where x is the number of the DMA channel:

Reg.	Name	Function	
$0A0	AUD0LCH	Pointer to the audio data	Bits 16-20
$0A2	AUD0LCL	for channel 0	Bits 0-15
$0B0	AUD1LCH	Pointer to the audio data	Bits 16-20
$0B2	AUD1LCL	for channel 1	Bits 0-15
$0C0	AUD2LCH	Pointer to the audio data	Bits 16-20
$0C2	AUD2LCL	for channel 2	Bits 0-15
$0D0	AUD3LCH	Pointer to the audio data	Bits 16-20
$0D2	AUD3LCL	for channel 3	Bits 0-15

The initialization of these address pointers can be accomplished with a MOVE.L command:

```
LEA $DFF000, A5              ;Base address of custom chips to A5
MOVE.L #Start, AUD0LCH(A5)   ;Write "Start" in AUD0LC
```

Next, the DMA controller must be told the length of the digitized cycle (i.e., how many samples it comprises). The appropriate registers are the AUDxLEN registers:

Reg.	Name	Function
$0A4	AUD0LEN	Number of audio data words for channel 0
$0B4	AUD1LEN	Number of audio data words for channel 1
$0C4	AUD2LEN	Number of audio data words for channel 2
$0D4	AUD3LEN	Number of audio data words for channel 3

The length is specified in words, not bytes. The number of bytes must be divided by two before being written to the AUDxLEN register.

The AUDxLEN register can be initialized with the following MOVE command. To avoid having to count all the words, two labels are defined: "Start" is the starting address of the data list, "End" the end address+1 (see the previous example data list). The base address of the custom chips ($DFF000) is stored in A5:

```
MOVE.W #(End-Start)/2, AUD0LEN(A5)
```

Now comes the volume of the sound. On the Amiga the volume for each channel can be set separately. A total of 65 levels are available, ranging from 0 (inaudible) to 64 (full volume). The corresponding registers are called AUDxVOL:

Reg.	Name	Function
$0A8	AUD0VOL	Volume of audio channel 0
$0B8	AUD1VOL	Volume of audio channel 1
$0C8	AUD2VOL	Volume of audio channel 2
$0D8	AUD3VOL	Volume of audio channel 3

Let's set our audio channel to half volume:

```
MOVE.W #32, AUD0VOL(A5)
```

The last parameter is the sampling rate. This determines how often a data byte (sample) is sent to the digital/analog converter. The sampling rate determines the frequency of the sound. As explained initially, the frequency equals the number of oscillations (cycles) per second. An oscillation consists of an arbitrary number of samples. In our example it is 16. If the sampling rate represents the number of samples read per second, the frequency of the sound corresponds to the sampling rate divided by the number of samples per cycle:

$$\text{Frequency} = \frac{\text{Sampling rate}}{\text{Samples per cycle}}$$

Unfortunately the sampling rate cannot be specified directly in Hertz. Instead, the DMA controller wants to know the number of bus cycles desired between the output of two samples. A bus cycle takes exactly 279.365 nanoseconds (billionths of a second) or 2.79365 * 10-7 seconds.

To get from the sampling rate to the number of bus cycles, first take the inverse of the sampling rate. This gives you the duration of the sample. Dividing this value by the duration of a bus cycle in seconds yields the number of bus cycles between two samples, called the sample period:

$$\text{Sample period} = \frac{1}{\text{Sampling rate} * 2.79365 * 10\text{-}7}$$

Let's assume that we want to play our sample tone with a frequency of 440 Hz, the standard A. The sampling rate is computed as follows:

Sampling rate = Frequency * Samples per cycle
Sampling rate = 440 Hz * 16 = 7040 Hz

We quickly obtain the required sample period by inserting the appropriate values:

$$\text{Sample period} = \frac{1}{7040 * 2.79365 * 10\text{-}7} = 508.4583$$

Since only integral values can be specified for the sample period, we round the result to 508. As a result, the output frequency is not exactly 440 Hz, but the deviation is minimal, namely 0.4 Hz.

The sample period can theoretically be anything between 0 and 65535. However, the actual range has an upper limit. As can be gathered from the figure in the "Fundamentals" section, each audio channel has one DMA slot per raster line (i.e., one data word (two samples) can be read from memory in each raster line). The smallest possible value for the sample period is 124. The sample frequency for this value is 28867 Hz. If the sample period is made shorter than 124, a data word can be output twice because the next one cannot be read in time.

The sample period registers are called AUDxPER:

Reg.	Name	Function
$0A6	AUD0PER	Sample period for audio channel 0
$0B6	AUD1PER	Sample period for audio channel 1
$0C6	AUD2PER	Sample period for audio channel 2
$0D6	AUD3PER	Sample period for audio channel 3

MOVE.W #508,AUD0PER(A5) puts the sampling rate we calculated into the AUD0PER register. Now all the registers for audio channel 0 have been supplied with the proper values for our sound. To make it audible, we still have to enable the DMA access for audio channel 0. Four bits in the DMACON register are responsible for the audio DMA channels:

DMACON bit no.	Name	Audio DMA channel no.
3	AUD3EN	3
2	AUD2EN	2
1	AUD1EN	1
0	AUD0EN	0

To enable the audio DMA for channel 0, we set the AUD0EN bit to 1. To be on the safe side, the DMAEN bit should be set along with it (see "Fundamentals"):

```
MOVE.W #$8201, DMACON(A5)         ;Set AUD0EN and DMAEN
```

Now the DMA starts to fetch the audio data from memory and output it through the digital/analog converter. The sound can be heard through the speaker. To turn it off again, simply set AUD0EN = 0.

Whenever AUDxEN is set to 1, the DMA starts at the address in AUDxLC. There is one exception: If the DMA channel was on (AUDxEN = 1) and the bit is briefly cleared and then set back to 1 without the DMA channel reading a new data word in the meantime, the DMA controller continues with the old address.

Audio interrupts

The audio DMA always starts with the data byte at the address in AUDxLC. Once the number of data words specified in AUDxLEN have been read from memory and output, the DMA starts over at the AUDxLC address. In contrast to the address registers for the Blitter or the bit-planes, the content of the AUDxLC register is not changed during the audio DMA. There is an additional address register for each audio channel. Before the DMA controller gets the first data byte from memory, it copies the value from the AUDxLC register to this internal address register.

It also transfers the AUDxLEN register value into an internal counter. When this happens, an interrupt is generated. As you may recall from the section on interrupts, there is a separate interrupt bit for each of the four audio channels. The level 4 processor interrupt is reserved exclusively for these bits.

While the DMA controller now reads data words from memory, the processor can supply AUDxLC and AUDxLEN with new data, since the values of both registers are stored internally. Not until the counter that was initially loaded with the value from AUDxLEN reaches 0 will the data from AUDxLC and AUDxLEN be read again.

The processor then has enough time to change the values of the two registers, if necessary. This allows uninterrupted sound output.

An interrupt is generated after each complete cycle. This means that for high frequency sounds interrupts occur very often. The interrupt enable bits (INTEN) for the audio interrupts should be set only when they are actually needed, or the processor may not be able to save itself from all the interrupt requests.

Modulation of volume and frequency

To create certain sound effects, it is possible to modulate the frequency and/or volume. One of the DMA channels acts as a modulator which changes the corresponding parameters of another channel. This can be done very simply: The modulation oscillator fetches its data from memory as usual, but instead of sending it to the digital/analog converter, it writes it to the volume or frequency register of the oscillator that it modulates (AUDxVOL or AUDxLEN). It can also affect both registers at once. In this case the data words read from its data list are written alternately to the AUDxVOL and AUDxLEN registers. The data words have the same format as their destination registers:

Volume:	Bits 7-15	Unused
	Bits 0-6	Volume value between 0 and 64
Frequency:	Bits 0-15	Sample period

The following shows the use of data words of the modulation oscillator for all three possible cases:

Data word No.	Oscillator modulates: Frequency	Volume	Frequency & volume
1	Period 1	Volume 1	Volume 1
2	Period 2	Volume 2	Period 1
3	Period 3	Volume 3	Volume 2
4	Period 4	Volume 4	Period 2

To activate an audio channel as a modulator, you must set the corresponding bit or bits in the audio disk control register (ADKCON). Each channel can modulate only its successor: channel 0 modulates channel 1, channel 1 modulates channel 2, and channel 2 modulates channel 3. Channel 3 can also be set as a modulator, but its data words are not used to modulate another channel and are lost. If an audio channel is used as a modulator, its audio output is disabled.

The ADKCON register contains, as its name suggests, control bits for the disk controller in addition to the audio circuitry. The disk controller bits are explained in more detail in another section.

ADKCON register $09E (write) $010 (read)

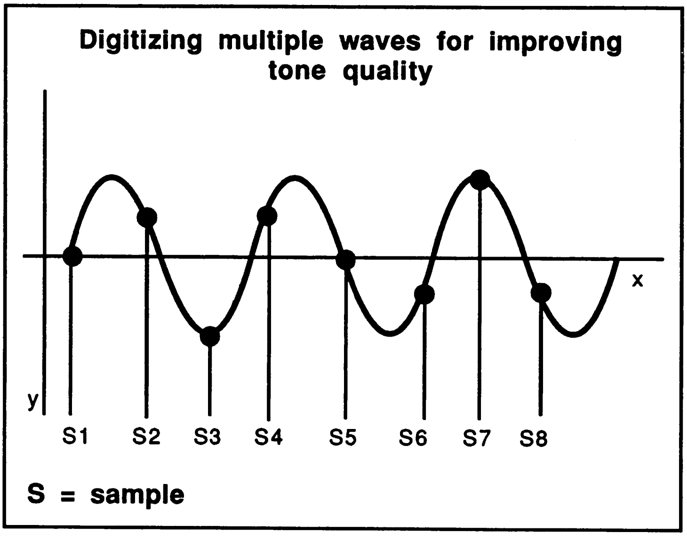
Bit no.	Name	Function
15	SET/CLR	Bits are set (SET/CLR=1) or cleared
14-8		Used by the disk controller
7	USE3PN	Audio channel 3 modulates nothing
6	USE2P3	Audio channel 2 modulates period of channel 3
5	USE1P2	Audio channel 1 modulates period of channel 2
4	USE0P1	Audio channel 0 modulates period of channel 1
3	USE3VN	Audio channel 3 modulates nothing
2	USE2V3	Audio channel 2 modulates volume of channel 3
1	USE1V2	Audio channel 1 modulates volume of channel 2
0	USE0V1	Audio channel 0 modulates volume of channel 1

To recap: If a channel is used for modulation, its data words are simply written into the corresponding register of the modulated channel. In other respects the two operate completely independently of each other.

Problems of digital sound generation on the Amiga

In our example we defined a cycle with 16 samples. The maximum sampling rate is 28867 Hz. This yields a maximum frequency of 28867 / 16 = 1460.4 Hz. This is close to a third-octave F sharp (1480 Hz).

If you want to go higher, you must decrease the number of samples per cycle. If we define our sine with half the samples, the maximum frequency increases to 3020.8 Hz. However, eight data bytes aren't enough for a good sine wave. For yet higher pitches, the number of samples decreases even more. For 6041.6 Hz there are only four. Waveforms can barely be recognized with just four samples.



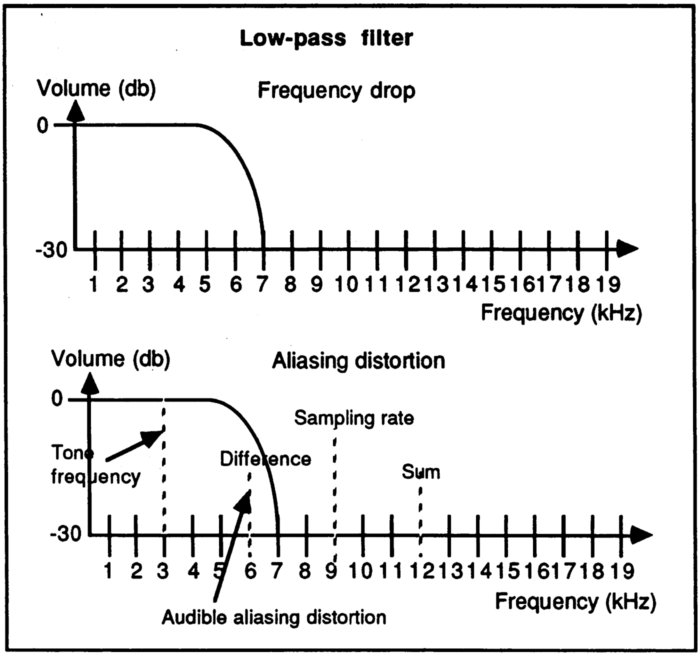
Digitizing multiple waves for improving tone quality

Cycles

However, this isn't very noticeable when heard. The ear reacts practically the same. The higher the frequency, the more difficult it is to identify sounds. Despite this, it can improve the sound quality to use multiple cycles to define the desired waveform at high frequencies.

The maximum frequency of the Amiga sound output is limited by another factor. When converting the digital sound data back to analog, two undesired interference frequencies occur due to interactions between the sampling rate and the desired sound frequency. One of these is the sum of the sampling rate and frequency and the other is their difference. This phenomenon is called "aliasing distortion."

For example, with a 3 kHz sound and a 12 kHz sampling rate, the difference is 9 kHz and the sum 15 kHz.



The low-pass filter

In order to eliminate the alias frequencies, a device called a low-pass filter has been placed between the output of the digital/analog converter and the audio connectors. All frequencies up to 4 kHz pass through undisturbed. Between 4 and 7 kHz the signal is weakened, until above 7 kHz nothing is allowed to pass. For example, the 3 kHz tone is not affected by the low-pass filter, but both the sum and the difference frequencies of 9 and 15 kHz lie above the filter's cut-off frequency of 7 kHz and cannot pass through. Also, they are not heard through the speaker. If you try to output the same 3 kHz tone with a sampling rate of 9 kHz, the difference frequency of 6 kHz (9 kHz - 3 kHz) is diminished by the filter but still passes through it.

To be sure that the difference frequency always lies above the cut-off frequency of the filter, we must observe the following rule:

```
Sampling rate > highest frequency component + 7 kHz
```

It is not enough to ensure that the difference between the sampling frequency and the desired output frequency is greater than 7 kHz. If a waveform with many harmonics is used, each of the harmonics produces its own difference frequency with the sampling rate. This is why the highest frequency of the waveform must be used in the previous expression.

Not only does the low-pass filter hold back the aliasing distortion, it also limits the frequency range of the Amiga. To be sure, tones with a fundamental frequency between 4 and 7 kHz rarely occur in a musical piece, but the harmonics of much lower fundamentals for certain waveforms lie within this range. This is especially clear for a square wave. The square waveform, as we saw earlier, consists of the combination of several sine waves having a set frequency relationship to each other. In the figure the square wave is shown to consist of just two harmonics and the fundamental tone. However, an actual square wave has an infinite number of harmonics. If the higher-order harmonics are limited or removed by the filter, a somewhat deformed square wave results. In the extreme case where the fundamental frequency of the square wave approaches the cut-off frequency of the filter, only the fundamental remains. This turns the original square wave into a sine wave.

Amplitude envelope of a sound

In addition to the waveform, the sound of an instrument is also influenced by its amplitude envelope. The Amiga can do almost anything in the area of waveforms. How are specific envelopes programmed?

The envelope of a sound can be divided into three sections: The attack, sustain, and decay phases.

As soon as the sound is played, the attack phase begins. It determines how quickly the volume rises from zero to the sustain value. During the sustain phase the sound remains at this volume. As the sound ends, it enters the decay phase, where the volume drops from the sustain value back to zero.

The amplitude curve that this process represents is generally called an envelope. How do you program such an envelope on the Amiga?

There are three possibilities:

Volume modulation

A second sound channel is used to modulate the volume of the sound. For example, channel 0 can be used to modulate channel 1. Channel 1 can continually output the desired sound with its volume set to 0.

The desired amplitude curve is divided into two parts: attack phase and decay phase. It is digitized (just like a waveform) and placed in memory in two data lists. When the sound is to be played, channel 0 is set to the address of the attack data and started. Since it modulates the volume of channel 1, the volume of the sound follows the desired attack phase exactly. When the attack phase reaches the sustain value, the data list for channel 0 has been processed. It then generates an interrupt, and the data list would normally be processed again from the beginning. The processor must react to the interrupt and turn off channel 0 by means of the AUD0EN bit in the DMACON register. Channel 1 remains at the desired sustain volume.

When the tone is to be turned off, you set channel 0 to the start of the decay data and start it again. Wait again for the interrupt, which signals that the decay phase is done, and turn channel 0 off.

The registers for channel 0 must be initialized as follows for this procedure:

USE0V1 — This bit in the ADKCON register should be set to 1 so that channel 0 modulates the volume of channel 1.

AUD0LC — First set to the data list for the attack phase and then to that of the decay phase.

AUD0LEN — Contains, depending on the address in AUD0LC, the length of either the attack or decay data.

AUD0VOL — Has no function here, since the audio output of channel 0 is turned off.

AUD0PER — The content of the AUD0PER register determines the speed at which the volume data is read from memory. This can be used to set the length of the attack/decay phase.

This method allows the desired envelope to be constructed perfectly. Unfortunately, it also has a big disadvantage: Two audio channels are required for one sound. If you want four different sound channels, you have to use an alternate method:

Controlling volume with the processor

The desired envelope is placed in memory as previously described. However, this time the processor changes the volume. It fetches the current volume from memory at regular intervals and writes it to the volume register of the corresponding sound channel.

The program must be run as an interrupt routine. This can be done in the vertical blanking interrupt or one of the timer interrupts from CIA-B can be used.

The disadvantage of this method is the amount of processor time that it requires, since the volume control is not performed by DMA. Since the amount of time needed is reasonably limited, this is usually the best method for most applications.

Constructing the envelope in the sample data

This method is best for short sounds or sound effects. Instead of digitizing just one cycle of the desired waveform, write the entire sound into memory. A program can calculate it, or you can use an audio digitizer, which performs hardware digitizing of sound with a microphone and analog/digital converter.

Several companies offer such devices for use with the Amiga. Once the data is in the Amiga, it can be played back at any pitch or speed. This allows complex effects, such as laughter or screams, to be reproduced by the Amiga with considerable accuracy.

This method also has its disadvantages: It involves either difficult calculations or additional hardware to put the complete sound in digitized form into memory. In addition, this method requires large amounts of memory. For example, if the sound is 1 second long with a sampling rate of 20 kHz, the sound data takes up 20K.

Tips, tricks and more

Sound quality

The value range of the digital data is from -128 to 127. This range should be used as fully as possible. It is best when the amplitude of the digital waveform equals 256.

Otherwise the sound quality deteriorates audibly, since a decrease in the range means relatively greater quantization error and noise that can quickly reach distortional proportions.

For this reason you should avoid using the amplitude of the digitized sound to control the volume. Each channel has its own AUDxVOL register for volume control. If the volume is reduced with this register, the relationship between the desired sound and the distortion remains the same and the Amiga's high sound quality is preserved.

Changing waveforms smoothly

To avoid annoying crackling or jumps in volume when changing waveforms, remember the following rules:

Each cycle should be digitized from zero-point to zero-point (i.e., it should start and end at a point where the waveform crosses the X axis).

If you follow this rule, all waveforms in memory have the same starting and ending value, namely zero. In transitions between consecutive waveforms of different shapes, there are no sudden level jumps which would be heard as noise.

Secondly, you should make sure that the total volumes of the two cycles are approximately equal. Volume refers to the effective value of the waveform. The effective value is equal to the amplitude of a square wave signal whose surface under the curve is exactly as large as that of the waveform.

This effective value determines the volume of an oscillation. Only for a square wave does it equal the amplitude. If you change from one waveform to another with a higher effective value, the second sounds louder than its predecessor.

The effective value of a cycle can be easily calculated from its digitized data:

You add up the values of all the data bytes and divide the result by the number of data bytes.

If you want to fully utilize the 8-bit value range of the digital/analog converter for all waveforms, the effective values will not always match. The volume must be adjusted accordingly with the AUDxVOL register when changing waveforms.

Playing notes

Normally a piece of music is written out in the form of notes. If you want to play it on the Amiga, you must convert the notes to the appropriate sample periods. To minimize the amount of calculation, it is generally best to use a table containing the sample period values for all the half-tones in an octave:

Table of sample period values for musical notes (for AUDxLEN = 16):

Note	Frequency [Hz]	Sample period	
C	261.7	427	(262.0)
C#	277.2	404	(276.9)
D	293.7	381	(293.6)
D#	311.2	359	(311.6)
E	329.7	339	(330.0)
F	349.3	320	(349.6)
F#	370.0	302	(370.4)
G	392.0	285	(392.5)
G#	415.3	269	(415.8)
A	440.0	254	(440.4)
A#	466.2	240	(466.0)
B	493.9	226	(495.0)
C	523.3	214	(522.7)

A value in parentheses represents the actual frequency for the corresponding sampling period. The frequency of a half-tone is always greater than its predecessor by the factor "twelfth root of 2." Thus 440 (A) * 2(1/12) = 466.2 (A#), 466.2 (A#) * 2(1/12) = 493.9 (B), etc.

An octave always corresponds to a doubling of frequency.

If you want to play a note from an octave that is not in the table, there are two options:

1. Change the sampling period. For each octave up the value must be halved. An octave lower means doubling the sampling period. This is simple, but one soon runs into certain limits. With a data field of 32 bytes (AUDxLEN = 16), as in our table, the smallest possible sampling period (124) is reached with the second A. The data list must be reduced in size.

   In this case you get problems with lower tones since the aliasing distortion then becomes audible.

A better solution is procedure 2:

2. Create a separate data list for each octave. The sampling period value remains the same for every octave. It is used only to select the half-tone. If a tone from an octave above that in the table is required, you use a data list that is only half as long. Correspondingly, a list twice as long is used for the next lower octave.

   The normal musical range comprises eight octaves, meaning that you need eight data lists per waveform.

In return for the extra work this method involves, you always get the optimal sound regardless of pitch.

Creating higher frequencies

The minimal sampling period is normally 124. The reason for this is that the audio DMA is not able to read the data words fast enough to support a shorter sampling period. The old data word is then output more than once. This effect can be used to our advantage. Since the data word read contains two samples, a high frequency square wave can be created with it. With a sampling period of 1 you get a sampling frequency of 3.58 MHz and an output frequency of 1.74 MHz. To be able to use this high frequency output signal, you must intercept it before it reaches the low-pass filter. The AUDIN input (pin 16) of the serial connector (RS232) allows you to do this. It is connected directly to the right audio output of Paula (see the section on interfaces).

In order to create such high frequencies, AUDxVOL must be set to the maximum volume (AUDxVOL = 64).

Playing polyphonic music

Since the Amiga has four independent audio channels, four different sounds can be created at once. This allows any four-voice musical pieces to be played directly.

But there can be more. Just because there are four audio channels doesn't mean that four voices is the maximum. As we mentioned, each waveform is actually a combination of sine signals. Just as these harmonics together make up the waveform, you can also combine multiple waveforms into a multi-voiced sound. The output signals for audio channels 0 and 3 are mixed together into one stereo channel inside Paula. The waveforms of both channels are combined into a single two-voiced channel.

The same thing that's done electronically with analog signals can be done by computation with digital data. Simply add the digital data of two completely different waveforms and output the new data to the audio channel as usual. Now you have two voices per audio channel. Theoretically, any number of voices can be played over a sound channel in this manner.

In practice the number of voices is limited by the speed of computation, but 16 voices are certainly possible.

Calculating the summed signal from the components is very simple. At each point in time the current values of all the sounds are added and the result is divided by the number of voices. This is how a square signal results from combining sine waves with the right relative frequencies.

Audio output without DMA

Like all DMA channels, the audio DMA channels have registers to which they write data and where data can also be written by the processor:

The audio data registers

Reg.	Name	Function
$0AA	AUD0DAT	These four registers always contain the current
$0BA	AUD1DAT	audio data word, consisting of two samples.
$0CA	AUD2DAT	The sample in the upper byte (bits 8-15)
$0DA	AUD3DAT	is always output first.

For the processor to be able to write to the audio data registers, the DMA must be turned off with AUDxEN = 0. This also changes the creation of audio interrupts. They will now always occur after the output of the two samples in the AUDxDAT register instead of at the start of each audio data list.

If a new data word is not loaded into AUDxDAT in time, the last two samples are not repeated as they are for DMA operation, but the output remains at the value of the last data byte (the lower half of the word in AUDxDAT).

The direct programming of the audio data registers costs a great deal of processing time. The audio DMA should be used except in special cases.

A few facts

AUDxVOL values in deciBels (0 dB = full volume):

AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB
64	0.0	48	-2.5	32	-6.0	16	-12.0
63	-0.1	47	-2.7	31	-6.3	15	-12.6
62	-0.3	46	-2.9	30	-6.6	14	-13.2
61	-0.4	45	-3.1	29	-6.9	13	-13.8
60	-0.6	44	-3.3	28	-7.2	12	-14.5
59	-0.7	43	-3.5	27	-7.5	11	-15.3
58	-0.9	42	-3.7	26	-7.8	10	-16.1
57	-1.0	41	-3.9	25	-8.2	9	-17.0
56	-1.2	40	-4.1	24	-8.5	8	-18.1
55	-1.3	39	-4.3	23	-8.9	7	-19.2
54	-1.5	38	-4.5	22	-9.3	6	-20.6
53	-1.6	37	-4.8	21	-9.7	5	-22.1
52	-1.8	36	-5.0	20	-10.1	4	-24.1
51	-2.0	35	-5.2	19	-10.5	3	-26.6
50	-2.1	34	-5.5	18	-11.0	2	-30.1
49	-2.3	33	-5.8	17	-11.5	1	-36.1

AUDxVOL = 0 corresponds to a dB value of minus infinity. If AUDxVOL = 64, then a digital value of 127 corresponds to an output voltage of about 400 millivolts, and -128 corresponds to -400 millivolts. A change of 1 LSB causes about a 3 millivolt variation in the output voltage.

Example programs

Program 1: Creating a simple sine wave

This program creates a sine wave tone with a frequency of 440 Hz. The sample table presented in the text is used. The largest portion of the program is used to allocate chip RAM for the audio data list. The sound is produced over channel 0 until the mouse button is pressed. The program then releases the occupied memory.

```
;*** Create a simple sinewave ***

;Custom chip registers
intena = $9A        ;Interrupt enable register (write)
dmacon = $96        ;DMA control register (write)

;Audio-Register
aud0lc = $A0        ;Address of audio data list
aud0len = $A4       ;Length of audio data list
aud0per = $A6       ;Sampling period
aud0vol = $A8       ;Volume
adkcon = $9E        ;Control register for modulation

;CIA-A Port register A (mouse button)
ciaapra = $bfe001

;Exec Library Base Offsets
AllocMem = -30-168      ;ByteSize,Requirements/d0,d1
FreeMem = -30-180       ;MemoryBlock,ByteSize/a1,d0

;Other labels
ALsize = ALend - ALstart   ;Length of audio data list
Execbase = 4
chip = 2            ;Allocate chip RAM

;*** Initialization ***

start:
;Allocate memory for audio data list
 move.l  Execbase,a6
 moveq   #ALsize,d0      ;Size of audio data list
 moveq   #chip,d1
 jsr     AllocMem(a6)    ;Allocate memory
```

```
 beq      Ende               ;Error -> End program

;Copy audio data list in chip RAM
 move.l  d0,a0              ;Address in chip RAM
 move.l  #ALstart,a1        ;Address in program
 moveq   #ALsize-1,d1       ;Loop counter

Loop:     move.b (a1)+,(a0)+ ;Data list in chip RAM
 dbf      d1,Loop

;*** Main program
;Initialize audio registers
 lea      $DFF000,a5
 move.w  #$000f,dmacon(a5) ;Audio DMA off
 move.l  d0,aud0lc(a5)      ;Set address of data list
 move.w  #ALsize/2,aud0len(a5) ;Length in words
 move.w  #32,aud0vol(a5)    ;Half volume
 move.w  #508,aud0per(a5)   ;Frequency: 440 Hz
 move.w  #$00ff,adkcon(a5) ;Disable modulation

;Enable audio DMA
 move.w  #$8201,dmacon(a5) ;Channel 0 on

;Wait for a mouse button
wait:  btst #6,ciaapra
 bne      wait

;Disable audio DMA
 move.w  #$0001,dmacon(a5) ;Channel 0 off

;*** End of program ***
 move.l  d0,a1              ;Address of data list
 moveq   #ALsize,d0         ;Length
 jsr      FreeMem(a6)       ;Release assigned memory

Ende: clr.l d0
 rts
;Audio data list

ALstart:
 dc.b     0,49
 dc.b     90,117
 dc.b     127,117
 dc.b     90,49
 dc.b     0,-49
 dc.b     -90,-117
 dc.b     -127,-117
 dc.b     -90,-49
```

```
ALend:
;Program end
 end
```

Program 2: Sine wave tone with vibrato

This program is an extension of the previous one. The same sine wave tone is output, but this time over channel 1. Channel 0 modulates the frequency of channel 1 and creates the vibrato effect. The data for the vibrato represents a digitized sine wave whose zero point has the value of the sampling period of a standard A (i.e., 508).

```
;*** Creating a vibrato ***

;Custom chip register

INTENA = $9A ;Interrupt enable register (write)
DMACON = $96 ;DMA control register (write)

;Audio registers

AUD0LC  = $A0 ;Address of audio data list
AUD0LEN = $A4 ;Length of audio data list
AUD0PER = $A6 ;Sampling period
AUD0VOL = $A8 ;Volume

AUD1LC  = $B0
AUD1LEN = $B4
AUD1PER = $B6
AUD1VOL = $B8

ADKCON  = $9E ;Control register for modulation

;CIA-A Port register A (mouse button)

CIAAPRA = $bfe001

;Exec Library Base Offsets

AllocMem = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem  = -30-180 ;MemoryBlock,ByteSize/a1,d0

;Other labels

Execbase = 4
chip     = 2    ;Allocate chip RAM
Vibsize = Vibend - Vibstart ;Length of vibrato table
```

```
ALsize = ALend - ALstart ;Length of audio data list
Size = ALsize + Vibsize     ;Total length of both lists


;*** Initialization ***
start:

;Allocate memory for data lists

 move.l  Execbase,a6
 move.l  #Size,d0       ;Length of both lists
 moveq   #chip,d1
 jsr     AllocMem(a6)   ;Allocate memory
 beq     Ende

;Copy audio data list in chip RAM

 move.l  d0,a0          ;Address in chip RAM
 move.l  #ALstart,a1    ;Address in program
 move.l  #Size-1,d1     ;Loop counter

Loop:    move.b (a1)+,(a0)+ ;Lists in chip RAM
 dbf     d1,Loop

;*** Main program

;Initialize audio registers

 move.l  d0,d1          ;Audio data list address
 add.l   #ALsize,d1     ;Address of vibrato table
 lea     $DFF000,a5
 move.w  #$000f,dmacon(a5) ;Audio DMA off
 move.l  d1,aud0lc(a5)  ;Set to  vibrato table
 move.w  #Vibsize,aud0len(a5) ;Length of vibrato table
 move.w  #8961,aud0per(a5)      ;Vibrato frequency

 move.l  d0,aud1lc(a5)  ;Channel 1 from audio data list
 move.w  #ALsize,aud1len(a5) ;Length of audio data list
 move.w  #32,aud1vol(a5)       ;Half volume

 move.w  #$00FF,adkcon(a5) ;Disable other modulation
 move.w  #$8010,adkcon(a5) ;Channel 0 modulates period from
;channel 1
;Audio DMA on

 move.w  #$8203,dmacon(a5) ;Channels 0 and 1 on

;Wait for a mouse button
```

```
wait:    btst #6,ciaapra
 bne     wait

;Audio DMA off

 move.w  #$0003,dmacon(a5) ;Channels 0 and 1 off

;*** End program  ***

 move.l  d0,a1          ;Address of lists
 move.l  #Size,d0       ;Length
 jsr     FreeMem(a6)    ;Release memory

Ende: clr.l d0
 rts

;Audio data list

ALstart:
 dc.b    0,49
 dc.b    90,117
 dc.b    127,117
 dc.b    90,49
 dc.b    0,-49
 dc.b    -90,-117
 dc.b    -127,-117
 dc.b    -90,-49
ALend:
;Vibrato table

Vibstart:
 dc.w    508,513,518,522,524,525,524,522,518,513
 dc.w    508,503,498,494,492,491,492,494,498,503
Vibend:
;Program end
 end
```

11.7.10 Mouse, Joystick and Paddles

Mouse, joystick and paddles -- all of these can be connected to the Amiga. We'll go through them in order, together with the corresponding registers. The pin assignment of the game ports, to which all of these input devices are connected, can be found in the section on interfaces. Let's start with the mouse:

The mouse

The mouse is the most-often used input device. It's an important device for using the user-friendly interfaces of the Amiga. But how does it work and how is the mouse pointer on the screen created and moved?

If you turn over the mouse, you'll see a rubber-coated metal ball that turns when the mouse is moved. These rotations of the ball are transferred to two shafts, situated at right angles to each other so that one turns when the mouse is moved along the X axis and the other when the mouse is moved along the Y axis. If the mouse is moved diagonally, both shafts rotate corresponding to the X and Y components of the mouse movement. Unfortunately, rotating shafts don't help the Amiga when it wants to determine the position of the mouse. The mechanical movement must be converted into electrical signals.

A wheel with holes around its circumference is attached to the end of each shaft for this purpose. When it rotates it repeatedly breaks a beam of light in an optical coupler. The signal that results from this is amplified and sent out over the mouse cable to the computer. Now the Amiga can tell when and at what speed the mouse is moved. But it still doesn't know in what direction (i.e., left or right, forward or backward).

A little trick solves this problem. Two optical couplers are placed on each wheel, set opposite each other and offset by half a hole. If the disk rotates in a given direction, one light beam is always broken before the other. If the direction is reversed, the order of the two signals from the optical coupler changes accordingly. This allows the Amiga to determine the direction of the movement.

Therefore, the mouse returns four signals, two per shaft. They are called Vertical Pulse, Vertical Quadrature Pulse, Horizontal Pulse and Horizontal Quadrature Pulse.

The next figure shows the phase relationship of the horizontal pulse (H) and horizontal quadrature pulse (HQ) signals, but it also holds for the vertical signals. It's easy to see how H and HQ differ from each other depending on the direction of movement. The Amiga performs logical operations on these two signals to obtain two new signals, X0 and X1. X1 is an inverted HQ, and X0 arises from an exclusive OR of H and HQ (i.e., X0 is 1 whenever H and HQ are at different levels).

With these two signals the Amiga controls a 6-bit counter which counts up or down on X1 depending on the direction. Together with X0 and X1 an 8-bit value is formed which represents the current mouse position.

If the mouse is moved right or down, the counter is incremented. If the mouse is moved left or up, it is decremented.

Denise contains four such counters, two per game port, since a mouse can be connected to each one. They are called JOYDAT0 and JOYDAT1:

JOY0DAT $00A - JOY1DAT $00C
(mouse on game port 0) - (mouse on game port 1)

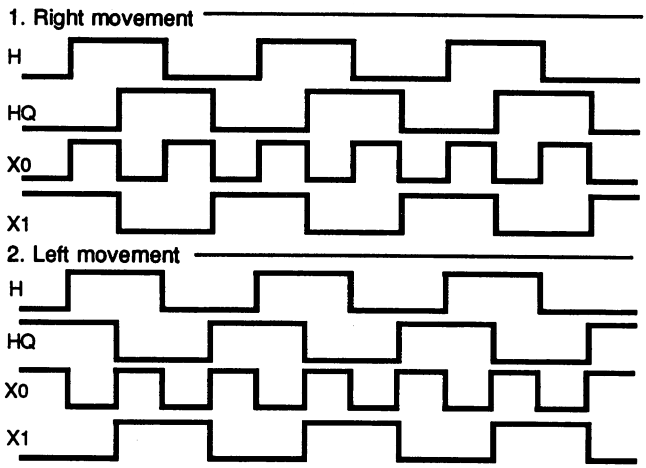
Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

Both registers are read-only.

Y0-7 Counter for vertical mouse movements (Y direction)

H0-7 Counter for horizontal mouse movements (X direction)

The mouse creates two hundred count pulses per inch, or about 79 per centimeter, which means that the limit of the mouse counter is soon reached. Eight bits yield a count range from 0 to 255. Moving the mouse over four centimeters overflows the counters. This can occur when counting up (the counter jumps from 255 to 0) as well as counting down (the counter jumps from 0 to 255). Therefore, the count registers must be read at given intervals to see if an overflow or underflow has occurred.



Truth tables for X0 and X1:

H	HQ	X0
0	0	0
0	1	1
1	0	1
1	1	0

HQ	X1
0	1
1	0

The mouse signals

The operating system usually does this during the vertical blanking interrupt. This is based on the assumption that the mouse is not moved more than 127 count steps between two successive reads. The new counter state is compared with the last value read. If the difference is greater than 127, then the counter overflowed and the mouse was moved right or down. If it's less than -127, an underflow occurred corresponding to a mouse movement left or up.

Old counter state	New counter state	Difference	Actual mouse movement	Under-/ Overflow
100	200	-100	+100	No
200	100	+100	-100	No
50	200	-150	-105	Underflow
200	50	+150	+105	Overflow
Difference = old counter state - new counter state				

If an underflow occurred, the actual mouse movement is calculated as follows:

-255 - difference, or in numbers: -255 - (50-200) = -105

For an overflow:

255 - difference, or in numbers: 255 - (200-50) = +105

A positive mouse movement corresponds to a movement right or up, a negative value to left or down.

The mouse counters can also be set through software. A value can be written to the counter through the JOYTEST register. JOYTEST operates on both game ports simultaneously, meaning that the horizontal and vertical counters of both mouse counters are initialized with the same value (JOY0DAT = JOY1DAT).

JOYTEST $036 (write-only)

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	Y7	Y6	Y5	Y4	Y3	Y2	xx	xx	X7	X6	X5	X4	X3	X2	xx	xx

As you can see, only the high-order six bits of the counters can be affected. This makes sense when you remember that the lower two bits are taken directly from the mouse signals and aren't in an internal memory location that can be changed.

The joysticks

When you look at the pin-out of the game ports, you see that the four direction lines for the joysticks occupy the same lines as those for the mouse. Therefore, it seems reasonable that they can also be read with the same registers. In fact, the joystick lines are processed exactly like the mouse signals (i.e., each pair of lines is combined into the X0 and X1 or Y0 and Y1 bits).

The joystick position can be determined from these four bits:

Joystick right	X1 = 1	(bit 1 JOYxDAT)
Joystick left	Y1 = 1	(bit 9 JOYxDAT)
Joystick backward	X0 EOR X1 = 1	(bits 0 and 1 JOYxDAT)
Joystick forward	Y0 EOR Y1 = 1	(bits 8 and 9 JOYxDAT)

In order to detect whether the joystick has been moved backward or forward, you must take the exclusive OR of X0 and X1 or Y0 and Y1, respectively. If the result is 1, the joystick is in the position. The following assembly language routine reads the joystick on game port 1:

```
TestJoystick:
 MOVE.W $DFF00C, D0          ;Move JOY1DAT to D0
 BTST #1, D0                 ;Test bit no. 1
 BNE RIGHT                   ;Set? If so, joystick right
 BTST #9, D0                 ;Test bit no. 9
 BNE LEFT                    ;Set? If so, joystick left
 MOVE.W D0,D1                ;Copy D0 to D1
 LSR.W #1,D1                 ;Move Y1 and X1 to position of Y0 and X0
 EOR.W D0,D1                 ;Exclusive OR: Y1 EOR Y0 and X1 EOR X0
 BTST #0, D1                 ;Test result of X1 EOR X0
 BNE BACK                    ;Equal 1? If so, joystick backward
 BTST #8, D1                 ;Test result of Y1 EOR Y0
 BNE FORWARD                 ;Equal 1? If so, joystick forward
 BRA MIDDLE                  ;Joystick is in middle position
```

The exclusive OR operation is performed as follows in this program:

A copy of the JOY1DAT register (previously moved to D0) is placed in D1 and is shifted one bit to the right. Now X1 in D1 and X0 in D0 have the same bit position, as do Y1 and Y0. An EOR between D0 and D1 exclusive ORs Y0 with Y1 and X0 with X1. Then all you have to do is test the result in D1 with the appropriate BTST commands.

This program does not support diagonal joystick positions.

The paddles

The Amiga has two analog inputs per game port, to which variable resistors called potentiometers can be connected. These have in each position a given resistance, which can be determined by the hardware in Paula. A paddle contains such a potentiometer which can be set with a knob. Analog joysticks also work this way. One potentiometer for the X and one for the Y direction determine the joystick position exactly.

Two registers contain the four 8-bit values of the analog inputs, POT0DAT for game port 0 and POT1DAT for game port 1:

POT0DAT $012 POT1DAT $014

Bit no.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

Both registers are read-only.

How is the resistance measured? Since a computer can process only digital signals, it needs a special circuit to convert any analog signals it wants to work with. On the Amiga the value of external resistance is determined as follows:

The potentiometers have a maximum resistance of 470 kilo Ohms (±10%). One side is connected to the +5-volt output and the other to one of the four paddle inputs of the game ports. These lead internally to the corresponding inputs of Paula and to one of four capacitors connected between the input and ground.

The measurement is started by means of a special start bit. Paula pulls all paddle inputs briefly to ground, discharging the capacitors. At this time the counters in the POTxDAT registers are also cleared. After this the counters increment by one with each screen line, while the capacitors are slowly recharged through the resistors. When the capacitor voltage exceeds a given value, the corresponding counter is stopped. The counter state corresponds exactly to the size of the resistance. Small resistances yield low counter values, greater ones yield higher values.

The start bit is located in the POTGO register:

POTGO $034 (write-only) - POTGOR $016 (read-only)

Bit no.	Name	Function
15	OUTRY	Switch game port 1 POTY to output
14	DATRY	Game port 1 POTY data bit
13	OUTRX	Switch game port 1 POTX to output
12	DATRX	Game port 1 POTX data bit
11	OUTLY	Switch game port 0 POTY to output
10	DATLY	Game port 0 POTY data bit
9	OUTLX	Switch game port 0 POTX to output
8	DATLX	Game port 0 POTX data bit
7 - 1		Unused
0	START	Discharge capacitors, begin measurement

A write access to POTGO clears both POTxDAT registers.

Normally you set the START bit to 1 in the vertical blanking gap. Then while the picture is being displayed, the capacitors charge up, reach the set value and stop the counters. The valid potentiometer readings can then be taken from the POTxDAT registers in the next vertical blanking gap.

The four analog inputs can also be programmed as normal digital input/output lines. The corresponding control and data bits are found together with the START bit in the POTGO register. Each line can be individually set to an output with the OUTxx bits (OUTxx = 1).

This separates them from the control circuit of the capacitors and causes the value in the DATxx bit of POTGO to be output over them. Reading a DATxx bit in POTGOR always returns the current state of the line. The following must be noted if the analog ports are used as outputs:

Since the four analog ports are internally connected to the capacitors for resistance measurement (47 nF), it can take up to 300 microseconds for the line to assume the desired level due to the charging/discharging of the capacitor required.

The input device buttons

Each of the three input devices mentioned so far has one or more buttons. The following table shows which registers contain the status of the mouse, paddle and joystick buttons:

Game port 0:

Left mouse button	CIA-A, parallel port A, port bit 6
Right mouse button	POTGOR, DATLY
(Third mouse button	POTGOR, DATLX)
Joystick fire button	CIA-A, parallel port A, port bit 6
Left paddle button	JOY0DAT, bit 9 (1 = button pressed)
Right paddle button	JOY0DAT, bit 1 (1 = button pressed)

Game port 1:

Left mouse button	CIA-A, parallel port A, port bit 7
Right mouse button	POTGOR, DATRY
(Third mouse button	POTGOR, DATRX)
Joystick fire button	CIA-A, parallel port A, port bit 7
Left paddle button	JOY1DAT, bit 9 (1 = button pressed)
Right paddle button	JOY1DAT, bit 1 (1 = button pressed)

Unless otherwise indicated, all bits are active-zero, meaning 0 = button pressed.

11.7.11 The Serial Interfaces

As we discussed earlier, the Amiga has a standard RS-232 interface. The various lines of this connector can be divided into two signal groups:

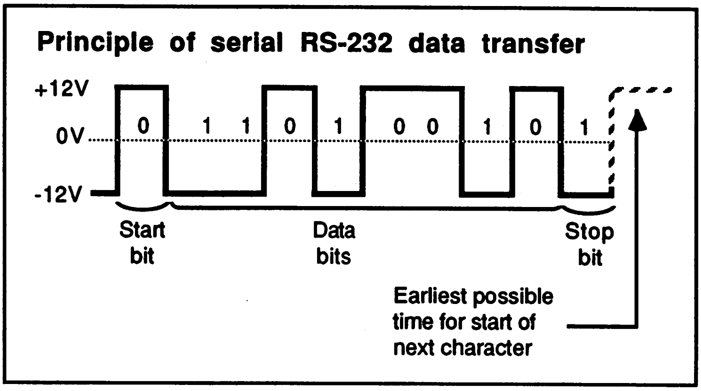
1. The serial data lines
2. The handshake lines

First about number 2: The RS-232 interface has a number of handshake lines. Normally they are not all used. However, the behavior of these signals is not always the same from one RS-232 device to another.

Now to number 1:

All data transfer takes place over the two data lines. The RXD line receives the data and it's sent out over TXD. RS-232 communication can take place in two directions at once when two devices are connected together through RXD and TXD. The RXD of one device is connected to the TXD of the other, and vice versa.

Principle of serial RS-232 data transfer



The RS-232 data transfer

Since only one line is available for the data transfer in each direction, the data words must be converted into a serial data stream which can then be transmitted bit by bit. No clock lines are provided in the RS-232 standard. So that the receiver knows when it can read the next bit, the time per bit must be constant (i.e., the speed at which the data is sent and received must be defined). This speed is called the baud rate, and it determines the number of bits transferred per second. For example, common baud rates are 300, 1200, 2400, 4800 and 9600 baud. You're not limited to these baud rates, but when using strange baud rates, remember that the sender and receiver must actually match.

One more thing required for successful transfer is that the receiver must know when a byte starts and ends. The above figure shows the timing of the transmission of a data byte on one of the data lines. Each byte begins with a start bit, which is no different from the normal data bits but always has a value of 0. Following this are the data bits in the order LSB to MSB. At the end are one or two stop bits, which have the value 1. The receiver recognizes the transition from one byte to the next by the level change from 1 to 0 that occurs when a start bit follows a stop bit.

The component that performs this serial transfer is called a Universal Asynchronous Receiver/Transmitter, or UART. In the Amiga it is contained in Paula, and its registers are in the custom chip register area:

The UART registers

SERPER $032 (write-only)

Bit no.	Name	Function
15	LONG	Set length of receive data to 9 bits
0 - 14	RATE	This 15-bit number contains the baud rate

SERDAT $030 (write-only)

SERDAT contains the send data.

SERDATR $018 (read-only)

Bit no.	Name	Function
15	OVRUN	Overrun of receive shift register
14	RBF	Receive buffer full
13	TBE	Transmit buffer empty
12	TSRE	Transmit shift register empty
11	RXD	Corresponds to level on RXD line
10	---	Unused
9	STP	Stop bit
8	STP or DB8	Depends on data length
7	DB7	Receive data buffer bit 7
6	DB6	Receive data buffer bit 6
5	DB5	Receive data buffer bit 5
4	DB4	Receive data buffer bit 4
3	DB3	Receive data buffer bit 3
2	DB2	Receive data buffer bit 2
1	DB1	Receive data buffer bit 1
0	DB0	Receive data buffer bit 0

One bit in the ADKCON register belongs to UART control:

ADKCON $09E (write) ADKCONR $010 (read) SERIELL

Bit no. 11: UARTBRK - interrupts the serial output and sets TXD to 0.

Data transfer with the Amiga UART

Receiving

The reception of the serial data takes place in two stages. The bits arriving on the RXD pin are received into the shift register at the baud rate and are combined there into a parallel data word. When the shift register is full, its contents are written into the receiver data buffer. It is then free for the next data. The processor can read only the receiver data buffer, not the shift register. The corresponding data bits in the SERDATR register are DB0 to DB7 or DB8.

The Amiga can receive both eight and nine-bit data words. The UART can be set to 9-bit words with the LONG bit (=1) in the SERPER register.

The data length determines the format in the SERDATR register. With 9 bits, bit 8 of SERDATR contains the ninth data bit, while the stop bit is found in bit 9. With eight data bits, bit 8 contains the stop bit. If two stop bits are present, the second lands in bit 9.

The state of the receiver shift register and the data buffer is given by two signal bits in SERDATR:

RBF stands for Receive Buffer Full. As soon as a data word is transferred from the shift register to the buffer, this bit changes to 1 and thereby signals the processor that it should read the data out of SERDATR.

This bit also exists in the interrupt registers (RBF, INTREQ/INTEN bit 11). After the processor has read the data, it must reset RBF in INTREQ. The bit then returns to 0 in SERDATR and in INTREQR.

```
MOVE.W #$0800,$DFF000+INTREQ      ;Clears RBF in INTREQ and SERDATR
```

If this is not done and the shift register has received another complete data word, the UART sets the OVRUN bit. This signals that no more data can be received because both the buffer (RBF = 1) and the shift register (OVRUN = 1) are full. OVRUN returns to 0 when RBF is reset. RBF then jumps back to 1 because the contents of the shift register are immediately transferred to DB0 through DB8 to free the shift register for more data.

Transmitting

The sending process also takes place in two stages. The transmit data buffer is found in the SERDAT register. As soon as a data word is written into this register it is transferred to the output shift register. This is signaled by the TBE bit. TBE stands for Transmit Buffer Empty and indicates that SERDAT is ready to take more data. TBE is also present in the interrupt registers (TBE, INTREQ/INTEN bit 0). Like RBF, TBF must also be reset in the INTREQ register.

Once the shift register has sent the data word, the next one is automatically loaded from the transmitter data buffer. If this is empty, the UART sets the TSRE bit (Transmit Shift Register Empty) to 1. This bit is reset when TBE is cleared.

The length of the data word and the number of stop bits are set by the format of the data in SERDAT. You simply write the desired data word to the lower eight or nine bits of SERDAT with one or two stop bits (1's) in front of it. An eight-bit data word with two stop bits would look like this, for example:

Bit no.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Function:	0	0	0	0	0	1	1	D7	D6	D5	D4	D3	D2	D1	DO

DO to D7 are the eight data bits.

The two ones represent the desired two stop bits. With a nine-bit data word and one stop bit the following data must be written into SERDAT:

Bit no.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Function:	0	0	0	0	0	1	D8	D7	D6	D5	D4	D3	D2	D1	DO

Eight bits plus one stop bit:

Bit no.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Function:	0	0	0	0	0	0	1	D7	D6	D5	D4	D3	D2	D1	DO

The LONG bit in the SERPER register affects only the length of the data received. The format of the transmitted data is determined only by the value in the SERDAT register.

Setting the baud rate

The baud rate for sending and receiving data must be written to the lower 15 bits of the SERPER register. Unfortunately, the baud rate cannot be set directly. You must select the number of bus cycles between two bits (1 bus cycle takes 2.79365 * 10-7 seconds). If a bit is to be output every n bus cycles, the value n-1 must be written to the SERPER register. The following formula can be used to calculate the necessary SERPER value from the baud rate:

$$SERPER = \frac{1}{\text{Baud rate} * 2.79365 * 10\text{-}7} - 1$$

For example, for a baud rate of 4800 baud:

SERPER = 1/(4800*2.79365*10-7)-1= 1/0.00134-1= 744.74

The calculated value is rounded and written to SERPER:

```
MOVE.W #745,$DFF000+SERPER          ;Set SERPER, LONG = 0
```

or

```
MOVE.W #$8000+745,$DFF000+SERPER    ;LONG = 1
```

11.7.12 The Disk Controller

The hardware control of the disk drives is divided into two parts. First there are the control lines which activate the desired drive, turn the motor on, move the read/write head, etc. They all lead to various port lines of the CIAs.

Excluded from these are the data lines. These carry the data from the read/write head to the Amiga and, when writing, in the opposite direction from the Amiga to the diskette. A special component in Paula, the disk controller, handles the processing of the data.

It has its own DMA channel and writes or reads data by itself to or from the disk.

Programming the disk DMA

Before you start the disk DMA you must be sure that the previous disk DMA is finished. If one interrupts a write access in progress, the data on the corresponding track can be destroyed. Let's assume that the last disk DMA is done.

First we must define the memory address of the data buffer. The disk DMA uses one of the usual address register pairs as a pointer to the chip RAM. The registers are called DSKPTH and DKSPTL:

$20 DSKPTH	Pointer to data from/to disk Bits 16-20
$22 DSKPTL	Pointer to data from/to disk Bits 0-15.

Next the DSKLEN register must be initialized. It is constructed as follows:

DSKLEN $024 (write-only)

Bit no.	Name	Function
15	DMAEN	Enable disk DMA
14	WRITE	Write data to disk
0-13	LENGTH	Number of data words to be transferred

LENGTH — The lower 14 bits of the DSKLEN register contain the number of data words to be transferred.

WRITE — WRITE = 1 switches the disk controller from read to write.

DMAEN — When DMAEN is set to 1 the data transfer begins.

A few things must be noted:

1. The Disk DMA Enable bit in the DMACON register (DSKEN, bit 4) must also be set.
2. To make it more difficult to write to the disk accidentally, the DMAEN bit must be set twice in succession. Only then does the disk DMA begin. Furthermore, for safety's sake the WRITE bit should only be 1 during a write operation.

An orderly initialization sequence for disk DMA appears as follows:

1. Write a 0 to DSKLEN to turn DMAEN off.
2. If DSKEN in DMACON is not yet set, do so now.
3. Store the desired address in DSKPTH and DSKPTL.
4. Write the correct value for LENGTH and WRITE along with a set DMAEN bit to DSKLEN.
5. Write the same value into DSKLEN again.
6. Wait until the disk DMA is done.
7. For safety's sake, set DSKLEN back to zero.

The DSKBLK interrupt (disk block finished, bit 1 in INTREQ/INTEN) is provided so that the processor knows when the disk controller has transferred the number of words defined in LENGTH. It is generated when the last data word is read or written. The current status of the disk controller can be read in the DSKBYTR register:

DSKBYTR $01A (read-only)

Bit no.	Name	Function
15	BYTEREADY	Signals that the data byte in the lower eight bits is valid.
14	DMAON	Indicates whether the disk DMA is enabled. To make DMAON = 1, both DMAEN in DSKLEN and DSKEN in DMACON must also be set.
13	DSKWRITE	Indicates the status of WRITE in DSKLEN.
12	WORDEQUAL	Disk data equals DSKSYNC
11-8		Unused
7-0	DATA	Current data byte from the disk

With the eight DATA bits and the BYTEREADY flag you can read the data from the disk with the processor rather than through DMA. Each time a complete byte is received the disk controller sets the BYTEREADY bit. The processor then knows that the data byte in the eight DATA bits is valid. After the DSKBYTER register is read the BYTEREADY flag is automatically reset.

Sometimes we don't want to read an entire track into memory at once. In this case the DMA transfer can be made to start at a given position. To do this, write the data word at which you want the disk controller to start into the DSKSYNC register:

DSKSYNC $07E (write-only)

DSKSYNC contains the data word at which the transfer is to begin. The disk controller then starts as usual after the disk DMA is enabled and reads the data from the disk, but it doesn't write it into memory. Instead, it continually compares each data word with the word in DSKSYNC. When the two match it starts the data transfer, which then continues as usual. The disk controller can be programmed to wait for the synchronization mark at the start of a data block.

The WORDEQUAL bit in the DSKBYTER register becomes 1 as soon as the data read matches DSKSYNC. Since this match only lasts two (or four) microseconds, WORDEQUAL is also set only during this time span. An interrupt is also generated at the same time WORDEQUAL goes to 1:

Bit 12 in the INTREQ and INTEN registers is the DSKSYN interrupt bit. It is set when the data from the disk matches DSKSYNC.

Setting the operating parameters

The data cannot be written to the disk in the same format as found in memory. It must be specially coded. Normally the Amiga works with MFM coding. However, it is also possible to use GCR coding. Two steps are necessary for selecting the desired coding:

1. An appropriate routine must encode the data before it is written to disk and decode the data as it is read from disk.
2. The disk controller must be set for the appropriate coding. This is done with certain bits in the ADKCON register.

ADKCON $09E (write) ADKCONR $010 (read) DISK

Bit no.	Name	Function
15	SET/CLR	Set (SET/CLR=1) or clear bits
14-13	PRECOMP	These bits contain the precompensation value: Bit 14 Bit 13 PRECOMP time 0 0 Zero 0 1 140 ns 1 0 280 ns 1 560 ns
12	MFMPREC	0 = GCR, 1 = MFM
11	UARTBRK	Not a disk controller bit, see UART
10	WORDSYNC	WORDSYNC = 1 turns on synchronization of the disk controller according to the word in the DSKSYNC register.
9	MSBSYNC	MSBSYNC = 1 enables GCR synchronization
8	FAST	Disk controller clock rate: FAST=1: 2 microseconds/bit (MFM) FAST=0: 4 microseconds/bit (GCR)
7-0	AUDIO	These bits do not belong to the disk controller.

The disk controller data registers

As usual the DMA controller transfers data in memory to and from the appropriate data registers. The disk controller has one data register for data read from the disk and one for data that is to be written to the disk.

DSKDAT $026 (write-only)

Contains the data to be written to the disk.

DSKDAT $008 (read-only)

Contains the data read from the disk. This is an early-read register and cannot be read by the processor.

Bibliography

O'Hara, R.P. and Gomberg, D.R.: Modern Programming Using REXX, Prentice-Hall London, 1985

Hawes, William S.: ARexx User's Reference Manual, Maynard MA, 1987

The design of the REXX language, IBM Systems Journal, Volume 23, No.4, 1984

Motorola: Motorola MC68030 Users Manual, Prentice-Hall London, 1990

Commodore: AMIGA Reference Manuals (Series), Addison-Wesley, 1990

Index

A

$00000004 10
SimpleRefresh 11
ABBREV() 524
AbortIO 207
ABS() 538
Acceleration 18
Accrued Exception Byte 690
accubuffered truetime clock 12
ActivateCxObj 39
ActivateGadget 398
ActivateWindow 366
AddAnimOb 317
AddAppIconA 461
AddAppMenuItemA 462
AddAppWindowA 464
AddBob 317
AddBootNode 225
AddBuffers 74
AddClass 414
AddClipNode() 614
AddConfigDev 226
AddDevice 208
AddDosEntry 66
AddDosNode 226
AddFont 307
AddFreeList 327
AddGadget 398
AddGList 399
AddHead 182
AddIEvents 51
AddIntServer 166
Addition (+) 489
ADDLIB() 544
AddLibrary 203
AddMemList 175
AddPart 128
AddPort 194
AddResource 213
ADDRESS 480, 507
Address bits and mask 672
ADDRESS() 544
AddRsrcNode() 614
AddSegment 95
AddSemaphore 214
AddTail 182
AddTask 186
AddVSprite 318
Alert 194
aliasing distortion 919
All Files 16
AllocAbs 176
AllocAslRequest 28
Allocate 176
AllocateTagItems 454
AllocConfigDev 228
AllocDosObject 63
AllocEntry 177
AllocExpansionMem 228
AllocFileRequest 27
AllocIFF 332
AllocLocalItem 347
AllocMem 177
ALLOCMEM() 574
AllocRaster 287
AllocRemember 414
AllocSignal 186
AllocTrap 187
AllocVec 178
alternate gadget 13
Amiga Operating System 5
AmigaOS 5
AndRectRegion 283
AndRegionRegion 283
Animate 318
AreaDraw 294

AreaEllipse ... 295
AreaEnd ... 295
AreaMove ... 296
ARexx:
Function calls ... 584
Parameter conversion ... 584
ARG ... 477, 496
ARG() ... 545
arp.library ... 11
AskFont ... 308
AskSoftStyle ... 308
ASL library ... 26
AslRequest ... 30
Aspect ... 20
AssignAdd ... 67
AssignLate ... 69
AssignLock ... 69
assignment clause ... 492
AssignPath ... 70
ATC (Address Translation Cache) ... 670
AttachCxObj ... 42
AttemptLockDosList ... 70
AttemptLockLayerRom ... 287
AttemptSemaphore ... 214
Audio device ... 12
Audio interrupts ... 916
AUL ... 887
Autopoint ... 21
AutoRequest ... 388
AvailFonts ... 55
AvailMem ... 178

B

B2C() ... 541
back/front gadget ... 13
BADDR() ... 581
barrel shifter ... 875
baud rate ... 942
BCPL-pointers ... 11
BEGIN..END ... 477
BeginRefresh ... 366
BeginUpdate ... 434
BehindLayer ... 434
BFD (Blitter Finish Disable) ... 801
bit-plane ... 786
bit-map graphic ... 684
BITAND() ... 535
BITCHG() ... 535
BITCLR() ... 535
BITCOMP() ... 536
BitMapScale ... 273
BITOR() ... 536
BITSET() ... 536
BITTST() ... 537
BITXOR() ... 537
Blanker ... 21
Blitter ... 684, 865
Blitter control registers ... 877
Blitter DMA ... 879
Blitter DMA cycles ... 893
BltBitMap ... 275
BltBitMapRastPort ... 276
BltClear ... 277
BltMaskBitMapRastPort ... 277
BltPattern ... 278
BltTemplate ... 278
boolean algebra ... 872
boolean equation ... 872
Boot Menu ... 6
BREAK ... 501
BrokerCommand ... 51
BuildEasyRequestArgs ... 388
BuildSysRequest ... 389
BumpRevision ... 327
burst-fill ... 703
bus cycles ... 788
BY ... 477

C

C2B() ... 541
C2D() ... 541
C2X() ... 542

Cache Address Register (CAAR) 704
Cache Control Register (CACR) 704
Cache inhibit (CI) 672
Cache Row Layout 703
cache-hit 703
cache-miss 703
CacheClearE 187
CacheClearU 188
CacheControl 188
CALL 507
Cause 168
CBump 254
CENTER() 524
CENTRE() 524
ChangeMode 113
ChangeSprite 318
ChangeWindowBox 368
CheckIO 208
CheckSignal 103
chip RAM 770
CIAs 772
Clauses 492
ClearCxObjError 41
ClearDMRequest 390
ClearEOL 308
ClearMem() 615
ClearMenuStrip 368
ClearPointer 369
ClearRectRegion 284
ClearRegion 284
ClearRexxMsg() 615
ClearScreen 309
CLI 22, 104
ClipBlit 279
clockcycle-optimized programs 12
CloneTagItems 454
Close 113
CLOSE() 520
CloseClipboard 332
CloseDevice 208
CloseF() 604
CloseFont 309
CloseIFF 332
CloseLibrary 204
CloseMonitor 254
CLOSEPORT() 575
ClosePublicPort() 615
CloseScreen 352
CloseWindow 369
CloseWorkBench 352
CMove 254
CmpString() 609
ColdCapture 10
ColdReboot 163
CollectionChunk 342
Color Correct 20
color palette 808
ColorMap 11
COMMAND 480
Command >NIL parameter 14
command clause 493
Command Keys 19
Command Line Interface 6
command lines 22
Comments 492
Commodities Library 36
COMPARE() 525
CompareDates 138
COMPRESS() 524
Condition Code Byte 688
Control Register (FPCR) 685
Control Registers 688, 825
COPIES() 525
Copper 797
Copper DMA 802
Copper interrupt 801
Copper list 11, 797
CopyBrokerList 51
CopyMem 178
CopyMemQuick 179
CopySBitMap 279
CreateArgString() 615

CreateBehindHookLayer 434
CreateBehindLayer 435
CreateContext 234
CreateCxObj 37
CreateDir 85
CreateDOSPkt () 604
CreateGadgetA 234
CreateIORequest 209
CreateMenusA 239
CreateMsgPort 194
CreateNewProc 96
CreateProc 96
CreateRexxMessage() 616
CreateUpfrontHookLayer 436
CreateUpfrontLayer 436
CurrentChunk 339
CurrentDir 86
CurrentEnv() 616
CurrentTime 416
custom chips 772
CVa2i() 611
CVc2x() 612
CVi2a() 612
CVi2arg() 612
CVi2az() 613
CVs2i() 613
CVx2c() 613
CWait 255
Cx library 36
CxBroker 37
CxMsgData 47
CxMsgID 48
CxMsgType 47
CxObjError 41
CxObjType 40

D

D2C() 542
D2X() 542
data fetch start 790
data fetch stop 790
DATATYPE() 525
DATE() 545
DateStamp 138
DateToStr 129
DDFSTRT 816
Deallocate 179
Debug 195
deciBels 906
Delay 139
DELAY() 575
DELETE() 581
DeleteArgstring() 617
DeleteCxObj 39
DeleteCxObjAll 40
DeleteDOSPkt() 605
DeleteFile 114
DeleteIORequest 209
DeleteLayer 437
DeleteMsgPort 195
DeleteRexxMsg() 617
DeleteVar 140
DELSTR() 526
DELWORD() 526
descriptors 675
DeviceProc 74
Devices 7
DFFSTOP 816
DIGITS() 538
Disable 168
DisownBlitter 280
DisplayAlert 390
DisplayBeep 410
DisposeCxMsg 50
DisposeFontContents 57
DisposeLayerInfo 437
DisposeObject 416
DisposeRegion 284
Dithering 20
DivertCxMsg 48
Division (/) 489
DIWSTOP 814

DIWSTRT 814
DMA access 781
DMA controller 781
DO 501
DO ...END 477
DoCollision 319
DoIO 209
DoPkt 75
dos.library 11
DOSRead() 605
DOSWrite() 605
double real 686
Double-Click 18, 416
Draw 296
DrawBevelBoxA 240
DrawBorder 410
DrawEllipse 296
DrawGList 319
DrawImage 411
DrawImageState 411
DROP 508
dual playfield mode 811
DupLock 64
DupLockFromFH 64
Dyadic Operations 693

E

Early Termination (ET) 680
Easy error trapping 474
Easy string manipulation 474
EasyRequestArgs 391
ECHO 496
ECS 5
Edit Standard Overscan 20
Edit Text Overscan 20
Elementary bit-map 684
ELSE 502
Enable 169
Enable bit (E) 672
END 503
EndNotify 75
EndRefresh 370
EndRequest 393
EndUpdate 438
Enhanced Chip Set 5
Enhanced Chip Set (ECS) 860
Enqueue 183
EnqueueCxObj 43
EntryHandler 343
EOF() 520
EraseImage 412
EraseRect 297
ErrorMsg() 613
ErrorReport 143
ERRORTEXT() 546
even cycles 788
ExAll 86
Examine 87
ExamineFH 114
Exception Byte 689
Exception Enable 689
Exception/Interrupt Table 10
exec.library 10
Execute 104
ExistF() 605
EXISTS() 520
Exit 97, 503
ExitHandler 344
ExNext 88
expansion cards 12
Expansion-library 11
Exponentiation (**) 488
EXPORT() 546
Expressions 487
extended real 686
ExtendFont 309
external commands 493
extra half-bright mode 809

F

fast RAM 770
FattenLayerInfo 11

Fault 130
FGetC 114
FGets 130
File Extension 596
FilePart 131
FillRexxMsg() 617
FilterTagChanges 454
FilterTagItems 455
FIND() 527
FindArg 131
FindBroker 45
FindCliProc 105
FindCollection 339
FindConfigDev 228
FindDevice() 606
FindDisplayInfo 255
FindDosEntry 71
FindLocalItem 347
FindName 183
FindPort 195
FindProp 340
FindPropContext 340
FindResident 163
FindRsrcNode() 617
FindSegment 97
FindSemaphore 214
FindTagItem 455
FindTask 188
FindToolType 328
FindVar 140
FKey 21
flickering 785
Floating-point format 685
floating-point numbers 684
floating-point representation 685
Flood 297
Flush 115
FontExtent 310
FOR 477
Forbid 169
FORBID() 576
FOREVER 477
FORM() 538
Format 76
FORWARD() 576
FPIAR Register 701
FPU exceptions:
- BSUN 699
- Cop.Prot. 699
- DZ 699
- F-Line 699
- FTRAPcc 699
- INEX 699
- INEX2 699
- OPERR 700
- OVFL 700
- SNAN 700
- UNFL 700

FPutC 115
FPuts 132
frames 785
FRead 116
FreeAslRequest 29
FreeBrokerList 51
FreeClass 417
FreeColorMap 288
FreeConfigDev 229
FreeCopList 255
FreeCprList 256
FreeDeviceProc 76
FreeDiskObject 328
FreeDosEntry 65
FreeDosObject 65
FreeEntry 180
FreeExpansionMem 229
FreeFileRequest 27
FreeFreeList 328
FreeGadgets 240
FreeGBuffers 319
FreeIFF 333
FreeLocalItem 348
FreeMem 180

FREEMEM() 576
FreeMenus 240
FreePort() 618
FreeRaster 288
FreeRemember 417
FreeScreenDrawInfo 352
FreeSignal 189
FREESPACE() 547, 618
FreeSprite 320
FreeSysRequest 393
FreeTagItems 456
FreeTrap 189
FreeVec 180
FreeVisualInfo 241
FreeVPortCopLists 256
frequency 906
FRESTORE 697
FSAVE 697
FUZZ() 538
FWrite 116

G

GadgetMouse 399
gadgets 11
GETARG() 577
GetArgStr 132
GetAttr 419
GetCC 169
GETCLIP() 547
GetColorMap 288
GetColorMap() 11
GetConsoleTask 77
GetCurrentBinding 229
GetCurrentDirName 132
GetDefaultPubScreen 353
GetDefDiskObject 329
GetDefPrefs 419
GetDeviceProc 77
GetDiskObject 329
GetDiskObjectNew 329
GetDisplayInfoData 256
GetFileSysTask 78
GetGBuffers 320
GetMsg 196
GETPKT() 577
GetPrefs 419
GetProgramDir 89
GetProgramName 133
GetPrompt 133
GetRGB4 289
GetScreenData 353
GetScreenDrawInfo 354
GETSPACE() 548, 618
GetSprite 321
GetTagData 456
GetVar 141
GetVisualInfoA 241
GetVPModeID 257
GfxBase:
  Row/cols 11
GoodID 333
GoodType 333
GT_BeginRefresh 241
GT_EndRefresh 242
GT_GetIMsg 243
GT_PostFilterIMsg 243
GT_RefreshWindow 243
GT_ReplyIMsg 244
GT_SetGadgetAttrsA 244

H

HAM 810
handshake lines 941
harmonic 907
HASH() 548
Hertz 906
HI 569
hold-and-modify mode 810
Hook 25-26
horizontal blanking gap 784
Horizontal Pulse 934
Horizontal Quadrature Pulse 934

Host address 595
Human Logic 474

I

I/O Functions 602
Icon 16, 17
  Copy 17
  Delete 17
  Empty Trash 18
  Format Disk 18
  Information 17
  Leave Out 17
  Open 17
  Put Away 17
  Rename 17
  Snapshot 17
  Unsnapshot 17
IDtoStr 334
IF 503
IF...THEN...ELSE 477
IFF file format 331
iffparse.library 331
IHelp 21
Image 20
IMPORT() 548
INDEX() 527
Indirect Descriptors 681
Info 89
Inhibit 78
InitArea 298
InitBitMap 290
InitCode 163
InitGels 321
InitGMasks 321
InitIFF 334
InitIFFasClip 334
InitIFFasDOS 335
InitLayers 11
InitList() 619
InitMasks 322
InitPort() 619
InitRastPort 290
InitRequester 394
InitResident 164
InitSemaphore 215
InitStruct 164
InitTmpRas 291
InitView 291
InitVPort 291
Input 105
Input and output data flow 596
input device buttons 940
Input task 5
Insert 183
INSERT() 527
InsertCxObj 43
InstallClipRegion 438
InstallLayerHook 438
Instruction Address Register (FPIAR) ... 685
Instruction Cache Design 702
Instructions 495
INT2, INT3, INT6 737
Integer division (%) 489
intensity 906
interlace mode 823
interlacing 785
InternalLoadSeg 98
InternalUnLoadSeg 99
INTERPRET 481, 508
interrupt mask register 795
interrupt request register 795
IntuiTextLength 412
Intuition tasks 5
InvertKeyMap 50
IoErr 144
IsFileSystem 79
IsInteractive 117
IsRexxMsg() 619
IsSymbol() 620
ItemAddress 371
ITERATE 504

J-L

joysticks 937
Key Repeat Delay 18, 19
Key Repeat Rate 19
Key Repeat Test 19
kilohertz 906
label marker 492
LASTPOS() 528
Layer 11
LayoutMenuItemsA 245
LayoutMenusA 246
LEAVE 504
LEFT() 528
LENGTH() 528
lengthargstring() 609, 620
Libraries 7
LINES() 520
ListNames() 620
LoadRGB4 257
LoadSeg 99
LoadView 258
LoadWB 14
LocalItemData 348
Lock 117
LockDosList 71
LockIBase 420
LockLayer 439
LockLayerInfo 439
LockLayerRom 292
LockLayers 439
LockPubScreen 354
LockPubScreenList 355
LockRecord 118
LockRexxBase() 620
Logical AND (&) 491
Logical exclusive OR (^ or &&) 491
Logical inclusive OR (|) 491
Logical NOT (~) 491
long frame 785
low-pass filter 920

M

MakeClass 420
MAKEDIR() 582
MakeDosEntry 66
MakeDosNode 230
MakeFunctions 164
MakeLibrary 165
MakeScreen 355
MakeVPort 258
mantissa 685
MapTags 456
Masking 876
MatchEnd 89
MatchFirst 90
MatchNext 90
MatchPattern 134
MatchToolValue 330
mathffp.library 8
MAX() 538
MaxCli 106
MemHeader 11
Message Packets 588
MIN() 539
minterm 873
Mode RM 691
Mode RN 691
Mode RZ 691
ModeNotAvailable 258
ModifyIDCMP 371
ModifyProp 400
modules 7
modulo values 865
Monadic Operations 694
mouse 934
Mouse Screen Drag keys 19
Mouse Speed slider 18
Move 298, 797, 799
MoveLayer 439
MoveLayerInFrontOf 440
MoveScreen 355
MoveSizeLayer 440

MoveSprite 322
MoveWindow 371
MoveWindowInFrontOf 372
MrgCop 259
Multiplication (*) 488

N

NameFromFH 134
NameFromLock 134
NewFontContents 57
NewLayerInfo 11, 441
NewLoadSeg 99
NewModifyProp 400
NewObjectA 421
NewRegion 285
NewScaledDiskFont 58
NEXT() 578
NextDisplayInfo 259
NextDosEntry 72
NextObject 421
NextPubScreen 356
NextTagItem 457
No declarations 474
NoCapsLock 21
NOP 504
null clauses 492
NULL() 578
NUMERIC 476, 509
NUMERIC DIGITS 490
NUMERIC FUZZ 490

O

ObtainConfigBinding 231
ObtainGIRPort 401
ObtainSemaphore 215
ObtainSemaphoreList 215
ObtainSemaphoreShared 215
odd cycles 788
OffGadget 401
OffMenu 372
OFFSET() 578
OldOpenLibrary 204
OnGadget 402
Only Icons 16
OnMenu 373
Open 119
OPEN() 521
OpenClipboard 335
OpenDevice 210
OpenDiskFont 55
OpenF() 606
OpenFont 310
OpenFromLock 120
OpenIFF 335
OpenLibrary 204
OpenMonitor 259
OPENPORT() 578
OpenPublicPort() 621
OpenResource 213
OpenScreen 356
OpenScreenTagList 357
OpenWindow 373
OpenWindowTagList 375
OpenWorkBench 358
Operating System Menu 6
Operators 485
OPTIONS 480, 510
OrRectRegion 285
OrRegionRegion 285
OTHERWISE 504
Output 106
OVERLAY() 528
OwnBlitter 280

P

PackBoolTags 457
Packed decimal real 686
paddles 938
Paged Memory Management Unit 668
ParentChunk 341
ParentDir 91
ParentOfFH 91

PARSE 481, 496
ParseIFF 336
ParseIX 47
PARSE PULL 481
ParsePattern 135
parsing data 555
PassPort 595
PathPart 135
Permit 169
PERMIT() 579
Pipelining 894
playfields 807
PointInImage 422
PolyDraw 299
PopChunk 341
POS() 529
potentiometers 938
PRAGMA() 549
Prefix conversion (+) 488
Prefix negation (-) 488
Prefs 18
  Font 20
  IControl 19
  Input 18
  OverScan 20
  Palette 19
  Pointer 20
  Printer 20
  PrinterGfx 20
  ScreenMode 20
  Serial 21
  Time 21
  WBpattern 19
Preserve colors 19
PrintFault 144
PrintIText 413
PROCEDURE 478, 510
Procure 216
Program Control Instructions 695
PropChunk 344
PubScreenStatus 358
PULL 476, 499
PUSH 499
PushChunk 341
PutDefDiskObject 330
PutDiskObject 330
PutMsg 196
PutStr 144

Q-R

QBlit 280
QBSBlit 280
quantization 911
QueryOverscan 359
QUEUE 499
QueueF() 607
Quotient Byte 688
R/W and RWM 672
RAM 770
RANDOM() 539
RANDU() 539
RawDoFmt 196
Read 120
ReadArgs 106
READCH() 522
ReadChunkBytes 336
ReadChunkRecords 337
ReadF() 607
ReadItem 107
READLN() 522
ReadPixel 299
ReadPixelArray8 300
ReadPixelLine8 301
ReadStr() 607
RectFill 301
refresh cycles 790
RefreshGadgets 402
RefreshGList 402
RefreshTagItemClones 460
RefreshWindowFrame 376
Relabel 79
ReleaseConfigBinding 231

ReleaseGIRPort 403
ReleaseSemaphore 216
ReleaseSemaphoreList 216
Remainder (//) 489
RemakeDisplay 359
RemClipNode() 621
RemConfigDev 231
RemDevice 211
RemDosEntry 72
RemFont 310
RemHead 184
RemIBob 322
RemIntServer 170
REMLIB() 550
RemLibrary 205
Remove 184
RemoveAppIcon 464
RemoveAppMenuItem 465
RemoveAppWindow 465
RemoveClass 422
RemoveCxObj 44
RemoveGadget 403
RemoveGList 404
RemPort 199
RemResource 213
RemRsrcList() 622
RemRsrcNode() 622
RemSegment 100
RemSemaphore 217
RemTail 184
RemTask 189
RemVSprite 323
Rename 120
RENAME() 582
REPLY() 579
ReplyMsg 199
ReplyPkt 79
ReportMouse 404
Request 394
RequestFile 28
reset vector 779
ResetMenuStrip 376
ResetWindows 11
Resource Nodes 589
Resources 7
RethinkDisplay 359
RETURN 504
REVERSE() 529
RIGHT() 529
ROM 779
romboot.library 10
rounding errors 911
RouteCxMsg 49
RunCommand 100
RX 570
RXADDCON [RXFB_NONRET] 598
RXADDFH [RXFB_NONRET] 598
RXADDLIB [RXFB_NONRET] 599
RXC 571
RXCOMM [RXFB_TOKEN] [RXFB_STRING] [RXFB_RESULT] [RXFB_NOIO] 599
RXFB_NOIO 600
RXFB_NONRET 601
RXFB_RESULT 601
RXFB_STRING 601
RXFB_TOKEN 601
RXFUNC [RXFB_RESULT] [RXFB_STRING] [RXFB_NOIO] 599
RXREMCON [RXFB_NONRET] 600
RXREMLIB [RXFB_NONRET] 600
RXSET 570
RXTCCLS [RXFB_NONRET] 600
RXTCOPN [RXFB_NONRET] 600

S

SameLock 121
sample period 914
samples 911
sampling rate 914
SAS C-compiler 11
SAY 476, 500

ScalerDiv 281
Scaling 20
Screen menu snap 19
Screens 11
ScreenToBack 360
ScreenToFront 360
ScrollLayer 441
ScrollRaster 281
ScrollVPort 260
Seek 121
SEEK() 522
SeekF() 608
SELECT 505
SELECT...WHEN...OTHERWISE...END 477
SelectInput 108
SelectOutput 108
SendIO 211
SendPkt 80
serial data lines 941
serial interface 12
SetAPen 302
SetArgStr 108
SetAttrsA 422
SetBPen 302
SETCLIP() 550
SetCollision 323
SetComment 122
SetConsoleTask 80
SetCurrentBinding 232
SetCurrentDirName 109
SetCxObjPri 42
SetDefaultPubScreen 360
SetDMRequest 394
SetDrMd 302
SetEditHook 404
SetExcept 190
SetFileDate 122
SetFileSize 123
SetFileSysTask 81
SetFilter 46
SetFilterIX 46
SetFont 311
SetFunction 206
SetGadgetAttrsA 405
SetIntVector 170
SetIoErr 145
SetLocalItemPurge 348
SetMenuStrip 377
SetMouseQueue 378
SetPointer 378
SetPrefs 423
SetProgramDir 109
SetProgramName 109
SetPrompt 110
SetProtection 123
SetPubScreenModes 360
SetRast 304
SetRGB4 260
SetRGB4CM 292
SetSignal 190
SetSoftStyle 311
SetSR 171
SetTaskPri 191
SetTranslate 45
SetVar 142
SetWindowTitles 379
Shade 20
SHELL 511
short frame 785
Show 17, 18
SHOW() 551
SHOWDIR() 582
SHOWLIST() 579
ShowTitle 361
SIGN() 540
Signal 192, 477, 505
SIGNAL ON 481
Single real 686
SizeLayer 441
SizeWindow 379
SKIP 797, 801

Smoothing 20
SortGList 323
sound pressure 906
SOURCELINE() 551
SPAbs/IEEESPAbs/IEEEDPAbs 444
SPACE() 530
SPAcos/IEEESPAcos/IEEEDPAcos 448
SPAdd/IEEESPAdd/IEEEDPAdd 445
SPAsin/IEEESPAsin/IEEEDPAsin 448
SPAtan/IEEESPAtan/IEEEDPAtan 449
SPCeil/IEEESPCeil/IEEEDPCeil 445
SPCmp/IEEESPCmp/IEEEDPCmp 445
SPCos/IEEESPCos/IEEEDPCos 449
SPCosh/IEEESPCosh/IEEEDPCosh 449
SPDiv/IEEESPDiv/IEEEDPDiv 445
Special 7
SPExp/IEEESPExp/IEEEDPExp 449
SPFieee/IEEESPFieee/IEEEDPFieee 450
SPFix/IEEESPFix/IEEEDPFix 446
SPFloor/IEEESPFloor/IEEEDPFloor 446
SPFlt/IEEESPFlt/IEEEDPFlt 446
SplitName 136
SPLog/IEEESPLog/IEEEDPLog 450
SPLog10/IEEESPLog10/IEEEDPLog10 450
SPMul/IEEESPMul/IEEEDPMul 446
SPNeg/IEEESPNeg/IEEEDPNeg 447
SPPow/IEEESPPow/IEEEDPPow 450
sprite data list 840
sprite DMA 840
sprites 807, 838
SPSin/IEEESPSin/IEEEDPSin 451
SPSincos/IEEESPSincos/IEEEDPSincos 451
SPSinh/IEEESPSinh/IEEEDPSinh 451
SPSprt/IEEESPSqrt/IEEEDPSqrt 451
SPSub/IEEESPSub/IEEEDPSub 447
SPTan/IEEESPTan/IEEEDPTan 452
SPTanh/IEEESPTanh/IEEEDPTanh 452
SPTieee/IEEESPTieee/IEEEDPTieee 452
SPTst/IEEESPTst/IEEEDPTst 447
StackF() 608
StartNotify 81
startup Copper list 806
Status Register 688
  Accrued Exception Byte 688
  Condition Code Byte 688
  Exception Byte 688
  Quotient Byte 688
Status Register (FPSR) 685
StcToken() 609
StopChunk 345
StopOnExit 346
STORAGE() 552
StoreItemInContext 349
StoreLocalItem 349
StrcmpN() 610
StrcpyA() 610
StrcpyN() 610
StrcpyU() 611
StrflipN() 611
Strings 485
STRIP() 530
StripFont 312
Strlen() 611
StrToDate 136
StrToLong 137
SUBSTR() 530
Subtraction (-) 489
SUBWORD() 531
SUD 887
SUL 887
SumKickData 166
SumLibrary 206
Super-HiRes mode 860
SuperBitmap Console 12
SuperState 171
Supervisor 172
SwapBitsRastPortClipRect 442
SYMBOL() 552
Symbols 483
SYS 6

SysReqHandler 395
System 21
  FixFonts 21
  NoFastMem 21
  SetMap 21
system directory 6
SystemTagList 110

T

tag-entry 702
tag-field 703
TagInArray 460
TagItem fields 25
TagItems 26
TCC 571
TCO 571
TE 571
template 555
Test 18
Text 312
Text gadget filter 19
TextExtent 312
TextFit 313
TextLength 314
THEN 506
ThinLayerInfo 11
timbre 907
TIME() 552
Tokens 483
tone color 907
ToolManager 18
Tools 18
  ResetWB 18
ToUpper() 614
TRACE 474, 511
TRACE() 553
tracing programs 555
Trackdisk 12
Translate 452
TRANSLATE() 531
Translation Control Register (TC) 673
translation tree 670
tremolo 907
TRIM() 532
TRUNC() 540
TS 571
Type-less data 473
TypeOfMem 181
TYPEPKT() 580

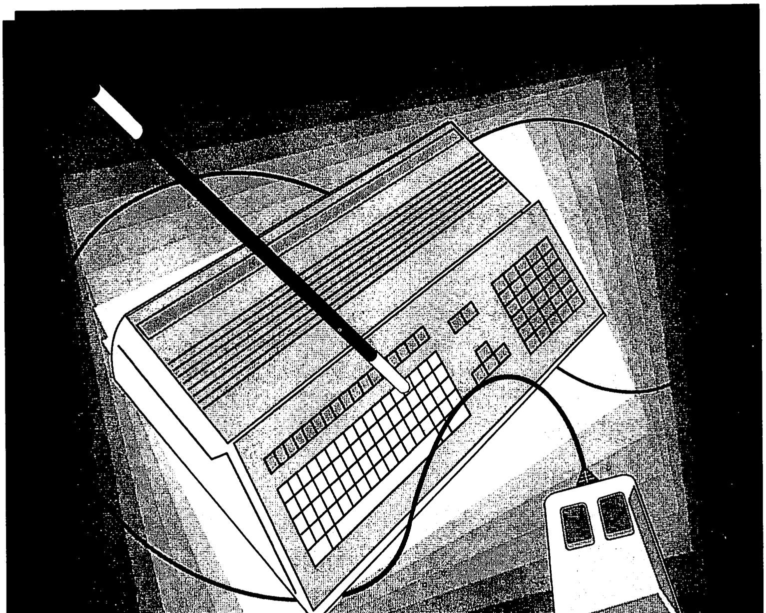
U

UART 942
UnGetC 124
Universal applicability 473
Universal Asynchronous Receiver/
Transmitter 942
UnLoadSeg 101
UnLock 124
UnLockDosList 72
UnlockIBase 423
UnlockLayer 442
UnlockLayerInfo 442
UnlockLayerRom 292
UnlockLayers 443
UnlockPubScreen 361
UnlockPubScreenList 361
UnLockRecord 124
UnLockRecords 125
UnlockRexxbase() 622
UNTIL 477
UpfrontLayer 443
UPPER 512
UPPER() 532
UserState 172
Utilities 21
  Clock 21
  Display 21
  Exchange and Commodities 21
  More 21
  Say 21

V-Z

Vacate 217
VALUE() 554
VBeamPos 261
Vector Base Register (VBR) 10
Verify Timeout 19
VERIFY() 532
vertical blanking gap 785
Vertical Pulse 934
Vertical Quadrature Pulse 934
VFPrintf 125
VFWritef 126
vibrato 906
VideoControl 261
ViewAddress 362
ViewPortAddress 380
volume 906
Volume modulation 922
VPrintf 110
Wait 192, 797, 800
WaitBOVP 262
WaitForChar 139
WAITFORPORT 572
WaitIO 211
WaitPkt 81
WAITPKT() 581
WaitPort 200
WaitTOF 263
WBenchToBack 362
WBenchToFront 362
WeighTAMatch 314
WHEN 506
WHILE 477
white noise 909
Window 16
  Clean Up 16
  Close 16
  New Drawer 16
  Open Parent 16
  Select Contents 16
  Show 16
  Snapshot 16
  Update 16
  View By 16
Window frames 11
WindowLimits 380
WindowToBack 380
WindowToFront 381
WORD() 533
WORDINDEX() 533
WORDLENGTH() 533
WORDS() 533
Workbench 11, 13, 15
  About 15
  Backdrop 15
  Execute Command 15
  Last Message 15
  Quit 15
  Redraw All 15
  Update All 15
workbench-task 8
workbench.library 14
Write 126
write-through cache 704
WRITECH() 523
WriteChunkBytes 337
WriteChunkRecords 338
WriteF() 608
WRITELN() 523
WritePixel 304
WritePixelArray8 305
WritePixelLine8 305
X2C() 543
X2D() 543
XorRectRegion 286
XorRegionRegion 286
XRANGE() 533
ZipWindow 381

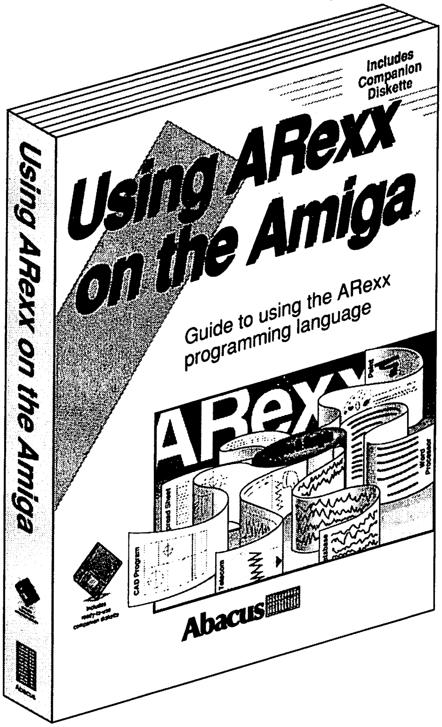
Abacus
Amiga Catalog



Order Toll Free 1-800-451-4319

Using ARexx on the Amiga

Using ARexx on the Amiga is the most authoritative guide to using the popular ARexx programming language on the Amiga. It's filled with tutorials, examples, programming code and a complete reference section that you will use over and over again. **Using ARexx on the Amiga** is written for new users and advanced programmers of ARexx by noted Amiga experts Chris Zamara and Nick Sullivan.



Topics include:

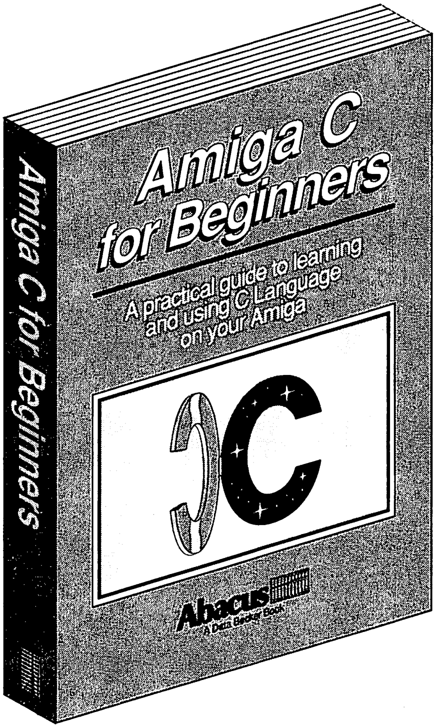
- What is Rexx/ARexx - a short history
- Thorough overview of all ARexx commands - with examples
- Useful ARexx macros for controlling software and devices
- How to access other Amiga applications with ARexx
- Detailed ARexx programming examples for beginners and advanced users
- Multitasking and inter-program communications
- Companion diskette included
- And much, much more!

Item #B114 ISBN 1-55755-114-6.
Suggested retail price: $34.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga C for Beginners

Amiga C for Beginners is an introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more. Topics include:



- Beginner's overview of C
- Particulars of C
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of the C language
- Input/Output using C
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions)
- Using the LATTICE and AZTEC C compilers

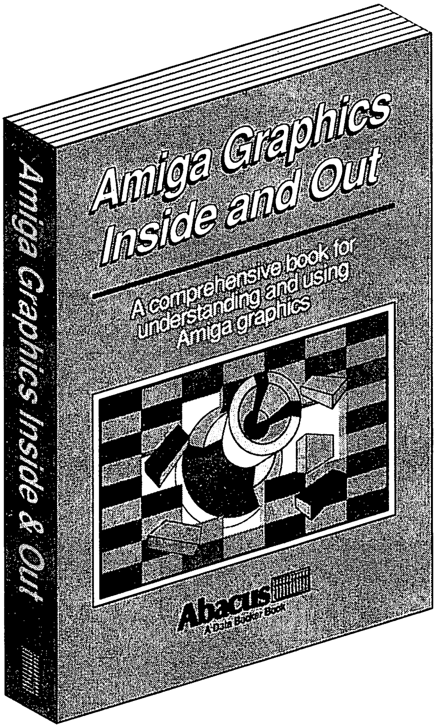
Item #B045 ISBN 1-55755-045-X. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **Item #S045. $14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga Graphics: Inside & Out

Amiga Graphics: Inside & Out will show you the super graphic features and functions of the Amiga in detail. Learn the graphic features that can be accessed from AmigaBASIC or C. The advanced user will learn how to call the graphic routines from the Amiga's built-in graphic libraries. Learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. Complete description of the Amiga graphic system- View, ViewPort, RastPort, bitmap mapping, screens and windows.



Topics include:

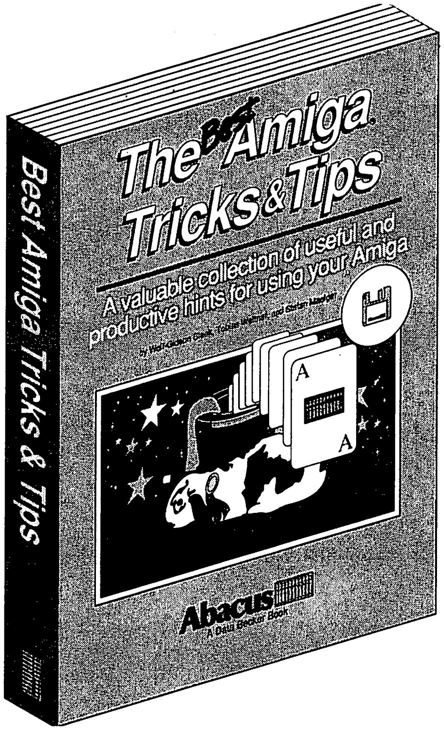
- Accessing fonts and type styles in AmigaBASIC
- Loading and saving IFF graphics
- CAD on a 1024 x 1024 super bitmap, using graphic library routines
- Access libraries and chips from BASIC- 4096 colors at once, color patterns, screen and window dumps to printer
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming

Item #B052 ISBN 1-55755-052-2. Suggested retail price: $34.95
Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **Item #S052. $14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

The Best Amiga Tricks & Tips

The Best Amiga Tricks & Tips is a great collection of Workbench, CLI and BASIC programming "quick-hitters", hints and application programs. You'll be able to make your programs more user-friendly with pull-down menus, sliders and tables. BASIC programmers will learn all about gadgets, windows, graphic fades, HAM mode, 3D graphics and more.



The Best Amiga Tricks & Tips includes a complete list of BASIC tokens and multitasking input and a fast and easy print routine. If you're an advanced programmer, you'll discover the hidden powers of your Amiga.

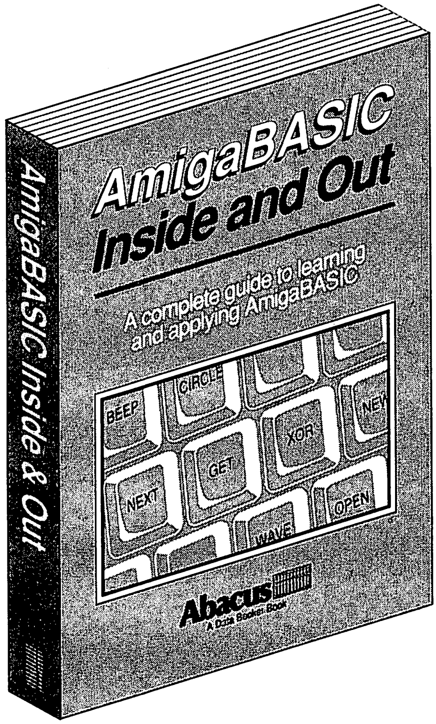
- Using the new AmigaDOS, Workbench and Preferences 1.3 and Release 2.0
- Tips on using the new utilities on Extras 1.3
- Customizing Kickstart for Amiga 1000 users
- Enhancing BASIC using ColorCycle and mouse sleeper
- Disabling FastRAM and disk drives
- Using the mount command
- Writing an Amiga virus killer program
- Disk drive operations and disk commands
- Learn machine language calls.

Item # B107 ISBN 1-55755-107-3.
Suggested retail price $29.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga BASIC: Inside and Out

Amiga BASIC: Inside and Out is the definitive step-by-step guide to programming the Amiga in BASIC. This huge volume should be within every Amiga user's reach. Every Amiga BASIC command is fully described and detailed. In addition, **Amiga BASIC: Inside and Out** is loaded with real working programs.



Topics include:

- Video titling for high quality object animation
- Bar and pie charts
- Windows
- Pull down menus
- Mouse commands
- Statistics
- Sequential and relative files
- Speech and sound synthesis

Item #B87X ISBN 0-916439-87-9. Suggested retail price: $24.95

Companion Diskette available: *Contains every program listed in the book complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **Item #S87X. $14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga® Intern

Includes Release 2.0 Workbench 2.0 information

Amiga Intern is the definitive reference library in one guide for all Amiga 500, 1000, 2000, 2500 and 3000 users. Amiga Intern will explain the internals of the Amiga 3000, Release 2.0 (Workbench 2.0) of the operating system and Kickstart 2.0. *Amiga Intern* teaches you important information about the ARexx programming language that is bundled with all Amiga 3000s. *Amiga Intern* is divided into three easy-to-use sections for hardware, operating system and ARexx programming. If you are interested in the Amiga hardware you'll learn all the essentials of the 68030 processor and its environment. *Amiga Intern* also contains an extensive reference section on Kickstart 2.0 and much more.

US $39.95/ CDN $49.95

ISBN 1-55755-148-0

The definitive reference book for all Amiga computers

A short overview of the contents:

Hardware:
- 68030 and 68881/82 specifications
- MMU, FPU and ECS
- Zorro II bus system
- SCSII controller
- FlickerFixer

System Software:
- Kickstart 2.0 innovations
- Workbench 2.0 innovations
- Overview of library functions
- Program samples

ARexx:
- History of development
- Syntax oriented command lists
- Basic elements
- Special language elements
- Function libraries
- Program samples

Computer Book Category
Computer: Amiga
Level: Intermediate/Advanced

Abacus
5370 52nd Street SE • Grand Rapids, MI 49512