

AMIGA<sup>®</sup> ROM KERNEL  
REFERENCE MANUAL:  
LIBRARIES & DEVICES

**REVISED & UPDATED**

AMIGA<sup>®</sup>

COMMODORE-AMIGA,  
INCORPORATED



**Amiga**  
**ROM Kernel Reference Manual:**  
**Libraries and Devices**

Commodore-Amiga, Incorporated

Amiga Technical Reference Series



Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California New York  
Don Mills, Ontario Wokingham, England Amsterdam Bonn  
Sydney Singapore Tokyo Madrid San Juan



## Amiga ROM Kernel Reference Manual: Libraries and Devices

This manual corresponds to the V1.3 Commodore Amiga system software release.

The text of the original version of this manual was written by Amiga engineers and writers including:

Bruce Barrett, Mark Barton, Dave Berezowski, Bob "Kodiak" Burns, Susan Deyl, Sam Dicker, Andy Finkel, Larry Hildenbrand, Neil Katin, Joe Katz, Dale Luck, Dave Lucas, Jim Mackraz, R.J. Mical, Bob Pariseau, Rob Peck, Tom Pohorsky, Carl Sassenrath, and Stan Shepard

This manual was revised and updated by Commodore Applications and Technical Support (CATS) and Commodore-Amiga engineers including:

Dan Baker, Steve Beats, Dave Berezowski, Ray Brand, Bob Burns, Peter Cherna, Eric Cotton, Ken Farinsky, Andy Finkel, Mark Green, Randell Jesup, David Junod, Kevin Klop, Adam K. Levin, Dale Luck, Jim Mackraz, Bryce Nesbitt, Nancy Rains, Marc Rifkin, Michael Sinz, Darius Taghavy, Ewout Walraven, Bart Whitebook, and Rob Wyesham. Special thanks to Dave Lucas for his contributions to this manual.

Project Leader: Ken Farinsky

Printed on the the NEC LC890 Silentwriter™ laser printer.

This manual is dedicated to the machine it was published on, our Amiga AMIX™ node "CBMCATS", and to the individual Amigas that allowed each writer to edit, correspond, compile, test, typeset, and print concurrently.

Copyright © 1990 by Commodore-Amiga, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps. Amiga is a registered trademark of Commodore-Amiga, Incorporated and is used herein with their permission. Amiga 500, Amiga 1000, Amiga 2000, AmigaDOS, Amiga Workbench, Amiga Kickstart, the Boing! and rainbow Checkmark logos are trademarks of Commodore-Amiga, Inc. 68000, 68020, 68030, 68040 and Motorola are trademarks of Motorola, Inc. CBM, Commodore, the Commodore logo, and AUTOCONFIG are registered trademarks of Commodore Electronics Limited. Alphacom is a registered trademark and Alphapro is trademark of Alphacom, Inc. Aztec C and Manx are trademarks of Manx Software Systems. Brother is a registered trademark of Brother Industries, Ltd. Canon is a registered trademark of Canon USA Inc. CAPE and Inovatronics are trademarks of Inovatronics, Inc. Centronics is a registered trademark of Centronics Data Computer Corp. ColorMaster is a trademark of CalComp. Diablo is a registered trademark of Xerox Corporation. Epson is registered trademark of Epson America, Inc. Hisoft and Devpac are trademarks of HiSoft. IBM is a registered trademark and Proprinter is a trademark of International Business Machines Corp. Imagewriter and Apple II are trademarks of Apple Computer, Inc. LaserJet and PaintJet are trademarks of the Hewlett Packard Company. Lattice is a registered trademark of Lattice, Inc. LetterPro 20 is a trademark of Qume Corporation. NEC is a registered trademark of NEC Information Systems. Okidata is a registered trademark of Okidata, a division of Oki America, Inc. Okimate 20 is a trademark of Okidata, a division of Oki America, Inc. Pinwriter is a registered trademark of NEC Information Systems. UNIX is a registered trademark of AT&T.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder. Printed in the United States of America from camera-ready mechanicals supplied by the authors. Published simultaneously in Canada.

The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only under the terms of that agreement.

Commodore item number: 363099-01

ISBN 0-201-18187-8

BCDEFGHIJ-AL-93210

Second Printing, June 1990

**WARNING:** The information described in this manual may contain errors or bugs, and may not function as described. An attempt has been made to warn software developers of known bugs, however, not all bugs will be so marked. All information is subject to enhancement or upgrade for any reason including to fix bugs, add features or change performance. As with all software upgrades, full compatibility, although a goal, cannot be guaranteed, and is in fact unlikely.

**DISCLAIMER:** COMMODORE-AMIGA, INC. ("COMMODORE") AND THE AUTHORS MAKE NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THE INFORMATION DESCRIBED HEREIN, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. SUCH INFORMATION IS PROVIDED ON AN "AS IS" BASIS. THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE USER. CURRENT INFORMATION IS SUBJECT TO FUTURE CHANGE WITHOUT NOTICE. SHOULD THE INFORMATION PROVE DEFECTIVE, THE USER (AND NOT THE AUTHORS, COMMODORE, THEIR DISTRIBUTORS NOR THEIR RETAILERS) ASSUMES THE ENTIRE COST OF ALL NECESSARY DAMAGES. IN NO EVENT WILL COMMODORE OR THE AUTHORS BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE INFORMATION EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION BY COMMODORE MAY NOT APPLY.



# Introduction

The Amiga family of computers consists of several models, each of which has been designed on the same premise - to provide the user with a low cost computer that features high cost performance. There are three distinct models that make up the the Amiga line: the A500, the A1000, the A2000. Though the models differ in price and features, they have a common hardware nucleus, and use the same powerful proprietary operating system software.

## About this book

The Amiga Technical Reference Series is the official guide to programming the Commodore-Amiga computers. This revised edition has been updated for version 1.3 of the Amiga operating system and the new Amiga computer systems. The series has been reorganized into three volumes. This volume, the *Amiga ROM Kernel Reference Manual: Libraries and Devices*, contains tutorial-style chapters on the use of the Amiga system library functions and device commands.

The other manuals in this series are the *Amiga Hardware Reference Manual*, a guide to hardware level programming of the Amiga custom and peripheral chips, and the *Amiga Rom Kernel Reference Manual: Includes and Autodocs*, an alphabetically organized reference of autodoc function summaries, listings of the Amiga system include files, and the IFF Interchange File Format.

## System Software Architecture

The Amiga kernel consists of a number of system modules, known as **Libraries** and **Devices**, some of which reside in ROM (or in the protected *kickstart* memory on an A1000) and others that are loaded as needed from the system disk. Each **Library** contains a set of functions for interacting with a particular part of the operating system. Each **Device** provides commands and functions for interacting with a particular parts of the Amiga hardware.

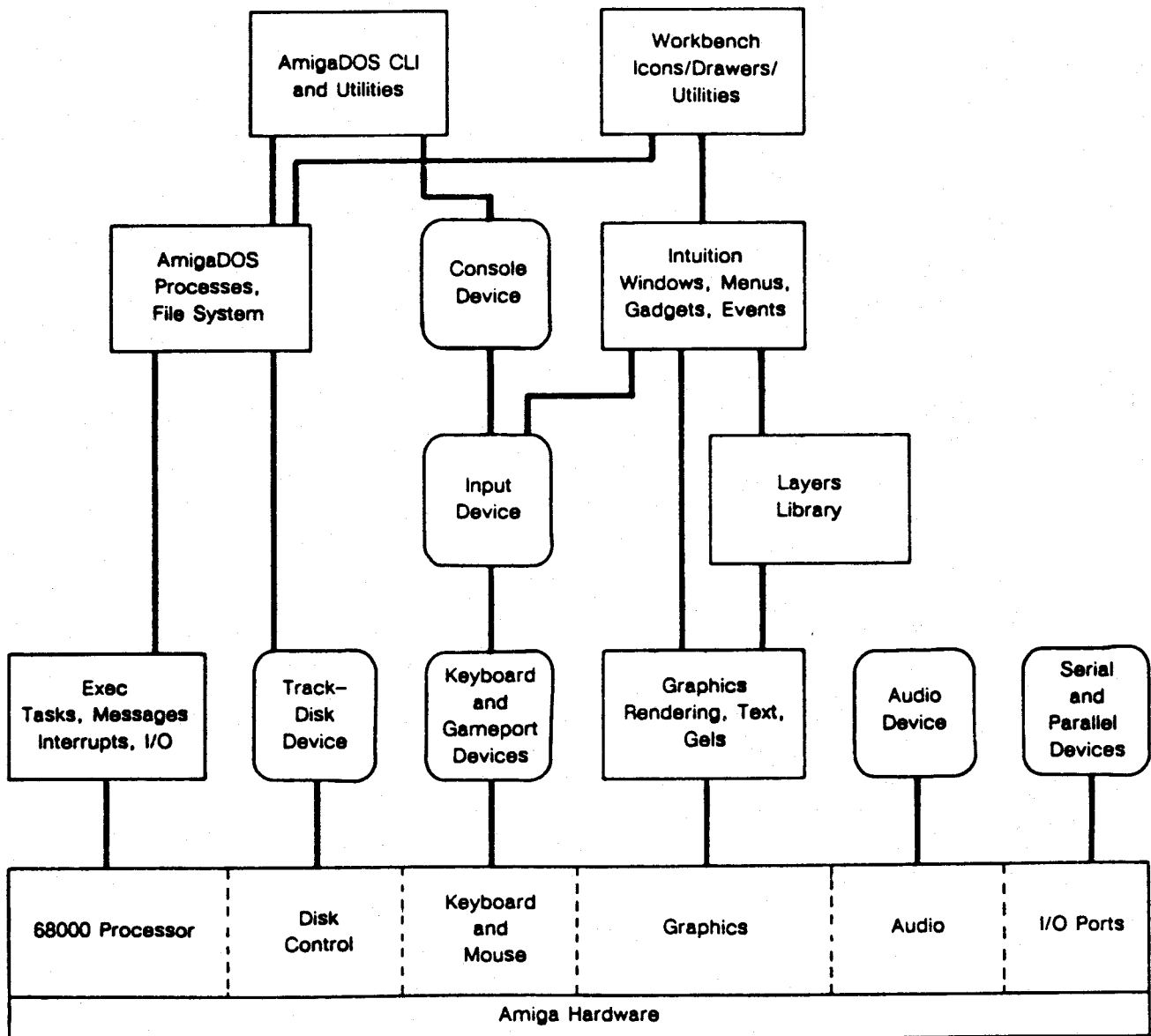
At the top of the hierarchy are Workbench and the Command Line Interface (CLI), the user-visible portions of the system. Workbench uses Intuition to produce its displays, and AmigaDOS to interact with the filing system. Intuition, in turn, uses the input device to retrieve its input and the graphics and layers library routines to produce its output.

AmigaDOS controls processes and maintains the filing system. It is in turn built upon Exec, which manages tasks, task switching, interrupt scheduling, message-passing, I/O, and many other functions.

At the lowest level of the hierarchy is the Amiga hardware itself. Just above the hardware are the modules that control the hardware directly. Exec controls the 68000, scheduling its time among tasks and maintaining its interrupt vectors, among other things. The trackdisk device is the lowest-level interface to the disk hardware, performing disk-head movement and raw disk I/O. The keyboard and gameport devices handle the keyboard and gameport hardware, queuing up input events for the input device to process. The audio device, serial device, and parallel device handle their respective hardware. Finally, the graphics library handles the interface to the graphics hardware.



The following diagram illustrates the hierarchy of the Amiga system software modules:





## About the Examples

68000 assembly language examples have been assembled under either the Metacomco assembler V11.0, the Inovatronics CAPE assembler V2.0, or the HiSoft Devpac assembler V1.2. No substantial changes should be required to switch between assemblers.

C examples have been compiled under Lattice C, version 5.02 or 5.04. Except as noted, C examples were generally compiled with the following Lattice compiler flags:

LC -b1 -cfist -v -y

where:

-b1 = small data model...  
-cf = Function prototypes...  
    i = Don't multi-include include files...  
    s = Make all literal strings that are the same  
        be stored in the same place...  
    t = Warnings for structures used before defined...  
-v = Turn off stack checking...  
-y = Load the LinkerDB (a4) at start of all functions...

The -v and -y flags are generally only needed when parts of the program code will be called directly by the system (for instance interrupt servers, handlers, and subtasks). as shown in some of the manual examples.

Note that some of these flags have been picked to make the C source source as vanilla as possible.

Code was generally linked by either adding a -L in the LC command line (ie. let the compiler select the startup and the linker libraries), or by explicitly using Blink to link with startup c.o and library LC.lib, amiga.lib. Notes on exact flags and linkage may be found in the initial comment of many manual examples. Note that most manual examples assume 32-bit ints. If your development environment assumes 16-bit ints, you may need to explicitly cast or type certain arguments as longs (for example: 1L << sigbit instead of 1 << sigbit).

An effort was made to keep the C code examples as standard as possible, for easy porting to other compilers. The examples should port fairly easily to the Manx Aztec C68K compiler. Some necessary modifications for porting to Manx would be:

1. Replace `#include <protos/all.h>` with `#include <functions.h>`
2. Replace CXBRK line (if any) which disables Lattice CTRL-C handling with:  

```
/* Before main(), reference abort enable */  
extern int Enable_Abort;  
/* As first line in main() turn off CTRL-C */  
Enable_Abort=0;
```
3. Check your compiler manual to chose compiler flags with similar effect to those the example was compiled with.



# Amiga Development Guidelines

The environment of the Amiga computer is quite different than that of many older computers. The Amiga is multitasking, which means multiple programs must share the same machine without interfering with each other. It also means that certain guidelines must be followed during programming.

- Always make sure you actually GET what you ask for. This applies to memory allocations, windows, screens, file handles, libraries, devices, ports, etc. Where an error value or return is possible, ensure that there is a reasonable failure path. Many poorly written programs will *appear* to be reliable, until some error condition (such as memory full or a disk problem) causes the program to continue with an invalid or null pointer, or branch to untested error handling code.
- Always clean up after yourself. This applies for both normal program exit and program termination due to error conditions. Anything that was opened must be closed, anything allocated must be deallocated. It is generally correct to do closes and deallocations in reverse order of the opens and allocations. Be sure to check your development language manual and startup code; some items may be closed or deallocated automatically, especially in abort conditions. If you write in the C language, make sure to provide your own CTRL-C handling to free any Amiga-specific resources and structures.
- Remember that memory, peripheral configurations, and ROMs differ between models and between individual systems. Do not make assumptions about memory address ranges, storage device names, or the locations of system structures or code. Never call ROM routines directly. Beware of any example code you find which calls routines at addresses in the \$F00000 range. These are ROM routines and they will move with every OS release. The only supported interface to system ROM code is through the provided library, device, and resource calls.
- Do not assume library bases or structures will exist at any particular memory location. The only absolute in the system is address 0x00000004, which contains a pointer to the exec.library base. Do not modify or depend on the format of private system structures. This includes the poking of copper lists, memory lists, and library bases.
- Do not assume that programs can access hardware resources directly. Most hardware is controlled by system software and resources that will not respond well to interference. Shared hardware requires programs to use the proper sharing protocols. Using the defined interface enhances the probability that your software will continue to operate on future Amiga computers.
- Do not access shared data structures directly without the proper mutual exclusion (locking). Remember that other tasks may be accessing the same structures.
- Do not assume that system flags and options are limited to current possible values or choices. For example, do not assume a display must be PAL if not NTSC, and do not assume an event must be SELECTUP if not SELECTDOWN. Explicitly check for the values or choices you support, and provide a default case for everything else (for example a default ReplyMsg() for unknown IntuiMessages).
- The system does not monitor the size of a program's stack. Take care that your program does not cause stack overflow, and provide enough leeway for the possibility that future revisions of system functions might require additional stack space.

- If your program waits for external events like menu selection or key-strokes, do not bog down the multitasking system by busy-waiting in a loop. Instead, let your task go to sleep by Wait()ing on its signal bits. For example:

```
ULONG windowSig = 1L << window->UserPort->mp_SigBit;
ULONG consoleSig = 1L << consolePort->mp_SigBit;
signals = (ULONG) Wait(windowSig | consoleSig);
```

This turns the signal bit number for each port into a mask, then combines them as the argument for the exec.library/Wait() function. When your task is awakened, handle all of the messages at each port where the SigBit is set. There may be more than one message per port, or no messages at the port. Make sure that you ReplyMsg() to all messages that are not replies themselves.

- Tasks (and Processes) execute in 68000 processor user mode. Supervisor mode is reserved for interrupts, traps, and task dispatching. Take extreme care if your code executes in supervisor mode. Exceptions while in supervisor mode are deadly.
- Most system functions require a particular execution environment. All DOS functions and any functions that might call DOS (such as the opening of a disk-resident library, font, or device) can only be executed from a process. A task is not sufficient. Most other ROM Kernel functions may be executed from tasks. Only a few may be executed from interrupts.
- Do not disable interrupts or multitasking for long periods. If you use Forbid() or Disable(), be aware that use of any system function that Waits will temporarily suspend your Forbidden or Disabled state, and allow multitasking and interrupts to occur. Such functions include almost all forms of DOS and device IO, including common "stdio" functions like "printf".
- Do not tie up system resources unless it is absolutely necessary. For example, if your program does not require constant use of the printer, open the printer.device only when you need it. This will allow other tasks to use the printer while your program is running. You must provide a reasonable error response if a resource is not available when you need it.
- Check for memory loss. Operate your program, then exit. Write down the amount of free memory. Repeat the operation of your program and exit. The amount of free memory remaining should be *exactly* the same. Any difference may signal some serious problem in your cleanup. A useful tool for memory testing is the "LoadWB -debug" command; this will start the Workbench tool with a special invisible debug menu. The "flushlibs" option of this menu can cause unused libraries and devices to be flushed out of memory. (The "debug" option invokes the ROM debugger, RomWack, on the serial port at 9600 baud.)
- All data for the custom chips *must* reside in CHIP type memory. This includes bitplanes, sound samples, trackdisk buffers, and images for sprites, bobs, pointers, and gadgets. (On the current generation of machines, CHIP memory is the lowest 512K or 1 Meg of memory in the system.) The AllocMem() call takes a flag (MEMF\_CHIP) for specifying CHIP type memory. On machines with expansion (FAST) memory, programs will by default load into FAST memory, and allocations which are not specified as MEMF\_CHIP will receive FAST memory. On machines with CHIP memory only, all program code and data, and all allocations will automatically be in CHIP ram, and this can mask the symptoms of improper placement of custom chip data or buffers. Most compilers have options to mark specific data structures or object modules so that they will load into CHIP ram. Some older compilers provide the Atom utility for marking object modules. If this method is unacceptable, use the AllocMem() call to dynamically allocate CHIP memory, and copy your data there. When making allocations that do not require CHIP memory, do not explicitly ask for MEMF\_FAST. Ask for memory type 0L or MEMF\_PUBLIC as appropriate. If FAST memory is available, you will get FAST memory.



- Do not use software delay loops! Under the multitasking operating system, the time spent in a loop can be better used by other tasks. Even ignoring the effect of multitasking, timing loops are inaccurate and will wait widely varying amounts of time depending on the configuration and processor of the computer. The timer.device can provide accurate timing for use under the multitasking system. The AmigaDOS Delay() function provides a simple interface for longer delays. The 8520 I/O chips provide timers for developers who are bypassing the operating system (see the *Amiga Hardware Reference Manual* for more information).
- Obey structure conventions!
  - All non-byte fields must be word aligned.
  - All address pointers should be 32 bits (not 24 bits). The upper byte must never be used for data.
  - Fields that are not defined to contain particular initial values *must* be initialized to zero. This includes pointer fields.
  - All reserved or unused fields *must* be initialized to zero for future compatibility.
  - Data structures to be accessed by the custom chips, public data structures (such as a task control block), and structures which must be longword aligned must NOT be allocated on a program's stack.
  - Dynamic allocation of structures with AllocMem provides longword aligned memory of a specified type with optional initialization to zero, which is useful in the allocation of structures.

#### **For 68010/68020/68030 compatibility**

Special care must be taken to be compatible with the entire family of 68000 processors:

- Do not use the upper 8 bits of a pointer for storing unrelated information. The 68020 uses all 32 bits for addressing.
- Do not use signed variables or signed math for addresses.
- Do not use self-modifying code.
- Do not use software delay loops, and do not make assumptions about the order in which asynchronous tasks will finish.
- The stack frame used for exceptions is different on each member of the 68000 family. The type identification in the frame must be checked! In addition, the interrupt autovectors may reside in a different location on processors with a VBR register.
- Do not use the "MOVE SR,..." instruction! This 68000 instruction acts differently on other members of the 68000 family. If you wish a copy of the processor condition codes, use the exec.library/GetCC() function.
- Do not use the CLR instruction on a hardware register which is triggered by access. The 68020 CLR instruction does a single Write access. The 68000 CLR instruction does a Read access first, then a Write access. This can cause a hardware register to be triggered twice. Use MOVE(.whatever) #0, address instead.

## **Hardware Programming Guidelines**

If you find it necessary to program the hardware directly, then it is your responsibility to write code which will work properly on various various models and configurations. Be sure to properly request and gain control of the hardware resources you are manipulating, and be especially careful in the following areas:

- All custom chip registers are READ ONLY or WRITE ONLY. Do not read Write-only registers, and do not write to Read-only registers.
- Do not write spurious data to, or interpret undefined data from, currently unused bits or addresses in the custom chip space. To be software-compatible with future chip revisions, all undefined bits must be set to zeros on writes, and must be masked out on reads before interpreting the contents of the register.
- Do not write past the current end of custom chip space. Custom chips may be extended or enhanced to provide additional registers, or to use bits which are currently undefined in existing registers.
- Do not read, write, or use any currently undefined address ranges. The current and future usage of such areas is reserved by Commodore and is subject to change.

## **Additional Assembler Development Guidelines**

- Do not use the "TAS" instruction on the Amiga. System DMA can conflict with this instruction's special indivisible read-modify-write cycle .
- System functions must be called with A6 containing the library or device base. Libraries and devices assume A6 is valid at the time of any function call. Even if a particular function does not currently require its base register, you must provide it for compatibility with future system software releases.
- Except as noted, system library functions use registers D0, D1, A0, and A1 as scratch registers and you must consider their former contents to be lost after a system library call. The contents of all other registers will be preserved. System functions which provide a result will return the result in D0.
- System functions that return a result may not necessarily affect the processor condition codes. The caller must test the returned value before acting on a condition code. This is usually done with a TST or MOVE instruction.

## Commodore Applications and Technical Support (CATS)

Commodore currently maintains a technical support group dedicated to helping developers achieve their goals with the Amiga. Available technical support programs are tailored both to the needs of smaller independent developers and larger corporations. Subscription to the support publication *AmigaMail* is available to anyone with an interest in the latest news, Commodore software and hardware changes, and tips for developers.

To request an application for the Commodore-Amiga Developer Programs, lists of CATS technical publications, or information regarding electronic developer support, send a self-addressed, stamped, 9" x 12" envelope to:

CATS-Information  
1200 West Wilson Drive  
West Chester, PA 19380-4231

## Error Reports

In a complex technical manual, errors are often found after publication. When errors in this manual are found or reported through the proper channels, they will be filed for reference during future revisions. Important updates or corrections may be published in the *AmigaMail* technical support publication.

Bug reports can be sent to Commodore electronically or by mail. Submitted reports must be clear, complete, and concise. Reports must include a telephone number and enough information so that the bug can be quickly verified from your report (i.e. please describe the bug *and* the steps that preceded it).

Amiga Software Engineering Group  
ATTN: BUG REPORTS  
Commodore Business Machines  
1200 Wilson Drive  
West Chester, PA 19380-4231  
USA

BIX: afinkel    USENET: bugs@commodore.COM or uunet!cbmvax!bugs  
Enhancement requests may be mailed to "suggestions" instead of "bugs".

## Cautions

Additional warnings and specifications on the usage of individual Amiga system software functions and commands may be found in the Addison-Wesley *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. Additional warnings and specifications regarding programming of the Amiga hardware may be found in the Addison-Wesley *Amiga Hardware Reference Manual*.

### WARNING

Failure to regard warnings and specifications in this and other manuals can result in system failures including, but not limited to, incorrect operation, corruption of memory, corruption of storage media, and incompatibility with operating system revisions and hardware configurations.



# Table of Contents

<b>Chapter 1 Intuition: Introduction .....</b>	<b>1</b>
How the User Sees an Intuition Application.....	2
The Right Approach to Using Intuition .....	5
Intuition Components.....	6
General Program Requirements and Information .....	6
THE INTUITION EVENT LOOP .....	7
<b>Chapter 2 Intuition: Screens.....</b>	<b>11</b>
About Screens .....	11
The Workbench Screen.....	13
Custom Screens.....	15
Screen Characteristics .....	16
DISPLAY MODES.....	16
DEPTH AND COLOR.....	17
TYPE STYLES .....	18
SCREEN POSITION AND DIMENSIONS.....	19
APPLICATION-MANAGED CUSTOM SCREENS .....	19
SCREEN TITLE .....	20
CUSTOM GADGETS .....	20
Screen Data Structures .....	20
NEWSCREEN STRUCTURE .....	20
SCREEN STRUCTURE .....	23
SCREEN FUNCTIONS.....	23
Examples.....	26
LOW-RES SCREEN EXAMPLE.....	26
DUAL-PLAYFIELD SCREEN EXAMPLE .....	28
<b>Chapter 3 Intuition: Windows.....</b>	<b>33</b>
About Windows .....	33
WINDOW INPUT/OUTPUT.....	35
OPENING WINDOWS .....	35
WINDOWS AND SCREENS.....	35
THE ACTIVE WINDOW .....	36
CLOSING WINDOWS.....	36
SPECIAL WINDOW TYPES.....	37
WINDOW GADGETS.....	39
WINDOW BORDERS .....	41
PRESERVING THE WINDOW DISPLAY .....	42
REFRESHING THE WINDOW DISPLAY .....	45
WINDOW POINTER .....	46

GRAPHICS AND TEXT IN WINDOWS .....	47
WINDOW COLORS .....	47
WINDOW DIMENSIONS.....	47
The New Window Structure.....	47
WINDOW STRUCTURE.....	52
WINDOW FUNCTIONS.....	52
SETTING UP A SUPERBITMAP WINDOW .....	56
SETTING UP A CUSTOM POINTER .....	57
Examples.....	59
BACKDROP WINDOW EXAMPLE.....	59
TWO WINDOW EXAMPLE .....	62
INVISIBLE POINTER EXAMPLE .....	66
SUPERBITMAP WINDOW EXAMPLE.....	66
<b>Chapter 4 Intuition: Gadgets.....</b>	<b>71</b>
About Gadgets.....	71
System Gadgets.....	73
SIZING GADGET .....	74
DEPTH-ARRANGEMENT GADGETS .....	74
DRAGGING GADGET .....	74
CLOSE GADGET.....	74
Application Gadgets.....	75
RENDERING GADGETS.....	75
USER SELECTION OF GADGETS .....	77
GADGET SELECT BOX .....	77
GADGET POINTER MOVEMENTS .....	78
GADGETS IN WINDOW BORDERS.....	79
MUTUAL EXCLUDE .....	79
ALLOWABLE TYPE OF GADGETS FOR MUTUAL EXCLUSION.....	79
ALLOWABLE TYPES OF HIGHLIGHTING FOR MUTUAL EXCLUSION .....	79
HANDLING OF MUTUALLY EXCLUSIVE GADGETS .....	80
GADGET HIGHLIGHTING .....	80
GADGET ENABLING AND DISABLING.....	81
GADGET REFRESHING BY INTUITION.....	81
GADGET REFRESHING BY YOUR PROGRAM.....	82
BOOLEAN GADGET TYPE .....	82
MASKED BOOLEAN GADGETS .....	83
PROPORTIONAL GADGET TYPE.....	83
SCROLLING THROUGH GRAPHICAL OR TEXTUAL INFORMATION.....	85
ADJUSTING A LEVEL .....	85
STRING GADGET TYPE.....	87
INTEGER GADGET TYPE .....	88
COMBINING GADGET TYPES .....	89
Gadget Activation Messages.....	89
GADGET STRUCTURE.....	90
FLAGS .....	92
ACTIVATION FLAGS .....	93
SPECIALINFO DATA STRUCTURES .....	94
GADGET FUNCTIONS .....	98
Example .....	100
<b>Chapter 5 Intuition: Menus .....</b>	<b>109</b>
About Menus.....	109
SUBMITTING AND REMOVING MENU STRIPS .....	111

ABOUT MENU ITEM BOXES .....	112
ACTION/ATTRIBUTE ITEMS AND THE CHECKMARK .....	113
TOGGLE-SELECTION.....	114
MUTUAL EXCLUSION .....	114
COMMAND-KEY SEQUENCES AND IMAGERY .....	115
ENABLING AND DISABLING MENUS AND MENU ITEMS .....	116
CHANGING MENU STRIPS .....	116
MENU NUMBERS AND MENU SELECTION MESSAGES .....	116
HOW MENU NUMBERS REALLY WORK .....	117
INTERCEPTING NORMAL MENU OPERATIONS .....	118
REQUESTERS AS MENUS .....	120
MENU STRUCTURES .....	120
MENU FUNCTIONS .....	124
Example .....	125
<b>Chapter 6 Intuition: Requesters and Alerts .....</b>	<b>135</b>
About Requesters .....	135
RENDERING REQUESTERS .....	137
REQUESTER DISPLAY POSITION .....	138
DOUBLE-MENU REQUESTERS .....	138
GADGETS IN REQUESTERS .....	138
IDCMP REQUESTER FEATURES.....	139
A SIMPLE, AUTOMATIC REQUESTER .....	139
User Rendering.....	140
REQUESTER STRUCTURE .....	141
THE VERY EASY REQUESTER .....	144
REQUESTER FUNCTIONS .....	144
Alerts.....	146
Examples .....	147
AUTOREQUEST EXAMPLE.....	147
DISPLAY ALERT EXAMPLE .....	149
DOUBLE MENU REQUEST EXAMPLE.....	151
<b>Chapter 7 Intuition: Input and Output Methods.....</b>	<b>155</b>
An Overview of Input and Output .....	155
About Input and Output .....	156
Using the IDCMP.....	161
INTUIMESSAGES .....	162
IDCMP FLAGS .....	163
SETTING UP YOUR OWN IDCMP MONITOR TASK AND USER PORT .....	167
Examples.....	168
<b>Chapter 8 Intuition: Images, Line Drawing, And Text .....</b>	<b>173</b>
Using Intuition Graphics.....	173
DISPLAYING BORDERS, INTUITEXT, AND IMAGES .....	174
CREATING BORDERS .....	174
CREATING TEXT .....	178
CREATING IMAGES .....	180
INTUITION GRAPHICS FUNCTIONS .....	187
<b>Chapter 9 Intuition: Mouse and Keyboard .....</b>	<b>189</b>
About the Mouse.....	189
Mouse Messages .....	191
About the Keyboard.....	191



Using the Keyboard as an Alternate to the Mouse.....	193
MOUSE AND KEYBOARD EXAMPLE .....	193
<b>Chapter 10 Intuition: Other Features.....</b>	<b>199</b>
Introduction.....	199
Locking IntuitionBase.....	199
Easy Memory Allocation and Deallocation .....	200
INTUITION HELPS YOU REMEMBER .....	200
HOW TO REMEMBER .....	201
THE REMEMBER STRUCTURE .....	201
AN EXAMPLE OF REMEMBERING .....	202
Preferences .....	204
PREFERENCES STRUCTURE .....	205
PREFERENCES FUNCTIONS.....	208
Remaking the ViewPorts .....	209
Current Time Values.....	209
Flashing the Display.....	210
Using Sprites in Intuition Windows and Screens.....	210
<b>Chapter 11 Intuition: Style.....</b>	<b>211</b>
Menu Style .....	212
PROJECT MENUS.....	213
EDIT MENUS.....	213
Gadget Style.....	214
Requester Style .....	215
The Sides of Good and Bad .....	215
Command Key Style .....	216
The HELP key.....	217
Cursor Key Style.....	217
Mouse Style.....	218
Window and Screen Style.....	218
Miscellaneous Style Notes .....	218
<b>Chapter 12 Intuition: Functions.....</b>	<b>221</b>
Assembly Language Conventions.....	223
<b>Chapter 13 Intuition: Internal Procedures.....</b>	<b>225</b>
SetPrefs().....	226
AlohaWorkbench().....	226
Intuition() .....	227
<b>Chapter 14 Exec: Libraries.....</b>	<b>229</b>
What Is a Library?.....	229
How To Access a Library .....	230
OPENING A LIBRARY.....	230
CALLING A LIBRARY FUNCTION.....	232
USING A LIBRARY TO REFERENCE DATA.....	233
CACHING LIBRARY POINTERS.....	233
CLOSING A LIBRARY .....	234
Adding a Library .....	234
RESIDENT (ROMTAG) STRUCTURE .....	234
MINIMUM SUBSET OF LIBRARY CODE VECTORS .....	235
STRUCTURE OF A LIBRARY NODE.....	236
CHANGING THE CONTENTS OF A LIBRARY .....	236

Relationship of Libraries to Devices.....	237
<b>Chapter 15 Exec: Memory Allocation.....</b>	<b>239</b>
Introduction.....	239
Using Memory Allocation Routines.....	240
MEMORY REQUIREMENTS.....	240
SAMPLE CALLS FOR ALLOCATING SYSTEM MEMORY.....	241
SAMPLE FUNCTION CALLS FOR FREEING SYSTEM MEMORY.....	241
Using Memory Information Routines.....	242
MEMORY REQUIREMENTS.....	242
SAMPLE CALLS INFORMATION ROUTINES.....	242
Using Memory Copy Routines.....	242
SAMPLE CALLS FOR COPYING SYSTEM MEMORY.....	243
SUMMARY OF SYSTEM CONTROLLED MEMORY HANDLING ROUTINES.....	243
Allocating Multiple Memory Blocks.....	243
Memory Allocation and Tasks.....	246
SUMMARY OF MULTIPLE MEMORY BLOCKS ALLOCATION ROUTINES.....	246
Managing Memory With Allocate() And Deallocate().....	246
Allocating Memory at an Absolute Address.....	248
<b>Chapter 16 Exec: Lists and Queues.....</b>	<b>249</b>
Introduction.....	249
List Structure.....	250
List Functions.....	250
INSERTION AND REMOVAL.....	250
SPECIAL CASE INSERTION.....	251
SPECIAL CASE REMOVAL.....	251
MINLIST / MINNODE OPERATIONS.....	251
PRIORITIZED INSERTION (QUEUES).....	251
SEARCHING BY NAME.....	252
NODE STRUCTURE DEFINITION.....	253
NODE INITIALIZATION.....	254
LIST HEADER STRUCTURE DEFINITION.....	254
HEADER INITIALIZATION.....	255
MORE ON THE USE OF NAMED LISTS.....	257
List Macros for Assembly Language Programmers.....	257
Empty Lists.....	257
Scanning a List.....	258
Important Note - Shared Lists.....	260
<b>Chapter 17 Exec: Tasks.....</b>	<b>261</b>
Introduction.....	261
Tasks on the Amiga.....	262
SCHEDULING.....	262
WAITING.....	263
TASK STATES.....	263
TASK QUEUES.....	264
PRIORITY.....	264
STRUCTURE.....	264
Creation.....	265
CREATETASK.....	266
STACK.....	267
Termination.....	268
Signals.....	269

ALLOCATION.....	269
WAITING FOR A SIGNAL .....	270
GENERATING A SIGNAL .....	271
Exclusion.....	271
FORBIDDING .....	271
DISABLING .....	272
SEMAPHORES .....	273
Exceptions.....	274
Traps.....	274
HANDLERS .....	275
TRAP INSTRUCTIONS.....	277
<b>Chapter 18 Exec: Messages and Ports .....</b>	<b>279</b>
Introduction.....	279
Ports .....	280
STRUCTURE .....	280
CREATION .....	281
DELETION.....	282
RENDEZVOUS .....	282
Messages.....	283
PUTTING A MESSAGE .....	283
WAITING FOR A MESSAGE.....	284
GETTING A MESSAGE.....	286
REPLYING .....	286
<b>Chapter 19 Exec: Input/Output.....</b>	<b>289</b>
Introduction.....	289
Request Structure .....	290
Interface Functions.....	291
Standard Device Commands .....	293
Performing I/O .....	294
PREPARATION FOR EXEC I/O.....	294
SYNCHRONOUS REQUESTS .....	294
ASYNCHRONOUS REQUESTS.....	295
COMPLETING THE USE OF A DEVICE .....	296
QUICK I/O.....	296
Example of Device Use.....	297
Standard Devices.....	298
<b>Chapter 20 Exec: Semaphores.....</b>	<b>299</b>
Introduction.....	299
The Signal Semaphore .....	300
<b>Chapter 21 Exec: Interrupts .....</b>	<b>305</b>
Introduction.....	305
SEQUENCE OF EVENTS DURING AN INTERRUPT .....	305
INTERRUPT PRIORITIES .....	306
NONMASKABLE INTERRUPT .....	308
Servicing Interrupts.....	308
INTERRUPT DATA STRUCTURE .....	308
ENVIRONMENT .....	308
INTERRUPT HANDLERS .....	309
INTERRUPT SERVERS .....	314
Software Interrupts.....	316



Disabling Interrupts.....	319
<b>Chapter 22 Exec: ROM-Wack.....</b>	<b>321</b>
Introduction.....	321
Getting to ROM-Wack.....	321
Keystrokes, Numbers, and Symbols .....	322
Register Frame .....	322
Display Frames .....	323
Relative Positioning .....	324
Absolute Positioning .....	324
Altering Memory & Stored Registers .....	324
Execution Control .....	325
Breakpoints .....	325
Other Commands .....	326
Returning to Multitasking After a Crash.....	326
<b>Chapter 23 Graphics: Primitives.....</b>	<b>327</b>
Introduction.....	327
COMPONENTS OF A DISPLAY .....	328
INTRODUCTION TO RASTER DISPLAYS.....	328
INTERLACED AND NON-INTERLACED MODES .....	330
HIGH- AND LOW-RESOLUTION MODES .....	332
FORMING AN IMAGE .....	332
ROLE OF THE COPPER (COPROCESSOR).....	334
Display Routines and Structures .....	335
LIMITATIONS ON THE USE OF VIEWPORTS .....	336
CHARACTERISTICS OF A VIEWPORT.....	337
VIEWPORT SIZE SPECIFICATIONS .....	337
VIEWPORT COLOR SELECTION.....	339
VIEWPORT DISPLAY MODES .....	340
VIEWPORT DISPLAY MEMORY .....	343
FORMING A BASIC DISPLAY .....	346
LOADING AND DISPLAYING THE VIEW.....	350
GRAPHICS EXAMPLE PROGRAM .....	350
Advanced Topics.....	354
CREATING A DUAL-PLAYFIELD DISPLAY.....	354
CREATING A DOUBLE-BUFFERED DISPLAY .....	355
EXTRA-HALF-BRITE MODE.....	356
HOLD-AND-MODIFY MODE.....	357
Drawing Routines .....	358
INITIALIZING A BITMAP STRUCTURE.....	358
INITIALIZING A RASTPORT STRUCTURE.....	358
USING THE GRAPHICS DRAWING ROUTINES .....	364
User Copper Lists.....	381
Advanced Graphics Examples .....	385
DUAL-PLAYFIELDS EXAMPLE .....	385
HOLD-AND-MODIFY MODE EXAMPLE .....	388
<b>Chapter 24 Graphics: Text.....</b>	<b>395</b>
Introduction.....	395
Printing Text into a Drawing Area.....	396
CURSOR POSITION .....	396
BASELINE OF THE TEXT .....	396
SIZE OF THE FONT .....	397

PRINTING THE TEXT .....	397
Selecting the Font.....	398
Selecting the Text Color .....	400
Selecting a Drawing Mode.....	400
Effects of Specifying Font Style .....	402
Using a Disk Font .....	403
USING FONTS IN OTHER DIRECTORIES .....	403
Finding Out Which Fonts Are Available .....	404
Contents of a Font Directory.....	408
The Disk Font.....	410
Defining a Font.....	410
THE TEXT NODE.....	410
FONT HEIGHT .....	410
FONT STYLE.....	411
FONT FLAGS.....	411
FONT WIDTH.....	412
FONT ACCESSORS .....	412
CHARACTERS REPRESENTED BY THIS FONT.....	412
THE CHARACTER DATA.....	412
A COMPLETE SAMPLE FONT .....	414
Wrapper code .....	417
<b>Chapter 25 Graphics: Sprites, Bobs and Animation .....</b>	<b>421</b>
Introduction — Some Terms.....	422
Types Of Animation.....	423
AnimComps .....	424
Preparing To Use Graphics Animation .....	425
Using Simple (Hardware) Sprites .....	427
Controlling Sprite DMA .....	429
Accessing A Hardware Sprite .....	429
Changing The Appearance Of A Simple Sprite.....	430
Moving A Simple Sprite .....	430
Relinquishing A Simple Sprite .....	431
Complete Sprite Example .....	431
True VSprites .....	433
VSprite Setup .....	433
SPECIFICATION OF VSPRITE STRUCTURE .....	433
RESERVED VSPRITE MEMBERS .....	435
USING VSPRITE FLAGS.....	436
VSPRITE POSITION .....	436
VSPRITE IMAGE SIZE.....	437
VSPRITE MEMBERS FOR COLLISION DETECTION.....	437
VSPRITE IMAGEDATA .....	437
SPECIFYING THE COLORS OF A VSPRITE.....	438
ADDING A VSPRITE.....	439
REMOVING A VSPRITE .....	439
CHANGING VSPRITES .....	440
VSPRITE ADVANCED TOPICS .....	440
VSPRITE MACHINE.....	440
GETTING THE VSPRITE LIST IN ORDER .....	442
DISPLAYING THE VSPRITES .....	443
Complete VSprite Example .....	445
Using Bobs.....	448
THE VSPRITE STRUCTURE AND BOBS .....	448

VSPRITE FLAGS AND BOBS.....	448
THE BOB STRUCTURE .....	450
LINKING A BOB TO A VSPRITE STRUCTURE .....	452
USING BOB FLAGS.....	453
SPECIFYING THE SIZE OF A BOB .....	454
SPECIFYING THE SHAPE OF A BOB .....	454
SPECIFYING THE COLORS OF A BOB.....	455
OTHER ITEMS INFLUENCING BOB COLORS .....	456
BOB PRIORITIES .....	458
ADDING A BOB .....	459
REMOVING A BOB .....	459
GETTING THE LIST OF BOBS IN ORDER.....	460
DISPLAYING BOBS .....	460
CHANGING BOBS.....	460
COMPLETE BOB EXAMPLE .....	461
DOUBLE-BUFFERING.....	464
Collisions And User Structure Extensions.....	465
DETECTING GEL COLLISIONS .....	465
SETTING UP FOR BOUNDARY COLLISIONS .....	470
ADDING USER EXTENSIONS TO GEL DATA STRUCTURES .....	471
Animation Concepts.....	472
Animation Structures and Controls.....	472
ANIMATION TYPES .....	473
CHARACTERISTICS OF THE ANIMATION SYSTEM .....	475
SPECIFYING ANIMATION COMPONENTS .....	475
SEQUENCING COMPONENTS .....	476
Sequence List Traversal.....	476
Component Ordering.....	476
ANIMATION SEQUENCING .....	478
SPECIFYING TIME FOR EACH IMAGE.....	478
ANOTHER LOOK AT THE ANIMOB .....	479
ADDING ANIMATION OBJECTS .....	479
YOUR OWN ANIMATION ROUTINE CALLS.....	480
MOVING THE OBJECTS.....	481
THE ANIMATE() SYSTEM CALL.....	481
THE ANIMKEY .....	481
STANDARD GEL RULES STILL APPLY .....	482
ANIMATIONS SPECIAL NUMBERING SYSTEM .....	482
Complete Example Program .....	482
 <b>Chapter 26 Layers Library .....</b>	 <b>505</b>
Introduction.....	505
DEFINITION OF LAYERS .....	506
TYPES OF LAYERS SUPPORTED.....	506
Layers Library Routines.....	506
ALLOCATING AND DEALLOCATING LAYER_INFO.....	507
ALLOCATING AND DEALLOCATING LAYERS .....	508
CREATING AND DELETING LAYERS.....	508
MOVING LAYERS.....	508
SIZING LAYERS .....	508
CHANGING A VIEWPOINT .....	508
REORDERING LAYERS .....	509
DETERMINING LAYER POSITION .....	509
INTERTASK OPERATIONS.....	509

SUB-LAYER RECTANGLE OPERATIONS.....	510
The Layer's RastPort.....	511
SIMPLE REFRESH LAYER.....	511
SMART REFRESH LAYER.....	512
SUPERBITMAP LAYER.....	512
BACKDROP LAYER.....	512
Using the Layers Library .....	512
OPENING THE LAYERS LIBRARY .....	512
OPENING THE GRAPHICS LIBRARY .....	513
CREATING A VIEWING WORKSPACE .....	513
CREATING THE LAYERS .....	513
GETTING THE POINTERS TO THE RASTPORTS .....	513
USING THE RASTPORTS FOR DISPLAY .....	514
Regions.....	514
CREATING AND DELETING REGIONS.....	515
INSTALLING REGIONS.....	515
CHANGING A REGION .....	517
CLEARING A REGION.....	518
LAYERS EXAMPLE .....	518
<b>Chapter 27 Expansion Library.....</b>	<b>527</b>
AUTOCONFIG™.....	527
The Expansion Sequence .....	528
Expansion Board Drivers .....	529
DISK BASED DRIVERS .....	529
MakeDosNode AND AddDosNode.....	530
ROM BASED AND AUTOBOOT DRIVERS.....	531
EVENTS AT DIAG TIME .....	532
EVENTS AT ROMTAG INIT TIME.....	537
EVENTS AT BOOT TIME .....	538
RigidDiskBlock and Alternate Filesystems .....	538
RigidDiskBlock.....	539
BadBlockBlock .....	541
PartitionBlock .....	542
FileSysHeaderBlock.....	543
LoadSegBlock.....	544
filesysres.h and i.....	545
<b>Chapter 28 Math Libraries.....</b>	<b>547</b>
Introduction.....	547
FFP Floating Point Data Format .....	548
FFP Basic Mathematics Library .....	549
FFP Transcendental Mathematics Library .....	554
FFP Mathematics Conversion Library .....	560
IEEE Double-Precision Data Format.....	562
IEEE Double-Precision Basic Math Library.....	563
IEEE Double-Precision Transcendental Math Library .....	568
<b>Chapter 29 Translator Library .....</b>	<b>575</b>
OPENING THE TRANSLATOR LIBRARY .....	575
USING THE TRANSLATE FUNCTION .....	576
CLOSING THE TRANSLATOR LIBRARY.....	576
ADDITIONAL NOTES ABOUT TRANSLATE.....	576

<b>Chapter 30 Workbench .....</b>	<b>579</b>
Introduction .....	579
The Icon Library .....	580
The Info File .....	581
THE DISKOBJECT STRUCTURE .....	581
THE GADGET STRUCTURE .....	583
ICONS WITH NO POSITION .....	584
Workbench Environment .....	584
WBSTARTUP MESSAGE .....	584
The ToolTypes Array .....	586
Example Code .....	587
PrArgs.c .....	587
RKM_Icon_Example.c .....	589
PROGRAM STARTUP CODE .....	593
Standard Amiga Startup Source .....	594
 <b>Chapter 31 Audio Device .....</b>	 <b>607</b>
Introduction .....	607
DEFINITIONS .....	608
Audio Functions and Commands .....	609
COMMAND TYPES .....	609
SCOPE OF COMMANDS .....	610
SYSTEM FUNCTIONS .....	610
ALLOCATION AND ARBITRATION .....	614
ALLOCATION/ARBITRATION COMMANDS .....	615
HARDWARE CONTROL COMMANDS .....	618
DOUBLE BUFFERED SOUND EXAMPLE .....	620
 <b>Chapter 32 Clipboard Device .....</b>	 <b>627</b>
Introduction .....	627
Clipboard Commands .....	627
Clipboard Data .....	629
Clipboard Messages .....	629
Multiple Clips .....	630
Example Program .....	631
Support Functions Called from Example Program .....	633
 <b>Chapter 33 Console Device .....</b>	 <b>637</b>
Introduction .....	637
System Functions .....	638
Console I/O .....	638
GENERAL CONSOLE SCREEN OUTPUT .....	638
CONSOLE KEYBOARD INPUT .....	638
Creating an I/O Request .....	640
Opening a Console Device .....	640
THE CONUNIT STRUCTURE .....	641
SENDING A CHARACTER STREAM TO THE CONSOLE DEVICE .....	641
Control Sequences for Window Output .....	642
READING FROM THE CONSOLE .....	647
Closing a Console Device .....	648
Console Device Example Code .....	648
INFORMATION ABOUT THE READ-STREAM .....	653
CURSOR POSITION REPORT .....	654
WINDOW BOUNDS REPORT .....	654

SELECTING RAW INPUT EVENTS.....	655
Complex Input Event Reports.....	656
Using the Console Device Without a Window .....	662
Keymapping .....	663
ABOUT QUALIFIERS.....	665
KEYTYPE TABLE ENTRIES .....	666
STRING-OUTPUT KEYS.....	666
CAPSABLE BIT TABLE.....	667
REPEATABLE BIT TABLE.....	668
KEY MAP STANDARDS .....	668
DEAD-CLASS KEYS .....	669
Double-Dead Keys .....	671
Complete Keymap Source Example .....	672
<b>Chapter 34 Gameport Device.....</b>	<b>683</b>
Introduction.....	683
Gameport Device Commands .....	684
GPD_ASKCTYPE .....	684
GPD_SETCTYPE .....	685
GPD_SETTRIGGER .....	686
GPD_ASKTRIGGER .....	688
GPD_READEVENT.....	688
Joystick Example Program.....	689
<b>Chapter 35 Input Device.....</b>	<b>693</b>
Introduction.....	693
Input Device Commands.....	694
IND_ADDHANDLER COMMAND.....	695
IND_REMHANDLER COMMAND.....	697
IND_WRITEEVENT COMMAND.....	697
IND_SETTHRESH COMMAND.....	699
IND_SETPERIOD COMMAND.....	699
Input Device and Intuition .....	700
Sample Program .....	700
<b>Chapter 36 Keyboard Device .....</b>	<b>705</b>
Introduction.....	705
Keyboard Device Commands .....	706
KBD_READMATRIX.....	706
KBD_ADDRESETHANDLER .....	708
KBD_REMRESETHANDLER .....	709
KBD_RESETHANDLERDONE .....	709
KBD_READEVENT .....	712
Example Keyboard Read-event Program .....	713
<b>Chapter 37 Narrator Device.....</b>	<b>715</b>
Introduction.....	715
The Translator Library .....	716
USING THE TRANSLATE FUNCTION .....	716
The Narrator Device.....	717
OPENING THE NARRATOR DEVICE.....	717
CONTENTS OF THE WRITE REQUEST BLOCK.....	717
CONTENTS OF THE READ REQUEST .....	719
PERFORMING A WRITE AND A READ.....	719



Example Program.....	720
How to Write Phonetically for Narrator.....	720
PHONETIC SPELLING .....	721
CHOOSING THE RIGHT VOWEL.....	721
CHOOSING THE RIGHT CONSONANT .....	722
CONTRACTIONS AND SPECIAL SYMBOLS .....	722
STRESS AND INTONATION .....	722
HOW AND WHERE TO PUT THE STRESS MARKS .....	723
WHAT STRESS VALUE DO I USE?.....	724
PUNCTUATION .....	724
HINTS FOR INTELLIGIBILITY.....	725
EXAMPLE OF ENGLISH AND PHONETIC TEXTS.....	725
CONCLUDING REMARKS .....	726
The More Technical Explanation.....	726
Table of Phonemes.....	727
 <b>Chapter 38 Parallel Device.....</b>	 <b>737</b>
Introduction.....	737
Opening & Closing the Parallel Device.....	738
Termination of Reads.....	740
Setting Parallel Parameters .....	740
PARALLEL FLAGS (io_ParFlags).....	741
SETTING THE PARAMETERS .....	741
Errors from the Parallel Device.....	742
 <b>Chapter 39 Printer Device.....</b>	 <b>743</b>
Introduction.....	743
Using the Printer Device as an AmigaDOS File.....	744
OPENING THE AMIGADOS PRINTER DEVICE.....	744
CLOSING THE AMIGADOS PRINTER DEVICE.....	745
Using the Printer Device Directly.....	745
DATA STRUCTURES USED DURING PRINTER I/O .....	745
CREATING AN I/O REQUEST .....	746
OPENING THE PRINTER DEVICE .....	746
SENDING I/O COMMANDS TO THE PRINTER DEVICE.....	747
WRITING TEXT TO THE PRINTER .....	748
SENDING PRINTER COMMANDS TO THE PRINTER .....	749
PRINTER COMMAND DEFINITIONS.....	749
DUMPING A RASTPORT TO THE PRINTER .....	752
PRINTER SPECIAL FLAGS .....	753
PRINTING WITH CORRECTED ASPECT RATIO.....	753
HANDLING PRINTER ERROR CODES .....	755
STRIP PRINTING .....	759
GETTING INFORMATION ABOUT THE PRINTER .....	760
CHANGING THE PRINTER PREFERENCES SETTINGS.....	762
ADDITIONAL NOTES ABOUT GRAPHIC DUMPS .....	763
Creating a Printer Driver.....	763
WRITING A GRAPHICS PRINTER DRIVER .....	766
WRITING AN ALPHANUMERIC PRINTER DRIVER .....	772
TESTING THE PRINTER DRIVER.....	776
Example Printer Driver Source Code.....	777
macros.i.....	777
EPSONX .....	779
EPSONX: PRINTERTAG.ASM .....	779

EPSONX_REV.I.....	782
EPSONX: INIT.ASM .....	782
EPSONX: DATA.C .....	784
EPSONX: DOSPECIAL.C .....	788
EPSONX: RENDER.C .....	791
EPSONX: DENSITY.C .....	796
EPSONX: TRANSFER.ASM.....	797
EPSONX: TRANSFER.C.....	801
EPSONQ .....	804
EPSONQ: PRINTERTAG.ASM .....	804
EPSONQ: EPSONQ_REV.I .....	806
EPSONQ: INIT.ASM .....	806
EPSONQ: DATA.C .....	809
EPSONQ: DOSPECIAL.C .....	812
EPSONQ: RENDER.C .....	815
EPSONQ: DENSITY.C .....	819
EPSONQ: TRANSFER.C.....	819
HP_LASERJET .....	822
HP_LASERJET: PRINTERTAG.ASM.....	822
HP_LASERJET: HP_REV.I.....	825
HP_LASERJET: INIT.ASM.....	825
HP_LASERJET: DATA.C.....	827
HP_LASERJET: DOSPECIAL.C.....	829
HP_LASERJET: RENDER.C.....	833
HP_LASERJET: DENSITY.C.....	836
HP_LASERJET TRANSFER.C .....	836
XEROX_4020 .....	838
XEROX_4020: PRINTERTAG.ASM .....	838
XEROX_4020: XEROX_4020_REV.I.....	840
XEROX_4020: INIT.ASM .....	841
XEROX_4020: DATA.C .....	843
XEROX_4020: DOSPECIAL.C .....	847
XEROX_4020: RENDER.C .....	850
XEROX_4020: TRANSFER.C.....	855
<b>Chapter 40 Serial Device .....</b>	<b>859</b>
Introduction .....	859
Opening the Serial Device .....	860
Closing the Serial Device.....	861
Writing to the Serial Device.....	861
Reading from the Serial Device .....	862
FIRST ALTERNATIVE MODE FOR INPUT OR OUTPUT .....	862
SECOND ALTERNATIVE MODE FOR INPUT OR OUTPUT .....	863
HIGH SPEED OPERATION.....	863
USE OF BEGINIO WITH THE SERIAL DEVICE.....	864
TERMINATION OF THE READ .....	864
Using Separate Read and Write Tasks .....	865
Setting Serial Parameters - SDCMD_SETPARAMS .....	866
SERIAL FLAGS (bit definitions for io_SerFlags) .....	868
SETTING THE PARAMETERS .....	869
Error codes from the Serial Device .....	869
Multiple serial port support.....	869
Taking Over the Hardware.....	870

<b>Chapter 41 Timer Device.....</b>	<b>871</b>
Introduction .....	871
Timer Device Units .....	871
Opening a Timer Device .....	873
Adding a Time Request.....	873
Aborting a Timer Request.....	875
Closing a Timer.....	875
Additional Timer Functions and Commands .....	875
SYSTEM TIME .....	876
USING THE TIME ARITHMETIC ROUTINES.....	877
WHY USE TIME ARITHMETIC? .....	878
Sample Timer Program .....	879
 <b>Chapter 42 Trackdisk Device.....</b>	 <b>883</b>
Introduction .....	883
The Amiga Floppy Disk.....	884
Trackdisk Device Commands .....	884
Creating an I/O Request.....	886
Opening a Trackdisk Device.....	887
Sending a Command to the Device.....	888
Terminating Access to the Device .....	888
Device-specific Commands.....	888
ETD_READ and CMD_READ .....	888
ETD_WRITE and CMD_WRITE.....	889
ETD_UPDATE AND CMD_UPDATE.....	889
ETD_CLEAR and CMD_CLEAR.....	890
ETD_MOTOR and TD_MOTOR.....	890
ETD_FORMAT and TD_FORMAT .....	891
Status Commands.....	891
TD_CHANGENUM .....	892
TD_CHANGESTATE .....	892
TD_PROTSTATUS .....	892
TD_GETDRIVETYPE .....	892
TD_GETNUMTRACKS .....	893
Being Notified of Disk Changes .....	893
TD_ADDCHANGEINT .....	893
TD_REMCHANGEINT .....	894
Commands for Low-Level Access.....	894
ETD_RAWREAD and TD_RAWREAD .....	894
ETD_RAWWRITE and TD_RAWWRITE.....	895
Commands for Diagnostics and Repair.....	896
ETD_SEEK and TD_SEEK.....	896
Trackdisk Device Errors .....	897
Example Program .....	897
 <b>Chapter 43 Resources .....</b>	 <b>903</b>
Introduction .....	903
Disk Resource .....	904
CIA Resource.....	904
Misc Resource.....	905
POTGO Resource.....	907

<b>Appendix A Troubleshooting Your Software.....</b>	<b>909</b>
GENERAL DEBUGGING TECHNIQUES .....	914
A FINAL WORD ABOUT TESTING .....	914
<b>Appendix B Linker Libraries .....</b>	<b>915</b>
Introduction.....	915
AMIGA.LIB.....	916
DEBUG.LIB .....	916
Amiga.lib.....	917
EXEC_SUPPORT.....	917
CLIB.....	917
OTHER .....	918
Debug.lib.....	919
<b>Appendix C Floppy Boot Process and Physical Layout .....</b>	<b>921</b>
<b>Glossary.....</b>	<b>925</b>
<b>Index.....</b>	<b>935</b>

## List of Figures

Figure 1-1 A Screen with Windows.....	3
Figure 1-2 Menu Items and Subitems .....	3
Figure 1-3 A Requester .....	4
Figure 1-4 An Alert.....	5
Figure 2-1 A Screen and Windows .....	12
Figure 2-2 Screen and Windows with Menu List Displayed .....	13
Figure 2-3 The Workbench Screen and the Workbench Application .....	14
Figure 2-4 Topaz Font in 60-column and 80-column Types .....	19
Figure 3-1 A High-resolution Screen and Windows.....	34
Figure 3-2 System Gadgets for Windows .....	40
Figure 3-3 Simple Refresh .....	43
Figure 3-4 Smart Refresh .....	44
Figure 3-5 SuperBitMap Refresh.....	45
Figure 3-6 The X-Shaped Custom Pointer.....	58
Figure 4-1 System Gadgets in a Low-resolution Window.....	73
Figure 4-2 Hand-drawn Gadget — Unselected and Selected .....	75
Figure 4-3 Line-drawn Gadget — Unselected and Selected.....	76
Figure 4-4 Example of Combining Gadget Types .....	89
Figure 5-1 Screen with Menu Bar Displayed.....	111
Figure 5-2 Example Item Box.....	112
Figure 5-3 Example Subitem Box.....	113
Figure 5-4 Menu Items with Command Key Shortcuts .....	115
Figure 6-1 Requester Deluxe .....	136
Figure 6-2 A Simple Requester Made with AutoRequest() .....	140
Figure 6-3 The “Out of Memory” Alert.....	146
Figure 7-1 Watching the Stream .....	156
Figure 7-2 Input from the IDCMP, Output through the Graphics Primitives.....	158
Figure 7-3 Input and Output through the Console Device.....	159
Figure 7-4 Full-system Input and Output (a Busy Program) .....	160
Figure 7-5 Output Only .....	161
Figure 8-1 Example of Border Relative Position.....	176
Figure 8-2 Intuition’s High-resolution Sizing Gadget Image .....	182
Figure 8-3 Example of PlanePick and PlaneOnOff.....	184
Figure 8-4 Example Image — the Front Gadget.....	186
Figure 11-1 The Dreaded Erase-Disk Requester.....	214
Figure 16-1 Simplified Overview of an Exec List .....	250
Figure 16-2 Complete Sample List Showing all Interconnections .....	252
Figure 16-3 List Header Overlap .....	256
Figure 16-4 Initializing a List Header Structure .....	256
Figure 23-1 How the Video Display Picture Is Produced.....	329
Figure 23-2 Display Overscan Restricts Usable Picture Area .....	330

Figure 23-3 Interlaced Mode — Display Fields and Data in Memory .....	331
Figure 23-4 Interlaced Mode Doubles Vertical Resolution .....	331
Figure 23-5 Sample Memory Words.....	332
Figure 23-6 A Rectangular “Look” at the Sample Memory Words .....	333
Figure 23-7 Bit-Plane for a Full-screen, Low-resolution Display .....	333
Figure 23-8 Bits from Each Bit-Plane Select Pixel Color .....	334
Figure 23-9 The Display Is Composed of ViewPorts .....	336
Figure 23-10 Correct and Incorrect Uses of ViewPorts.....	337
Figure 23-11 Size Definition for a ViewPort .....	338
Figure 23-12 A Single-playfield Display .....	341
Figure 23-13 A Dual-playfield Display.....	342
Figure 23-14 How HIRES Affects Width of Pixels .....	342
Figure 23-15 How LACE Affects Vertical Resolution .....	343
Figure 23-16 ViewPort Data Area Parameters.....	344
Figure 23-17 Example of Drawing Through a Stencil .....	374
Figure 23-18 Example of Extracting from a Bit-Packed Array .....	375
Figure 23-19 Modulo .....	376
Figure 24-1 Text Baseline.....	396
Figure 24-2 Complement Mode.....	401
Figure 24-3 CharSpace.....	413
Figure 24-4 CharKern .....	414
Figure 25-1 GEL Structure Layout .....	425
Figure 25-2 Sprite Color Registers .....	428
Figure 25-3 A Collision Mask .....	467
Figure 25-4 Ring Motion Control .....	474
Figure 25-5 Linking AnimComps For a Multiple Component AnimOb .....	477
Figure 25-6 Specifying an AnimOb Position.....	479
Figure 25-7 Linking of an AnimOb .....	480
Figure 33-1 Amiga Character Set .....	639
Figure 33-2 Amiga 1000 Keyboard Showing Keycodes in Hex.....	658
Figure 33-3 Amiga 500/2000 Keyboard Showing Keycodes in Hex .....	658
Figure 36-1 Raw Key Matrix .....	707

# List of Tables

Table 2-1 Screen Depth and Color .....	18
Table 4-1 System Gadget Placement in Windows and Screens.....	73
Table 4-2 Editing Keys and Their Functions .....	88
Table 9-1 Mouse Activities.....	190
Table 9-2 Special Command Keys.....	192
Table 11-1 Project Menus .....	213
Table 11-2 Edit Menus.....	213
Table 11-3 Selection Shortcuts .....	216
Table 11-4 Information (Menu) Shortcuts .....	217
Table 11-5 Cursor Keys .....	217
Table 17-1 Traps (68000 Exception Vector Numbers).....	275
Table 21-1 Interrupts by Priority.....	307
Table 23-1 Depth Values and Number of Colors in the ViewPort .....	339
Table 23-2 Single-playfield Mode (DUALPF <i>not</i> specified in <b>Modes</b> variable).....	340
Table 23-3 Dual-playfield Mode (DUALPF specified in <b>Modes</b> variable) .....	340
Table 23-4 Bit-Plane Assignment in Dual-playfield Mode.....	354
Table 23-5 Minterm Logic Equations.....	378
Table 23-6 Some Common Logic Equations for Copying .....	378
Table 24-1 Default Character Fonts .....	398
Table 30-1 Workbench Object Types .....	581
Table 31-1 Suggested Precedences for Channel Allocation .....	614
Table 31-2 Possible Channel Combinations .....	615
Table 33-1 Console Control Sequences .....	642
Table 33-2 Amiga Console-control Sequences.....	645
Table 33-3 Special Key Report Sequences .....	654
Table 33-4 Raw Input Event Types .....	655
Table 33-5 Input Event Qualifiers.....	657
Table 33-6 ROM Default (USA0) and USA1 Console Key Mapping.....	659
Table 33-7 High Key Map Hex Values.....	664
Table 33-8 Keymap Qualifier Bits .....	666
Table 35-1 Input Device Commands .....	694
Table 37-1 Recommended Stress Values.....	724
Table 37-2 Phonemes.....	727
Table 38-1 Parallel Parameters (IOExtPar).....	740
Table 38-2 Parallel Flags (io_ParFlags).....	741
Table 38-3 Parallel Device Errors .....	742
Table 39-1 Printer Device Command Functions.....	750
Table 40-1 Serial Parameters .....	866
Table 40-2 Serial Flags (io_SerFlags).....	868
Table 42-1 Trackdisk Device Error Codes.....	897



# Chapter 1

## Intuition: Introduction

Welcome to Intuition, the Amiga user interface.

What is a user interface? This sweeping phrase covers all aspects of getting input from and sending output to the user. It includes the innermost mechanisms of the computer and rises to the height of defining a philosophy to guide the interaction between man and machine. Intuition is, above all else, a philosophy turned into software.

Intuition's user interface philosophy is simple to describe: the interaction between the user and the computer should be simple, enjoyable, and consistent; in a word, intuitive. Intuition supplies a bevy of tools and environments that can be used to meet this philosophy.

Intuition was designed with two major goals in mind. The first is to give users a convenient, constant, colorful interface with the functions and features of both the Amiga operating system and the programs that run in it. The other goal is to give application designers all the tools they need to create this colorful interface and to free them of the responsibility of worrying about any other programs that may be running at the same time, competing for the same display and resources.

The Intuition software manages a many-faceted windowing and display system for input and output. This system allows full and flexible use of the Amiga's powerful multitasking, multi-graphic, and sound synthesis capabilities. Under the Amiga Executive operating system, many programs can reside in memory at the same time, sharing the system's resources with one another. Intuition allows these programs to display their information in overlapping windows without interfering with one another; in addition, it provides an orderly way for the user to decide which program to work with at any given instant, and how to work with that program.

Intuition is implemented as a library of functions. These functions are available to high-level language programmers via interface libraries and to assembly-language programmers. Application programmers use these routines along with simple data structures to generate program displays and to interface with the user.

A program can have access to all the functions and features of the machine by opening its own *virtual terminal*. When a virtual terminal is opened, your program will seem to have the entire machine and display to itself. It may then display text and graphics to its terminal, and it may ask for input from any number of sources, ignoring the fact that any number of other programs may be performing these same operations. In fact, your program can open several of these virtual terminals and treat each one as if it were the only program running on the machine.

The user sees each virtual terminal as a *window*. Many windows can appear on the same display. Each window can be the virtual terminal of a different application program, or several windows can be created by the same program.

The Amiga also gives you powerful graphics and audio tools for your applications. There are many display modes and combinations of modes (for instance, four display resolutions, hold-and-modify mode, dual-playfield mode, different color palettes, double-buffering, and more) plus animation and speech and music synthesis. You can combine sound, graphics, and animation in your Intuition windows. As you browse through the Intuition chapters, you'll find many creative ways to turn Intuition and the other Amiga tools into your own personal kind of interface.

## How the User Sees an Intuition Application

From the user's viewpoint, the Amiga environment is colorful and graphic. Application programs can use graphics as well as text in the windows, menus, and other display features described below. You can make liberal use of *icons* (small graphic objects symbolic of an option, command, or object such as a document or program) to help make the user interface clear and attractive.

The user of an Amiga application program, or of the AmigaDOS operating system, sees the environment through windows, each of which can represent a different task or context (see figure). Each window provides a way for the user and the program to interact. This kind of user interface minimizes the context the user must remember. The user manipulates the windows, *screens* (the background for windows), and contents of the windows with a mouse or other controller. At his or her convenience, the user can switch back and forth between different tasks, such as coding programs, testing programs, editing text, and getting help from the system. Intuition remembers the state of partially completed tasks while the user is working on something else.

The user can change the shape and size of these windows, move them around on the screen, bring a window to the foreground, and send a window to the background. By changing the arrangement of the windows, the user can select which information is visible and which terminal will receive input. While the user is shaping and moving the windows around the display, your program can ignore the changes. As far as the application is concerned, its virtual terminal covers the entire screen, and outside of the virtual terminal there's nothing but a user with a keyboard and a mouse (and any other kind of input device, including joysticks, graphics tablets, light pens, and music keyboards).

Screens can be moved up or down in the display, and they can be moved in front of or behind other screens. In the borders of screens and windows there are control devices, called *gadgets*, that allow the user to modify the characteristics of screens and windows. For instance, there is a gadget for changing the size of a window and a gadget for arranging the depth of the screens.

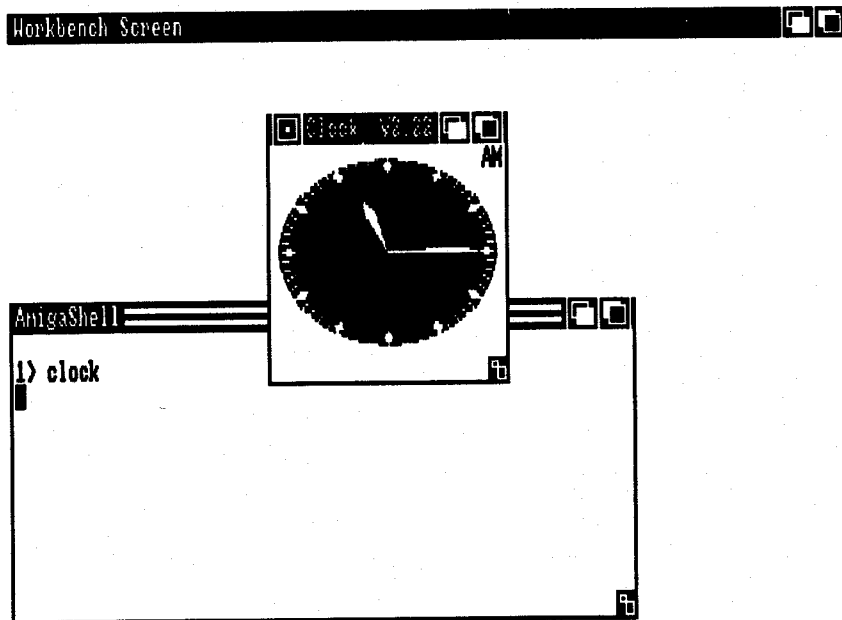


Figure 1-1: A Screen with Windows

Applications can use a variety of custom gadgets. For example, the program might use a gadget to request that the user type in a string of characters. Another gadget might be used to adjust the sound volume or the color of the screen.

At any time, only one window is active in the sense that only one window receives input from the user. Other windows, however, can work on some task that requires no input. For the active window, the screen's title bar can be used to display a list of menus (called the *menu bar*) at the user's command. By moving the mouse pointer along the menu bar, the user can view a list of menu items for each menu category on the menu bar. Each item in the list of menus can have its own subitem list (see figure).

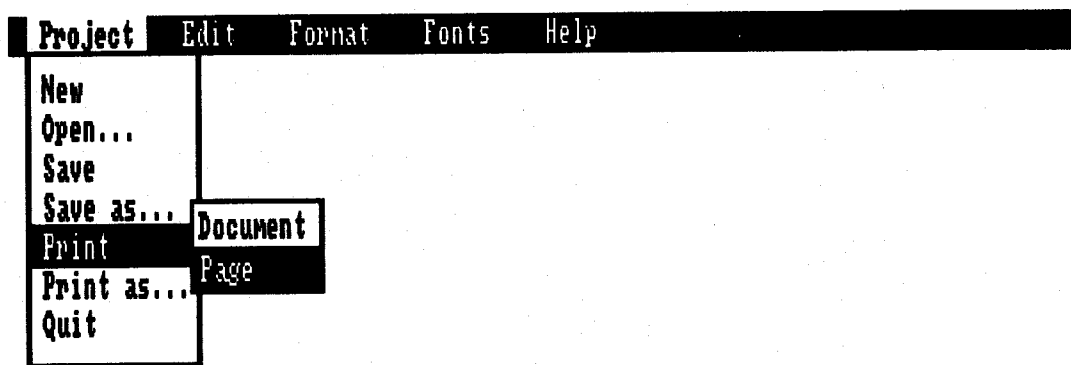


Figure 1-2: Menu Items and Subitems

Menus present lists of options and commands. The user can make choices from menus by using the mouse pointer and buttons. Applications can also provide the user with key-sequence shortcuts, as an alternative to the mouse. Intuition supplies certain key-sequence shortcuts automatically.

Windows can present the user with special *requester* boxes, invoked by the system or by applications (see figure). Requesters provide extended communication between the user and the application. When a requester is displayed, interaction with that window is halted until the user takes some action. The user, however, can make some other window active and deal with the requester later. If you wish, you can let the user bring up a requester on demand.

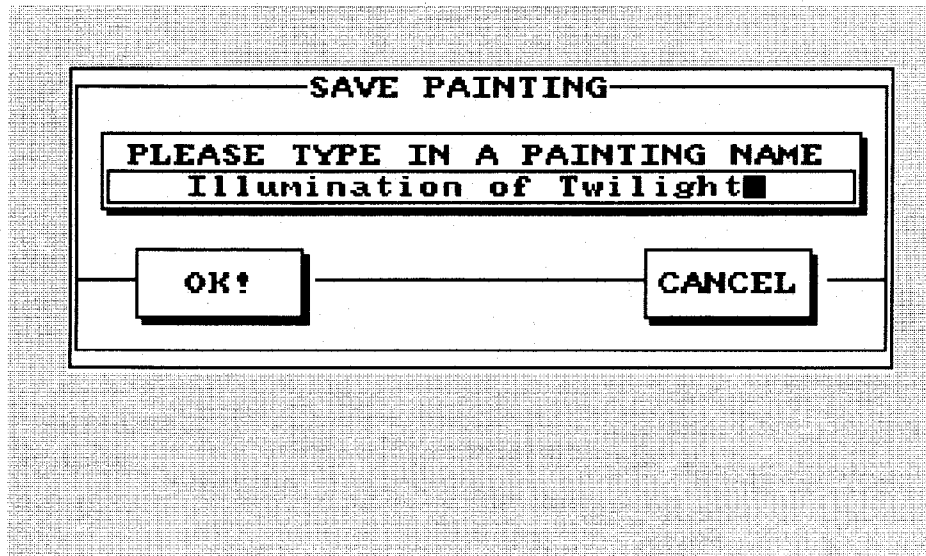
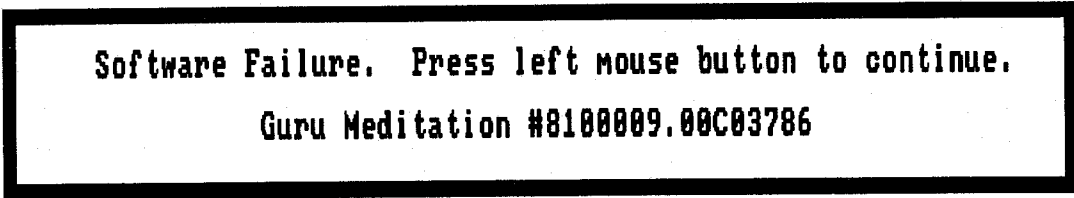


Figure 1-3: A Requester

The *alert* (see figure) is another kind of special information exchange device invoked by the system or an application. The alert display is dramatic. It appears in red and black at the top of the display, with text and a blinking border. Alerts are meant to be used when a serious problem has occurred or when the user must take some action immediately. The application may also try to get the user's attention by flashing the screen or windows in a complementary color.



**Software Failure. Press left mouse button to continue.**  
**Guru Meditation #81000009.00C03786**

Figure 1-4: An Alert

## The Right Approach to Using Intuition

Intuition is a very flexible program environment, with a vast number of features and defaults. The tools and devices are well defined and easily accessible. Although many default values are provided for you to rely on, few restrictions are placed on you. You are encouraged to let your creativity flow. Taking advantage of the many Intuition features enables you to spend less time implementing user-interaction mechanisms of your own, since Intuition already provides a wide range of them for you; in addition, the user of your code gets to work in an environment that does not change radically from one application to another.

For example, you can define the windows for your program in the standard Workbench screen provided by Intuition. Then you can use the standard system requesters and gadgets and simple menu facilities. Alternatively, you can design a custom screen using your own choice of modes and colors. You can use Intuition's standard imagery for your windows and gadgets, or you can design completely custom graphics. Intuition allows you to create your own pointer and to combine elaborate graphic images and text strings in menu items. You can also choose to mix predefined features and custom designs. Your creative freedom is practically limitless under Intuition.

No matter how simple, complex, or fanciful your program design, it will fit within the basic Intuition framework of windows and screens, gadgets, menus, requesters, and alerts. The users of the Amiga will come to understand these basic Intuition elements and to trust that the building blocks remain constant. This consistency ensures that a well-designed program will be understandable to the naive user as well as to the sophisticate. This is the essence and the beauty of the Intuition philosophy.

# Intuition Components

These are Intuition's major components:

- **Windows** provide the means for obtaining input from the user; they are also the normal destination for the program's output.
- **Screens** provide the background for opening windows.
- Numerous mechanisms exist for interaction between users and applications:
  - **Menus** present users with options and give them an easy way of entering commands.
  - **Requesters** provide a menu-like exchange of information.
  - **Gadgets** are the main method of communication.
  - **Alerts** are for emergency communications.
  - The mouse is the user's primary tool for making selections and entering commands.
  - The keyboard is used for entering text and as an alternate shortcut method of entering commands.
  - Other input devices, like graphics tablets or music keyboards, provide additional means of user input.
- The methods of program input and output are as follows:
  - Input is received through the console device or Intuition Direct Communication Message Ports (known as the IDCMP).
  - Output is transmitted through the console device or directly to the graphics, text, and animation library functions, as well as through speech and sound.

## General Program Requirements and Information

The sample Intuition shell program that follows shows all of the basic requirements for an Intuition application. There are three important points:

- You must open the Intuition library before you can use the Intuition functions.
- Certain languages such as "C" require the pointer to the Intuition library be assigned to a variable called "IntuitionBase".
- Resources *must* be returned to the System.

## THE INTUITION EVENT LOOP

The Intuition event loop that we'll use is called "main.c". It opens a window on a custom screen, then waits for you to click on the close gadget. When you do, it closes the window, the screen, and Intuition. We will use main.c for most of our Intuition examples. It will vary from example to example, as our needs dictate. Each different window has its own header file. So, we'll sometimes "#include "sandstoneWindow.h"" or "agateWindow.h", instead of the customary "graniteWindow.h". (All of the window header files are named after rocks, for consistency in naming. The names convey nothing about the window characteristics.)

```
/* sysgads.h */
/* These are, respectively, the sum of the widths of the */
/* close gadget and depth arrangement gadgets; and the */
/* sum of the heights of the sizing gadget and the */
/* depth arrangement gadgets. These values are merely */
/* advisory, since the height depends on the font */
/* height, and the width depends on the screen width. */

#define SYSGADSWIDTH 80
#define SYSGADSHHEIGHT 19

/* end of sysgads.h */

/* hires.h -- Declare and initialize a NewScreen structure */
struct NewScreen fullHires =
{
    0, /* the LeftEdge must be zero */
    0, /* TopEdge */
    640, /* Width (high-resolution) */
    STDSCREENHEIGHT, /* Height (non-interlace) */
    2, /* Depth (4 colors will be available) */
    -1,-1, /* Default DetailPen and BlockPen */
    HIRES, /* the high-resolution display mode */
    CUSTOMSCREEN, /* the screen type */
    NULL, /* no special font */
    "Our Own Screen", /* the screen title */
    NULL, /* no special screen gadgets */
    NULL /* no CustomBitMap */
};

/* end of hires.h */

/* graniteWindow.h -- This file implements a fairly ordinary window. */
#include "sysgads.h"

#define GRAN_LEFTEDGE 20
#define GRAN_TOPEDGE 20
#define GRAN_WIDTH 400
#define GRAN_HEIGHT 150

struct NewWindow graniteWindow =
{
    GRAN_LEFTEDGE,
    GRAN_TOPEDGE,
    GRAN_WIDTH,
    GRAN_HEIGHT,
    0,1, /* Plain vanilla DetailPen and BlockPen. */
    CLOSEWINDOW, /* Tell program when close gadget has been hit */
    WINDOWCLOSE | SMART_REFRESH | ACTIVATE | WINDOWDRAG |
    WINDOWDEPTH | WINDOWIZING | NOCAREREFRESH,
    NULL, /* Pointer to the first gadget -- */
    /* may be initialized later. */
    NULL, /* No checkmark. */
}
```



```

    "graniteWindow", /* Window title. */
    NULL,            /* Attach a screen later. */
    NULL,            /* No bitmap. */
    SYSGADSWIDTH,    /* Minimum width. */
    SYSGADSHHEIGHT,  /* Minimum height. */
    0xFFFF,          /* Maximum width. */
    0xFFFF,          /* Maximum height. */
    CUSTOMSCREEN      /* A screen of our own. */
};

/* end of graniteWindow.h */

/* main.c - This is the program shell we'll be using with our examples. */
/* Compiled with Lattice C v5.02 */
/* Compiler flags were "-bl -cfist -L -v -w" */
/* where the file Include.q is a precompiled header file of all of the */
/* Amiga "include" files, plus the Lattice-supplied "proto" files */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. For illustration, we show */
/* two header files which we will use frequently. */
#include "hires.h"
#include "graniteWindow.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33

extern VOID cleanExit( struct Screen *, struct Window *, int );
extern UBYTE handleIDCMP( struct Window *);

struct IntuitionBase *IntuitionBase = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;
    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library",INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Open any other required libraries */

    /* Make the assignments that were postponed above */

    /* Open the screen */
    screen1 = OpenScreen(&fullHires);
    if (screen1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Attach the window to the open screen ... */
    graniteWindow.Screen = screen1;

    /* ... and open the window */
    window1 = OpenWindow(&graniteWindow);
    if (window1 == NULL)

```

```

    cleanExit(screen1, window1, RETURN_WARN);

/* Set up the signals that you want to hear about ... */
signalmask = 1L << window1->UserPort->mp_SigBit;

/* Call the functions that do the main processing */

/* And wait to hear from your signals */
while( !done )
{
    signals = Wait(signalmask);
    if (signals & signalmask)
        done = handleIDCMP(window1);
};

/* Exit the program */
cleanExit(screen1, window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
    {
        class = message->Class;

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);

        /* See what events occurred */
        switch( class )
        {
            case CLOSEWINDOW:
                flag = 1;
                break;
            default:
                break;
        }
    }
    return(flag);
}

VOID cleanExit( scrn, wind, returnValue )
struct Screen *scrn;
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Close the window and the screen */
    if (wind) CloseWindow( wind );
    if (scrn) CloseScreen( scrn );

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}

```

## Chapter 2

# Intuition: Screens

Screens are the basis for all Intuition displays. They set up the environment for overlapping windows and they give you easy access to all the Amiga display modes and graphics features. In this chapter you will learn how to use the Workbench screen provided by Intuition and how to create your own custom screens.

### About Screens

The screen is Intuition's basic unit of display. By using an Intuition screen, you can create a video display with any combination of the many Amiga display modes. Certain basic parameters of the video display (such as fineness of vertical and horizontal resolution, number of colors, and color choices) are defined by these modes. By combining modes, you can have many different types of displays. For example, the display may show eight different colors in low-resolution mode or 32 colors in interlaced mode (high resolution of lines). For a description of all the different display modes, see the "Custom Screens" section below.

Every other Intuition display component is defined with respect to the screen in which it is created. Each screen's data structure contains definitions that describe the modes for the particular screen. Windows inherit their display parameters from the screens in which they open, so a window that opens in a given screen always has the same display modes and colors as that screen. If your program needs to open windows that differ from one another in their display characteristics, you can open more than one screen.

Screens are always the full width of the display. This is because the Amiga hardware allows very flexible control of the video display, but imposes certain restrictions. Sometimes it is not possible to change display modes in the middle of a scan line. Even when it is possible, it is usually aesthetically unpleasant or visually jarring to do so. To avoid these problems, Intuition imposes its own display restriction, allowing only one screen (one collection of display modes) per video line. Because of this, screens can be dragged vertically but not horizontally. This allows screens with different display modes to overlap, but prevents any changes in display mode within a video line.

Screens provide display raster memory, which is the RAM in which all imagery is first rendered and then translated by the hardware into the actual video display. The Amiga graphics structure that describes how rendering is done into display memory is called a **RastPort**. The **RastPort** also has pointers into the actual display memory locations. The screen's display memory is also used by Intuition for windows and other high-level display components that overlay the screen. Application programs that open custom screens can use the screen's display memory in any way they choose.

Screens are rectangular in shape. When they first open they usually cover the entire surface of the video display, although they can be shorter than the height of the display. Like windows, screens can be moved up or down and arranged at different depths by using special control mechanisms called gadgets. Unlike windows, however, screens cannot be made larger or smaller, and they cannot be moved left or right.

The dragging and depth-arrangement gadgets reside in the title bar at the top of all Intuition screens. In the title bar there may also be a line of text identifying the screen and its windows.

The figure shows a screen with open windows. The depth-arrangement gadgets (front gadget and back gadget) are at the extreme right of the screen title bar. The drag gadget (for moving the screen) occupies the entire area of the screen title bar not occupied by other gadgets. The user changes the front-to-back order of the displayed screens by using a controller (such as a mouse) or the keyboard cursor control keys to move the Intuition pointer within one of the depth-arrangement gadgets. When the user clicks the left mouse button (known as the *select button*), the screen's depth arrangement is changed.

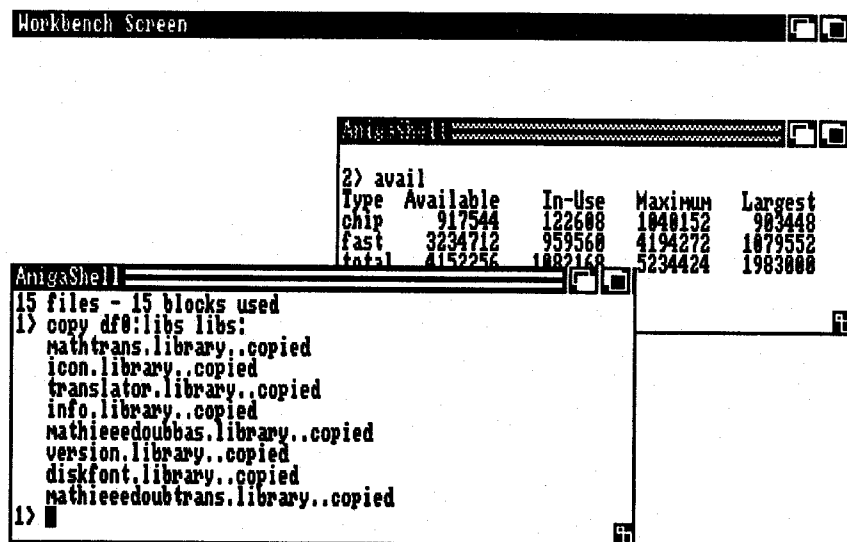


Figure 2-1: A Screen and Windows

The user moves the entire screen up or down on the video display by moving the pointer within the drag gadget, holding down the left mouse button while moving the pointer, and finally releasing the button when the screen is in the desired location.

The screen's title bar is also used to display a window's menus when the user asks to see them. Typically, when the user presses the right mouse button (the *menu button*), a list of menu topics called a menu list appears across the title bar. The figure shows a screen after the user has displayed the menu list.

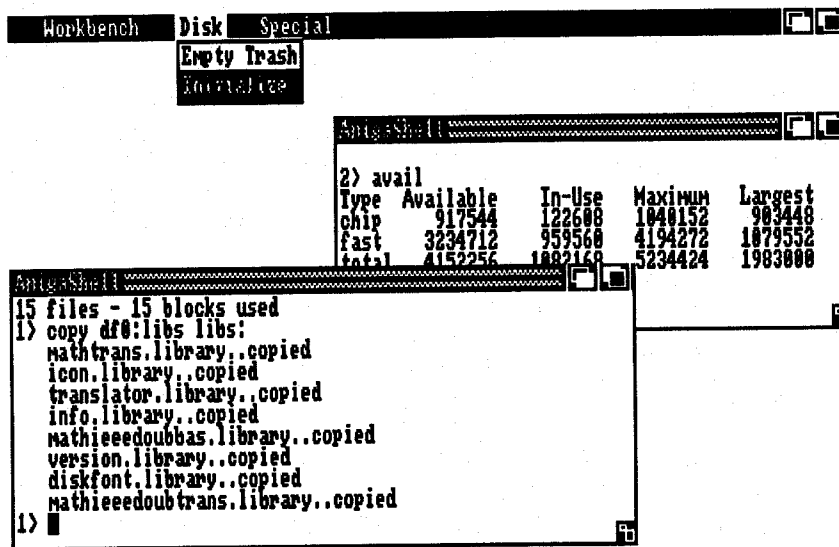


Figure 2-2: Screen and Windows with Menu List Displayed

By further mouse movement and mouse button manipulation, the user can see a list of menu items and subitems for each of the topics in the menu list. The menu list, menu items, and subitems that are displayed pertain to the currently active window, which is the window receiving the user's input. Only one window is active at any time. The screen containing the active window can be thought of as the active screen. Because there is only one active window, there can be only one active menu list at a time. The menu list appears on the title bar of the active screen. Menus are handled by the Intuition menu system. See the chapter entitled "Intuition: Menus," for more information about putting menus together and attaching them to windows.

Both you and the user will find working with screens much like working with windows—for you, the data structures and the functions for manipulating screens and windows are similar. For the user, moving and arranging screens will require the same steps as moving and arranging windows. However, the user will be less aware of screens than of windows, since user input and application output occur mostly through windows.

There are two kinds of screens—the standard Workbench screen supplied by Intuition and custom screens created by you.

## The Workbench Screen

Workbench is both a screen and an application. It is a high-resolution four-color screen. On An NTSC Amiga, the nominal dimensions of the Workbench screen are 640 pixels x 200 lines (400 lines if the user has chosen an interlaced Workbench screen using Preferences), but see the section below, "Screen Position and Dimensions" for more details. The default colors are blue for the background, white and black for details, and orange for the text cursor and highlighting (see figure).

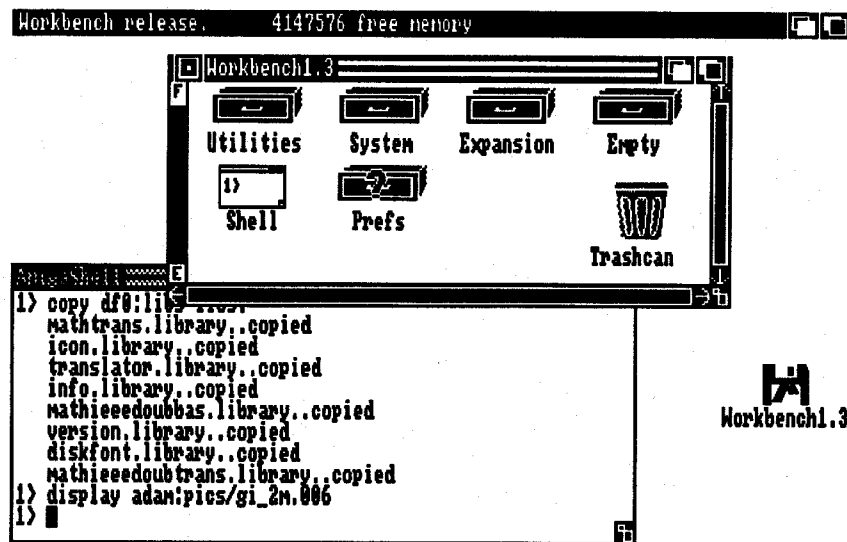


Figure 2-3: The Workbench Screen and the Workbench Application

The Workbench screen is used by both the Amiga Command Line Interface (CLI) and the Workbench application. If you want to use the Workbench as a screen for the windows of your program, you just specify a window type of `WBENCHSCREEN` in the data structure called `NewWindow`, which you initialize when opening a window.

Any application program can use the Workbench screen for opening its windows. Developers of text-oriented applications are especially encouraged to open in the Workbench screen. This is convenient for the user because many windows will open in the same screen, requiring less movement between screens. Using the Workbench screen is also memory-efficient, because you will not be allocating the memory for your own custom screen.

Your application should not change the colors of the Workbench screen, because other applications may depend on the constancy of the Workbench colors. For instance, a business package that runs on the Workbench screen may expect the colors to be reasonable for a dither pattern in a graph. If you change the colors, that program's graphics display may not look as intended, making the program harder to use, and defeating the purpose of the Workbench screen.

Generally, an application which creates a custom screen is responsible for closing it, but there is a special function, `CloseWorkBench()`, which allows your program to close the Workbench screen. If your application needs more memory than is available, it can attempt to reclaim the memory used by the Workbench screen by calling `CloseWorkBench()`. If you use this, you should call `OpenWorkBench()` as your program exits.

The Workbench screen does not close even if all the windows in it are closed, and it automatically reopens when all other screens close down.

The Workbench application program allows users to interact with the Amiga file system, using icons (small graphic images) to represent files. Intuition treats the Workbench application as a special case, communicating with it in extraordinary ways. For example, you can open or close the Workbench screen by calling the Intuition functions `OpenWorkBench()` and `CloseWorkBench()`, even though the Workbench application may have open windows in the screen. `CloseWorkBench()` will fail, however, if the user has other windows open on the Workbench screen.

Library functions allow you to create and manipulate the Workbench application's objects and icons. The functions in the library allow you to create disk files that the user can handle within the context of the Workbench program.

The user can change the colors of the Workbench screen via Preferences. For more information about Preferences, see the chapter entitled “Intuition: Other Features.”

## Custom Screens

Typically, you create your own screen when you need a specific kind of display that is not offered by the Workbench screen or when you want to modify the screen or its parameters directly—as in changing colors or directly modifying the *Copper* list or display memory. The Copper is the display-synchronized coprocessor that handles the actual video display by directly affecting the hardware registers. For example, you might need a display in which you can have movable sprite objects. Alternatively, you might have your own display memory that you want to use for the screen's display memory or you may want to allow the user to play with the colors of a display that you've created. If you want to do these sorts of things, you'll have to create a custom screen.

If you have opened a custom screen, you must call `CloseScreen()` to close it, before your program exits. Otherwise, your screen would stay around forever.

When you create a custom screen, you have a great deal of latitude in specifying screen parameters, including:

- Height of the screen and starting point of the screen when it first opens.
- Depth of the screen, which determines how many colors you can use for the display.
- Choice of the available colors for drawing details, such as gadgets, and for doing block fills, such as the title bar area.
- Display modes—high or low resolution, interlaced or non-interlaced, sprites, and dual playfields.
- Initial display memory.

You can also use the special Intuition graphics, line, and text structures and functions within the windows in your custom screen. See the chapter entitled “Intuition: Images, Line Drawing, and Text,” for details about these.

With some care, you may also render directly into your screen's display memory using the full complement of graphics primitives—or even directly manipulating the screen display memory using the processor—bypassing the protocol of Intuition windows. You can do color animation, scrolling, patterned line drawing and patterned fills, and much more. Although you can still combine such use of a screen with other Intuition features—for example, windows, menus, and requesters—these features draw into your display. The interactions described in the next paragraph are those that take place when you write to the custom screen while windows and menus are being displayed and moved over the screen.

First, Intuition does not save background screen information when a window is opened, sized, or moved. Screen areas that are subsequently revealed are restored to a blank background color, obliterating any data you might have written into the display memory area of your screen. Second, menus are protected from data being output to the windows behind them but not from data being output to screens. When a menu is on the screen, all underlying windows are locked against graphical output to prevent such output from trashing the menu display. Menus cannot, however, lock direct graphical output to the display memory of a screen. Therefore, be careful about writing to a screen that has or can have menus displayed in it. You can easily overrun the menus and obliterate the information contained in them.



In summary, keep in mind that the user can modify the display by moving things around (by using window gadgets) or making things appear and disappear (menus and requesters). If you want to write directly to a custom screen's display memory, you have to design the pieces carefully so that they interact without conflict. If you want complete control of the screen display memory and are willing to give up some windowing capabilities (such as menus and window sizing and dragging), you can use a custom screen. If you want to control the display memory *and* run windows and menus in the custom screen, you need to deal with the hazards. Always bear in mind that playing with screen displays in this way requires detailed knowledge of how screens and windows work. You should not attempt it lightly.

What if you want a screen with your own full-screen display, one you can manipulate any way you choose, but you still want access to all the windowing and menu capabilities without worry? A special kind of window satisfies all of these needs—the Backdrop window, which always stays in the background and can be fashioned to fill the entire display area. Writing to this kind of window is almost as flexible as writing directly to display memory and requires only a little more overhead in memory management and performance. Menus and ordinary windows can safely reside over this window. You can also cause the screen's title bar to disappear behind a Backdrop window by calling the `ShowTitle()` function, thereby filling the entire video display with your display memory. This is the Intuition-blessed way to fill the entire display and still exist in an Intuition environment. For more information about setting up Backdrop windows, see the "Intuition: Windows" chapter.

When you are using the graphics primitives (functions) in your custom screen, the functions sometimes require pointers to the graphics control structures that lie beneath the Intuition display. These graphics structures are the **RastPort**, **ViewPort**, and **View**. For more information and details about how to get the pointers into the display memory, see the chapter entitled "Intuition: Images, Line Drawing, and Text."

## Screen Characteristics

The following characteristics apply to both the Workbench screen and custom screens. Keep in mind, however, that you should not change the characteristics of the Workbench screen.

### DISPLAY MODES

You can use any or all of the following display modes in your custom screens. The windows that open in a screen inherit the screen's display modes and colors.

There are two modes of horizontal display: low resolution and high resolution. In low-resolution mode, there are nominally 320 *pixels* across a horizontal line. In high-resolution mode, there are 640 pixels across. A pixel is the smallest addressable part of the display and corresponds to one bit in a bit-plane. Twice as many pixels are displayed in high-resolution mode. However, low-resolution mode gives you twice as many potential colors, 32 instead of 16.

There are two choices of vertical resolution: interlaced and non-interlaced. You can have nominally 200 vertical lines of display in non-interlaced mode (256 vertical lines in PAL), and 400 lines in interlaced mode (512 lines in PAL). See also the section, "Screen Position and Dimensions." Twice as many display rows are displayed in interlaced mode. Typically, applications use non-interlaced mode, which requires half as much memory and creates a display that does not have the potential for flickering, as interlaced displays tend to do. Intuition supports interlaced mode because some applications will want to use it; for instance, a computer-aided design package running on a high-persistence monitor will want to use it, and it is often a requirement for video applications.

In *sprite* mode, you can have up to eight small moving objects on the display. You define sprites with a simple data structure and move them by specifying a series of x,y coordinates. Sprites can be up to sixteen bits wide and any number of lines tall, can have three colors (plus transparent), and pairs of sprites can be joined to create a fifteen-color (plus transparent) sprite. They are also reusable vertically, so you can really have more than eight at one time. The Amiga GELS system, described elsewhere in this manual, provides just such a *multiplexing*, or interleaving, of sprites for you. The chapter entitled "Intuition: Windows" contains a brief description of a sprite used as a custom pointer.

*Dual-playfield* mode is a special display mode that allows you to have two display memories. This gives you two separately controllable and separately scrollable entities that you can display at the same time, one in front of the other. With this mode, you can have some really interesting displays, because wherever the front display has a pixel that selects color register 0, that pixel is displayed as if it were transparent. You can see through these transparent pixels into the background display. In the background display, wherever a pixel selects color register 0, that pixel is displayed in whatever color is in color register 0. You should not try to implement a dual playfield display by setting the DUALPF flag in the NewScreen structure *before* opening your screen. We illustrate the correct method in our examples.

*Hold-and-modify* mode gives you extended color selection.

*Extra-halfbright* mode provides one extra bitplane that defines colors with the EHB bit set as half the RGB level of colors without that bit set. This can be used, together with the blitter, as an easy way to produce shadows, for example.

If you want to use sprites, hold-and-modify mode, or Extra-half Bright, you should read about all of their features elsewhere in this manual.

## DEPTH AND COLOR

Screen *depth* refers to the number of bit-planes in the the screen display. This affects the colors you can have in your screen and in the windows that open in that screen.

Display memory for a screen is made up of one or more of bit-planes, each of which is a contiguous series of memory words. When they are displayed, the planes are overlapped so that each pixel in the final display is defined by one bit from each of the bit-planes. For instance, each pixel in a three-bit-plane display is defined by three bits. The binary number formed by these three bits specifies the color register to be used for displaying a color at that particular pixel location. In this case, the color register would be one of the eight registers numbered 0 through 7. The thirty-two system color registers are completely independent of any particular display. You load colors into these registers by specifying the amounts of red, green, and blue that make up the colors. To load colors into the registers, you use the graphics primitive `SetRGB40`. The table shows the relationship between screen depth, number of possible colors in a display, and the color registers used.

Table 2-1: Screen Depth and Color

Depth	Maximum Number of Colors	Color Register Numbers
1	2	0-1
2	4	0-3
3	8	0-7
4	16	0-15
5	32	0-31
6	64	0-31*
6	4096	0-15+

\* Extra-halfbright

+ Hold-and-modify

The maximum number of bit-planes in a screen depends upon the dual-playfields display mode, and the HIRIS flag. The first four lines in the previous table apply to all display modes. Any of the display modes can have up to four bitplanes. Five or six bitplanes are possible only in the low resolution mode. In particular, both the extra-halfbright and hold-and-modify modes require a lo-res display. For dual playfields, you can have from two to six bitplanes, which are divided between the two playfields (see the dual playfields example, below). For hold-and-modify mode you need six bit-planes.

The color register numbers are also known as “pen” colors. If you specify a depth of 5, for instance, then you also have 32 choices (in low-resolution mode) for the **DetailPen** and **BlockPen** fields in the structure. **DetailPen** is used for details such as gadgets and title bar text. **BlockPen** is used for block fills, such as all of the title bar area not taken up by text and gadgets.

## TYPE STYLES

When you open a custom screen, you can specify a text font for the text in the screen title bar and the title bars of all windows that open in the screen. A font is a specification of type size and type style. The system default font is called “Topaz.” Topaz is a fixed-width font and comes in at least two sizes:

- Eight display lines tall with 80 characters per line in a 640-pixel high-resolution display (40 characters in low resolution).
- Nine display lines tall with 64 characters per line in a high-resolution display (32 characters in low resolution).

On a television screen, you may not be able to see all 640 pixels across a horizontal line. On any reasonable television, however, a width of 600 pixels is a safe minimum, so you should be able to fit 60 columns of the large Topaz font.

## NOTE

Font is a Preferences item and the user can choose either the 80- or 64-column (8- or 9-line) default, whichever looks best on his or her own monitor (see figure). You can use or ignore the user’s choice of default font size. See the chapter entitled “Intuition: Other Features,” for more information about Preferences items.

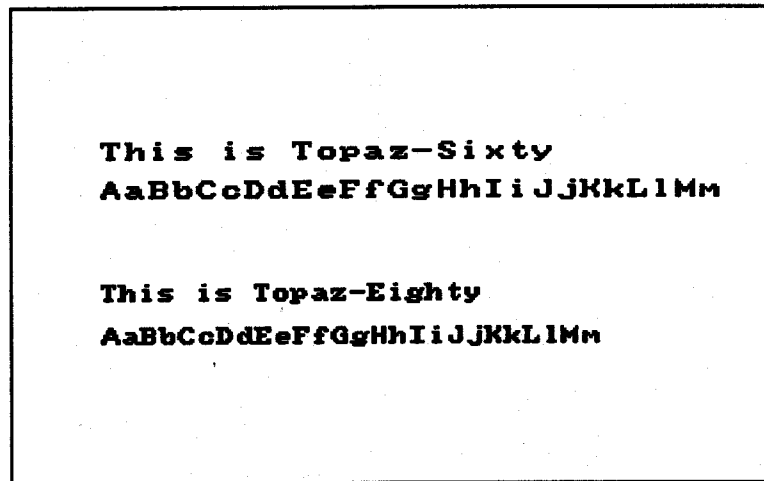


Figure 2-4: Topaz Font in 60-column and 80-column Types

If you want the default Topaz font in the default size currently selected by the user, set the **Font** field in the screen structure to **NULL**. If you want some other font, you specify it by creating a **TextAttr** structure and setting the screen's **Font** field to point to the structure. See elsewhere in this manual for further information about text-support primitives.

## SCREEN POSITION AND DIMENSIONS

When you open a custom screen, you specify the initial starting location for the top line of the screen in the **TopEdge** and **LeftEdge** fields of the screen structure. After that, the user can drag the screen up or down. You must always set the **LeftEdge** field (the x coordinate) to 0. (This parameter is included only for upward compatibility with future versions of Intuition.)

## APPLICATION-MANAGED CUSTOM SCREENS

You specify the dimensions of the screen in the **Height** and **Width** fields. You can set the screen height to values less than the maximum allowed, but do not make it too small for the title bar to appear. The width may also be less than the maximum allowed value, but, again, don't make it extraordinarily small.

The default maximum screen dimensions are 640 x 200 (640 x 400 interlaced) for NTSC, and 640 x 256 (640 x 512 interlaced) for PAL. Using the utility **MoreRows** (which induces the System to overscan its display, the user can slightly increase these maximum values.

The true current maximum screen dimensions are stored in the graphics library fields **GfxBase->NormalDisplayColumns** and **GfxBase->NormalDisplayRows**. These fields have values defined for a high-resolution interlaced display (e.g. 650 x 412) and must be scaled to provide maximum dimensions for other modes.

There is a simple method to open screens to their full height on any display. You do this by specifying the value `STDSCREENHEIGHT` in the `NewScreen.Height` field (this constant is defined in *intuition/screens.h*).

## SCREEN TITLE

The screen title is used for two purposes: to identify the screen like an identification tab on a file folder and to designate which window is the active one.

Although the initial screen title is set in the `NewScreen` structure, it can change according to the preferences of the windows that open in the screen. Each screen has two kinds of titles that can be displayed in the screen title bar:

- A “default” title, which is specified in the `NewScreen` structure and is always displayed when the screen first opens.
- A “current” title, which is associated with the currently active window. When the screen is first opened, the current title is the same as the default title. The current title depends upon the preferences of the currently active window.

Each window can have its own title, which appears in its own title bar, and its “screen title,” which appears in the *screen's* title bar. When the window is the active window, its screen title will be displayed in the screen's title bar. The function `SetWindowTitles()` allows you to specify, change, or delete both the window's own title and its screen title.

Screen title display is also affected by calls to `ShowTitle()`, which coordinates the display of the screen title and windows that overlay the screen title bar. Depending upon how you call this function, the screen's title bar can be behind or in front of any special Backdrop windows that open at the top of the screen. By default, the title bar is displayed in front of a Backdrop window when the screen is first opened. Non-Backdrop windows always appear in front of the screen title bar.

## CUSTOM GADGETS

You cannot attach custom gadgets directly to a screen. You can, however, attach custom gadgets to a borderless backdrop window and monitor their activity through the window's input/output channels. See the chapter on “Intuition: Gadgets,” for information about custom gadgets.

## Screen Data Structures

Below, we describe the most important fields in the two data structures pertaining to screens.

### NEWSCREEN STRUCTURE

Here are the specifications for the `NewScreen` structure:

```

struct NewScreen
{
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
    UBYTE DetailPen, BlockPen;
    USHORT ViewModes;
    USHORT Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap;
};

```

The meanings of the variables and flags in the **NewScreen** structure are as follows:

#### **LeftEdge**

Initial x position for the screen. This field is not currently used by Intuition; however, for upward compatibility, always set this field to 0.

#### **TopEdge**

Initial y position of the screen. Set this field to a value representing the screen offset, in scan lines, from the top of the display. Most often, this value should be zero.

#### **Width**

Width of the screen. Values of 320 for low-resolution mode or 640 for high-resolution mode, are acceptable.

#### **Height**

Height of the screen in number of lines. Values of 200 for non-interlaced mode and 400 for interlaced mode, are acceptable.

#### **Depth**

Number of bit-planes in the screen. Set this field from 1 to 6, but be careful to specify a value which is appropriate for the screen's display mode.

#### **DetailPen**

Color register number for details such as gadgets and text in the title bar.

#### **BlockPen**

Color register number for block fills, such as the title bar area.

#### **ViewModes**

These flags select display modes. You can set any or all of them:

##### **HIRES**

Selects high-resolution mode (640 pixels across). The default is 320 pixels across.

##### **LACE**

Selects interlaced mode (400 lines). The default is 200 lines. For a PAL Amiga, the recommended values are 512 and 256, respectively.

##### **SPRITES**

Set this flag if you want to use sprites in the display.

##### **DUALPF**

Set this flag if you want two playfields, but it is better to convert a normal screen into dual-playfield after it is opened. See the example below.

## HAM

Set this flag if you want hold-and-modify mode.

## EXTRA\_HALFBRITE

Set this flag if you want the additional colors provided by the extra-halfbright mode. This mode is supported by all but the earliest Amiga 1000's.

## Type

### CUSTOMSCREEN

Indicates that this will be a custom screen. All screens you open will be custom screens, since the Workbench screen is managed by Intuition and the functions **OpenWorkBench()** and **CloseWorkBench()**.

### SCREENBEHIND

Indicates that when the screen is opened, it should be behind all other screens. Among other uses, this method allows a program to prepare imagery in the screen, change its colors, and so on, bringing it to the front when it becomes presentable.

### SCREENQUIET

Allows fancy viewport operations in your custom screen. A screen opened with this flag will not have a title bar nor visible gadget rendering, but dragging and depth arrangement still function. In order to prevent Intuition from rendering into your screen, you must intercept the menu button in each window in the screen, using **MENUVERIFY** or **RMBTRAP**.

### CUSTOMBITMAP

Set this flag if you want to use your own bitmap and display memory for this screen.

## Font

A pointer to the default **TextAttr** structure for this screen and all Intuition-managed text that appears in the screen and its windows. Set this to **NULL** if you want to use the default Intuition font. NOTE: Intuition will not be able to load a font from disk, so to guarantee that the font will be available, it is a good idea to open it yourself, before calling **OpenScreen()**. Don't forget to close the font later, if you do this.

## DefaultTitle

A pointer to a null-terminated line of text that will be displayed in the screen's title bar; this should be set to **NULL** if you want a blank title bar. Null-terminated means that the last character in the text string is **NULL**, which is automatic in the C language.

## Gadgets

This field is not used at this time. It should be set to **NULL**.

## CustomBitMap

If you intend to provide your own **CUSTOMBITMAP** for the screen, you need to know how big the screen will be before opening it. This is done by examining the public field **GfxBase->NormalDisplayRows** (see *graphics/gfxbase.h*). **NormalDisplayRows** specifies the non-interlaced count, which is 200 on NTSC machines. The **CustomBitMap** field holds a pointer to a **BitMap** structure, used if you want your own display memory to be used as the display memory for this screen. You inform Intuition that you want to supply your own display memory by setting the flag **CUSTOMBITMAP** in the **Type** variables above, creating a **BitMap** structure that points to your display memory and having this variable point to it.

## SCREEN STRUCTURE

You create a custom screen by passing the address of your **NewScreen** structure to the function **OpenScreen()**, which, if successful, will return a valid pointer to a **Screen** structure. The following list shows the variables of the **Screen** structure that may be of interest to you. This is not a complete list of the **Screen** variables; only the more useful ones are described. Also, most of these variables are for use by advanced programmers, so you may choose to ignore them at first.

### **TopEdge**

Examine this to see where the user has positioned your screen.

### **MouseX, MouseY**

You can look here to see where the mouse is with respect to the upper left corner of your screen.

### **ViewPort, RastPort, BitMap, LayerInfo**

For hard-core graphics users, these are actual instances of these graphics structures (Note: *not* pointers to structures). For normal use of custom screens, these structures can be ignored.

### **BarLayer**

This is the pointer to the **Layer** structure for the screen's title bar.

## SCREEN FUNCTIONS

Here is a quick rundown of Intuition screen functions.

### **Opening a Screen**

This is the basic function to open an Intuition custom screen according to the parameters specified in **NewScreen**. This function sets up the screen structure and substructures, does all the memory allocations, and links the screen's **ViewPort** into Intuition.

- **OpenScreen (NewScreen)**

The argument is a pointer to an instance of a **NewScreen** structure. The function returns a pointer to a **Screen** structure.

### **Showing a Screen Title Bar**

This function causes the screen's title bar to be displayed or concealed, according to your specification of the **ShowIt** parameter and the position of the various types of windows that may be opened in the screen.

- **ShowTitle (Screen, ShowIt)**

The screen's title bar can be behind or in front of any **Backdrop** windows that are opened at the top of the screen. The title bar is always concealed by other windows, no matter how this function sets the title bar. The parameter **Screen** is a pointer to a **Screen** structure. Set the variable **ShowIt** to boolean **TRUE** or **FALSE**



according to whether the title is to be hidden behind Backdrop windows. When **ShowIt** is TRUE, the screen title bar is shown in front of Backdrop windows. When **ShowIt** is FALSE, the screen title bar is always behind any window. The function returns nothing.

## Moving a Screen

With this function, you can move the screen vertically.

- **MoveScreen (Screen, DeltaX, DeltaY)**

Moves the screen in a vertical direction by the number of lines specified in the **DeltaY** argument. (**DeltaX** is here for upward compatibility only. You should pass zero for **DeltaX**. **Screen** is a pointer to the screen structure. The function returns nothing. Calls to **MoveScreen** are asynchronous; when you call the function, the screen is not necessarily moved immediately. If the calls happen too quickly, there may be unexpected results. One way to pace these calls is to call the function one time for each INTUITICK event. For information on INTUITICKS, see the chapter "Input and Output Methods".

## Changing Screen Depth Arrangement

These functions change the screen's depth arrangement with respect to other displayed screens. **Screen** is a pointer to the screen structure.

- **ScreenToBack (Screen)**

Sends the specified screen to the back of the display. Returns nothing.

- **ScreenToFront (Screen)**

Brings the specified screen to the front of the display. Returns nothing.

## Closing a Screen

The following function unlinks the screen and its substructures and deallocates everything that Intuition allocated when it opened the screen. It ignores any windows attached to the screen. All windows must be closed first. Attempting to close a screen with open windows will crash the system. If this is the last screen displayed, Intuition attempts to reopen the Workbench. The function returns nothing.

- **CloseScreen (Screen)**

The variable **Screen** is a pointer to the screen to be closed.

## Finding Out About a Screen

This function is typically used to find out about the Workbench screen.

- **GetScreenData(Buffer,Size,Type,Screen)**

Programs opening windows on the Workbench screen may inquire to find out its size, the size of its menu bar area, and so on, by using the function **GetScreenData()**. This function will open the Workbench screen if it happens to be closed. This is best suited for use by a program about to open a window on the Workbench screen. Workbench screen inquiry also applies to custom screens, but the need for that will be rare. **Buffer** is a CPTR (see *exec/types.h*), **Size** and **Type** are USHORT's, and **Screen** is a pointer to a screen structure.

## Handling the Workbench

These functions are for opening, closing, and modifying the Workbench screen.

- **OpenWorkBench()**

This routine attempts to open the Workbench screen. If not enough memory exists to open the screen, this routine fails. Also, if the Workbench tool is active, it will attempt to reopen its windows. This function takes no arguments, and returns a pointer to a **Screen** structure.

- **CloseWorkBench()**

This routine attempts to close the Workbench screen. If another application (other than the Workbench tool) has windows opened in the Workbench screen, this routine fails, and returns FALSE. If only the Workbench tool has opened windows in the Workbench screen, the Workbench tool will close its windows, allow the screen to close, and return TRUE. This function takes no arguments, and returns a **BOOL**.

- **WBenchToFront(), WBenchToBack()**

If the Workbench screen is opened, calling these routines will cause it to be in front or in back of other screens, depending on which command is used. If the Workbench screen is closed, these routines have no effect. These functions take no arguments, and return **BOOL**.

## Advanced Screen and Display Functions

These functions are for advanced users of Intuition and graphics. They are used primarily in programs that make changes in their custom screens (for instance, in the Copper instruction list). These functions cause Intuition to incorporate a changed screen and merge it with all the other screens in a synchronized fashion. These functions return nothing. For more information about these functions, see the chapter "Intuition: Other Features."

- **MakeScreen(Screen)**

This function is the Intuition equivalent of the lower-level **MakeVPort()** graphics library function. **MakeScreen()** performs the **MakeVPort()** call for you, synchronized with Intuition's own use of the screen's **ViewPort**. The variable **Screen** is a pointer to the screen that contains the **ViewPort** that you want remade.

- **RethinkDisplay()**

This procedure performs the Intuition global display reconstruction, which includes massaging some of Intuition's internal state data, rethinking all of the Intuition screen **ViewPorts** and their relationship to one another, and, finally, reconstructing the entire display by merging the new screens into the Intuition **View**

structure. This function calls the graphics primitives **MrgCop()** and **LoadView()**. It takes no arguments.

- **RemakeDisplay()**

This routine remakes the entire Intuition display. It performs a **MakeVPort()** (graphics primitive) on every Intuition screen and then calls **RethinkDisplay()** to recreate the view. It takes no arguments.

## Examples

### LOW-RES SCREEN EXAMPLE

This example is called "greetings.c". It opens a window on a low-resolution screen, and prints a message in it.

```
/* greetings.c -- Opens a window on a low-res screen and writes a greeting. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. */
#include "hires.h"
#include "graniteWindow.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33
#define GRAPHICS_REV 33

extern VOID cleanExit( struct Screen *, struct Window *, int );
extern UBYTE handleIDCMP( struct Window *);

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;
    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit( screen1, window1, RETURN_WARN );

    /* Open any other required libraries */
    GfxBase = (struct GfxBase *)
        OpenLibrary( "graphics.library", GRAPHICS_REV );

    if ( GfxBase == NULL )
        cleanExit( screen1, window1, RETURN_WARN );
}
```

```

/* Make the assignments that were postponed above */
fullHires.Width = 320;      /* Make custom screen low-res */
fullHires.ViewModes = NULL;
graniteWindow.Width /= 2;   /* Cut the window to fit */

/* Open the screen */
screen1 = OpenScreen(&fullHires);
if (screen1 == NULL)
    cleanExit(screen1, window1, RETURN_WARN);

/* Attach the window to the open screen ... */
graniteWindow.Screen = screen1;

/* ... and open the window */
window1 = OpenWindow(&graniteWindow);
if (window1 == NULL)
    cleanExit(screen1, window1, RETURN_WARN);

/* Set up the signals that you want to hear about ... */
signalmask = 1L << window1->UserPort->mp_SigBit;

/* Call the functions that do the main processing */

/* Write the message into the window */
Move(window1->RPort, 20, 20);
Text(window1->RPort, "Hello World!", 12);

/* And wait to hear from your signals */
while( !done )
{
    signals = Wait(signalmask);
    if (signals & signalmask)
        done = handleIDCMP(window1);
};

/* Exit the program */
cleanExit(screen1, window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
    {
        class = message->Class;

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);

        /* See what events occurred */
        switch( class )
        {
            case CLOSEWINDOW:
                flag = 1;
                break;
            default:
                break;
        }
    }
    return(flag);
}

VOID cleanExit( scrn, wind, returnValue )
struct Screen *scrn;
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

```

```

/* Close the window and the screen */
if (wind) CloseWindow( wind );
if (scrn) CloseScreen( scrn );

/* Close the library, and then exit */
if (GfxBase) CloseLibrary( (struct Library *)GfxBase );
if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

exit(returnValue);
}

```

## DUAL-PLAYFIELD SCREEN EXAMPLE

This example shows how to create a dual-playfield display. It makes a dual-playfield out of the Workbench screen, but we hasten to warn that this is only for illustration. You would normally make a dual-playfield display out of a custom screen. Try it!

Setting the DUALPF flag in the NewScreen.Flags field is not the best method of obtaining a dual playfield viewport for your screen. It is better to open a standard screen, passing to Intuition (or letting Intuition create) only one of your playfield bitmaps (the front one). Then you allocate and set up a second BitMap, its bit-planes, and a RasInfo structure. Install these into new screen's viewport, change the viewport modes to include DUALPF, MakeScreen(), and RethinkDisplay(). This method keeps Intuition rendering (gadgets, menus, windows) in a single playfield. This is the method that we demonstrate here.

```

/* dualpf.c - Shows a dual-playfield. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */
#include <exec/memory.h>

/* Include user-written header files here. */
#include "graniteWindow.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33
#define GRAPHICS_REV 33

extern VOID cleanExit( struct Screen *, struct Window *, struct RasInfo *,
                      struct BitMap *, struct RastPort *, int, int );
extern VOID drawSomething( struct RastPort * );
extern UBYTE handleIDCMP( struct Window * );

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

VOID main(int argc, char *argv[])
{
/* Declare variables here */
ULONG signalmask, signals;
UBYTE done = 0;
struct Screen *wbscreen = NULL;
struct Window *window1 = NULL;
struct RasInfo *rinfo2 = NULL; /* Second playfield rasinfo */
struct BitMap *bmap2 = NULL; /* Second playfield bitmap */
struct RastPort *rport2 = NULL; /* Used to render into bmap2 */
int it_is_done = 0; /* Success flag */

```

```

/* Open the Intuition Library */
IntuitionBase = (struct IntuitionBase *)
    OpenLibrary( "intuition.library", INTUITION_REV );

if (IntuitionBase == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

/* Open any other required libraries */
GfxBase = (struct GfxBase *)
    OpenLibrary("graphics.library", GRAPHICS_REV);

if ( GfxBase == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

/* Make the assignments that were postponed above */
graniteWindow.DetailPen = -1;
graniteWindow.BlockPen = -1;
graniteWindow.Title = " Dual Playfield Mode ";
graniteWindow.Type = WBENCHSCREEN;

/* Open the screen */
/* Workbench is already open */

/* Open the window */
window1 = OpenWindow(&graniteWindow);
if (window1 == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

/* Set up the signals that you want to hear about ... */
signalmask = 1L << window1->UserPort->mp_SigBit;

/* Call the functions that do the main processing */
wbscreen = window1->WScreen; /* Find the Workbench screen */

/* Allocate the second playfield's rasinfo, bitmap, and bitplane */
rinfo2 = (struct RasInfo *) AllocMem( (ULONG)sizeof(struct RasInfo),
    (ULONG) MEMF_PUBLIC|MEMF_CLEAR);

if (rinfo2 == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

bmap2 = (struct BitMap *) AllocMem( (ULONG)sizeof(struct BitMap),
    (ULONG) MEMF_PUBLIC|MEMF_CLEAR);

if (bmap2 == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

InitBitMap(bmap2, (BYTE)1, (LONG)wbscreen->Width, (LONG)wbscreen->Height);

/* We'll use one bitplane */
bmap2->Planes[0] = (PLANEPTR) AllocRaster( (LONG)wbscreen->Width,
    (LONG)wbscreen->Height);

if (bmap2->Planes[0] == NULL)
    cleanExit(wbscreen, window1, rinfo2, bmap2, rport2, it_is_done, RETURN_WARN);

/* Get a rastport, and set it up for rendering into bmap2 */
rport2 = (struct RastPort *) AllocMem( (ULONG)sizeof(struct RastPort),
    (ULONG) MEMF_PUBLIC);

if (rport2 == NULL)
    cleanExit( wbscreen, window1, rinfo2, bmap2, rport2,
        it_is_done, RETURN_WARN);

InitRastPort(rport2);
rport2->BitMap = bmap2;
SetRast(rport2, (UBYTE)0);

```

```

/* Manhandle the viewport: install second playfield and change modes */
Forbid();
rinfo2->BitMap = bmap2;      /* Install new bitmap into new rasinfo */

/* Install rinfo for viewport's second playfield */
wbscreen->ViewPort.RasInfo->Next = rinfo2;

wbscreen->ViewPort.Modes |= DUALPF; /* Convert viewport */
it_is_done = 1;
Permit();

/* Set foreground color; color 9 is color 1 for second */
/* playfield of hi-res viewport */
SetRGB4(&wbscreen->ViewPort, (SHORT)9, (UBYTE)0, (UBYTE)0xF, (UBYTE)0);

/* Put viewport change into effect */
MakeScreen(wbscreen);
RethinkDisplay();

/* Now ... */
drawSomething(rport2);

/* And wait to hear from your signals */
while( !done )
{
    signals = Wait(signalmask);
    if (signals & signalmask)
        done = handleIDCMP(window1);
};

/* Exit the program */
cleanExit( wbscreen, window1, rinfo2, bmap2, rport2, it_is_done, RETURN_WARN);
}

VOID drawSomething(struct RastPort *rp)
{
    int width, height;
    int r, c;

    width = rp->BitMap->BytesPerRow * 8;
    height = rp->BitMap->Rows;

    SetAPen(rp, 1L);

    for (r = 0; r < height; r += 40)
        for (c = 0; c < width; c += 40)
        {
            Move(rp, 0L, (LONG) r);
            Draw(rp, (LONG) c, 0L);
        }
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
    {
        class = message->Class;

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);

        /* See what events occurred */
        switch( class )
        {
            case CLOSEWINDOW:
                flag = 1;
                break;

```

```

        default:
            break;
    }
}

return(flag);
}

VOID cleanExit( scrn, wind, rasi, bitm, rasp, flag, returnValue )
struct Screen *scrn;
struct Window *wind;
struct RasInfo *rasi;
struct BitMap *bitm;
struct RastPort *rasp;
int flag, returnValue;
{
    /* clean up dual-playfield trick */
    if (flag)
    {
        Forbid();
        scrn->ViewPort.RasInfo->Next = NULL;
        scrn->ViewPort.Modes &= ~DUALPF;
        Permit();
        MakeScreen(scrn);
        RethinkDisplay();
    }

    /* Close things in the reverse order of opening */
    if (rasp) FreeMem(rasp, (ULONG)sizeof(struct RastPort));

    if (bitm)
    {
        if (bitm->Planes[0])
            FreeRaster(bitm->Planes[0], (LONG)scrn->Width, (LONG)scrn->Height);
        FreeMem(bitm, (ULONG)sizeof(struct BitMap));
    }

    if (rasi) FreeMem(rasi, (ULONG)sizeof(struct RasInfo));

    /* Close the window and NOT the screen */
    if (wind) CloseWindow( wind );

    /* Close the libraries, and then exit */
    if (GfxBase) CloseLibrary( (struct Library *)GfxBase );
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}

```



## **Chapter 3**

### **Intuition: Windows**

This chapter provides a general description of windows: how to handle the I/O of the virtual terminal; how to preserve the display when windows get overlapped; how to open windows and define their characteristics; and how to get the system gadgets for shaping, moving, closing, and depth-arranging windows. It also explains the special windows, and how to customize windows by adding touches like a custom pointer.

#### **About Windows**

The windows you open can be colorful, lively, and interesting places for the user to work. You can use all of the standard Amiga graphics, text, and animation primitives (functions) in every one of your windows. You can also use the quick and easy Intuition structures and functions for rendering images, text, and lines into your windows. The special Intuition features that go along with windows, like the gadgets and menus, can be visually exciting as well.

Each window can open an Intuition Direct Communications Message Port (IDCMP), which offers a direct communication channel with the underlying Intuition software, or the window can open a console device for input and output. Either of these communication methods turns the window into a visual representation of a virtual terminal, where your program can carry on its interaction with the user as if it had the entire machine and display to itself. Your program can open more than one window and treat each separately as a virtual terminal.

Both you and the user deal with each individual window as if it were a complete terminal. The user has the added benefit of being able to arrange the windows front to back, shrink and expand them, or overlap them.

Windows are rectangular display areas whose size and location can be adjusted in many ways. The user can shape windows by making them wider or longer or both to reveal more of the information being output by the program. He can also shrink windows into long, narrow strips or small boxes to reveal other windows or to make room for other windows to open. Multiple windows can be overlapped, and the user can bring a window up front or send it to the bottom of the stack with a click of the mouse button. While the user is doing all this shaping and rearranging and stacking of windows, your program need not pay any attention. To the program, there is nothing out there but a user with a keyboard and a mouse (or, in place of a mouse, there could be a joystick, a graphics tablet, or practically any other input device).

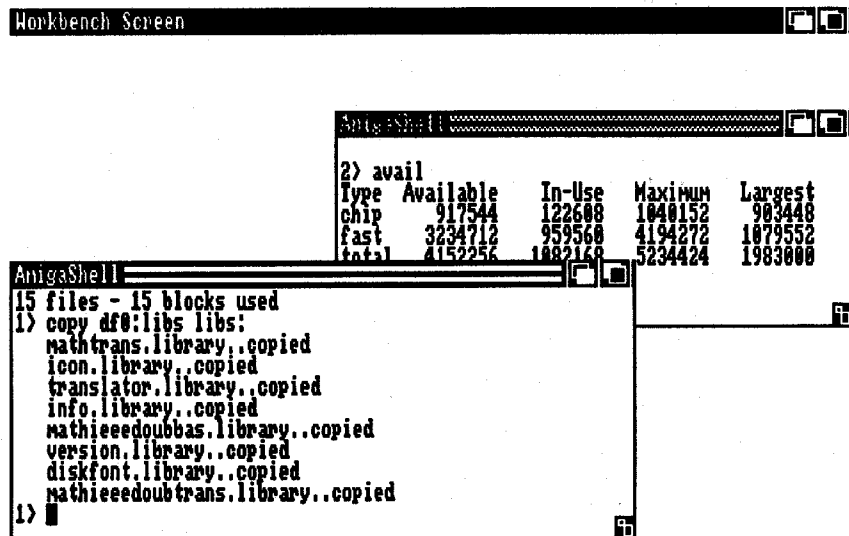


Figure 3-1: A High-resolution Screen and Windows

Your program can open as many of these virtual terminal windows as the memory configuration of your Amiga will allow. Each window opens in a specific screen, and several windows may open in the same screen. Even windows opened by different programs may coexist in the same screen.

Your program can open windows for any purpose. For example, different windows of an application can represent:

- Different interpretations of an object, such as the same data represented as a bar chart and a pie chart.
- Related parts of a whole, such as the listing and output of a program.
- Different parts of a document or separate documents being edited simultaneously.

You open a window by specifying its structure and issuing a call to a function that opens windows. After that, you can output to the user and receive input while Intuition manages all the user's requests to move, shape, and depth-arrange the window. Intuition lets you know if the user makes a menu choice, chooses one of your own custom gadgets, or wants to close the window. If you need to know when the user changes the window's size or moves the pointer, Intuition will tell you about that, too.

Custom gadgets, menus, input/output, and controllers are dealt with in later chapters. The balance of this section deals with some important concepts you'll need to know before attempting to open your own windows.

## WINDOW INPUT/OUTPUT

You can choose from two different paths for input and two for output. Each path satisfies particular needs. The two paths for user input are as follows:

- **Intuition Direct Communications Message Ports (IDCMPs).** The message ports give you mouse (or other controller) events, keyboard events, and Intuition messages in their most raw form; in addition, these ports supply the way for your program to send messages *to* Intuition.
- **Console device.** The console ports give you processed input data, including keycodes translated to ASCII characters and Intuition event messages converted to ANSI escape sequences. If you wish, you can also get raw (untranslated) input through the console device.

There are also two paths for program output:

- **Text** is output through the console device, which formats and supplies special text primitives and text functions, such as automatic line wrapping and scrolling.
- **Graphics** are output through the general-purpose Amiga graphics primitives, which provide rendering functions such as area fill and line-drawing and animation functions.

If you use the console device for input, output, or both, you need to open it after opening your window. If you want the IDCMP for input, you specify one or more of the IDCMP flags in the **NewWindow** structure. This automatically sets up a pair of message ports, one for Intuition and one for you. Although the IDCMP does not offer text formatting or character positioning, it has many special features that you may want, and it requires less RAM and less processing overhead.

For more information about I/O methods read the chapter entitled "Input and Output Methods."

## OPENING WINDOWS

Before your program can open a window, you need to initialize a **NewWindow** structure. This structure contains all the arguments needed to define and open a window, including initial position and size, sizing limits, color choices for window detailing, gadgets to attach, how to preserve the display, IDCMP flags, window type if it is one of the special windows, and the screen in which the window should open.

A window is opened and displayed by a call to the **OpenWindow()** function, whose only argument is a pointer to the **NewWindow** structure. After successfully opening a window, you receive a pointer to another structure, the **Window** structure.

## WINDOWS AND SCREENS

You may open your window on the Workbench screen or on your own custom screen. To use the Workbench screen, set the **Type** field in your **NewWindow** structure to **WBENCHSCREEN**. If you want to use a custom screen, first open your screen, then put a pointer to the resulting screen structure in the **Screen** field of your **NewWindow** structure, and set the **Type** field to **CUSTOMSCREEN**.

System requesters appear on the Workbench screen by default. It is possible to redirect system requesters to appear on your custom screen, instead. You must open a window on your screen in order to do this. You assign your window pointer to the `pr_WindowPtr` field of your `Process` structure, right after you open your window. Just before you close your window, you restore the old value of the `pr_WindowPtr` field. Examples one and two at the end of this chapter show how to do this.

When the system wants to open a requester for your process, it looks in the `pr_WindowPtr` field of the `Process` structure to find out where your window is. It then tries to put the requester in the same place as your window. If the `pr_WindowPtr` field is `NULL`, then the system will open the requester on the default screen. If the `pr_WindowPtr` field is `-1`, then the system assumes that the user pressed cancel, and the requester does not open.

System requesters not associated with your process can't be redirected to your screen. One example is the system requester that informs you that "You MUST replace volume x in unit y." This requester is associated with a filesystem instead of your process.

## THE ACTIVE WINDOW

Only one window is active in the system at a time. The active window is the one that is receiving user input through a keyboard and mouse (or some other controller). Some areas of the active window are displayed more boldly than those on inactive windows. In particular, the title bars of inactive windows are covered with a faint pattern of dots, rendering them slightly less distinct. This is called ghosting. When the user brings up a menu list in the screen title bar, the active window's menu list is displayed.

Your program need not worry about whether or not one of its windows is active. The inactive windows can just wait for the user to get back to them, or they can be doing some background task that requires no user input. The job of activating windows is mostly left up to the user, who activates a window by moving the pointer into the window and clicking the left mouse button. There is, however, an `ACTIVATE` flag in the `NewWindow` structure. Setting this flag causes the window to become active when it opens. If the user is doing something else when a window opens with the `ACTIVATE` flag set, input is immediately redirected to the newly opened window. In general, you should set the `ACTIVATE` flag only in a window which opens as a direct response to the user's action, such as starting your program or asking for some operation which necessitates a new window.

After your window is opened, you can discover when it is activated and when it is inactivated by setting the `IDCMP` flags `ACTIVEWINDOW` and `INACTIVEWINDOW`. If you set these flags, the program will receive a message every time the user activates your window or causes your window to become inactive by activating some other window.

## CLOSING WINDOWS

Although there is a window closing gadget, a window does not automatically close when the user selects this gadget. Intuition sends the program a message about the user's action. The program can then do whatever clean-up is necessary, such as replying to any outstanding Intuition messages or verifying that the user really meant to close the window, and then call `CloseWindow()`.

When the active window is closed, the previously active window may become the active window. The window (call it window A) that was active when this one was *opened* will become the active window. If window A is already closed, then the window (if any) that was active when window A opened will become the active window, and so on. However, you are not to count on this behavior. If you care whether your window becomes active, set the `ACTIVEWINDOW` `IDCMP` flag, in order to receive messages from Intuition to that effect.

## SPECIAL WINDOW TYPES

Intuition's special windows give you some very useful bonus features, in addition to all the normal window features. The Backdrop window stays anchored to the back of the display and provides a way to take over the display without taking over the machine. The Borderless window supplies a window with no drawn border lines. Gimmezerozero windows give you all the border features *plus* the freedom to ignore borders altogether when you are drawing into the window.

### NOTE

There is a good deal of overhead in using Gimmezerozero windows as they use a separate layer for the border display. See the "Layers" chapter for information on setting up user clip regions to limit graphics display in the window.

Finally, the SuperBitMap window not only gives you your own display memory in which to draw, but also frees you from ever worrying about preserving the window when the user sizes it or overlaps it with another window.

Notice that these are not necessarily separate, discrete window types. You can combine them for even more special effects. For instance, you can create a Backdrop, Borderless window that fills the entire screen and looks like a normal computer display terminal.

### Borderless Window Type

This window is distinguished from other windows by having no default borders. With normal windows, Intuition creates a thin border around the perimeter of the window, allowing the window to be easily distinguished from other windows and the background. When you ask for a Borderless window, you do not get this default thin border; however, your window can still have borders. It can have borders based solely on the location of border gadgets and whether or not you have supplied title text, or it may have no gadgets or text and thus no visible borders and no border padding at all. You can use this window to cover the entire video display. It is especially effective combined with a Backdrop window. This combination forms a window that you can render in almost as freely as writing directly to the display memory of a custom screen. It has the added benefit that you can render in it without running the risk of trashing menus or other windows in the display.

If you use a Borderless window that does not cover the entire display, be aware that its lack of borders may cause visual confusion on the screen. Since windows and screens share the same color palette, borders are often the only way of distinguishing a window from the background.

Set the BORDERLESS flag in the **NewWindow** structure to get this window type.

### Gimmezerozero Window Type

The unique feature of a Gimmezerozero window is that there are actually two "planes" to the window: a larger, outer plane in which the window title, gadgets, and border are placed; and a smaller, inner plane (also called the *inner window*) in which you can draw freely without worrying about the window border and its contents. The top left coordinates of the inner window are always (0,0), regardless of the size or contents of the outer window; thus the name "Gimmezerozero."

The area in which you can draw is formally defined as the area within the variables **BorderLeft**, **BorderTop**, **BorderRight**, and **BorderBottom**. These variables are computed by Intuition when the window is opened. To draw in normal windows with the graphics primitives (for instance to draw a line from the top left to somewhere else in the window), you have to start the line away from the window title bar and borders. Otherwise, you risk drawing the line over the title bar and any gadgets that may be in the borders. In a Gimmezerozero window, you can just draw a line from (0,0) to some other point in the window without worrying about the window borders.

The Gimmezerozero window uses more RAM than other window types and degrades performance in the moving and sizing of windows. There can be a noticeable performance lag, especially when several Gimmezerozero windows are open at the same time.

There are some special variables in the **Window** structure that pertain only to Gimmezerozero windows. The **GZZMouseX** and **GZZMouseY** variables can be examined to discover the position of the mouse relative to the inner window. The **GZZWidth** and **GZZHeight** variables can be used to discover the width and height of the inner window.

The console device gives you another kind of encumbrance-free window. If you are using the console device, any formatted text you output goes into an inner window automatically; you need not worry about gadgets. Therefore, you do not need a Gimmezerozero window just for the purpose of text output. See the chapter entitled "Input and Output Methods," for more information about this aspect of the console device.

Requesters in a Gimmezerozero window are positioned relative to the inner window. If you are bringing up requesters in the window, you may wish to take this into consideration when deciding where to put them. See the chapter entitled "Requesters and Alerts," for more information about requester location.

To specify a Gimmezerozero window, set the **GIMMEZEROZERO** flag in the window structure's flags. All system gadgets you attach to this type of window will go into the outer window automatically; however, if you are attaching custom gadgets and you want the gadgets to appear in the border (*not* in the inner window), be sure to set the **GZZGADGET** flag in your gadget structures. If you do not, Intuition will draw custom gadgets in the display of the inner window.

### **Backdrop Window Type**

The Backdrop window is always in back of any other kind of window. Its great advantage is that other windows can overlap it and be depth-arranged without ever going behind the Backdrop window. Because of this characteristic, you can use the Backdrop window as a primary display surface while opening other auxiliary windows on top of it.

The Backdrop window is like normal windows except that:

- It always opens behind all other windows (including other Backdrop windows that you might already have opened).
- The only system gadget you can attach is the close-window gadget. (You can attach your own gadgets as usual.)
- Normal windows in the same screen open in front of all Backdrop windows and always stay in front of them. No amount of depth arranging will ever send a non-Backdrop window behind a Backdrop window.

You might want to use a Backdrop window, for example, in a simulation program in which the environment is rendered in the Backdrop window while the simulation controls exist in normal windows that float above the environment. Another example is a sophisticated graphics program where the primary work surface is on the Backdrop window while auxiliary tools are made available in normal windows in front of the work surface.

You can often use a Backdrop window instead of drawing directly into the display memory of a custom screen. If you want to draw in your background with the graphics primitives, you may even prefer a Backdrop window to a custom screen because you do not run the danger of writing to the window at the wrong time and trashing a menu that is being displayed. In fact, if you also set the BORDERLESS flag and you create a window that is the full-screen width and height, you get a window that fills the entire screen and stays in the background. If you also specify no gadgets, there will be no borders. Finally, if you add a call to `ShowTitle()` with an argument of FALSE, the window will conceal the screen title. (See *The Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a complete list of arguments for `ShowTitle()`.) All of these steps result in a window that fills the entire video display, has no borders, and stays in the background.

To use the Backdrop feature, you set the BACKDROP flag in the window structure.

## SuperBitMap Window

SuperBitMap is both a window type and a way of preserving and redrawing the display. This window is like other windows except that you supply your own bit-map instead of using the one belonging to the screen. The windowing system displays some portion of the window's bit-map in the screen's raster according to the dimensions and limits you specify and the user's actions. You can make the bit-map any size as long as the window sizing limits are set accordingly.

This window is handy when you want to give the user the flexibility of scrolling around and revealing any portion of the bit-map. You can do this because the entire bit-map is always available to be displayed.

To get this type of window, set the SUPER\_BITMAP flag in the window structure and set up a BitMap structure. You probably want to set the GIMMEZEROZERO flag also, so that the borders and gadgets will be rendered in a separate bit-map. You need to be certain that the size-limiting variables in the window structure are properly set, considering the size of the bit-map and how much of it you want to display.

For complete information about SuperBitMap, see "Setting Up a SuperBitMap Window" later in this chapter.

## WINDOW GADGETS

The easiest way for a user to communicate with a program running under Intuition is through the use of window gadgets. There are two basic kinds of window gadgets—system gadgets that are predefined and managed by Intuition and your own custom application gadgets.

### System Gadgets

System gadgets are supplied to allow the user to manage the following aspects of window display: size and shape of windows, location of windows on the screen, and depth arrangement. Also, there is a system gadget for the user to tell the application when he or she is ready to close the window. These gadgets save you a lot of work because, with the exception of the close gadget, your program never has to pay any attention to what the user does with them. On the other hand, if you want to be notified when the user sizes the window because of some special drawing you may be doing in the window, Intuition will let you know. For more information, read about the IDCMP verify functions in the chapter "Input and Output Methods."

In the NewWindow structure, you define the starting location and starting size of a window and a maximum and minimum height and width for sizing the window. When the window opens, it appears in the location and in the

size you have specified. After that, however, the user normally has the option of shaping the window within the limits you have set, moving it about on the screen and sending it into the background behind all the other displayed windows or bringing it into the foreground. To give the user this freedom, plus the ability to request that the window be closed, you can attach system gadgets to the window. The graphic representations of these gadgets are predefined, and Intuition always displays them in the same standard locations in the window borders. In the window structure, you can set flags to request that all, some, or none of these system gadgets be attached to your window. The system gadgets and their locations in the window are:

- A *sizing* gadget in the lower right of the window. With the sizing gadget, the user can stretch or shrink the height and width of the window. You set the maximum and minimum limits for sizing. You can specify whether this gadget is located in the right border or bottom border, or in both borders.
- Two *depth-arrangement* gadgets in the upper right of the window. One sends the window behind all other displayed windows (back gadget) and the other brings the window to the front of the display (front gadget).
- A *drag* gadget, which occupies every part of the window title bar not taken up by other gadgets. The drag gadget allows the user to move the window to a new location on the screen. A title in the title bar does not interfere with drag gadget operation.
- A *close* gadget in the upper left of the window, which allows the user to request that the window be closed.

The figure shows how all the system window gadgets look and where they are located in the window borders.

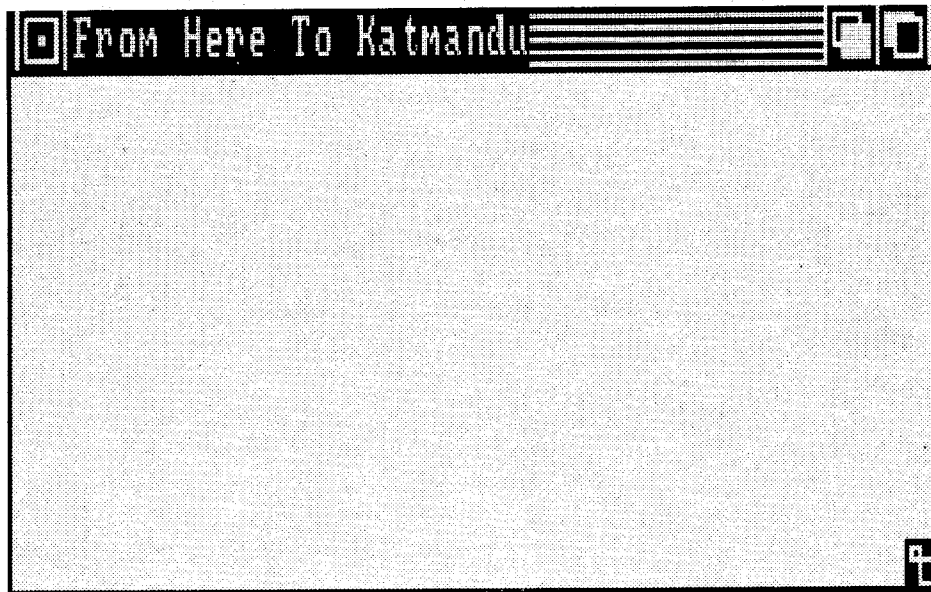


Figure 3-2: System Gadgets for Windows



## Application Gadgets

Four types of application gadgets are available—proportional, boolean, string, and integer. You can use application gadgets to request various kinds of input from the user, and that input can affect the application in any way you like. You design gadgets as text and graphic images to go anywhere in the window. For application gadgets, you define a data structure for each one and create a linked list of these structures. To attach your list of gadgets to a window, set a pointer in the `NewWindow` structure to point to the first gadget in the list. For details about creating gadgets, see the chapter “Gadgets.”

## WINDOW BORDERS

Intuition offers you several possibilities for handling window borders. You can take advantage of the fancy border features, such as automatic double border lines around the window and automatic padding of borders to allow for gadgets. If you'd rather, you can eliminate borders completely, or you can use the Gimmezerozero window, which gives you all the border features and then lets you ignore them.

The actual border *lines* are drawn around the perimeter of the window and are mostly distinct from the border *area* in which border gadgets are placed. Intuition automatically draws a double border around a window unless you ask for something different (such as by setting the `BORDERLESS` flag.) This nominal border consists of an outer line around the entire window, rendered in the `BlockPen` color, and within this a second line, rendered in the pen 0 color, the background color of Workbench. The inner line is most easily seen in a program like Notepad, the Workbench notepad utility, which uses a different background color than Workbench. `BlockPen` is defined, along with its mate, `DetailPen`, in the `NewWindow` structure.

The default minimum thickness of the border areas depends upon certain parameters set in the definition of the underlying screen, certain choices the user has made with Preferences, and the default font. If the window is not a special Borderless window, the borders will be at least the default thickness. Intuition adjusts the size of a window's border areas to accommodate system gadgets or your own application gadgets.

You can find the thickness of the border areas in the variables `BorderLeft`, `BorderTop`, `BorderRight`, and `BorderBottom`. These variables are computed when the window is opened and can be found in the `Window` structure. You may want to use them to position visual elements within your window, for example, if you are drawing lines in the window with graphics primitives, which require you to specify a set of coordinates as the beginning and ending points for the line. In a typical window, you cannot specify a line from (0,0) to (50,50) because you may draw a line over the window title bar. Instead, you would use the border variables to specify a line from (0+`BorderLeft`, 0+`BorderTop`) to (50+`BorderLeft`, 50+`BorderTop`). This may look clumsy, but it offers a way of avoiding a Gimmezerozero window, which—although much more convenient to use—requires extra memory and degrades performance.

For the top border, in addition to the system gadgets and your own gadgets, you can specify a window title. The window title bar does not appear unless you specify one of the following:

- A window title.
- Any of the system gadgets for window dragging, window depth arranging, or window closing.

Usually, borders are drawn automatically and adjusted *within* the dimensions you specify in the `NewWindow` structure. In the special Borderless and Gimmezerozero windows, however, borders are handled differently. A Borderless window has no drawn borders and no automatic border spacing or padding. If you have system gadgets

or your own gadgets with a border flag set, borders may be visually defined by the gadgets. A Gimmezerozero window places the borders and gadgets in their own bit-map, separate from the window's bit-map. This means you can draw freely across the entire surface of the window without worry about scribbling over the gadgets.

You can specify whether or not your application gadgets reside in the borders, and in which border, by setting a flag in the **Gadget** structure. See the chapter "Gadgets," for more information about gadgets and how to place them where you want them.

## **PRESERVING THE WINDOW DISPLAY**

When a window is revealed after having been overlapped, the display has to be redrawn. Intuition offers three ways of preserving the display:

- In the **Simple Refresh** method, your program redraws the display.
- In the **Smart Refresh** method, Intuition keeps a copy of the display in RAM buffers.
- In the **SuperBitMap** method, you allocate an entirely separate display memory for your window.

Smart Refresh and SuperBitMap use the window's idea of its display memory space to save the parts of the window that are not currently being displayed. Windows and other high-level display components, such as menus and gadgets, have a "virtual" understanding of their display memory. The application can ignore other windows being displayed and write into its own virtual memory area. The Amiga graphics software then takes these requests to draw in virtual display memory and translates them into real operations that are placed in save buffers (for Smart Refresh) or in areas of a private bit-map (for SuperBitMap) maintained by the application.

The three methods of preservation are explained below. You must choose one of them.

### **Simple Refresh**

With the Simple Refresh redrawing method, Intuition does not need to remember anything about windows that are overlapped. For the most part, the program is responsible for redrawing the window. If the user sizes the window larger on either axis or reveals a window that was overlapped, the program must redraw the display. However, if the user merely drags the window around, Intuition preserves it and redisplay it in the new location. Simple Refresh tends to be slower than other methods, but it is memory-efficient, since no RAM is consumed in saving the obscured portions of a window. Simple Refresh uses the screen's display memory for the window's display.

Your program can be notified by Intuition when part of a window needs to be redrawn. In addition, Intuition supplies functions that limit the redrawing to the "damaged" area, without your program having to know which part of the window was affected. This greatly speeds up the refresh process. A Simple Refresh window is appropriate when your application can redraw its visuals relatively rapidly.

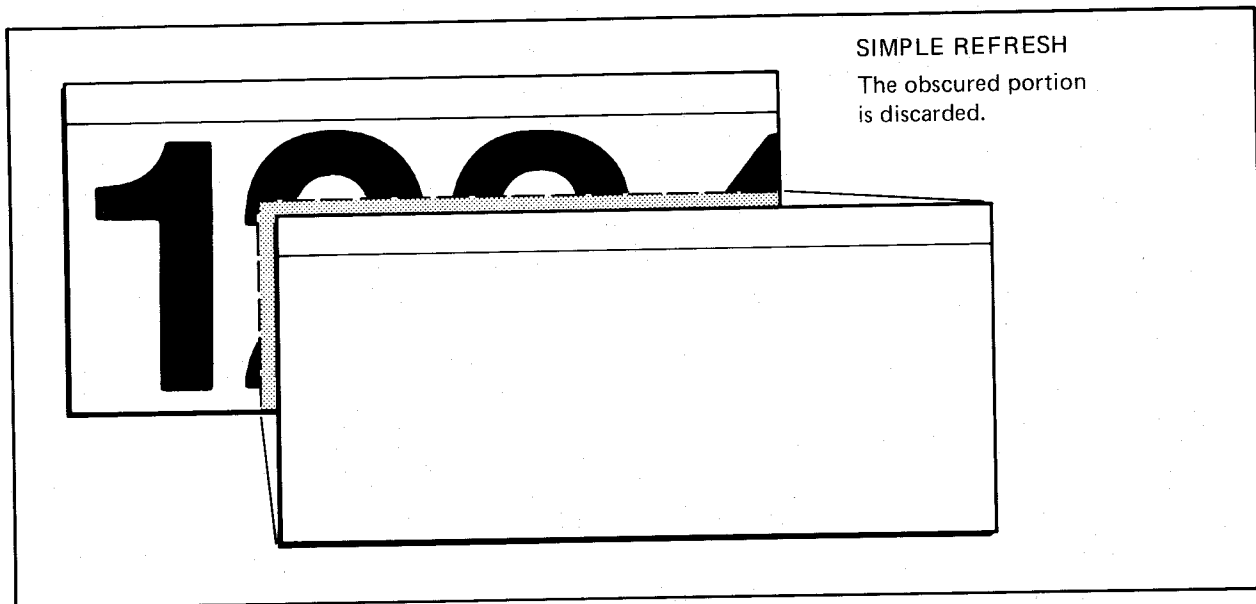


Figure 3-3: Simple Refresh

### Smart Refresh

With the Smart Refresh redrawing method, Intuition keeps all information about the window in RAM, whether the window is currently concealed or is up front. If the user reveals a window that was overlapped, Intuition recreates the display. If the window has a sizing gadget, then when the user makes the window larger, the application is still responsible for creating the display in the new portion of the window. Intuition will notify your application and offers the same ability to constrain the redrawing as is done for Simple Refresh. Smart Refresh windows are appropriate when regenerating the display would take too long. Smart Refresh uses the screen's display memory for the window display and requires extra buffers for the off-screen portions of the window (portions not currently being displayed). Smart Refresh uses more display memory but redraws the display faster than Simple Refresh.

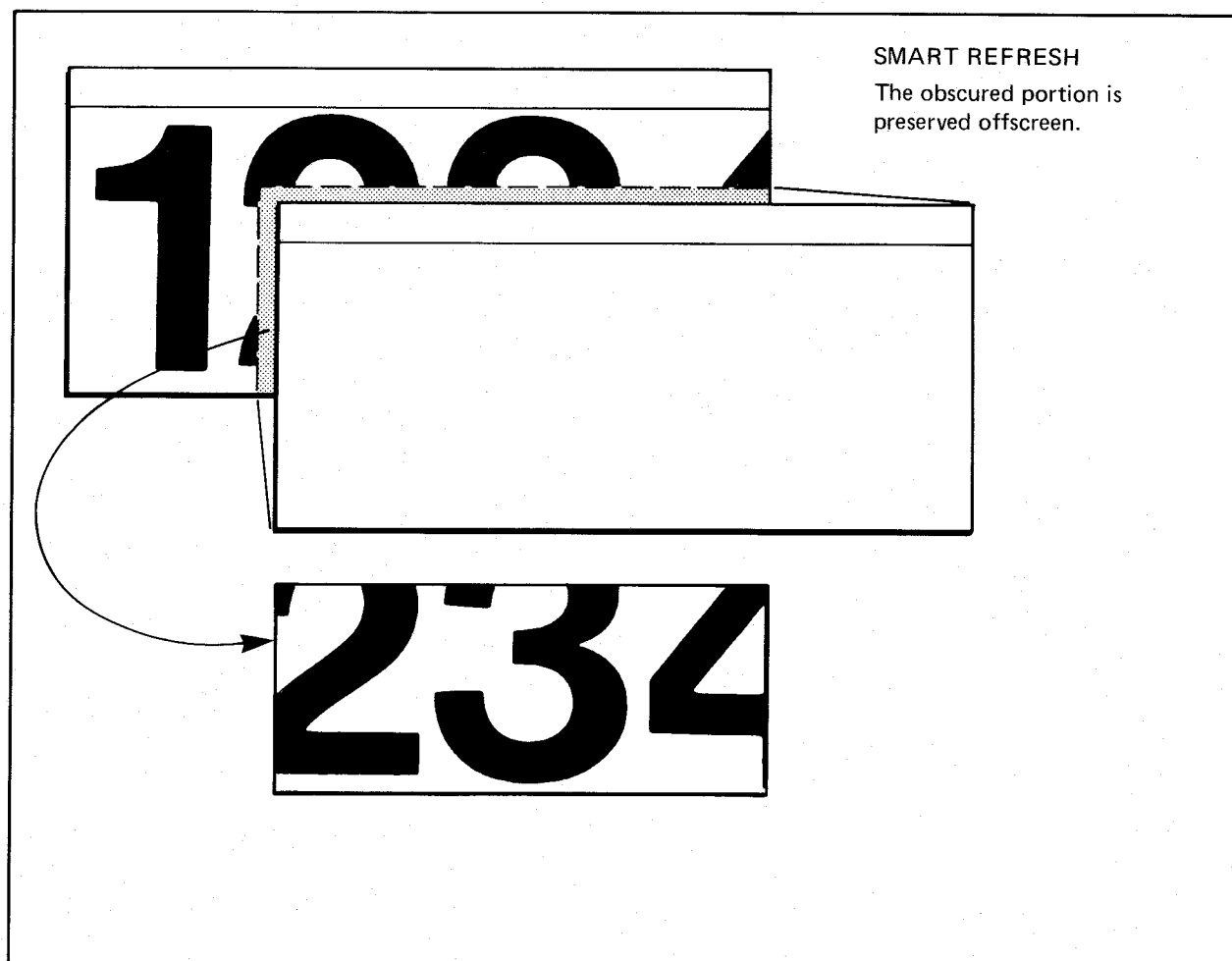


Figure 3-4: Smart Refresh

### SuperBitMap

This is both a special type of window and a method of redrawing the display. When you choose this method of redrawing, you get your own bit-map to use as display memory instead of using the screen's display memory. You make this bit-map as large as the window can get (or larger). You never have to worry about redisplay after the window is uncovered because the entire display is always there in RAM.

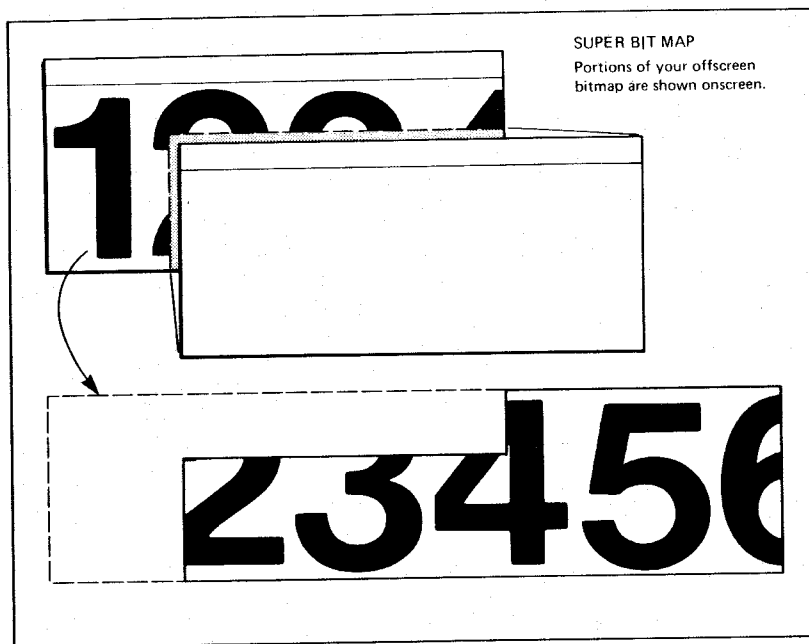


Figure 3-5: SuperBitMap Refresh

## REFRESHING THE WINDOW DISPLAY

If you open either a Simple Refresh or a Smart Refresh window, your program may be asked to refresh part of your display at some time. When a Simple Refresh window is moved or sized, or when other windows are moved or sized in such a way that areas of a Simple Refresh window are revealed, the window will have to be refreshed. With Smart Refresh windows, the window must be sized larger on either axis to generate a REFRESHWINDOW event.

The program finds out that the window needs refreshing via either source of input, the IDCMP or the console device. A message of the class REFRESHWINDOW arrives at the IDCMP, telling the program that the window needs to be refreshed. Every time the program learns that it should refresh a window, it must take some action, even if it is just the acceptable minimum action described below. (See the chapter "Input and Output Methods" for further information.)

When the program is asked to refresh a window, before actually starting to refresh it the program should call the Intuition function **BeginRefresh()**. This function makes sure that refreshing is done in the most efficient way, only redrawing those portions of the window that really need to be redrawn. The rest of the rendering commands are discarded.

After **BeginRefresh()** is called, the program should redraw its display. Then, call **EndRefresh()** to restore the state of the internal structures.

When using **Begin/EndRefresh()** restrict your operations to simple rendering. All of the rendering functions in `intuition.library` and `graphics.library` are safe. **RefreshGadgets()** is also permissible, but probably unnecessary. Avoid calls that may lock the `LayerInfo`, or get complicated in Intuition, since **BeginRefresh()** leaves the window's layer or layers locked. Avoid **AutoRequest()**, and therefore all direct or indirect disk-related DOS calls. Though **BeginRefresh()** and **EndRefresh()** provide good results in refreshing a smart window, the refreshing of gadgets can be more complicated. This topic is covered in the gadgets chapter.

Even if you don't want the program to redraw immediately, you should make sure the program at least calls **Begin/EndRefresh()** each time it is asked to refresh a window. This helps Intuition and the layer library keep things sorted and organized.

If you are opening a window that you will never care to refresh, no matter what happens to it or around it, then you can avoid having to call **BeginRefresh()** and **EndRefresh()** by setting the **NOCAREREFRESH** flag in the **NewWindow** structure when you open your window.

## WINDOW POINTER

The active window contains a pointer to allow the user to make selections from menus, choose gadgets, and so on. The user moves the pointer around with a mouse controller, other kinds of controllers, or the keyboard cursor keys.

### Pointer Position

If your program needs to know about pointer movements, you can either look at the position variables or arrange to receive messages each time the pointer moves. The **MouseX** and **MouseY** fields of the **Window** structure always contain the current pointer x and y coordinates, whether or not your window is the active one. If you elect to receive messages, you get a set of x,y coordinates each time the pointer moves. These coordinates are relative to the upper left corner of your window and are reported in the resolution of your screen, even though the pointer's visible resolution is always in low-resolution mode (note that the pointer is actually a sprite).

If your window is a Gimmezerozero window, you can examine the variables **GZZMouseX** and **GZZMouseY** in the **Window** structure to find the position of the mouse relative to the upper left corner of the inner window.

To get messages about pointer movements, either **InputEvents** or message-port messages, you must set the **REPORTMOUSE** flag in your window structure. Thereafter, whenever your window is active, you'll get a broadcast every single time the pointer moves. This can be a lot of messages, so be prepared to handle them efficiently. If you want to change whether or not you are following mouse movements, you can call **ReportMouse()**.

You can also get messages about pointer movements by setting the flag **FOLLOWMOUSE** in your application gadget structures. If this flag is set in a gadget, the current pointer position is reported as long as that gadget is selected by the user. This can result in a lot of messages, too.

### Custom Pointer

You can set up your window with a custom pointer to replace the default arrow pointer. Your custom pointer will be displayed whenever your window is the active one. Good uses include a cross-hair pointer for a paint-program, or a "busy" pointer to indicate that your program is performing some operation and is not ready to respond to new requests from the user. To define the pointer, set up a sprite data structure (sprites are one of the general-purpose Amiga graphics structures). To place your custom pointer in the window, call **SetPointer()**. To remove your custom pointer from the window, call **ClearPointer()**. Both of these functions take effect immediately if yours is the active window.

Also, you can change the colors of the Intuition pointer. The Intuition pointer is always sprite 0. To change the colors of sprite 0, call the graphics library routine **SetRGB4()**. Refer to chapter on style for more information about this.

See the last section of this chapter for a complete example of a custom pointer.

## GRAPHICS AND TEXT IN WINDOWS

There are two ways of rendering graphics, lines, and text into windows. You can use all of the Amiga graphics, animation, and text primitives in any window. Also, you can use the quick and easy Intuition structures and functions to display Intuition **Image**, **Border**, or **IntuiText** structures in windows. See the chapter entitled "Images, Line Drawing, and Text," for more information about **Images**, **Borders**, and **IntuiText**.

## WINDOW COLORS

The number of colors you can use for the window display and the actual colors that will appear in the color registers are defined by the screen in which the window opens. In the window structure, you specify two color register numbers ("pens"), one for the border outline, text and gadgets and one for block fills (such as the title bar and menu backgrounds). These pen colors are also a function of the screen. You can specify different colors for the pens than those used by the screen or you can use the screen's pen colors.

## WINDOW DIMENSIONS

In the **NewWindow** structure, you define the dimensions and the starting location of your window on the screen. The position and dimensions of the window undergo error checking when the window is opened. The maximum dimensions of the window, specified as **NewWindow.MaxWidth** and **NewWindow.MaxHeight**, are unsigned and may legally be set to the maximum by using the value 0xFFFF, better expressed as ~0.

If you are letting the user change the size and shape of the window, you also need to specify the minimum size to which the window can shrink and the maximum size to which it can grow. If you do not ask that the window sizing gadget be attached to the window, then you need not initialize any of these maximum and minimum values.

In setting all these size dimensions, bear in mind the horizontal and vertical resolutions of the screen in which you are opening the window.

If you want to change the sizing limits after you have opened the window, you can call **WindowLimits()** with the new values.

## The NewWindow Structure

Here are the specifications for the **NewWindow** structure:

```
struct NewWindow
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    UBYTE DetailPen, BlockPen;
    ULONG IDCMPFlags;
    ULONG Flags;
    struct Gadget *FirstGadget;
    struct Image *CheckMark;
    UBYTE *Title;
```

```

struct Screen *Screen;
struct BitMap *BitMap;
SHORT MinWidth, MinHeight;
USHORT MaxWidth, MaxHeight;
USHORT Type;
};

```

The fields in the **NewWindow** structure are explained below. Some of the fields contain variables to which you need to assign a value, some contain flag bits to set or unset, and some are pointers to other structures.

### **LeftEdge, TopEdge, Width and Height**

These fields describe where your window will first appear on the screen and how large it will be initially. These dimensions are relative to the top left corner of the screen, which has the coordinates (0,0):

- LeftEdge**    The initial x position, which represents the offset from the first pixel on the line, pixel 0.
- TopEdge**    The initial y position, which represents how many lines down from the top (line 0) you want the window to begin.
- Width**       The initial width in pixels.
- Height**      The initial height in lines.

### **DetailPen and BlockPen**

These fields contain the “pen” numbers used to render details of the window. The colors associated with the pens are a function of the screen. If you supply a value of -1 for either of these, you will get the screen’s value for that pen by default.

- DetailPen**   The pen number (or -1) for the rendering of window details like gadgets or text in the title bar
- BlockPen**    The pen number (or -1) for window block fills (like the title bar) and the outer rim of the window border.

### **Flags**

#### **System Gadget Flags**

##### **WINDOWSIZING**

This flag allows the user to change the size of the window. Intuition places the window’s sizing gadget in the lower right of your window. By default, the right border is adjusted to accommodate the sizing gadget, but you can change this with the following two flags, which work in conjunction with **WINDOWSIZING**. The sizing gadget can go in either the right or bottom border (or both) of the window.

- The **SIZEBRIGHT** flag, which is the default, puts the sizing gadget in the right border.



- The SIZEBOTTOM flag puts the sizing gadget in the bottom border. You might wish to set this flag to put the sizing gadget in the bottom border if you want all possible horizontal bits—for instance, for 80-column text—and are willing to sacrifice vertical space.

#### WINDOWDEPTH

Setting this flag adds both the UPFRONT gadget to bring the window into the foreground and the DOWNBACK gadget to send it behind other currently displayed windows.

This allows the user to change the window's depth arrangement with respect to all other currently displayed windows. Intuition places the window depth-arrangement gadgets in the upper right of the window.

#### WINDOWCLOSE

Setting this flag attaches the standard close gadget to the upper left of the window. When the user selects this gadget, Intuition transmits a message to your application. It is up to the application to call `CloseWindow()` when ready.

#### WINDOWDRAG

This flag turns the entire title bar of the window into a drag gadget, allowing the user to move the window into a different position on the screen by clicking anywhere in the window title bar and moving the mouse or other controller.

#### NOTE

Even if you do not specify a text string in the Text variable shown below, a title bar appears if you use any one of the system gadgets WINDOWDRAG, WINDOWDEPTH, or WINDOWCLOSE. If no text is provided, the title bar is blank.

#### GIMMEZEROZERO

Set this flag if you want a Gimmezerozero window.

### Window Refresh Flags

The following four flags determine how Intuition preserves the display when an overlapped window is uncovered by the user. You *must* select one of the first three.

#### SIMPLE\_REFRESH

When this flag is set, every time a portion of the window is revealed the application program must redraw its display.

#### SMART\_REFRESH

When this flag is set, the only time you have to redraw your display is when the window's sizing gadget is used to make the window larger.

#### NOTE

If you open a SMART\_REFRESH window without asking for the sizing gadget, then Intuition *never* tells you to redraw this window.

#### SUPER\_BITMAP

Setting this flag means you are allocating and maintaining your own bit-map. You must also set the `BitMap` field to point to your own `BitMap` structure. For complete information about SuperBitMap, see "Setting Up a SuperBitMap Window" later in this chapter, and the example "dualpf.c", at the

end of the chapter.

**OTHER\_REFRESH**  
Reserved.

### Special Window Flags

**BACKDROP**  
Set this flag if you want a Backdrop window.

**BORDERLESS**  
This flag creates a window with none of the default border padding and border lines.

#### NOTE

Be careful when you set this flag. It may cause visual confusion on the screen. Also, there may still be some borders if you have selected some of the system gadgets, supplied text for the window's title bar, or specified that any of your custom gadgets go in the borders.

### Message Flags

**REPORTMOUSE**  
This flag sets the window to receive pointer movements as x,y coordinates. Also see the description of the IDCMP flag, MOUSEMOVE, in the chapter entitled "Input and Output Methods."

**ACTIVATE**  
When this flag is set, the window automatically becomes active when it is opened.

#### NOTE

Use this flag carefully. It can change where the user's input is going.

**NOCAREREFRESH**  
Set this flag if you do not want to receive messages telling you to refresh your window.

**RMBTRAP**  
Set this flag if you do not want any menu operations at all for your window. Whenever the user presses the right mouse button while this window is active, the program will receive normal MOUSEBUTTON events.

This flag may be modified on-the-fly by your program. The recommended way to set or clear this flag is as an atomic operation. Caution: Intuition can preempt a multistep set or clear operation, i.e. read contents of address into register, perform bit operation on register, and write register out to address. This can cause Intuition to become confused. An atomic operation could be done in assembler, using 68000 instructions that operate directly on memory, or it could be done by locking out Intuition with a **Forbid()/Permit()** pair. Here, you would call **Forbid()**, do the operation on **Window.Flags**, and then call **Permit()**.

### IDCMPFlags

The IDCMPFlags are listed and described in *The Amiga ROM Kernel Reference Manual: Includes and Autodocs* for the **OpenWindow()** function and in the chapter in this manual entitled "Input and Output

Methods.” If any of these flags are set, Intuition creates a pair of message ports and uses them selectively for sending input to the task opening this window instead of using the console device.

**FirstGadget**

This is a pointer to the first in the linked list of custom **Gadget** structures that you want included in the window.

**CheckMark**

This is a pointer to an instance of a custom image to be used when menu items selected by the user are to be checkmarked. If you just want to use the default checkmark (✓), set this field to **NULL**.

**Title**

This is a pointer to a null-terminated text string, which becomes the window title and is displayed in the window title bar. Intuition draws the text using the colors in the **DetailPen** and **BlockPen** fields and displays as much as possible of the window title, depending upon the current width of the title bar. You get the screen’s default font.

**NOTE**

The window title is not an instance of **IntuiText**; it is simply a string ending in a **NULL**.

**Type**

This contains the screen type for this window. The currently available types are **WBENCHSCREEN** and **CUSTOMSCREEN**.

**IMPORTANT**

If you choose **CUSTOMSCREEN**, you must have already opened your custom screen via a call to **OpenScreen()**, and you must copy that pointer into the **Screen** field (see next entry), before you open your window.

**Screen**

If your **type** is **WBENCHSCREEN**, then this argument is ignored. If **Type** is **CUSTOMSCREEN**, point this to your custom screen structure.

**BitMap**

If you specify **SUPER\_BITMAP** as the refresh type, this flag must be a pointer to your own **BitMap** structure. If you specify some other refresh type, Intuition ignores this field.

The following four variables are used to set the minimum and maximum size to which you allow the user to size the window. If you do not set the flag **WINDOWSIZING**, then these variables are ignored by Intuition.

If you set any of these variables to 0, that means you want to use the initial setting for that dimension. For example, if **MinWidth** is 0, Intuition gives this variable the same value as the opening **Width** of the window.

**NOTE**

To change the limits after the window is opened, call **WindowLimits()**.

**MinWidth**

The minimum width for window sizing, in pixels.

**MinHeight**

The minimum height for window sizing, in lines.

**MaxWidth**

The maximum width for window sizing, in pixels. Use (~0) to allow a window as wide as the screen.

**MaxHeight**

The maximum height for window sizing, in lines. Use (~0) to allow a window as high as the screen.

**WINDOW STRUCTURE**

If you have successfully opened a window by calling the **OpenWindow()** function, you receive a pointer to a **Window** structure. This section describes some of the more useful variables of the **Window** structure. A complete description of the **Window** structure is given in *The Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

**LeftEdge, TopEdge, Width and Height**

As the user moves and sizes your window, these variables will change to reflect the new parameters.

**MouseX, MouseY, GZZMouseX, GZZMouseY**

These variables always reflect the current position of the Intuition pointer, whether or not your window is currently the active one. The **GZZMouse** variables reflect the position of the pointer relative to the inner window of Gimmezerozero windows and the offset into normal windows after taking the borders into account.

**ReqCount**

You can examine this variable to discover how many requesters are currently displayed in the window.

**WScreen**

This variable points to the data structure for this window's screen. If you have opened this window in a custom screen of your own making, you should already know the address of the screen. However, if you have opened this window on the Workbench screen, this variable will point you to that screen's data structure.

**RPort**

This variable is a pointer to this window's **RastPort**. You may need the address of the **RastPort** when using the graphics, text, and animation functions.

**BorderLeft, BorderTop, BorderRight, BorderBottom**

These variables describe the current size of the respective borders that surround the window.

**BorderRPort**

With Gimmezerozero windows, this variable points to the **RastPort** for the outer window, in which the border gadgets are kept.

**UserData**

This is a memory location that is reserved for your use. You can attach your own block of data to the window structure by setting this variable to point to your data.

**WINDOW FUNCTIONS**

Here's a quick rundown of Intuition functions that affect windows. For a complete description of these functions, see *The Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

## Opening the Window

Use the following function to open a window:

- **OpenWindow(NewWindow)**

**NewWindow** is a pointer to a **NewWindow** structure. If successful, a pointer to a **Window** structure is returned. This pointer is required by many of the other functions listed below.

## Activating a Window

Use the following function to activate windows:

- **ActivateWindow(Window)**

**Window** is a pointer to a **Window** structure. This function call may have its action deferred. Don't assume that the selected window has become active just because you called this function for it. You can detect when this window has become active by using the **ACTIVEWINDOW IDCMP** message. We suggest that you use this function only in response to some user action. An example of a system program using this function on a (long, narrow) window along with the related function **ActivateGadget()**, is the Workbench **Rename** function.

## Menus

Use the following functions to attach and remove menus:

- **SetMenuStrip(Window, Menu)**

This function attaches menus to a window.

- **ClearMenuStrip(Window)**

This function removes the menu strip from a window. After this is done, the user can no longer access menus for this window. If you have called **SetMenuStrip()**, you should call **ClearMenuStrip()** before closing your window.

See the chapter "Menus," for complete information about setting up your menus.

## Changing Pointer Position Reports

Although you decide when opening the window whether or not you want messages about pointer position, you can change this later with the following function:

- **ReportMouse(Window, Boolean)**

- **ReportMouse(Boolean, Window)**

This function determines whether or not mouse movements in this window are reported. While most code will use the first form of the call, some compilers require the second. Consult your compiler manual for the correct calling sequence. From assembler, the interface is always the same: Boolean in D0, Window in A0. It is still endorsed to simply set, or reset, the REPORTMOUSE flag bit in Window->Flags on your own, in an atomic way, as explained for RMBTRAP, above.

### **Closing the Window**

After the user selects the close gadget, the program can do whatever it needs to do to clean up and then actually close the window with the **CloseWindow (Window)** function. This function closes a window.

### **Requesters in the Window**

The following two functions allow requesters to become active:

- **Request (Requester, Window)**

This function activates a requester in the window.

- **SetDMRequest (Window, Requester)**

This function sets up a requester that the user can bring up in the window by clicking the menu button twice.

These two functions disable requesters:

- **EndRequest (Requester, Window)**

This function removes a requester from the window.

- **ClearDMRequest (Window, Requester)**

This function clears the double-click requester, so that the user can no longer access it.

### **Custom Pointers**

The following functions apply if you have a custom pointer:

- **SetPointer (Window, Pointer, Height, Width, Xoffset, Yoffset)**

This function sets up the window with a sprite definition for a custom pointer. If the window is active, the change takes place immediately.

- **ClearPointer (Window)**

This function clears the sprite definition from the window and resets to the default Intuition pointer.

## Changing the Size Limits

The following function changes the limits for window sizing:

- **WindowLimits (Window, MinWidth, MinHeight, MaxWidth, MaxHeight)**

This function changes the maximum and minimum sizing of the window from the initial dimensions in the **NewWindow** structure. If you do not want to change a dimension, set the corresponding argument to 0. Out-of-range numbers are ignored. If the user is currently sizing the window, new limits take effect after the user releases the select button.

## Changing the Window or Screen Title

The following function changes the window title after the window has already been displayed:

- **SetWindowTitles (Window, WindowTitle, ScreenTitle)**

This function changes the window title (and screen title, if this is the active window) immediately. **WindowTitle** or **ScreenTitle** can be -1, 0, or a null-terminated string:

- 1        Do not change this title.
- 0        Leave a blank title bar
- string    Change to the title given in this string.

## Refresh Procedures

The following functions allow you to refresh your window in an optimized way:

- **BeginRefresh (Window)**

This function initializes Intuition and layer library internal states for optimized refresh. After you call this procedure, you may redraw your entire window. Only those portions that need to be refreshed will actually be redrawn; the other drawing commands will be discarded.

- **EndRefresh (Window, boolean)**

After you've refreshed your window, call **EndRefresh()** to restore the internal states of Intuition and the layer library. The boolean value determines whether you are completely finished with refreshing or not. If you set it to **FALSE**, you may perform further refreshing between subsequent **BeginRefresh()/EndRefresh()** pairs. You should set the boolean to **TRUE** for the last call to **EndRefresh()**.

- **RefreshWindowFrame(Window)** Refreshes the border of a window, including the title region and the gadgets. It's provided in case your program has been trashing borders and you want to clean up.

## Program Control of Window Arrangement

These functions allow you to modify the arrangement of your window as if the user were activating the associated system window gadgets. These four are among the Intuition functions that are asynchronous. The window will not be affected by them immediately; rather, Intuition will act on it the next time Intuition receives an input event, which happens currently at a minimum rate of ten times per second, and a maximum of sixty times per second. Remember that the actions you call for with these commands may not occur immediately. In some cases, there are IDCMP messages you can request and wait for, which will let you know when the change has occurred (for example NEWSIZE).

- **MoveWindow (Window, DeltaX, DeltaY)**

This function allows you to move the window to a new position in the screen.

- **SizeWindow (Window, DeltaX, DeltaY)**

You can change the size of your window with a call to this procedure.

- **WindowToFront (Window)**

This function causes your window to move in front of all other windows in this screen.

- **WindowToBack (Window)**

This function causes your window to move behind all other windows in this screen.

## SETTING UP A SUPERBITMAP WINDOW

For a SuperBitMap window, you need to set up your own bit-map, since you will not be using the screen's display memory. To set up the bit-map, you need to create a BitMap structure and allocate memory space for it.

The general-purpose graphics function **InitBitMap()** prepares a **BitMap** structure, which describes how a linear memory area is organized as a series of one or more rectangular bit-planes. Here is the specification for this function:

- **InitBitMap (bitmap, depth, bitwidth, bitheight)**

**bitmap**

This is a pointer to the **BitMap** structure to be initialized.

**depth**

This specifies the number of bit-planes to set up.

**bitwidth**

This specifies how wide each bit-plane should be, in bits. Should be a multiple of 16.

**bitheight**

This specifies how high each bit-plane should be, in lines.



The general-purpose graphics function **AllocRaster()** allocates the memory space for the **BitMap**. Here is the specification for this function:

- **AllocRaster (width, height)**

The arguments **width** and **height** are the maximum dimensions of the array in bits. An example of the use of **AllocRaster()** appears in the example "dualpf.c", in the previous chapter. One of the examples below demonstrates a SuperBitMap window.

## SETTING UP A CUSTOM POINTER

Follow these procedures to replace the default pointer with your own custom pointer:

1. Create a sprite data structure. (This is explained below.)
2. Call **SetPointer()**. If your window is active, the new pointer will be attached to the window.

An extra requirement is imposed on sprite data (and Image data). It must be located in chip memory, which is memory that can be accessed by the special Amiga hardware chips. Expansion memory cannot be addressed by the custom chips.

To write a program that will survive in multiple configurations of Amiga hardware, you must ensure that your sprite and Image data reside in this chip memory. You can make sure that your data is in chip memory by using the tools or flags provided by your compiler for this purpose. If none are provided, see the last paragraph in the "Other Features" chapter for another method.

To allocate chip memory, call the Exec function **AllocMem()** with **MEMF\_CHIP** as the requirements argument. See the chapter entitled "Exec: Memory Allocation", for more information.

### The Sprite Data Structure

A sprite data structure is made up of words of data. In a pointer sprite, the first two words and the last two words are reserved for the system. These should be set to 0's. All the other words represent the sprite image.

The example X-shaped custom pointer is nine lines high and two bit planes deep. So, the sprite image consists of 18 words (2 planes x 9 lines = 18 words). If we add the four words reserved for the system we get the following definition:

```
#define XPTR_WIDTH      9
#define XPTR_HEIGHT    9
#define XPTR_XOFFSET   -4
#define XPTR_YOFFSET   -4

USHORT XPointer[]=
{
    0x0000,0x0000,      /* Position and control words */
    0xC180,0x4100,      /* 1st line of the sprite image */
    0x6380,0xA280,      /* 2nd line of the sprite image */
    0x3700,0x5500,
    0x1600,0x2200,
    0x0000,0x0000,
    0x1600,0x2200,
    0x2300,0x5500,
```

```

0x4180,0xA280,
0x8080,0x4100, /* 9th line of the sprite image */

0x0000,0x0000 /* Reserved for system */
};

```

The first two words of image data, 0xC180 and 0x4100, represent the top line of the sprite. To find out what colors will appear on the top line of the sprite, take a bit from 0xC180 and the corresponding bit from 0x4100. This will give a 2-bit number from 0-3 representing the color register for the given pixel. For instance, the top left pixel of our example pointer gets color info from color register 01.

	LSB	MSB
First two lines of sprite data in hex:	0xC180	0x4100
In binary:	1100...	0100...
Color register used:	01 11 00 00....	

#### NOTE

The first word in a line gives the least significant bit of the color register and the second word gives the most significant bit. As you can see, sprites get their color information from the color registers much like screens do.

This example sprite creates an Intuition pointer that looks like the one shown in the figure.

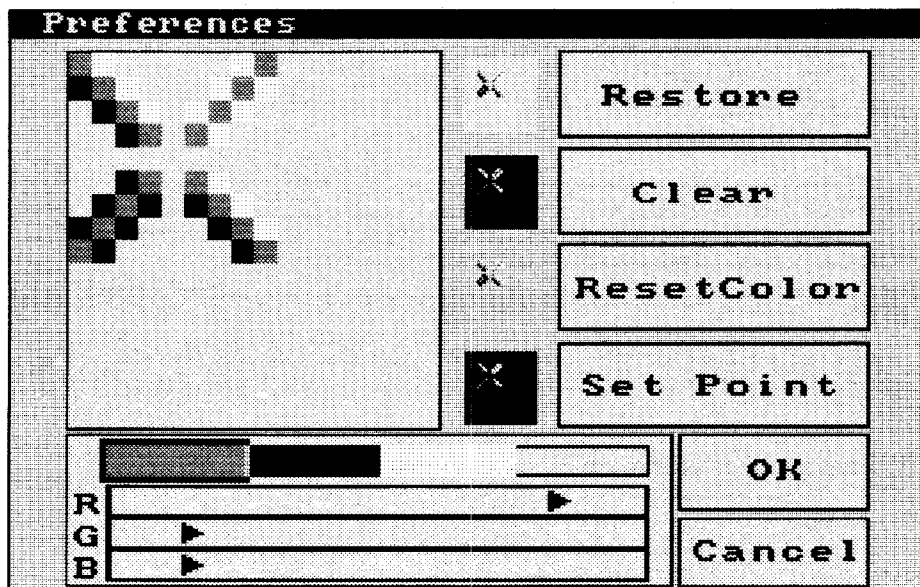


Figure 3-6: The X-Shaped Custom Pointer

#### Attaching the Pointer to the Window

You call `SetPointer()` with the following arguments:

**Window**

This is a pointer to the window that is to receive this pointer definition.

**Pointer**

This is a pointer to the data definition of a sprite.

**Height**

This specifies the height of the pointer; it can be as tall as you like.

**Width**

This specifies the width of the sprite (must be less than or equal to 16).

**XOffset, YOffset**

The **XOffset** and **YOffset** are used to offset the top-left corner of the hardware sprite imagery from what Intuition regards as the current position of the pointer. Another way of describing it is as the offset from the “hot spot” of the pointer to the top-left corner of the sprite. For instance, if you specify offsets of zero, zero, then the top-left corner of your sprite image will be placed at the pointer position. On the other hand, if you specify an **XOffset** of -7 (remember, sprites are 16 pixels wide) then your sprite will be centered over the pointer position. If you specify an **XOffset** of -15, the right edge of the sprite will be over the pointer position.

**NOTE**

For compatibility, you must tell Intuition that the “hot spot” of the pointer sprite is one pixel to the left of the position you actually intend. Changes to the pointer done by your program must compensate for this. The Preferences pointer editor correctly handles this situation.

## Examples

### BACKDROP WINDOW EXAMPLE

This program opens a borderless backdrop window, and writes a greeting to the location normally occupied by the screen title bar. After a pause of about three seconds, it goes away.

```
/* agateWindow.h -- This file implements a borderless backdrop */
/* window. */

#define AGAT_LEFTEDGE 0
#define AGAT_TOPEDGE 0
#define AGAT_WIDTH -1 /* Width and height are supposed to be the */
#define AGAT_HEIGHT -1 /* same as the screen width and height. */

struct NewWindow agateWindow =
{
    AGAT_LEFTEDGE,
    AGAT_TOPEDGE,
    640,
    200,
    0,1, /* Plain vanilla DetailPen and BlockPen. */
    NULL, /* IDCMP Flags can be added later. */
    SMART_REFRESH | ACTIVATE | NOCAREREFRESH | BORDERLESS
    | BACKDROP,
    NULL, /* Pointer to the first gadget -- */
}
```

```

        NULL,      /* may be initialized later. */
        NULL,      /* No checkmark. */
        NULL,      /* No title. */
        NULL,      /* Attach a screen later. */
        NULL,      /* No BitMap. */
        AGAT_WIDTH, /* Minimum width. */
        AGAT_HEIGHT, /* Minimum height. */
        AGAT_WIDTH, /* Maximum width. */
        AGAT_HEIGHT, /* Maximum height. */
        CUSTOMSCREEN /* A screen of our own. */
    };
/* End of agateWindow.h */

/* hellotext.h */

struct IntuiText hello =
{
    1,      /* Use color register 1 (BlockPen) for the FrontPen */
    2,      /* Color register 2, but not used in JAM1 mode. */
    JAM1,   /* Use the background color */
    0,      /* As far to the left as possible. */
    NULL,   /* I want to use the font height -- */
           /* postpone this till later. */
    NULL,   /* Font to use: the default. */
    "Hello, World! ", /* The text */
    NULL    /* No more IntuiText */
};
/* End of hellotext.h */

/* borderless.c - Opens a borderless backdrop window, writes a message. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif

/* Include user-written header files here. */
#include "hires.h"
#include "agateWindow.h"
#include "hellotext.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33
#define GRAPHICS_REV 33

/* TICKS_PER_SECOND is defined in libraries/dos.h
   NEVER call Delay() with an argument of 0 !
*/
#define PAUSE(seconds)    (Delay((seconds) * TICKS_PER_SECOND))

extern VOID cleanExit( struct Screen *, struct Window *, int );

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

/* The next two declarations are for redirecting system requesters. */
struct Process *myProcess = NULL;
APTR oldwindowptr = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    SHORT i;
    SHORT txWidth;
    struct Screen *screen1 = NULL;

```

```

struct Window *window1 = NULL;

/* Open the Intuition Library */
IntuitionBase = (struct IntuitionBase *)
    OpenLibrary( "intuition.library", INTUITION_REV );

if (IntuitionBase == NULL)
    cleanExit( screen1, window1, RETURN_WARN );

/* Open any other required libraries */
GfxBase = (struct GfxBase *)
    OpenLibrary( "graphics.library", GRAPHICS_REV );

if (GfxBase == NULL)
    cleanExit( screen1, window1, RETURN_WARN );

/* Open the screen */
screen1 = OpenScreen( &fullHires );
if (screen1 == NULL)
    cleanExit( screen1, window1, RETURN_WARN );

/* Attach the window to the open screen ... */
agateWindow.Screen = screen1;

/* Conceal the screen title */
ShowTitle( screen1, (BOOL)FALSE );

/* ... and open the window */
window1 = OpenWindow( &agateWindow );
if (window1 == NULL)
    cleanExit( screen1, window1, RETURN_WARN );

/* Now is the time to redirect system requesters. */

myProcess = (struct Process *)FindTask( NULL ); /* Finds our process */
oldwindowptr = myProcess->pr_WindowPtr;
myProcess->pr_WindowPtr = (APTR)window1;

/* Write at the top edge of the window */
/* Get the text width to space the greeting properly. */

txWidth = IntuiTextLength( &hello );

/* Print text five times, one pixel down from the top. */
for ( i = 0; i < 5; i++ )
    PrintIText( window1->RPort, &hello, i*txWidth, 1 );

PAUSE( 6 );

/* Set up the signals that you want to hear about ... */
/* No signals in this program */

/* Exit the program */

cleanExit( screen1, window1, RETURN_OK );

}

VOID cleanExit( scrn, wind, returnValue )
struct Screen *scrn;
struct Window *wind;
int returnValue;
{

```

```

/* Close things in the reverse order of opening */

/* Restore the old window pointer to our process, */
/* and close the window and the screen.          */

if (oldwindowptr)
    myProcess->pr_WindowPtr = oldwindowptr;

if (wind) CloseWindow( wind );
if (scrn) CloseScreen( scrn );

/* Close the library, and then exit */
if (GfxBase) CloseLibrary( (struct Library *)GfxBase );
if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

exit(returnValue);
}

```

## TWO WINDOW EXAMPLE

The following program, `twowindows.c`, opens two windows, and writes messages into them with `IntuiText`. It shows the function of the `ACTIVATE` flag, and illustrates how to get messages from two or more windows. It also shows how to redirect system requesters, that were initiated by your program, to your custom screen.

```

/* hellogoodbye.h */

struct IntuiText hello =
{
    1,          /* Use color register 1 (BlockPen) for the FrontPen */
    2,          /* Color register 2, but not used in JAM1 mode.    */
    JAM1,       /* Use the background color */
    0,          /* As far to the left as possible. */
    NULL,       /* I want to use the font height -- */
               /* postpone this till later.      */
    NULL,       /* Font to use: the default.      */
    "Hello, World! ", /* The text */
    NULL        /* No more IntuiText */
};

struct IntuiText other[3] =
{
    { 1, 3, JAM2, 0, 0, NULL, "You clicked in the", NULL },
    { 0, 2, JAM2, 8, 0, NULL, "other window!", NULL },
    { 3, 2, JAM2, 40, 0, NULL, "GOODBYE!", NULL }
};

/* End of hellogoodbye.h */

/* twowindows.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif

/* Include user-written header files here. For illustration, we show */
/* two header files which we will use frequently.                    */
#include "hires.h"
#include "graniteWindow.h"
#include "hellogoodbye.h"

```

```

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33
#define GRAPHICS_REV 33

/* TICKS_PER_SECOND (defined in libraries/dos.h)
   NEVER call Delay() with an argument of 0 !
*/
#define PAUSE(seconds)      (Delay((seconds) * TICKS_PER_SECOND))

extern VOID cleanExit( struct Screen *, struct Window *, struct Window *,
                      int returnValue);
extern UBYTE handleIDCMP( struct Window * );

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
/* The following two lines are for system requester redirection. */
APTR oldwindowptr = NULL;
struct Process *myProcess = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */

    ULONG aSignalmask, bSignalmask, signals;
    USHORT aDone = FALSE, bDone = FALSE, i, fontHeight;
    struct Screen *screen1 = NULL;
    struct Window *aWindow = NULL, *bWindow = NULL;

    /* Open the Intuition Library */

    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit( screen1, aWindow, bWindow, RETURN_WARN);

    /* Open any other required libraries */

    GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library", GRAPHICS_REV);

    if ( GfxBase == NULL)
        cleanExit( screen1, aWindow, bWindow, RETURN_WARN);

    /* Make the assignments that were postponed above */

    graniteWindow.Width = 300;
    graniteWindow.Height = 100;
    graniteWindow.Title = "aWindow";

    /* Open the screen */

    screen1 = OpenScreen(&fullHires);
    if (screen1 == NULL)
        cleanExit( screen1, aWindow, bWindow, RETURN_WARN);

    /* Attach the window to the open screen ... */

    graniteWindow.Screen = screen1;

    /* ... and open the window */

    aWindow = OpenWindow(&graniteWindow);
    if (aWindow == NULL)
        cleanExit( screen1, aWindow, bWindow, RETURN_WARN);

    /* Now is the time to redirect system requesters. */

```

```

myProcess = (struct Process *)FindTask(NULL); /* Finds our process */
oldwindowptr = myProcess->pr_WindowPtr;
myProcess->pr_WindowPtr = (APTR)aWindow;

/* Now find out how big the font is, and write the greeting */
fontHeight = (SHORT)aWindow->RPort->Font->tf_YSize;
hello.TopEdge = fontHeight;

PrintIText( aWindow->RPort, &hello, 5, (LONG)fontHeight );

PAUSE(3L);

/* The NewWindow structure is now free to be modified for */
/* the other window. */

graniteWindow.LeftEdge = 330;
graniteWindow.Title = "bWindow";

bWindow = OpenWindow(&graniteWindow);
if (bWindow == NULL)
    cleanExit( screen1, aWindow, bWindow, RETURN_WARN);

PrintIText( bWindow->RPort, &hello, 5, (LONG)fontHeight );

/* Now's a good time to finish initializing the IntuiText. */

other[0].NextText = &other[1];
other[1].NextText = &other[2];

/* Fill in the IntuiText vertical offset for the message */
for ( i = 0; i < 3; i ++ )
    other[i].TopEdge = ( i + 1 ) * fontHeight;

/* Set up the signals that you want to hear about ... */

aSignalmask = 1L << aWindow->UserPort->mp_SigBit;
bSignalmask = 1L << bWindow->UserPort->mp_SigBit;

/* Call the functions that do the main processing */

/* And wait to hear from your signals */

while( !aDone || !bDone ) {

    signals = Wait(aSignalmask | bSignalmask);
    if (signals & aSignalmask)
        aDone = handleIDCMP(aWindow);
    if (signals & bSignalmask)
        bDone = handleIDCMP(bWindow);
    if (aWindow && aDone) { /* Close bWindow! */

        PrintIText(bWindow->RPort, &other[0], 5L, 20L);
        PAUSE( 3L );
        CloseWindow(bWindow);
        bWindow = NULL;
        bSignalmask = 0L;
    }
    if (bWindow && bDone) { /* Close aWindow! */

        PrintIText(aWindow->RPort, &other[0], 5L, 50L);
        PAUSE( 3L );

        /* We're about to close the window that our Process
         * is pointing to, so we must switch our Process
         * to the other window, first.
         */
        myProcess->pr_WindowPtr = (APTR)bWindow;

        CloseWindow(aWindow);
    }
}

```



```

        aWindow = NULL;
        aSignalmask = 0L;
    }

    /* If either window has been closed, then the user cannot */
    /* close the remaining window, so we must close it and    */
    /* go away.                                                */

    if ( !aWindow || !bWindow )
        break;
}

/* Exit the program */

PAUSE( 3L );
cleanExit( screen1, aWindow, bWindow, RETURN_WARN);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) ) {

        class = message->Class;
        ReplyMsg( (struct Message *)message);

        switch( class ) {

            case CLOSEWINDOW:

                flag = 1;
                break;

            default:

                break;

        }

    }

    return(flag);
}

VOID cleanExit( scrn, aWind, bWind, returnValue )
struct Screen *scrn;
struct Window *aWind, *bWind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Restore the old window pointer in our process,
    /* and close the window and the screen */
    if (oldwindowptr)
        myProcess->pr_WindowPtr = oldwindowptr;

    if (bWind) CloseWindow( bWind );
    if (aWind) CloseWindow( aWind );

    if (scrn) CloseScreen( scrn );

    /* Close the library, and then exit */
    if (GfxBase) CloseLibrary( (struct Library *)GfxBase );
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}

```

## INVISIBLE POINTER EXAMPLE

The following fragment shows how to make your pointer invisible. It allocates six words, 12 bytes, corresponding to the four leading and trailing control words, together with the two data words, which are both zero.

```
/* For AllocMem() to define new pointer */
#define PDATASZ 12
UWORD  *pdata;

/* Allocate 6 words for blank pointer sprite data in chip ram */
if(pdata = (UWORD *)AllocMem(12, MEMF_CHIP|MEMF_CLEAR))
{
    SetPointer(window1, pdata, 1, 16, 0, 0);
}

/* restore the default pointer image */
ClearPointer(window1);

/* free our data for blank pointer sprite data */
if (pdata)    FreeMem(pdata, PDATASZ);
```

## SUPERBITMAP WINDOW EXAMPLE

This example shows how to implement a superbitmap, and uses a host of Intuition facilities. We suggest that you look over it quickly for now, and come back to it when you have digested all of the other Intuition material.

```
/*
 * Lines.c -- implements a superbitmap with scroll gadgets
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>

#include <proto/all.h>

#define    WIDTH_SUPER    800
#define    HEIGHT_SUPER   600
#define    DEPTH_SUPER    2

extern struct GfxBase *GfxBase;
extern struct IntuitionBase *IntuitionBase;

struct LayersBase *LayersBase;

/* WindowInfo helps keep track of where the line is */
struct WindowInfo
{
    SHORT LineX1;
    SHORT LineY1;
    SHORT LineX2;
    SHORT LineY2;
    SHORT LineX1d;
    SHORT LineY1d;
    SHORT LineX2d;
    SHORT LineY2d;
    SHORT pen;
};

#define GetGadgetID(x) (((struct Gadget *) (msg->IAddress))->GadgetID)
```

```

#define GetLayerXOffset(x) (x->RPort->Layer->Scroll_X)
#define GetLayerYOffset(x) (x->RPort->Layer->Scroll_Y)

#define UP_DOWN_GADGET      0
#define LEFT_RIGHT_GADGET  1
#define NO_GADGET           2

#define MAXVAL 0xFFFFL

struct Image Images[2];

/* The special data needed for the two proportional gadgets */
struct PropInfo GadgetsSInfo[2] =
{
    {FREEVERT|AUTOKNOB,0,0,-1,-1},
    {FREEHORIZ|AUTOKNOB,0,0,-1,-1},
};

/* The usual data needed for any gadget */
struct Gadget Gadgets[2] =
{
    /* Gadgets[0] */
    {&Gadgets[1],-15,10,16,-18,
    GRELRIGHT|GRELHEIGHT,
    RELVERIFY|GADGIMMEDIATE|RIGHTBORDER,PROPGADGET|GZZGADGET,
    (APTR)&Images[0],NULL,NULL,NULL,
    (APTR)&GadgetsSInfo[0],UP_DOWN_GADGET,NULL},

    /* Gadgets[1] */
    {NULL,0,-8,-14,9,
    GRELBOTTOM|GRELWIDTH,
    RELVERIFY|GADGIMMEDIATE|BOTTOMBORDER,PROPGADGET|GZZGADGET,
    (APTR)&Images[1],NULL,NULL,NULL,
    (APTR)&GadgetsSInfo[1],LEFT_RIGHT_GADGET,NULL}
};

static struct NewWindow NewLinesWindow =
{
    150,55,          /* window XY origin relative to TopLeft corner of screen */
    165,94,          /* window width and height */
    0,1,            /* detail and block pens */

    GADGETUP | GADGETDOWN | NEWSIZE | INTUITICKS | CLOSEWINDOW, /* IDCMP flags */

    WINDOWSIZEING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE |
    SUPER_BITMAP | GIMMEZEROZERO | NOCAREREFRESH, /* other window flags */

    Gadgets,          /* first gadget in gadget list */
    NULL,              /* custom CHECKMARK imagery */
    "Lines 2.0",       /* window title */
    NULL,              /* custom screen pointer */
    NULL,              /* custom bitmap */
    90,40,             /* minimum width and height */
    WIDTH_SUPER,HEIGHT_SUPER, /* maximum width and height */
    WBENCHSCREEN        /* destination screen type */
};

ULONG Seed=0x1437289L;
SHORT Rand(SHORT max) /* A simple random number generator */
{
    ULONG tmp;

    tmp=(Seed<<8) + (Seed>>8);
    Seed=tmp;
    return(tmp % max);
}

/* Checks to see if a delta would cause a point to be outside of
 * its range, and adjusts the delta accordingly.
 */
VOID CheckBounce(SHORT point,SHORT *delta,SHORT *pen,SHORT max)
{

```

```

    point+=*delta;
    if (point < 0)
    {
        *delta=Rand(8)+1;
        *pen=(*pen % 3) + 1;
    }
    if (point > max)
    {
        *delta=-(Rand(8)+1);
        *pen=(*pen % 3) + 1;
    }
}

/* This function does all the work of drawing the lines */
VOID Do_DrawStuff(struct Window *window)
{
    struct RastPort *rp;
    struct WindowInfo *myinfo;

    rp=window->RPort;
    myinfo=(struct WindowInfo *) (window->UserData);

    Move(rp,myinfo->LineX1,myinfo->LineY1);
    Draw(rp,myinfo->LineX2,myinfo->LineY2);

    CheckBounce(myinfo->LineX1,&myinfo->LineX1d,&myinfo->pen,WIDTH_SUPER);
    CheckBounce(myinfo->LineY1,&myinfo->LineY1d,&myinfo->pen,HEIGHT_SUPER);
    CheckBounce(myinfo->LineX2,&myinfo->LineX2d,&myinfo->pen,WIDTH_SUPER);
    CheckBounce(myinfo->LineY2,&myinfo->LineY2d,&myinfo->pen,HEIGHT_SUPER);

    SetAPen(rp,myinfo->pen);

    myinfo->LineX1+=myinfo->LineX1d;
    myinfo->LineY1+=myinfo->LineY1d;
    myinfo->LineX2+=myinfo->LineX2d;
    myinfo->LineY2+=myinfo->LineY2d;
}

/* This function provides a simple interface to ScrollLayer */
VOID Slide_BitMap(struct Window *window,SHORT Dx,SHORT Dy)
{
    ScrollLayer(0,window->RPort->Layer,Dx,Dy);
}

VOID Do_NewSize(struct Window *window)
{
    ULONG tmp;

    tmp=GetLayerXOffset(window) + window->GZZWidth;
    if (tmp>=WIDTH_SUPER) Slide_BitMap(window,WIDTH_SUPER-tmp,0);

    NewModifyProp(&Gadgets[LEFT_RIGHT_GADGET],window,NULL,AUTOKNOB|FREEHORIZ,
        ( (GetLayerXOffset(window) * MAXVAL) /
          (WIDTH_SUPER - window->GZZWidth) ),
        NULL,
        ( (window->GZZWidth * MAXVAL) / WIDTH_SUPER ),
        MAXVAL,
        1);

    tmp=GetLayerYOffset(window) + window->GZZHeight;
    if (tmp>=HEIGHT_SUPER) Slide_BitMap(window,0,HEIGHT_SUPER-tmp);

    NewModifyProp(&Gadgets[UP_DOWN_GADGET],window,NULL,AUTOKNOB|FREEVERT,
        NULL,
        ( (GetLayerYOffset(window) * MAXVAL) /
          (HEIGHT_SUPER - window->GZZHeight) ),
        MAXVAL,
        ( (window->GZZHeight * MAXVAL) / HEIGHT_SUPER ),
        1);
}

VOID Check_Gadget(struct Window *window,USHORT gadgetID)

```

```

{
    ULONG tmp;
    SHORT dX=0;
    SHORT dY=0;

    switch (gadgetID)
    {
    case UP_DOWN_GADGET:    tmp=HEIGHT_SUPER - window->GZZHeight;
                           tmp=tmp*GadgetsSInfo[UP_DOWN_GADGET].VertPot;
                           tmp=tmp / MAXVAL;
                           dY=tmp - GetLayerYOffset(window);
                           break;

    case LEFT_RIGHT_GADGET: tmp=WIDTH_SUPER - window->GZZWidth;
                           tmp=tmp*GadgetsSInfo[LEFT_RIGHT_GADGET].HorizPot;
                           tmp=tmp / MAXVAL;
                           dX=tmp - GetLayerXOffset(window);
                           break;

    }
    if (dX || dY) Slide_BitMap(window,dX,dY);
}

VOID Do_MainLoop(struct Window *window)
{
    struct IntuiMessage *msg;
    SHORT flag=TRUE;
    USHORT CurrentGadget=NO_GADGET;

    SetDrMd(window->RPort,JAM1);
    Do_NewSize(window);
    while (flag)
    {
        /* Whenever you want to wait on just one message port */
        /* you can use WaitPort(). WaitPort() doesn't require */
        /* the setting of a signal bit. The only argument it */
        /* requires is the pointer to the window's UserPort */
        WaitPort(window->UserPort);
        while (msg=(struct IntuiMessage *)GetMsg(window->UserPort))
        {
            switch (msg->Class)
            {
            {
            case CLOSEWINDOW: flag=FALSE;
                             break;

            case NEWSIZE:     Do_NewSize(window);
                             break;

            case GADGETDOWN:  CurrentGadget=GetGadgetID(msg);
                             break;

            case GADGETUP:    Check_Gadget(window,CurrentGadget);
                             CurrentGadget=NO_GADGET;
                             break;

            case INTUITICKS:  Check_Gadget(window,CurrentGadget);
                             break;
            }
            ReplyMsg((struct Message *)msg);
        }
        Do_DrawStuff(window);
    }
}

VOID main(VOID)
{
    struct BitMap *BigOne;
    struct Window *window;
    struct WindowInfo MyWindowInfo;
    ULONG RasterSize;
    SHORT Loop;
    SHORT Flag;

    if (IntuitionBase=(struct IntuitionBase *)
        OpenLibrary("intuition.library",33L))
    {
        if (GfxBase=(struct GfxBase *)
            OpenLibrary("graphics.library",33L))
        {

```

```

if (LayersBase=(struct LayersBase *)
    OpenLibrary("layers.library",33L))
{
    if (BigOne=AllocMem(sizeof(struct BitMap),MEMF_PUBLIC|MEMF_CLEAR))
    {
        InitBitMap(BigOne,DEPTH_SUPER,WIDTH_SUPER,HEIGHT_SUPER);
        RasterSize=BigOne->BytesPerRow * BigOne->Rows;
        Flag=TRUE;
        for (Loop=0;Loop<DEPTH_SUPER;Loop++)
        {
            BigOne->Planes[Loop]=AllocMem(RasterSize,
                MEMF_CHIP|MEMF_CLEAR|MEMF_PUBLIC);
            if (!BigOne->Planes[Loop]) Flag=FALSE;
        }
        if (Flag)
        {
            NewLinesWindow.BitMap=BigOne;
            if (window=OpenWindow(&NewLinesWindow))
            {
                window->RPort->Layer->Window=(APTR)window;

                MyWindowInfo.LineX1=0;
                MyWindowInfo.LineY1=0;
                MyWindowInfo.LineX1d=5;
                MyWindowInfo.LineY1d=2;
                MyWindowInfo.LineX2=WIDTH_SUPER >> 2;
                MyWindowInfo.LineY2=HEIGHT_SUPER >> 2;
                MyWindowInfo.LineX2d=2;
                MyWindowInfo.LineY2d=-5;
                MyWindowInfo.pen=3;

                window->UserData=(BYTE *)&MyWindowInfo;

                Do_MainLoop(window);

                CloseWindow(window);
            }
        }
        for (Loop=0;Loop<DEPTH_SUPER;Loop++)
        {
            if (BigOne->Planes[Loop])
            {
                FreeMem(BigOne->Planes[Loop],RasterSize);
            }
        }
        FreeMem(BigOne,sizeof(struct BitMap));
    }
    CloseLibrary((struct Library *)LayersBase);
}
CloseLibrary((struct Library *)GfxBase);
CloseLibrary((struct Library *)IntuitionBase);
}
/* End of lines.c */

```

## **Chapter 4**

### **Intuition: Gadgets**

This chapter describes the workhorses of Intuition—the multipurpose input devices called gadgets. Most of the user's input to an Intuition application can take place through the gadgets in your windows and requesters. Gadgets are also used by Intuition itself for handling screen and window movement and depth arrangement, as well as window sizing and closing.

#### **About Gadgets**

Gadgets can make the user's interaction with your application consistent, easy, and fun. There are two kinds of gadgets: predefined system gadgets and custom application gadgets. The system gadgets help to make the user interface consistent. They are used for dragging and arranging the depth of screens and for dragging, sizing, closing and arranging the depth of windows. Since they always have the same imagery and always reside in the same location, they make it easy for the user to manipulate the windows and screens of any application.

Application gadgets add power and fun to Intuition-based programs. These gadgets can be used in a multitude of ways in your programs. You can design your own gadgets for your windows and requesters.

There are four basic types of application gadgets:

- Boolean gadgets elicit true/false or yes/no kinds of answers from the user.
- Proportional gadgets are flexible devices that you use to get some kind of proportional setting from the user or to simply display proportional information. With the proportional gadget, you can use imagery furnished by Intuition or design any kind of image you want for the slider or knob used to pick a proportional setting.
- String gadgets are used to get text from the user. A number of editing functions are available for users of string gadgets.
- The integer gadget is a special class of string gadget that allows the user to enter integer values only.

Although system gadgets are always in the borders of windows and screens, your own gadgets can go anywhere in windows or requesters and can be any size or shape.

Application gadgets are not supported in screens. Placing a gadget in a backdrop window allows you to receive gadget-related messages through that window's input/output channels. See the chapter "Intuition: Input and Output Methods," for details.

You can choose from the following ways of highlighting gadgets to emphasize that the gadget has been selected:

- Alternate image or alternate border.
- A box around the gadget.
- Color change (by complementing the colors).

You can elect to have your gadgets change in size as the user sizes the window. Also, window gadgets can be located relative to one of the window's borders so that they move with the borders as the user shapes or sizes the window. If you want the gadget in the border, as are the system gadgets, Intuition can adjust the border size accordingly.

Typically, the user selects a gadget by moving the pointer within an area called the select box; you define the dimensions of this area. Next, the user takes some action that varies according to the type of gadget. For a boolean gadget, the user may simply choose an action by clicking the mouse button. For a string or integer gadget, a cursor appears and the user enters some data from the keyboard. For a proportional gadget, the user might either move the knob with the mouse or click the mouse button to move the knob by a set increment.

Although you attach a list of predefined application gadgets when you define a window or requester structure, you can make changes to this list later. You can enable or disable gadgets, add or remove gadgets, modify the internal states of gadgets, and redraw some or all of the gadgets in the list.

When one of your application gadgets is selected by the user, your program learns about it from either the IDCMP or the console device. See the chapter "Intuition: Input and Output Methods," for details about these messages.



## System Gadgets

Intuition automatically attaches system gadgets to every screen. For windows, you specify which system gadgets you want. The system gadgets for screens are for dragging and depth arrangement. The system gadgets for windows are for dragging, depth arrangement, sizing, and closing.

System gadgets have fixed, standard locations in screens and windows, as shown in the following table and figure.

Table 4-1: System Gadget Placement in Windows and Screens

System Gadget	Location
Sizing	Lower right
Dragging	Entire title bar in all areas not used by other gadgets
Depth arrangers	Top right
Close	Top left

Your program need never know that the user selected a system gadget (with the exception of the close gadget); you can let Intuition attach these gadgets to your windows and do the work of responding to the user's wishes.

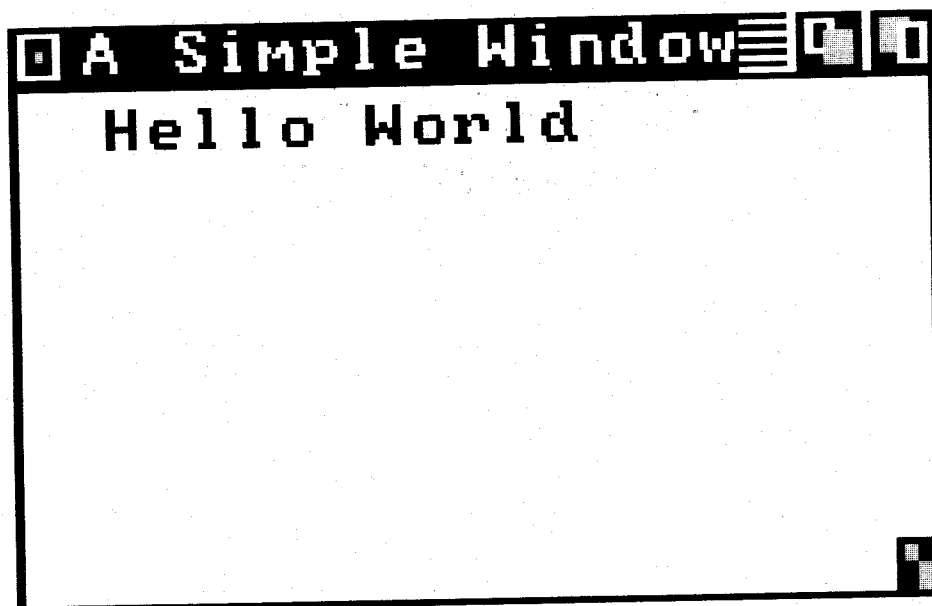


Figure 4-1: System Gadgets in a Low-resolution Window

## **SIZING GADGET**

When the user selects the window-sizing gadget, Intuition is put into a special state. The user is allowed to elongate or shrink a rectangular outline of the window until the user achieves the desired new shape of the window and releases the select button. The window is then reestablished in the new shape, which may involve asking the application to redraw part of its display. For more information about the application's responsibilities in sizing, see the discussion about preserving the display in the chapter on windows.

You attach the sizing gadget to your window by setting the **WINDOWSIZING** flag in the **Flags** variable of the **NewWindow** structure when you open your window. If you are using the **IDCMP** for input, you can elect to receive a message when the user attempts to size the window. A special **IDCMP** flag, **SIZEVERIFY**, allows you to hold off window sizing until you are ready for it. See the chapter "Intuition: Input and Output Methods," for more information about **SIZEVERIFY**.

## **DEPTH-ARRANGEMENT GADGETS**

The depth arrangers come in pairs—one for bringing the window or screen to the front of the display and one for sending the window or screen to the back. Notice that the actual depth arrangement of windows and screens is transparent to your program. The only time you might learn about it even indirectly is when Intuition notifies your program that it needs to refresh its display.

You attach the depth arrangement gadgets to your window by setting the **WINDOWDEPTH** flag in the **Flags** variable of the **NewWindow** structure when you open your window. You get screen depth arrangement gadgets automatically with every screen you open.

## **DRAGGING GADGET**

The dragging gadgets are also known as drag bars because they occupy the entire title bar area that is not taken up by other gadgets. Users can slide screens up and down, much as some classroom blackboards can be moved, to reveal more pertinent information. They can slide windows around on the surface of the screen to arrange the display any way they want.

In dragging a window, the user actually drags a rectangular outline of the window to the new position and releases the select button. The window is then reestablished in its new position. As in window sizing, this may involve asking the application to redraw part of its display. If you want the window drag gadget, set the **WINDOWDRAG** flag in the **Flags** variable of the **NewWindow** structure when you open your window. You get the screen drag gadget automatically with every screen you open.

## **CLOSE GADGET**

The close gadget is a special case among system gadgets, because Intuition notifies your program about the user's intent but doesn't actually close the window. When the user selects the close gadget, Intuition broadcasts a message to your program. It is then up to the program to call **CloseWindow()** when ready. You may want or need to take some actions before the window closes; for instance, you may want to bring up a requester to verify that the user really wants to close that window. To get the window close gadget, set the **WINDOWCLOSE** flag in the **Flags** variable of the **NewWindow** structure when you open your window.

## Application Gadgets

Intuition gadgets imitate real-life gadgets. They are the switches, knobs, controllers, gauges, and keys of the Intuition environment. You can create almost any kind of gadget that you can imagine, and you can have it do just about anything you want it to do. You can create any visual imagery that you like for your gadgets, including combining text with hand-drawn imagery or supplying coordinates for drawing lines.

You can also choose a highlighting method to change the appearance of the gadget after it is selected. All of this flexibility gives you the freedom to create gadgets that mimic real devices, such as light switches or joysticks, as well as the freedom to create devices that satisfy your own unique needs.

### RENDERING GADGETS

You can draw your gadgets by hand, specify a series of lines for a simple line gadget, or have no imagery at all.

#### Hand-drawn Gadgets

Because you are allowed to supply a hand-drawn image, there is no limit to the designs you can create for your gadgets. You can make them simple and elegant or whimsical and outrageous. You design the imagery using one of Amiga's many art tools and then translate your design into an instance of an **Image** structure. The following figure shows an example of a gadget made of hand-drawn imagery. It also shows how you can use an alternate image when the gadget is selected.

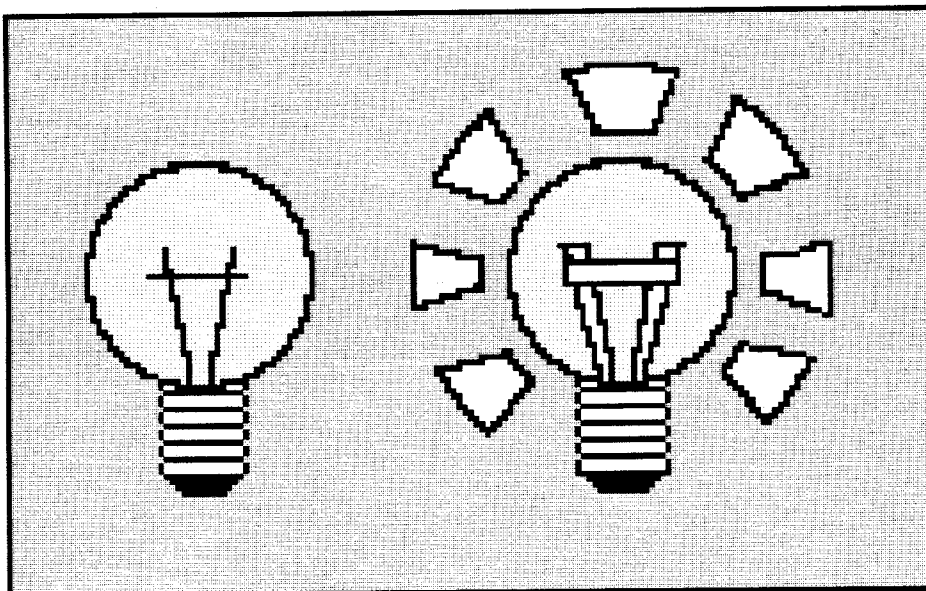


Figure 4-2: Hand-drawn Gadget — Unselected and Selected

You incorporate a hand-drawn image into your gadget by setting the **GADGIMAGE** flag in the gadget variable **Flags** to indicate that this gadget should be rendered as an **Image**. Then you put the address of your **Image** structure into the gadget variable **GadgetRender**. For more information about creating an **Image** structure, see the chapter “Intuition: Images, Line Drawing, and Text.”

### Line-Drawn Gadgets

You can also create simple designs for gadgets by specifying a series of lines to be drawn as the imagery of your gadget. These lines can go around or through the select box of your gadget, and you can specify more than one group of lines, each with its own color and drawing mode.

The following figure shows an example of a gadget that uses line-drawn imagery. It also shows an example of the complement-mode method of highlighting a gadget when it is selected. Furthermore, it shows additional text that has been included in the gadget imagery.

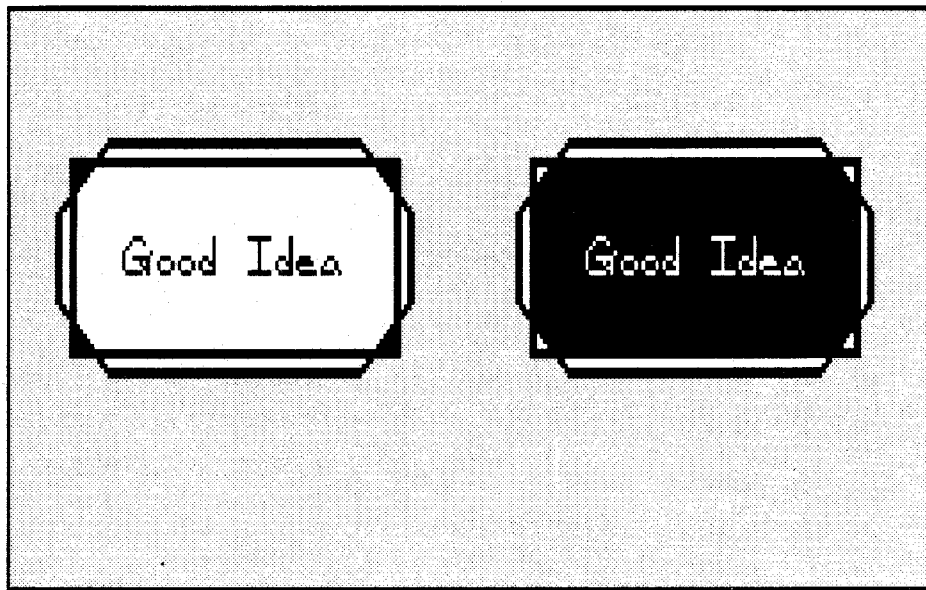


Figure 4-3: Line-drawn Gadget — Unselected and Selected

After deciding on the placement and color of your lines, you create an instance of a **Border** structure to describe your design. You incorporate the **Border** structure of your line-drawn imagery into your gadget by *not* setting the **GADGIMAGE** flag in the gadget’s **Flags** variable, thus specifying that this is a **Border**, not an **Image**. Also, you put the address of your **Border** structure into the gadget variable **GadgetRender**. For more information about creating a **Border** structure, see the chapter “Intuition: Images, Line Drawing, and Text.”

### Gadgets without Imagery

You can also create gadgets that have no imagery at all. For instance, you may want to follow the user’s mouse activity without cluttering the display with unnecessary graphics. An example of such a gadget is the window and screen dragging gadget, which displays no actual imagery. The title bar itself sufficiently implies the imagery of the gadget. You specify no imagery by *not* setting the gadget’s **GADGIMAGE** flag and by setting the **GadgetRender** variable to **NULL**.

## USER SELECTION OF GADGETS

When the user positions the pointer over a gadget and presses the select button, that gadget becomes “selected” and is immediately highlighted. Intuition has two different ways of notifying your program about gadget selection.

If you want the program to find out immediately when the gadget has been selected, you can set the **GADGIMMEDIATE** flag in the **Activation** field of the **Gadget** structure. When the user selects that gadget, an IDCMP event of class **GADGETDOWN** will be received. If you set only this flag, the program will hear nothing more about that gadget until it is selected again.

On the other hand, if you want to be absolutely sure that the user wanted to select the gadget, you can set the **RELVERIFY** flag (for “release verify”). When **RELVERIFY** is set and the user selects the gadget, the program will learn that the gadget was selected only if the user still has the pointer over the select box of the gadget when the select button is released. You may want to know this about some gadget selections whose consequences may be serious—for instance, the window close gadget. If you set the **RELVERIFY** flag, the program will learn about these events via an IDCMP message of the class **GADGETUP**, which you must set in the **IDCMPFlags** field of the **NewWindow** structure. There are two main benefits to **RELVERIFY**: the unsure user gets one last chance to reconsider, and using **RELVERIFY** helps avoid casual errors caused by the user brushing against or resting fingers on the mouse button.

If you want the program to receive both a **GADGETDOWN** and **GADGETUP** message, set both the **GADGIMMEDIATE** and **RELVERIFY** flags.

## GADGET SELECT BOX

To use a gadget, the user begins by moving the pointer into the gadget *select box*. You define the location and dimensions of the select box in the **Gadget** data structure. The location is an offset from one of the corners of the display element (window or requester) that contains the gadget. You place the left and top coordinates in the **LeftEdge** and **TopEdge** fields of the gadget structure.

**LeftEdge** describes a coordinate that is either an absolute offset from the left edge of the element or a negative offset (with an explicit minus sign) from the *current* right edge. The offset method is determined by the **GRELRIGHT** flag. For instance:

- If **GRELRIGHT** is cleared and **LeftEdge** is set to 25, the select box of the gadget starts 25 pixels from the left edge of the display element.
- If **GRELRIGHT** is set and **LeftEdge** is set to -25, the select box of the gadget starts 25 pixels left of the (current) right edge.

In the same way, **TopEdge** is either an absolute offset from the top of the element or a negative offset from the current bottom edge, according to how the flag **GRELBOTTOM** is set:

- If **GRELBOTTOM** is cleared, **TopEdge** is an absolute offset from the top of the element.
- If **GRELBOTTOM** is set, **TopEdge** is a negative offset (with an explicit minus sign) from the current bottom edge.

Similarly, the height and width of the gadget can be absolute or relative to the height and width of the display element in which it resides. If you set the width of a window gadget to -28, for example, and you set the gadget's GRELWIDTH flag, then the gadget's select box will always be 28 pixels less than the width of the window. If GRELWIDTH is not set and you set the width of the gadget to 28, the gadget's select box will always be 28 pixels wide. The GRELHEIGHT flag has the same effect on the height of the gadget select box.

Here are some examples of how you can take advantage of the special relativity modes of the select box.

- Consider the Intuition window sizing gadget. The **LeftEdge** and **TopEdge** of this gadget are both defined relative to the right and bottom edges of the window. No matter how the window is sized, the gadget always appears in the lower right corner.
- In the window-dragging gadget, the **LeftEdge** and **TopEdge** are always absolute in relation to the top left corner of the window. Also, **Height** is always an absolute quantity. **Width** of the gadget, however, is defined to be zero. When **Width** is combined with the effect of the GRELWIDTH flag, the dragging gadget is always as wide as the window.
- Assume that you are designing a program that has several requesters, and each requester has a pair of "OK" and "CANCEL" gadgets in the lower left and lower right corners of the requester. You can design "OK" and "CANCEL" gadgets that can be used in any of the requesters simply by virtue of their positions relative to the lower left and lower right corners of the requester. Regardless of the size of the requesters, these gadgets appear in the same relative positions.

The GRELRIGHT, GRELBOTTOM, GRELWIDTH, and GRELHEIGHT flags belong to the **Flags** field of the **Gadget** structure.

## GADGET POINTER MOVEMENTS

If you set the FOLLOWMOUSE flag for a gadget, you will receive mouse movement broadcasts as long as the gadget is selected. You may want to follow the mouse, for example, in a sound-effects program in which you use the mouse movement to change some quality of the sound.

The broadcasts received differ according to the following flag settings (remember, these examples assume you've set FOLLOWMOUSE):

- If you set the GADGIMMEDIATE *and* RELVERIFY flags, the program learns that the gadget was selected, gets some mouse reports (if the mouse moves), and finds out that the mouse button was released over the gadget.
- If you set only the GADGIMMEDIATE flag, the program learns that the gadget was selected and gets some mouse reports. Then the mouse reports will stop (when the user releases the select button), although the program will have no way of knowing for sure that this has happened.
- If you set only the RELVERIFY flag, the program gets some mysterious, anonymous mouse reports (which may be just what you want to get) followed, perhaps, by a release event for a gadget.
- If you set neither the GADGIMMEDIATE nor the RELVERIFY flag, the program gets only mouse reports. This may be exactly what you want the program to receive.

The FOLLOWMOUSE, GADGIMMEDIATE, and RELVERIFY flags belong to the **Activation** field of the **Gadget** structure.

## GADGETS IN WINDOW BORDERS

You can put your own gadgets in the borders of your window. In the **Gadget** structure, you set one or more of the border flags to tuck your gadget away into the window border. Setting these flags also tells Intuition to adjust the size of the window's borders to accommodate the gadget.

### NOTE

Borders are adjusted only when the window is opened. Although you can add and remove window gadgets after the window is opened, with **AddGadget()/AddGList()** and **RemoveGadget()/RemoveGList()**, Intuition does not readjust the borders.

You can put a given gadget in more than one border by setting more than one border flag. Ordinarily, it makes sense to put a gadget only into two adjoining borders. If you set both side border flags or both the top and bottom border flags for a particular gadget, you get a window that is all border.

The border flags are called **RIGHTBORDER**, **LEFTBORDER**, **TOPBORDER**, and **BOTTOMBORDER**; they belong to the **Activation** field of the gadget structure.

## MUTUAL EXCLUDE

Though mutual exclusion of boolean gadgets is not supported by Intuition, we can recommend the following flexible method of doing it yourself: it is up to your application to handle turning off excluded gadgets *in a way that is friendly to Intuition*. Intuition owns your gadgets and knows how to render them. You must proceed with caution so as not to get yourself or your gadget imagery out of synchronization with Intuition.

## ALLOWABLE TYPE OF GADGETS FOR MUTUAL EXCLUSION

When performing mutual exclusion, you must use hit-select (not **TOGGLESELECT**) boolean gadgets, with the **GADGIMMEDIATE** activation type (not **RELVERIFY**). You must execute your state changes upon receiving the **GADGETDOWN** Intuition message for these gadgets.

## ALLOWABLE TYPES OF HIGHLIGHTING FOR MUTUAL EXCLUSION

If you choose complement mode highlighting for these gadgets (gadget **Flags** of **GADGHCOMP**), you must supply an **Image** that is at least the size of the complemented area (the gadget select area). You may use an extended boolean gadget with a mask, to constrain the area that is highlighted.

You may use an **Image** and an alternate **Image** (gadget **Flags** of **GADGIMAGE** and **GADGHIMAGE**) provided these two images have exactly the same size and position. Likewise, you may use a **Border** and an alternate **Border** (gadget **Flags** of **GADGHIMAGE**), provided the two **Borders** are identical in shape, differing only in color.

You may NOT use other combinations such as a gadget with a **Border** that uses complement mode highlighting, or any gadget which uses **GADGHBOX** (highlighting by drawing a box).

## HANDLING OF MUTUALLY EXCLUSIVE GADGETS

Use **RemoveGList()** to remove a boolean gadget from the window or requester it is attached to. Set or clear the **SELECTED** flag to reflect the state of the gadget you desire to display to the user. Replace the gadget using **AddGList()** and refresh its imagery with **RefreshGList()**. You may of course handle several gadgets with a single call to each of these functions.

## GADGET HIGHLIGHTING

In general, the appearance of a selected gadget changes to tell the user that the gadget has indeed been selected. You select a highlighting method by setting one of the highlighting bits in **Flags**. There are three methods of highlighting after selection: highlighting by color complementing, by drawing a box and by an alternate image or border.

### NOTE

You *must* specify one of the highlighting values. If you do not want any highlighting, set the **GADGHNONE** bit.

### Highlighting by Color Complementing

You can highlight by complementing all of the colors in the gadget's select box. In this context, complementing means the complement of the binary number used to select a particular color register. For example, if the color in color register 2 is used (binary 10) in some of the pixels in the selected gadget, those pixels get changed to whatever color is in color register 1 (binary 01).

Only the select box of the gadget is complemented; any portion of the text, image, or border which is outside of the select box is not disturbed. See the chapter "Intuition: Images, Line Drawing, and Text," for more information about complementing and about color in general.

### Highlighting by Drawing a Box

To highlight by drawing a simple border around the gadget's select box, set the **GADGHBOX** bit in the **Flags** field.

### Highlighting with an Alternate Image or Alternate Border

You can supply alternate **Image** or **Border** imagery as highlighting. When the gadget is selected, the alternate **Image** or **Border** is displayed in place of the non-highlighted **Image** or **Border**, respectively. For this highlighting method, you should set the **SelectRender** field of the **Gadget** structure to point to the **Image** structure or **Border** structure for the alternate display.

An **Image** or **Border** structure contains a set of coordinates that specifies its location when displayed. Intuition renders the image or border relative to the top left corner of the gadget's select box.



In the same way as you set the **GadgetRender** field of the Gadget structure to point to your normal gadget imagery, you should set the **SelectRender** field to point to the alternate **Image** or **Border** of your design. You must also indicate that highlighting is to be done with alternate imagery by setting the **GADGHIMAGE** flag in the **Flags** field of the Gadget structure. If you are using a pair of images, then set **GADGIMAGE**, as well.

For information about how to create an **Image** or **Border** structure, see the chapter “Intuition: Images, Line Drawing, and Text.”

## GADGET ENABLING AND DISABLING

You can disable a gadget so that it cannot be selected by the user. When a gadget is disabled, its image is *ghosted*, and it cannot be selected. Ghosted means that the normal image is overlaid with a pattern of dots, thereby making the image less distinct. Before you first submit your gadget to Intuition, you initialize whether your gadget is disabled by setting or not setting the **GADGDISABLE** flag in the gadget's **Flags** field. If you always want the gadget to be enabled, you can ignore this flag.

After you have submitted a gadget for Intuition to display, you can change its current enable state by calling **OnGadget()** or **OffGadget()**. If it is a requester gadget, the requester must currently be displayed. If you use **OnGadget()** to enable a previously disabled gadget, its image is returned to its normal, nonghosted state.

You may also enable or disable multiple gadgets by removing them, changing the state of the **GADGDISABLED** flag, putting them back, and refreshing them.

## GADGET REFRESHING BY INTUITION

Intuition refreshes gadgets whenever a layer operation has damaged the layer of the window or requester they are attached to. In the processing of the **REFRESHWINDOW** message, the typical program doesn't need to call **RefreshGadgets()**, or **RefreshGList()**, at all.

Intuition's refreshing of the gadgets of a damaged layer is done *through the layer's damage list*. This means that rendering is clipped to the layer's damage region — the part of the window's layer which needs refreshing because it has been exposed by a layer operation.

To be precise, Intuition calls the layers.library functions **BeginUpdate()** and **EndUpdate()**, so that rendering is restricted to the **Region Layer.DamageList**. Your equivalents to these library functions are **BeginRefresh()** and **EndRefresh()**. For more information on **BeginRefresh()** and **EndRefresh()**, see the “Intuition: Windows” chapter, and *The Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

Gadgets which are positioned, using **GRELBOTTOM** or **GRELRIGHT**, or sized, using **GRELWIDTH**, or **GRELHEIGHT**, relative to the dimensions of their window, pose a problem when the window is sized, since the images for these gadgets must change, even though they are not necessarily in the damage region.

Therefore, Intuition must add the original and new visual regions for such relative gadgets to the damage region before it refreshes gadget rendering.

The result of this is that you should ensure that any gadgets with relative position do not have **Border**, **Image**, or **IntuiText** imagery that extends beyond their respective select boxes.

## GADGET REFRESHING BY YOUR PROGRAM

If you add gadgets to your window or requester, using one of the functions **AddGlist()** or **AddGadgets()**, you must subsequently call **RefreshGList()** or **RefreshGadgets()** to get the image of your gadget drawn.

New gadget refreshing functions have been added since the V1.1 release of the system software. These new functions are more efficient than the old set, since the old functions refreshed all the gadgets in the gadget list starting with the specified gadget, while the new functions allow you to specify the number of gadgets to be refreshed, which could be one or more.

The new functions are **RefreshGList()**, which is the alternative to **RefreshGadgets()**, and **NewModifyProp()**, the alternative to **ModifyProp()**. The last two functions of the older set, **OnGadget()** and **OffGadget()**, have no new equivalents, since they can each be implemented by manually modifying the **GADGDISABLED** flag and calling **RefreshGList()**, as described below.

Some programs use **RefreshGadgets()** (or **RefreshGList()**), to update the display after they have made state changes to the gadgets. The types of changes include: the **SELECTED** flag for boolean gadgets to implement mutually exclusive gadgets, the **GadgetText** of some gadget to change its label, the **GADGDISABLED** flag, and the contents of the **StringInfo.Buffer** of a string gadget. When performing these state changes, be sure to **RemoveGadget()**, or **RemoveGList**, any gadget before altering it. Boolean gadgets rendered with borders, instead of images, or highlighted with surrounding boxes (**GADGHBOX**) are handled very simply by Intuition, and complicated transitions done by your program (and in some cases the user's own actions) can get the rendering out of phase.

## BOOLEAN GADGET TYPE

Boolean gadgets are simple **TRUE** or **FALSE** gadgets. You can choose from two methods of selecting such gadgets—*hit select* or *toggle select*:

- Hit select means that when the gadget is hit (that is, when the user moves the pointer into the select box and presses the mouse select button) the gadget becomes selected and the select highlighting method is employed. When the mouse select button is released, the gadget is unselected and unhighlighted.
- Toggle select means that when the gadget is hit, it toggles between selected and unselected. That is, if the user selects the gadget, it remains selected when the user releases the button. To “unselect” the gadget, the user has to repeat the process of hitting the gadget. You can have the imagery reflect the selected/unselected state of the gadget by supplying an alternate image as the highlighting mode of the gadget. When the gadget is selected, the chosen highlighting method is employed.

You need to set the **TOGGLESELECT** flag in the **Activation** field of the **Gadget** structure if you want the gadget to be toggle-selected. The **SELECTED** flag in **Gadget** structure **Flags** determines the initial and current on/off selected state of a toggle-selected gadget. If **SELECTED** is set, the gadget will be highlighted. You can set the **SELECTED** flag before submitting the gadget to Intuition if you like. The program can examine this flag at any time to determine whether the user has selected this gadget.

If a boolean gadget is selected by the user, the application will hear about it. If it is never selected, the application will never know.

## MASKED BOOLEAN GADGETS

The simplest imagery for boolean gadgets is rectangular, but non-rectangular boolean gadgets are possible, with some restrictions. An auxiliary bit plane called a *mask* may be associated with a boolean gadget. When the user clicks within the select box of the gadget, a further test is made to see if the selection point is contained in the mask. Only if it is, does the interaction count as a gadget-hit.

If the gadget has highlight type `GADGHCOMP` then the complement rendering is restricted to the mask, which allows, for example, an oval gadget which highlights nicely, only within the oval.

However, there are some shortcomings to all non-rectangular boolean gadgets. The gadget image is not rendered through the mask. For example, in the case of an oval mask the image is still a rectangle, and when it is displayed, it will clobber the corner areas even though they are outside of the oval. Therefore, gadgets can't be crowded together without care.

Likewise, the ghosting of a disabled gadget does not respect the mask, so ghosting of the corners around an oval may be visible, depending on the colors involved.

To use a masked boolean gadget, you must fill out an instance of the `BoolInfo` structure. The `BoolInfo` structure contains a pointer to the mask plane data. You must also set the `BOOLEXTEND` flag in the gadget's `Activation` field.

## PROPORTIONAL GADGET TYPE

Proportional gadgets are enormously flexible input devices. You can use one of these to get a proportional setting from the user or to display a proportional value to the user. Best of all, you can use the same gadget to accomplish both of these feats.

The user can adjust the setting of a proportional gadget to specify how much of some measurable data or attribute is desired. For instance, the user may adjust a proportional gadget to specify a location in a text file or a desired volume setting. The current setting of a proportional gadget may also be set by the program as an indicator of how much of some measurable data or attribute is visible or available. For instance, the proportional gadget of a text editor's window might show how many lines are currently being displayed out of the total lines in the text file. A graphics program may allow the user to set the amount of red, green, and blue in a color, providing a proportional gadget for each of the three hues. The graphics program would initialize these settings to designate how much red, green, and blue is already contained in the color. An audio program may deal with the volume of the sound being produced by providing a gadget that allows the user to set the volume and to see what the current volume is in relation to the highest and lowest possible volume settings.

Proportional gadgets can do all of these things and more because they can take many shapes and sizes and get fractional settings on either the vertical or horizontal axis or both.

A proportional gadget has several parts that work together to give the gadget its flexibility. They are the *pot* variables, the *body* variables, the *knob*, and the *container*.

- The `HorizPot` and `VertPot` variables contain the actual proportional values. The word *pot* is short for *potentiometer*, which is an electrical analog device that can be used to adjust some variable value. The proportional gadget pots enable the user or program to set how much of the total data is visible or available. Because they represent fractional parts of a whole, the values in these variables ranges from 0 to (almost) 1.

The data, then, ranges from none visible or available to all of it visible or available.

There are two pot variables because proportional gadgets are adjustable on the horizontal axis or the vertical axis or both. For example, a gadget that allows the user to center the screen on the video display or to center his sunsights on a fleeing enemy must be adjustable on both axes.

Pot values change while the user is playing with the gadget. You can initialize the pot variables to whatever you want. In the case of the color gadgets, you might want to initialize them to some current color. The program may read the values in the pots at any time after it has submitted the gadget to the user via Intuition. The values will always have the current settings as adjusted by the user.

- The **HorizBody** and **VertBody** variables describe the increment, or typical step value, by which the pot variables change. For example, the proportional gadgets for color mixing might allow the user to add or subtract a color by 1/16 of the full value each time, as there are 16 possible settings for each RGB (red, green, blue) component of a color on the Amiga. The proportional gadget for centering the screen might allow the user to move the screen vertically a line at a time, or you may choose to set the step increment to a large number of lines, leaving the fine-resolution tuning to the use of the gadget's knob.

Body variables are also used in conjunction with the auto-knob (described below) to display for the user how much of the total quantity of data is directly available. For instance, if the user is working on a text file that is fifteen lines long, and five lines of the file are currently visible in the window, then you can graphically represent the total size of the file by setting the body variable to one-third ( $0xFFFF / 3 = 0x5555$ ). In this case, the auto-knob would fill one-third of the container (the gadget box), which represents the proportion of the visible text lines to the total number of text lines. Also, the user can tell at a glance that clicking the mouse button with the cursor in the container (not on the knob) will advance the text file by one-third in any direction, to the next "window" of data.

You can set the two body variables to the same or different increments. When the user clicks the mouse button in the container, your pot variables are adjusted by the amount set in the body variables.

- The **knob** is the object actually manipulated by the user to change the pot variables by the increments specified in the body variables. The knob is directly analogous to proportional controls, such as the volume knob on a radio, if the Intuition knob is restricted to one axis of movement. If the knob is free to move on both axes, it is more analogous to, say, a control-stick of an airplane. The user can move the knob by placing the pointer on it and dragging it on the vertical or horizontal axis or by moving the pointer near it (within the select box) and clicking the mouse button. With each click, the pot variable is increased or decreased by one increment, defined by the settings of the body variables. The current position of the knob reflects the pot value. For instance, in the color-selection gadget, the knob slides in a long narrow container. As the user moves the knob to the right, more of that color is added. When the knob is halfway along the container, the value in **HorizPot** is also halfway.

You can design your own imagery for the knob or use Intuition's handy *auto-knob*. The auto-knob is a rectangle that can move on either axis and changes its length or height according to the current body settings. The auto-knob is proportional to the size of the gadget. Therefore, you can place an auto-knob in a proportional gadget that adjusts its size relative to the size of a window, and the auto-knob will always be proportionally correct. For example, consider a proportional gadget with auto-knob being used as a scroll bar in the right border of a window. If the **VertBody** variable is set to show that one-third of a text file is being displayed in the window, the auto-knob fills one-third of the container. If the user makes the window (and therefore the container) larger, the auto-knob gets larger, too, so that it still visually represents one-third. This is yet another visual aid for the user, one that helps make the user interface of the Amiga as intuitive to use as possible.

- The **container** is the area in which the knob can move. It is actually the select box of the gadget. The size of the container, like that of any other gadget select box, can be relative to the size of the window.

The pot variable is a 16-bit word that contains a value ranging from 0 to 0xFFFF. For clarity, you may wish to use the constant MAXPOT, which is equivalent to 0xFFFF. This value range represents a fixed-point fraction that ranges from 0 to (almost) 1. You need to convert the current setting of the pot variable to a number that you can use.

There are two general ways in which proportional gadgets are used, namely to scroll through graphical or textual information (such as in a scrolling list or a text editor) or to adjust some level (such as a volume control or a color palette).

The Body and Pot values of a proportional gadget are "Intuition-friendly" numbers, in that they represent concepts convenient to Intuition, and not to your application. Fortunately, it is not too hard to convert the numbers you would like to deal with into Body and Pot values.

The following code fragment illustrates this conversion. You must supply four variables that describe your setup, namely "topLine", "visibleLines", "totalLines", and "Overlap". These will be defined by example:

## SCROLLING THROUGH GRAPHICAL OR TEXTUAL INFORMATION

If a text-editor has a 25-line view of a 100-line document, then "visibleLines" would be 25 and "totalLines" would be 100. If the first visible line was the 10th line of the file, the "topLine" would be 9 (since for topLine, we count from zero). It is a good idea to arrange things so that when the user clicks in the container of a proportional gadget, the view shifts by a bit less than one full view's size (say only 24 lines), creating an overlap between successive views. The extra line is the overlap, hence "Overlap" should be 1.

## ADJUSTING A LEVEL

If a volume control may go from 0 to 49, then choose "totalLines" of 50. If the current volume is 23, then "topLine" is 23. For correct behavior of level controls, always set "visibleLines" to 1 and "Overlap" to zero.

```
/*
** Finding VertPot and VertBody based on "application-friendly"
** parameters.
*/

/* You must supply values for these four: */
UWORD Overlap, totalLines, visibleLines, topLine;

UWORD hidden;

/* Find the number of hidden lines, those that don't fit in the
   visibleLines portion. It turns out to be useful in further
   calculations: */
hidden = MAX(totalLines - visibleLines, 0);

/* If topLine is so great that the remainder of the lines won't even
   fill the displayable area, reduce topLine: */
if (topLine > hidden)
    topLine = hidden;

/* Body is the relative size of the proportional gadget's body.
   Its size in the container represents the fraction of the total
   that is in view. If there are no lines hidden, then Body
   should be full-size (MAXBODY). Otherwise, Body should be the
   fraction of (the number of displayed lines - Overlap) over
   (the total number of lines - Overlap). */

if (hidden > 0)
    VertBody = (UWORD) (((ULONG) (visibleLines - Overlap) * MAXBODY) /
        (totalLines - Overlap));
```

```

else
    VertBody = MAXBODY;

/* Pot is the position of the proportional gadget body, with zero
   meaning that the scroll gadget is all the way up (or left),
   and full (MAXPOT) meaning that the scroll gadget is all the way
   down (or right). If we can see all the lines, Pot should be zero.
   Otherwise, Pot is the top displayed line divided by the number of
   unseen lines. */

if (hidden > 0)
    VertPot = (UWORD) (((ULONG) topLine * MAXPOT) / hidden);
else
    VertPot = 0;
}

```

After the user has adjusted the proportional gadget, you will want to determine the new value of `topLine`. Of course, you should only redraw your display if `topLine` actually changed from its previous value. You do not want to do any rendering if the proportional gadget moved, but not far enough to actually change `topLine`.

```

/*
** Finding the new topLine after the user has adjusted a
** proportional gadget.
*/

/* Again, we need the number of hidden lines */
UWORD hidden;

hidden = MAX(totallines - visibleLines, 0);

/* Pot can be thought of as the fraction of the hidden lines that
   are before the displayed part, in other words a Pot of zero
   means all hidden lines are after the displayed part
   (i.e. topLine = 0), and a Pot of MAXPOT means all
   the hidden lines are ahead of the displayed part
   (i.e. topLine = hidden). */
topLine = (((ULONG) hidden * VertPot) + (MAXPOT/2)) / MAXPOT;

```

You set up a proportional gadget as you do any other gadget, except for the extra **PropInfo** data structure (shown below under "Using Application Gadgets"). Carry out the following procedures to set up the **PropInfo** structure:

- If you want the auto-knob, set the **AUTOKNOB** flag in the **Flags** field. If you want your own knob imagery instead, see below.
- Set either or both of the **FREEHORIZ** and **FREEVERT** flags according to the direction(s) you want the knob to move.
- Initialize either or both of the **HorizPot** and **VertPot** variables to their starting values.
- Set either or both of the **HorizBody** and **VertBody** variables to the increment you want. If there is no data to show or the total amount displayed is less than the area in which to display it, set the body variables to the maximum (0xFFFF, or equivalently, **MAXBODY**).
- The remaining variables and flags are used by Intuition.

In the **Gadget** structure, set the **GadgetType** field to **PROPGADGET** and set the **SpecialInfo** field to point to your **PropInfo** structure (i.e., the one just described).

If you chose to use the auto-knob, set **GadgetRender** to point to an **Image**. In this case, you do not initialize the **Image** structure. You simply declare, for example:

```
struct Image im1;  
<propgadgetname>.GadgetRender = &im1
```

where **<propgadgetname>** is the name of the gadget structure.

To use your own knob imagery, set **GadgetRender** to point to a real, filled-out, **Image** or **Border** structure. If your highlighting will be by an alternate knob image (**GADGHIMAGE**), be sure to make the alternate image the same size as the normal knob image.

To change the flags and the pot and body variables, your program can call **ModifyProp()**, or the better **NewModifyProp()**, after the gadget is displayed. The gadget's internal state will be recalculated and the imagery will be redisplayed to show the new state.

If the program receives a message saying that the user has played with this gadget, the program can examine the **KNOBHIT** flag in the **PropInfo** structure. This flag indicates whether the user hit the knob or hit in the container but not on the knob itself. If the flag is set, the user hit the knob and moved it.

## STRING GADGET TYPE

A string gadget prompts the user to enter some text. Like a proportional gadget, a string gadget can be used in many different ways. String gadgets also require their own special structure, called the **StringInfo** structure.

A string gadget consists of a container and buffers to hold the strings. You supply two buffers for the string gadget. The input buffer contains the initial string, and the other is an optional undo buffer. The string you place in the initial buffer will be displayed, and can be edited by the user. When the user selects a string gadget with the mouse, the gadget's cursor moves to the position of the mouse.

If a string gadget has an undo buffer, the current string is copied into the undo buffer when the user selects the gadget. The user can revert to this initial string at any time by typing "Right-AMIGA-Q." (To type this key sequence, the user holds down the right AMIGA key while pressing the Q-key.) Because there is only one active gadget at a time, all string gadgets can share the same undo buffer as long as the undo buffer is as large as the largest input buffer.

You specify the size of the container into which the user types the string. Like the container for the proportional gadget, the container for the string gadget is its select box. As the user types text into a string gadget, the characters appear in the gadget's container.

You can change the justification of the string as it is displayed in the container. The default is left justification. If the flag **STRINGCENTER** is set, the text is center-justified; if **STRINGRIGHT** is set, the text is right-justified.

An important and useful feature of the string gadget is that you can supply a buffer to contain more text than will fit in the container. This allows the program to get text strings from the user that are much longer than the visible portion of the buffer. Intuition maintains the cursor position and scrolls the text in the container as needed.

You can initialize the input buffer to any starting value, as long as the initial string is terminated with a null. If you want to initialize the buffer to the null string (no characters), you must put a null character in the first position of the buffer. After the gadget is deselected by the user (either by hitting the RETURN key or by using the mouse to select some other operation), the program can examine this buffer to discover the current string.

String gadgets feature *auto-insert*, which allows the user to insert ASCII characters wherever the cursor is. The simple editing functions shown in the following table are available to the user.

Table 4-2: Editing Keys and Their Functions

Key(s)	Function
← or →	Move the cursor around the current string.
SHIFT ← or →	Move the cursor to the beginning or end of current string.
DEL	Delete the character under the cursor.
BACKSPACE	Delete the character to left of cursor.
RETURN	Terminate input and deselect the gadget. If the RELVERIFY activation flag is set, the program will receive a GADGETUP event for this gadget.
Right - AMIGA - Q	Undo (cancel) the last editing change to the string.
Right - AMIGA - X	Clears the input buffer. The undo buffer is left undisturbed.

You can supply any type of image for the rendering of this gadget—**Image**, **Border**, or no image at all. For this release of Intuition, you must specify that the highlighting is of type GADGHCOMP (complementary), and you cannot supply an alternate image for highlighting.

The string gadget inherits the input attributes and the font of the screen in which it appears. If you have not done anything fancy, the strings will appear in the default font with simple ASCII key translations. If you are using the console device for input, you can set up alternate key-mapping any way you like. If you do, Intuition will use your key map. See the “Console Device” chapter for more information about the console device and key-mapping.

For a string gadget, you set the **GadgetType** field to **STRGADGET** in the **Gadget** structure. Also set the **SpecialInfo** field to point to an instance of a **StringInfo** structure, which you must fill in with buffer and container information.

## INTEGER GADGET TYPE

The integer gadget is really a special sort of string gadget. You initialize it as you do a string gadget, except that you also set the flag **LONGINT** in the gadget’s **Activation** field. The user interacts with an integer gadget using exactly the same rules as for a string gadget, but Intuition filters the input and allows the user to enter only a plus or minus sign and digits. The integer gadget returns a signed 32-bit integer in the **StringInfo** variable **LongInt**.

To initialize an integer gadget you need to preset the buffer by putting an initial integer string in it. It is *not* sufficient to initialize an integer gadget by merely setting a value in the **LongInt** variable.



To specify that this string gadget is an integer gadget, set the flag **LONGINT** in the gadget's **Activation** variable. String gadgets of integer type have the **LongInt** value updated whenever the textual contents of the gadget changes, and again, when the gadget is deactivated.

## COMBINING GADGET TYPES

You can make some very useful gadgets by combining gadgets of several types. As an example, you can make a horizontal or vertical scroll bar with a proportional gadget and two boolean gadgets.

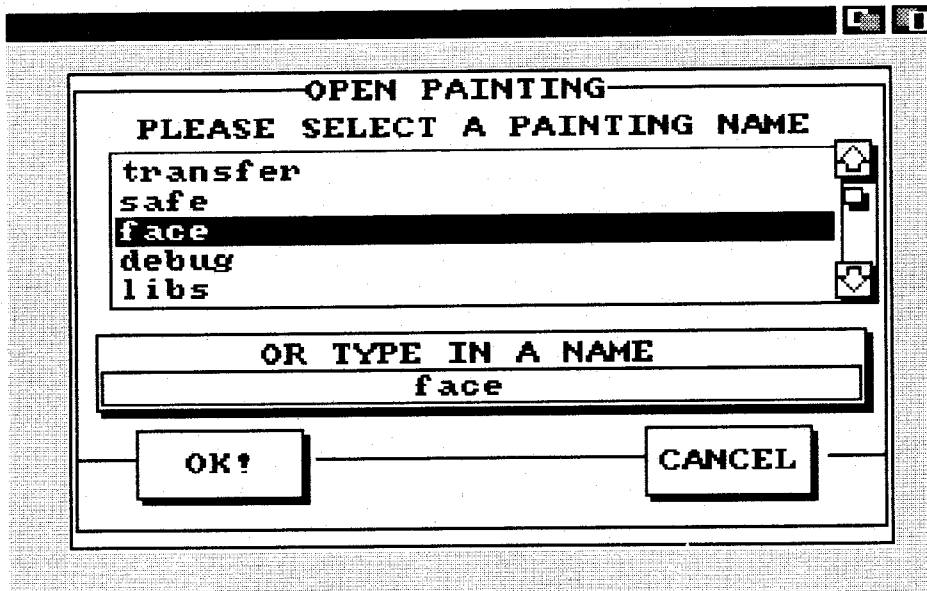


Figure 4-4: Example of Combining Gadget Types

If the scroll bar goes in the right border of the window, you may wish to place the system sizing gadget in the right border by setting the flag **SIZEBRIGHT** in the **NewWindow** structure. Remember that the sizing gadget has to fit in either the right or the bottom border. If you are going to cause the right edge border to be wide enough to accommodate a scroll bar, then you might as well put the sizing gadget there, too.

## Gadget Activation Messages

The Intuition **IntuiMessage** structure has a field named **IAddress**. If the user presses the select button over the select box of a gadget, the **IAddress** field of the **GADGETDOWN** IDCMP message will contain the **Gadget** structure address of the selected gadget. If you have three gadgets in your window, all with the **GADGIMMEDIATE** flag set, and with **GadgetID**'s of 10, 11, and 12, the following code fragment shows the correct way to process a gadget event from one of these gadgets:

```

class = msg->Class;
iaddress = msg->IAddress;
ReplyMsg(msg);
switch (class)
{
    case GADGETDOWN:
        switch (((struct Gadget *)iaddress)->GadgetID)
        {
            case 10:
                /* Perform gadget ten's function */
                break;
            case 11:
                /* Perform gadget eleven's function */
                break;
            case 12:
                /* Perform gadget twelve's function */
                break;
        }
        break;
    case default:
        /* Take default action */
        break;
}

```

## GADGET STRUCTURE

Here is the general specification for a Gadget structure:

```

struct Gadget
{
    struct Gadget *NextGadget;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    USHORT Activation;
    USHORT GadgetType;
    APTR GadgetRender;
    APTR SelectRender;
    struct IntuiText *GadgetText;
    LONG MutualExclude;
    APTR SpecialInfo;
    USHORT GadgetID;
    APTR UserData;
}

```

The variables and flags in the **Gadget** structure are explained below.

### NextGadget

This is a pointer to the next gadget in the list. The last gadget in the list should have a **NextGadget** value of **NULL**.

### LeftEdge, TopEdge, Width, Height

These variables describe the location and dimensions of the select box of the gadget. Both location and dimensions can be either absolute or relative to the edges and size of the window, or requester that contains the gadget.

**LeftEdge** and **TopEdge** are relative to one of the corners of the display element, according to how **GRELRIGHT** and **GRELBOTTOM** are set in the **Flags** variable (see below).

**Width** and **Height** can be either absolute dimensions or a negative increment to the width and height of a requester, or alert or the current width and height of a window, according to how the **GRELWIDTH** and **GRELHEIGHT** flags are set (see below).

## Flags

The **Flags** field is shared by your program and Intuition. See the section below called “Flags” for a complete description of all the flag bits.

## Activation

This field is used for information about some gadget attributes. See the “Activation Flags” section below for a description of the various flags.

## GadgetType

This field contains information about gadget type and in what sort of display element the gadget is to be displayed. You *must* set one of the following flags to specify the type:

### BOOLGADGET

Boolean gadget type.

### STRGADGET

String gadget type.

For an integer gadget, also set the **LONGINT** flag. See the “Flags” section below.

### PROPGADGET

Proportional gadget type.

The following flags tell Intuition if the gadget is for a requester or a Gimmezerozero window:

### GZZGADGET

If this gadget is for a Gimmezerozero window, setting this flag puts the gadget in the special bit-map for gadgets and borders (and out of your inner window). If you do not set this flag, the gadget will go into your inner window. If the destination of this gadget is not a Gimmezerozero window, do not set this bit.

### REQGADGET

Set this bit if this is a requester gadget; otherwise, be sure this bit is clear.

## GadgetRender

This is a pointer to the **Image** or **Border** structure containing the graphics of this gadget. If this field is set to **NULL**, no rendering will be done.

If the graphics of this gadget are to be implemented with an **Image** structure, this field must be made to point to that structure, and the **GADGIMAGE** bit must be set in the **Flags** field. If a **Border** structure is to be used instead, this field must be made to point to that **Border** structure, and the **GADGIMAGE** bit must not be used.

## SelectRender

If you don't want alternate graphics to indicate highlighting, set this field to **NULL**. If you do want alternate graphics to indicate highlighting, set the **GADGHIMAGE** flag in the **Flags** field (see below), and set the **SelectRender** field to point to the **Image** or **Border** structure that implements your alternate image.

## GadgetText

If you want text printed after this gadget is rendered, set this field to point to an **IntuiText** structure. The offsets in the **IntuiText** structure are relative to the top left of the gadget's select box.

Set this field to **NULL** if the gadget has no associated text.

### **MutualExclude**

This field is currently ignored by Intuition, but is reserved. If this is a boolean gadget with **BOOLEXTEND** activation, this variable must point to an instance of a **BoolInfo** data structure.

### **SpecialInfo**

If this is a proportional gadget, this variable must point to an instance of a **PropInfo** data structure. If this is a string or integer gadget, this variable must point to a **StringInfo** data structure. Otherwise, this variable is ignored. The structure contains the special information needed by the gadget. If the gadget is not of type proportional, string, or integer, this variable is ignored.

### **GadgetID**

This variable is strictly for your own use. Assign any value you would like here. This variable is ignored by Intuition. Typical uses in C are in **switch** and **case** statements, and in assembly language, table lookup.

### **UserData**

A pointer to any general data you would care to associate with this particular gadget. This variable is ignored by Intuition.

## **FLAGS**

The following are the flags you can set in the **Flags** variable of the **Gadget** structure.

### **GADGHIGHBITS**

Combinations of these bits describe what type of highlighting you want when the user has selected this gadget. There are four highlighting methods to choose from. You **must** set one of the four flags below.

### **GADGHCOMP**

This flag selects highlighting by complementing all of the bits contained within this gadget's select box.

### **GADGHBOX**

This flag selects highlighting by drawing a box around this gadget's select box.

### **GADGHIMAGE**

If you intend to indicate highlighting with alternate graphics, set this flag.

### **GADGHNONE**

Set this flag if you want no highlighting.

### **GADGIMAGE**

If your gadget has a graphic, and it is implemented with an **Image** structure, set this bit. If the graphic is implemented with a **Border** structure, make sure this bit is clear. This bit is also used by **SelectRender**.

### **GRELBOTTOM**

Set this flag if the gadget's **TopEdge** variable describes an offset relative to the bottom of the display element (window or requester) containing it. Clear this flag if **TopEdge** is relative to the top.

### **GRELRIGHT**

Set this flag if the gadget's **LeftEdge** variable describes an offset relative to the right edge of the display element containing it. Clear this flag if **LeftEdge** is relative to the left edge.

## GRELWIDTH

Set this flag for “relative gadget width” if you want your gadget’s width to change automatically whenever the width of its window changes. When this flag is set, you set the gadget’s **Width** field to a negative value (including minus sign). This value is added to the width of the gadget’s display element, to determine the actual width of the gadget. Do not set this flag if **Width** is an absolute value.

## GRELHEIGHT

Set this flag for “relative gadget height” if you want your gadget’s height to change automatically whenever the height of its window changes. When this flag is set, you set the gadget’s **Height** field to a negative value (including minus sign). This value is added to the height of the gadget’s display element, to determine the actual height of the gadget. Do not set this flag if **Height** is an absolute value.

## SELECTED

Use this flag to preselect the on/off selected state for a toggle-selected gadget. If the flag is set, the gadget starts off being on and is highlighted. If the flag is clear, the gadget starts off in the unselected state.

## GADGDISABLED

If this flag is set, this gadget is disabled. If you want to enable or disable a gadget later on, you can change the current state with the routines **OnGadget()** and **OffGadget()**, or you may remove one or more gadgets, change the state of this flag, put them back, and refresh them.

You do not need to use this flag if you want the gadget to always remain enabled.

## ACTIVATION FLAGS

Here are the flags you can set in the **Activation** variable of the **Gadget** structure:

### TOGGLESELECT

This flag applies only to a boolean gadget, and tells Intuition that it is to be a toggle-select gadget, not a hit-select one.

You preset the selection state with the gadget **Flag SELECTED** (see above); the program later discovers the selected state by examining **SELECTED**.

### GADGIMMEDIATE

Set this bit if you want the program to know immediately, via a **GADGETDOWN IDCMP** message, when the user selects this gadget.

### RELVERIFY

This is short for “release verify.” Set this bit if you want this gadget selection broadcast to your program only if the user still has the pointer positioned over this gadget when releasing the select button.

### ENDGADGET

This flag pertains only to gadgets attached to requesters. To make a requester go away, the user must select a gadget that has this flag set.

See the chapter “Intuition: Requesters and Alerts,” for more information about requester gadget considerations.

### FOLLOWMOUSE

When the user selects a gadget that has this flag set, the program will receive mouse position broadcasts every time the mouse moves at all.

You can use the following flags in window gadgets to adjust the size of a window's borders when you want to tuck your own window gadgets out of the way into the window border:

**RIGHTBORDER**

If this flag is set, the width and position of this gadget are used in deriving the width of the window's right border.

**LEFTBORDER**

If this flag is set, the width and position of this gadget are used in deriving the width of the window's left border.

**TOPBORDER**

If this flag is set, the height and position of this gadget are used in deriving the height of the window's top border. It can also be set to tell Intuition that this gadget must be refreshed after Intuition has rendered in the top border area of a window.

**BOTTOMBORDER**

If this flag is set, the height and position of this gadget are used in deriving the height of the window's bottom border.

The following flags apply to string gadgets:

**STRINGCENTER**

If this flag is set, the text in a string gadget is centered when rendered.

**STRINGRIGHT**

If this flag is set, the text in a string gadget is right-justified when rendered.

**LONGINT**

If this flag is set, the user can construct a 32-bit signed integer value in a normal string gadget. You must also preset the string gadget input buffer by putting an initial integer string in it.

**ALTKEYMAP**

This flag specifies that you have an alternate keymap. You also need to put a pointer to the keymap in the **StringInfo** structure variable **AltKeyMap**.

The following flag applies to Boolean gadgets:

**BOOLEXTEND**

If this flag is set, then this boolean gadget has a **BoolInfo** structure associated with it.

## **SPECIALINFO DATA STRUCTURES**

The following are the specifications for the structure pointed to by the **SpecialInfo** pointer in the **Gadget** structure.

## BoolInfo Structure

This is the special data required for a masked boolean gadget.

```
struct BoolInfo
{
    USHORT Flags;
    UWORD *Mask;
    ULONG Reserved;
};
```

The meanings for the fields in this structure are as follows:

### Flags

Flags must be given the value BOOLMASK.

### Mask

This is a bit mask for highlighting and selecting the gadget. Construct the mask the way you would construct a single plane of Image data. The image's width and height are determined by the width and height of the gadget's select box. The mask data must be in chip memory.

### Reserved

Set this field to NULL.

## PropInfo Structure

This is the special data required by the proportional gadget.

```
struct PropInfo
{
    USHORT Flags;
    USHORT HorizPot;
    USHORT VertPot;
    USHORT HorizBody;
    USHORT VertBody;
    USHORT CWidth;
    USHORT CHeight;
    USHORT HPotRes, VPotRes;
    USHORT LeftBorder;
    USHORT TopBorder;
};
```

The meanings of the fields in this structure are as follows:

### Flags

In the Flags variable, these flag bits can be specified:

#### AUTOKNOB

Set this if you want to use the auto-knob.

#### FREEHORIZ

If this is set, the knob can move horizontally.

**FREEVERT**

If this is set, the knob can move vertically.

**KNOBHIT**

This is set by Intuition when this knob is hit by the user.

**PROPBORDERLESS**

Set this if you want your proportional gadget to appear without a border drawn around its container.

Initialize these variables before the gadget is added to the system; then look here for the current settings:

**HorizPot**

Horizontal quantity fraction.

**VertPot**

Vertical quantity fraction.

These variables describe what fraction of the entire body is actually shown at one time:

**HorizBody**

Horizontal body.

**VertBody**

Vertical body.

Intuition sets and maintains the following variables, which are private to Intuition:

**CWidth**

Container real width.

**CHeight**

Container real height.

**HPotRes, VPotRes**

Pot increments.

**LeftBorder**

Container real left border.

**TopBorder**

Container real top border.

**StringInfo Structure**

This is the special data required by the string gadget.



```

struct StringInfo
{
    UBYTE *Buffer;
    UBYTE *UndoBuffer;
    SHORT BufferPos;
    SHORT MaxChars;
    SHORT DispPos;
    SHORT UndoPos;
    SHORT NumChars;
    SHORT DispCount;
    SHORT CLeft, CTop;
    struct Layer *LayerPtr;
    LONG LongInt;
    struct KeyMap *AltKeyMap;
};

```

The meanings of the fields in this structure are given below.

You initialize the following variables and Intuition maintains them:

#### **Buffer**

This is a pointer to a buffer containing the start and final string. The string you write into this buffer must be null-terminated.

#### **UndoBuffer**

This is an optional pointer to a buffer for undoing the current entry. If you are supplying an undo buffer, the memory location should be as large as the buffer for the start and final string. Because only one string gadget can be active at a time under Intuition, all of your string gadgets can share the same undo buffer. However, the undo buffer must be large enough to hold the largest buffer for starting and final strings.

#### **MaxChars**

This must be set to the maximum number of characters that will fit in the buffer, including the terminating NULL.

#### **BufferPos**

This specifies the initial character position of the cursor in the buffer.

#### **DispPos**

This specifies the buffer position of the first displayed character.

Intuition initializes and maintains these variables for you:

#### **UndoPos**

This specifies the character position in the undo buffer.

#### **NumChars**

This specifies the number of characters currently in the buffer.

#### **DispCount**

This specifies the number of whole characters visible in the container.

#### **CLeft, CTop**

This Intuition-private field specifies the top left offset of the container.

#### **LayerPtr**

This Intuition-private field specifies the layer containing this gadget.

### **LongInt**

After the user has finished entering an integer, you can examine this variable to discover the value if this is an integer string gadget.

### **AltKeyMap**

This variable points to your own alternate keymap; you must also set the ALTKEYMAP bit in the **Activation** flags of the gadget.

## **GADGET FUNCTIONS**

These are brief descriptions of the functions you can use to manipulate gadgets. For complete descriptions see *The Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

### **Adding and Removing Gadgets from Windows or Screens**

Use the following functions to add a gadget to or remove a gadget from the gadget list of a window.

- **AddGadget(AddPtr, Gadget, Position)**

This function adds one gadget to the gadget list of a window.

- **RemoveGadget(RemPtr, Gadget)**

This function removes one gadget from the gadget list of the specified window.

Use the following functions to add a sublist of gadgets to or remove a sublist of gadgets from the gadget list of a window or requester. A sublist may be the entire gadget list. A sublist of gadgets is a collection of gadgets that are linked by the Gadget.NextGadget field.

- **AddGList(Window, Gadget, Position, Numgad, Requester)**

This function adds up to Numgad gadgets, from a sublist beginning with the specified gadget, to a window or requester. You also supply the position in the gadget list where these gadgets should go. Use a position of -1 (ie. (USHORT) ~0) to denote the end of the gadget list.

- **RemoveGList(Window, Gadget, Numgad)**

This function removes up to Numgad gadgets from a window or requester, beginning with the specified one.

### **Disabling or Enabling a Gadget**

The following functions disable or enable a gadget in a window, screen, or requester.

- **OnGadget(Gadget,Ptr,Requester)**

This function enables the specified gadget.

- **OffGadget(Gadget, Ptr, Requester)**

This function disables the specified gadget.

### Redraw the Gadget Display

- **RefreshGList(Gadgets, Window, Requester, NumGad)**

Redraws no more than NumGad gadgets, starting with the specified gadget, in a window or requester. You should refresh any gadgets after you add them. You might want to use this if you have modified the imagery of your gadgets and want to display the new imagery. You might also use it if you think some graphic operation has trashed the imagery of the gadgets.

- **RefreshGadgets(Gadgets, Ptr, Requester)**

Redraws all of the gadgets in the gadget list of a window or requester, starting with the specified gadget. In a requester, *all* of the gadgets are redrawn. This function is superseded by the more flexible **RefreshGList()**.

### Modifying a Proportional Gadget

Use the following functions to modify the current parameters of a proportional gadget.

- **NewModifyProp(Gadget, Window, Requester, Flags, HorizPot, VertPot, HorizBody, VertBody, NumGad)**

This function modifies the parameters of a proportional gadget. The gadget's internal state is recalculated and the imagery is redisplayed.

- **ModifyProp(Gadget, Ptr, Requester, Flags, HorizPot, VertPot, HorizBody, VertBody)**

This is the same as **NewModifyProp()**, except that it refreshes all the gadgets, beginning with the specified one. You will want to use the more flexible **NewModifyProp()**, instead.

### Activating a String Gadget via Program

- **ActivateGadget(Gadget, Window, Request)**

This function allows your program to activate a string gadget. If successful, this function has the same effect as the user clicking the SELECT button when the mouse pointer is within the gadget's select box. Subsequent keystrokes accomplish entry and editing on the gadget's string. This function will fail if the user is in the middle of some other interaction, such as menu or proportional gadget operations.

The window or requester containing the string gadget to be activated must itself be open and active. Since some operations in Intuition may occur after the function that initiates them completes, calling **ActivateGadget()** after **OpenWindow()** or **Request()** is no guarantee that the gadget will actually activate. Instead, you should call **ActivateGadget()** only after having received an **ACTIVEWINDOW** or **REQSET** IDCMP message for a newly opened window or requester, respectively. Of course, you must set the window's IDCMP flags such that you will hear those messages, too.

NOTE: It is incorrect to simply insert a small delay between the call to **OpenWindow()** or **Request()** and the call to **ActivateGadget()**. Such schemes fail under various conditions, including processor speed and CPU loading.

## Example

This example implements string, proportional, and boolean gadgets. The boolean gadget is masked. It also includes examples of **Image**, **Border**, and **IntuiText** structures.

```
/* Gadgets.h -- has all of the structures needed for the gadgets */

struct TextAttr TOPAZ80 =
{
    (STRPTR) "topaz.font", TOPAZ_EIGHTY, 0, 0
};

UBYTE Buffer[512]; /* This is for showing any messages or entries */

struct IntuiText Messages =
{
    1, 0, JAM2, 0, 0, &TOPAZ80, &Buffer[0], NULL
};

struct Image BackImage1 =
{
    0, 0, /* X, Y origin relative to TopLeft of Gadget */
    150, 90, /* Image width and height in pixels */
    0, /* number of bitplanes in Image */
    NULL, /* pointer to ImageData, NULL for Rectangle */
    0x0000, 0x0002, /* PlanePick and PlaneOnOff */
    NULL /* next Image structure */
};

struct Gadget BackDrop = /* This is just for our blank images */
{
    NULL, 5, 1, 1, 1, GADGIMAGE, NULL, BOOLGADGET, (APTR) &BackImage1,
    NULL, NULL, NULL, NULL, 0, NULL
};

USHORT chip MaskData1[] = /* No Border Button Mask Data */
{
    0x07FF, 0xFFFF, 0xFFFF, 0xFFFF, 0xF000, 0x3FFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFE00, 0x7FFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFF00, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFF80, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFF80, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFF80, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFF80, 0x7FFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFF00,
    0x3FFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFE00, 0x07FF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xF000
};

struct Image ButtonI1 =
{
    0, 0, 73, 10, 1, MaskData1, 0x0001, 0x0000, NULL
};

/* Text for our buttons */
struct IntuiText IText1 = {2, 0, JAM1, 14, 1, &TOPAZ80, "Cancel", NULL};
struct IntuiText IText2 = {2, 0, JAM1, 20, 1, &TOPAZ80, "OKAY", NULL};

/* Mask information for gadget */
struct BoolInfo OkayMask = {BOOLMASK, MaskData1, 0};

struct Gadget ButtonGads[] =
{
    {
```

```

    &BackDrop,      100,106, 73,10, GADGHCOMP|GADGIMAGE,
    RELVERIFY|GADGIMMEDIATE|BOOLEXTEND, BOOLGADGET, (APTR)&Button11,
    NULL, &IText1, NULL, (APTR)&OkayMask, 7, NULL
    },
    {
    &ButtonGads[0],  5,106, 73,10, GADGHCOMP|GADGIMAGE,
    RELVERIFY|GADGIMMEDIATE|BOOLEXTEND, BOOLGADGET, (APTR)&Button11,
    NULL, &IText2, NULL, (APTR)&OkayMask, 6, NULL
    }
};

SHORT BorderVectors3[] =
{
    0,0,
    151,0,
    151,91,
    0,91,
    0,0
};

struct Border Border3 =
{
    -1,-1,          /* XY origin relative to TopLeft */
    1,0,JAM1,       /* front pen, back pen and drawmode */
    5,              /* number of XY vectors */
    BorderVectors3, /* pointer to XY vectors */
    NULL           /* next border in list */
};

struct Gadget EntryBox =
{
    &ButtonGads[1], /* next gadget */
    5,1,            /* origin XY of hit box relative to window TopLeft */
    150,90,         /* hit box width and height */
    GADGHBOX|GADGHIMAGE, /* gadget flags */
    RELVERIFY|GADGIMMEDIATE, /* activation flags */
    BOOLGADGET,     /* gadget type flags */
    (APTR)&Border3,  /* gadget border or image to be rendered */
    NULL,           /* alternate imagery for selection */
    NULL,           /* first IntuiText structure */
    NULL,           /* gadget mutual-exclude long word */
    NULL,           /* SpecialInfo structure */
    1,              /* user-definable data */
    NULL           /* pointer to user-definable data */
};

SHORT BorderVectors4[] =
{
    0,0,
    169,0,
    169,10,
    0,10,
    0,0
};

struct Border Border4 =
{
    -1,-1,          /* XY origin relative to TopLeft */
    1,0,JAM1,       /* front pen, back pen and drawmode */
    5,              /* number of XY vectors */
    BorderVectors4, /* pointer to XY vectors */
    NULL           /* next border in list */
};

UBYTE UNDOBUFFER[255];
UBYTE NameGadSIBuff[255];

struct StringInfo NameGadSInfo =
{
    NameGadSIBuff, /* buffer where text will be edited */
    UNDOBUFFER,    /* optional undo buffer */
    0,             /* character position in buffer */
    25,           /* maximum number of characters to allow */
    0,            /* first displayed character buffer position */
};

```

```

0,0,0,0,0, /* Intuition initialized and maintained variables */
0, /* Rastport of gadget */
0, /* initial value for integer gadgets */
NULL /* alternate keymap (fill in if you set the flag) */
);

struct Gadget NameGad =
{
    &EntryBox,
    5,94,
    168,9,
    NULL,
    RELVERIFY|GADGIMMEDIATE|LONGINT, /* Make into a Integer gadget */
    STRGADGET, /* String gadget */
    (APTR)&Border4,
    NULL,
    NULL,
    NULL,
    (APTR)&NameGadSInfo, /* SpecialInfo structure */
    5,
    NULL
};

USHORT chip DArrowData[] = /* Down Arrow */
{
    0xFFFF,0xF81F,0xF81F,0xF81F,0xF81F,0x8001,0xE007,0xF81F,0xFE7F
};

USHORT chip UArrowData[] = /* Up Arrow */
{
    0xFE7F,0xF81F,0xE007,0x8001,0xF81F,0xF81F,0xF81F,0xF81F,0xFFFF
};

USHORT chip RArrowData[] = /* Right Arrow */
{
    0xFFFF,0xFF3F,0xFF0F,0xC003,0xC000,0xC003,0xFF0F,0xFF3F,0xFFFF
};

USHORT chip LArrowData[] = /* Left Arrow */
{
    0xFFFF,0xFCFF,0xF0FF,0xC003,0x0003,0xC003,0xF0FF,0xFCFF,0xFFFF
};

struct Image Arrows[] =
{
    {0,0,16,9,2,UArrowData,0x0001,0x0000,NULL},
    {0,0,16,9,2,DArrowData,0x0001,0x0000,NULL},
    {0,0,16,9,2,LArrowData,0x0001,0x0000,NULL},
    {0,0,16,9,2,RArrowData,0x0001,0x0000,NULL}
};

struct Gadget ArrowGads[] =
{
    {
        &NameGad, 158, 73, 16,9, GADGIMAGE, RELVERIFY|GADGIMMEDIATE,
        BOOLGADGET,(APTR)&Arrows[0], NULL, NULL, NULL, NULL, 4, NULL
    },
    {
        &ArrowGads[0],158, 83,16,9,GADGIMAGE, RELVERIFY|GADGIMMEDIATE,
        BOOLGADGET,(APTR)&Arrows[1], NULL, NULL, NULL, NULL, 3, NULL
    }
};

struct PropInfo VertSliderSInfo =
{
    AUTOKNOB|FREEVERT, /* PropInfo flags */
    -1,-1, /* horizontal and vertical pot values */
    -1,-1, /* horizontal and vertical body values */
};

struct Image Image3 =
{0,0,7,72,0, NULL,0x0000,0x0000, NULL};

```

```

struct Gadget VertSlider =
{
    &ArrowGads[1], 158,0, 16,72, NULL,
    RELVERIFY|GADGIMMEDIATE,
    PROPGADGET,          /* Proportional Gadget */
    (APTR)&Image3,        /* Slider Imagry */
    NULL,NULL,NULL,
    (APTR)&VertSliderSInfo, /* SpecialInfo structure */
    2, NULL
};

struct NewWindow NewWindow =
{
    160,25, 178,150, 0,1,
    INTUITICKS|GADGETDOWN|GADGETUP|CLOSEWINDOW,
    WINDOWDRAG|WINDOWDEPTH|
    WINDOWCLOSE|ACTIVATE|NOCAREREFRESH,
    NULL,NULL, "Gadgets", NULL, NULL, 0,0, -1,-1, WBENCHSCREEN
};

/* End of gadgets.h */

/* Gadgets.c 10/89
 * Compiled with Lattice 5.04: LC -bl -cfist -L -v -w
 */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <libraries/dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef LATTICE
#include <proto/all.h>
#endif

#define RP window->RPort
#define TOT_DISPLAY 10
#define LINE_HEIGHT 9
#define MAX_VALUE 9999

#include "Gadgets.h"

/* function declarations */
VOID OpenAll(VOID);
VOID cleanExit(int);
USHORT HandleUGad(struct IntuiMessage *);
USHORT HandleDGad(struct IntuiMessage *);
VOID InitSlider(struct Gadget *g);
VOID GetSlider(struct Gadget *, int);
VOID DisplayEntries(VOID);
VOID PrintText(struct RastPort *rp, UBYTE *msg,
    LONG x, LONG y, USHORT f, USHORT b);

/* global declarations */
struct GfxBase *GfxBase = NULL;
struct IntuitionBase *IntuitionBase = NULL;
struct Window *window = NULL;

/* for showing the list information */
LONG numentries;
LONG topeentry;

VOID main(int argc, char *argv[])
{
    struct IntuiMessage *msg = NULL;
    ULONG class;
    USHORT flagi;

    OpenAll();

```

```

flagi=TRUE;
while(flagi) {
    Wait(1L << window->UserPort->mp_SigBit);
    while(msg=(struct IntuiMessage *)GetMsg(window->UserPort)) {
        class=msg->Class;
        switch(class) {
            case INTUITICKS:
                if(ArrowGads[0].Flags&SELECTED) GetSlider(&VertSlider,-1);
                if(VertSlider.Flags&SELECTED) GetSlider(&VertSlider, 0);
                if(ArrowGads[1].Flags&SELECTED) GetSlider(&VertSlider, 1);
                break;
            case GADGETDOWN:
                flagi=HandleDGad(msg); break;
            case GADGETUP:
                flagi=HandleUGad(msg); break;
            case CLOSEWINDOW:
                flagi=FALSE; break;
            default: break;
        }
        ReplyMsg((struct Message *)msg);
    }
}
cleanExit(0);
}

/* For this example, this is just used to clear the
 * message areas. You could use it to start a function
 * on the down-press of a gadget
 */

USHORT HandleDGad(struct IntuiMessage *m)
{
    struct Gadget *g;
    USHORT id;
    UBYTE msg[12];
    USHORT retval;

    retval=TRUE;
    strcpy(msg,"          00");
    g = (struct Gadget *)m->IAddress;
    id = g->GadgetID;

    /* clear the entry number area */
    PrintText(RP, msg, 10, 139, 1, 0);

    /* clear the button number area */
    sprintf(msg, "ID %-4dd", id);
    PrintText(RP, msg, 104, 139, 1, 0);

    /* return */
    return(retval);
}

USHORT HandleUGad(struct IntuiMessage *m)
{
    struct Gadget *g;
    USHORT id;
    UBYTE msg[12];
    USHORT retval;
    USHORT entry;

    retval=TRUE;
    g = (struct Gadget *)m->IAddress;
    id = g->GadgetID;
    switch(id) {
        case 1: /* The ENTRY area */
            entry = topeentry + ((m->MouseY - g->TopEdge + 1) / LINE_HEIGHT) + 1;
            if(entry > numentries) entry = 0;
            sprintf(msg, "Entry %-4d", entry);
            PrintText(RP, msg, 10, 139, 1, 0);
    }
}

```



```

        break;
    case 2: /* Clicked in the body of the Vertical PROP */
        GetSlider(&VertSlider, 0);
        break;
    case 5: /* The STRING gadget */
        /* Get the number of entries. Filter it out, put it back
         * into the String gadget and display it.
         */
        numentries = NameGadSInfo.LongInt;
        if(numentries>MAX_VALUE) numentries=MAX_VALUE;
        sprintf(NameGadSIBuff, "%d\000", numentries);
        RefreshGList(&NameGad, window, NULL, 1);
        InitSlider(&VertSlider);
        break;
    case 6: /* OKAY button */
    case 7: /* CANCEL button */
        retval = FALSE;
        break;
    default:
        break;
}
/* show the button number that we pushed */
sprintf(msg, "ID %-4du", id);
PrintText(RP, msg, 104, 139, 1, '0');

/* return */
return(retval);
}

/* Initialize the proportional gadget
 */
VOID InitSlider(struct Gadget *g)
{
    topentry=0;
    if(numentries>TOT_DISPLAY)
    {
        NewModifyProp(g,window,NULL,AUTOKNOB|FREEVERT,
            NULL, ((MAXBODY*topentry)/(numentries)),
            MAXBODY, ((MAXBODY*TOT_DISPLAY)/numentries),1L);
    }
    else
    {
        NewModifyProp(g,window,NULL,AUTOKNOB|FREEVERT,
            NULL,NULL,MAXBODY,MAXBODY,1L);
    }
    /* clear the entry area */
    RefreshGList(&BackDrop,window,NULL,1);
    /* display the current entries */
    DisplayEntries();
}

/* Get the current entry, based on either the movement of the
 * proportional gadget, or the pressing of the arrow keys.
 */
VOID GetSlider(struct Gadget *g, int dir)
{
    USHORT potv;
    struct PropInfo *p;
    static USHORT update=0;

    p=(struct PropInfo *)g->SpecialInfo;
    if(dir!=0)
    {
        topentry += dir;
        if(topentry>0 && topentry<(numentries-TOT_DISPLAY))
        {
            potv=((MAXBODY*topentry)/(numentries-TOT_DISPLAY));
            update=0;

```

```

    }
    else
    {
        /* It is necessary that the check for toplevtry>=numentries
        * is before toplevtry<=0 to catch the instance when
        * numentries < TOT_DISPLAY
        */
        if (toplevtry >= (numentries - TOT_DISPLAY))
        {
            potv = MAXBODY;
            toplevtry = (numentries - TOT_DISPLAY);
            update++;
        }
        if (toplevtry <= 0)
        {
            potv = 0;
            toplevtry = 0;
            update++;
        }
    }
    if (numentries > TOT_DISPLAY && update < 3)
    {
        NewModifyProp(g, window, NULL, AUTOKNOB | FREEVERT,
            NULL, potv, MAXBODY,
            ((MAXBODY * TOT_DISPLAY) / numentries - TOT_DISPLAY), 1L);
    }
}
else
{
    if (numentries > TOT_DISPLAY)
        toplevtry = (p->VertPot * (numentries - TOT_DISPLAY)) / MAXBODY;
    else
        toplevtry = 0;
}
DisplayEntries();
}

```

```

/* Update the display to show the current view of entries.
 * This example DOESN'T show how to handle situations
 * where the number of entries are less than what fits
 * in the view.
 */

```

```

VOID    DisplayEntries(VOID)
{
    UBYTE  msg[5];
    register USHORT i;

    /* update the display */
    for (i = 0; (i < TOT_DISPLAY && i < numentries); i++)
    {
        sprintf(msg, "%4d", i + toplevtry + 1);
        PrintText(RP, msg,
            7, (USHORT)(EntryBox.TopEdge + (i * LINE_HEIGHT)), 1, 2);
    }

    /* put the pen back the way it was */
    SetBPen(RP, 0);
}

```

```

VOID    PrintText(struct RastPort *rp, UBYTE *msg,
    LONG x, LONG y, USHORT f, USHORT b)
{
    strcpy(Buffer, msg);
    Messages.FrontPen = f;
    Messages.BackPen = b;
    PrintIText(rp, &Messages, x, y);
}

```

```

VOID    cleanExit(int retval)
{

```

```

    if(window)          CloseWindow(window);
    if(GfxBase)          CloseLibrary((struct Library *)GfxBase);
    if(IntuitionBase)    CloseLibrary((struct Library *)IntuitionBase);
    exit(retval);
}

VOID    OpenAll(VOID)
{
    struct Gadget *g;

    if(!(IntuitionBase=(struct IntuitionBase *)
        OpenLibrary("intuition.library",33)))
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);
    if(!(GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",33)))
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);

    /* center the window */
    NewWindow.TopEdge = (GfxBase->NormalDisplayRows - NewWindow.Height) / 2;
    NewWindow.LeftEdge= (GfxBase->NormalDisplayColumns-NewWindow.Width) / 2;

    /* open the window */
    if(!(window=OpenWindow(&NewWindow)))
        cleanExit(ERROR_NO_FREE_STORE);

    /* Adjust the top of the gadgets relative to the upper border.
     * Usually is 11 if using Topaz80, 12 if using Topaz60
     */
    g=&VertSlider;
    while(g)
    {
        g->TopEdge += window->BorderTop;
        g = g->NextGadget;
    }

    /* pre-initialize the list variables */
    numentries=100;

    /* just to show how many entries we're working with */
    sprintf(NameGadSIBuff,"%d\000", numentries);

    /* hook the gadgets to the window */
    AddGList(window, &VertSlider, 0, -1, NULL);

    /* update the display to show the new gadgets */
    RefreshGList(&VertSlider, window, NULL, -1);

    /* update the vertical proportional gadget information */
    InitSlider(&VertSlider);
}

```

# **Chapter 5**

## **Intuition: Menus**

This chapter shows how to set up the menus that let the user choose from your program's commands and options. The Intuition menu system handles all of the menu display from menu data structures that you set up. If you wish, some or all of your menu selections can be graphic images instead of text.

### **About Menus**

Intuition's menu system provides you with a convenient way to group together and display the functions and options that your application presents to the user. For instance, in a word-processor environment, menus may provide the following functions:

- Access to text files.
- Editing functions.
- Search and replace facilities.
- Formatting capabilities.

- Multiple fonts.
- A general help facility.

In a game, menus may provide the user with choices about how to:

- Load a new game or save the current one.
- Get hints.
- Bring up special information windows.
- Set the difficulty level.
- Auto-annihilate the enemy.

Menu commands are either actions or attributes. Actions are represented by verbs and attributes by adjectives. An attribute stays in effect until canceled, while a command is executed and then forgotten. You can set up menus so that some attribute items are mutually exclusive (selecting an attribute cancels the effects of one or more other attributes), or you can allow a number of attributes to be in effect at the same time. For example, an adventure game might have a menu list for things that the hero is holding in his hand. He could hold several small, lightweight objects, but holding the heavy sword excludes holding anything else. In a database program, you might be able to choose to send a report to a file, to the window, or to a printer. You could, for example, send it to both a window and a printer, while the "file" option excludes the other two.

After you set up a linked list of menu structures (called a *menu strip*) and attach the list to a window, the menu system handles the menu display. Using this list and any graphic images you have designed, the menu system displays the menu bar text that appears across the screen title bar when requested by the user. It also creates the lists of menu items and sub-menus that appear at the user's request. The application does not have to worry about menus until Intuition sends a message with news that the user has selected a menu item. This message gives the application the number of the selected item.

You can enable and disable menu items or whole menus, and make changes to the menus you previously attached to a window. Disabling an item prevents the user from selecting it, and disabled items are ghosted to look different from enabled items.

Menu items can be graphic images or text. When the user positions the pointer over an item, the item can be highlighted through a variety of techniques. Items can also show that they have been selected by having an image rendered next to them, usually a checkmark. Next to the menu items, you can display command-key alternatives.

To activate the menu system, the user presses the mouse menu button (or an appropriate command-key sequence) to display the menu bar in the screen title area. The menu bar displays a list of topics (called *menus*) that have menu items associated with them (see the figure).

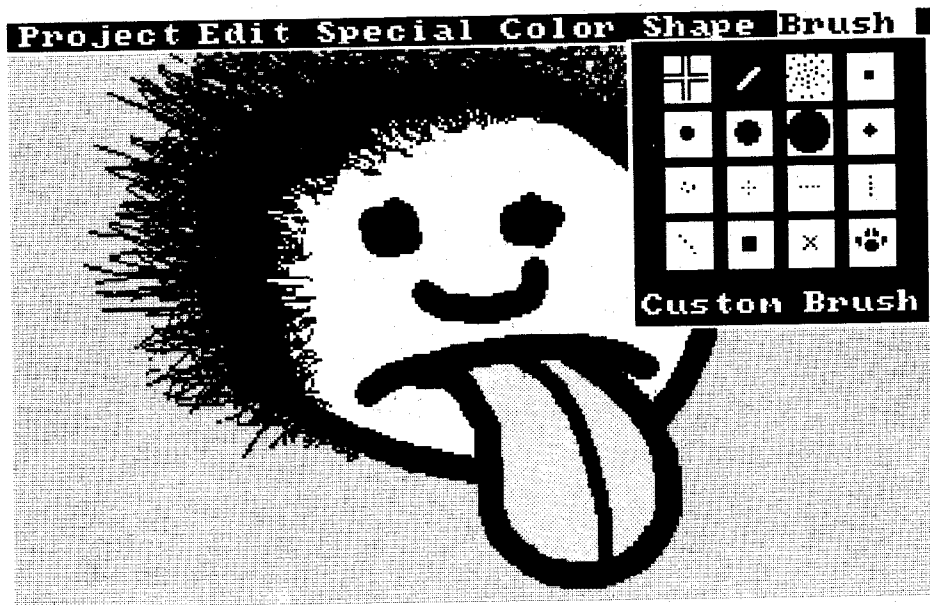


Figure 5-1: Screen with Menu Bar Displayed

When the user moves the mouse pointer to a topic in the menu bar, a list of menu items appears below the topic name. To select an item, the user moves the mouse pointer in the list of menu items while holding down the menu button, releasing the button when the pointer is over the desired item. If an item has a subitem list, moving the pointer over the item reveals a list of subitems. The user moves the pointer over one of the subitems and makes a selection in the same way as an item is selected. If there is a command-key sequence alternative, the user can make menu selections with the keyboard instead of the mouse. Furthermore, the user can select multiple items by:

- Pressing and releasing the mouse select button without releasing the menu button. This selects that item and keeps the user in “menu state” so that other items can be selected.
- Holding down both mouse buttons and moving the pointer over several items. This is called drag-selecting.

## SUBMITTING AND REMOVING MENU STRIPS

Once you have constructed a menu strip, you submit it to Intuition using the function **SetMenuStrip()**. You must ultimately remove every menu strip that you have submitted. When you want to remove the menu strip, you call **ClearMenuStrip()**. If you want to change the menu strip, you call **ClearMenuStrip()**, change the menu, and resubmit it with **SetMenuStrip()**.

The flow of events for menu operations should be:

1. **OpenWindow()**.
2. Zero or more iterations of **SetMenuStrip()** and **ClearMenuStrip()**.

### 3. CloseWindow().

You *must* clear the menu strip before closing the window.

## ABOUT MENU ITEM BOXES

The item box is the rectangle containing your menu items or subitems. You do not have to describe the size and location of the item or subitem boxes directly. You describe the size indirectly by how you place items and subitems. Intuition figures out the size of the minimum box required. It then adjusts the size of the box to make sure your menu display conforms to certain design philosophy constraints for items and subitems. See the following figures for examples of item and subitem box structures.

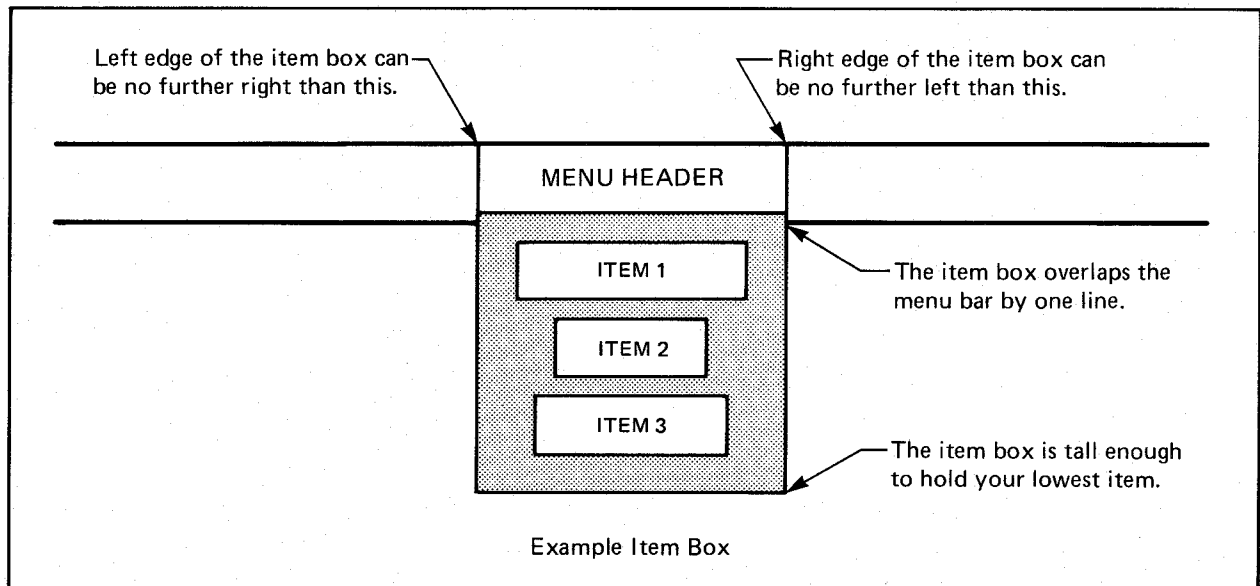


Figure 5-2: Example Item Box

The item box must start no further right than the leftmost position of the menu header's select box. It must end no further left than the rightmost position of the menu header's select box. The top edge of the menu box must overlap the screen's title bar by one line. The subitem box must overlap its item's select box somewhere.

### NOTE

Do not leave space between the select boxes of your menu items and (especially) your subitems. When the pointer moves off one subitem into the gap between it and the next subitem, the entire submenu is erased and redrawn, which causes ugly flickering. Even a space of a single line will cause flickering. For example, if the **Height** field of a menu item is ten, and the **TopEdge** field of the next menu item is 12, flickering will occur. The **TopEdge** field should be 11, in this case.

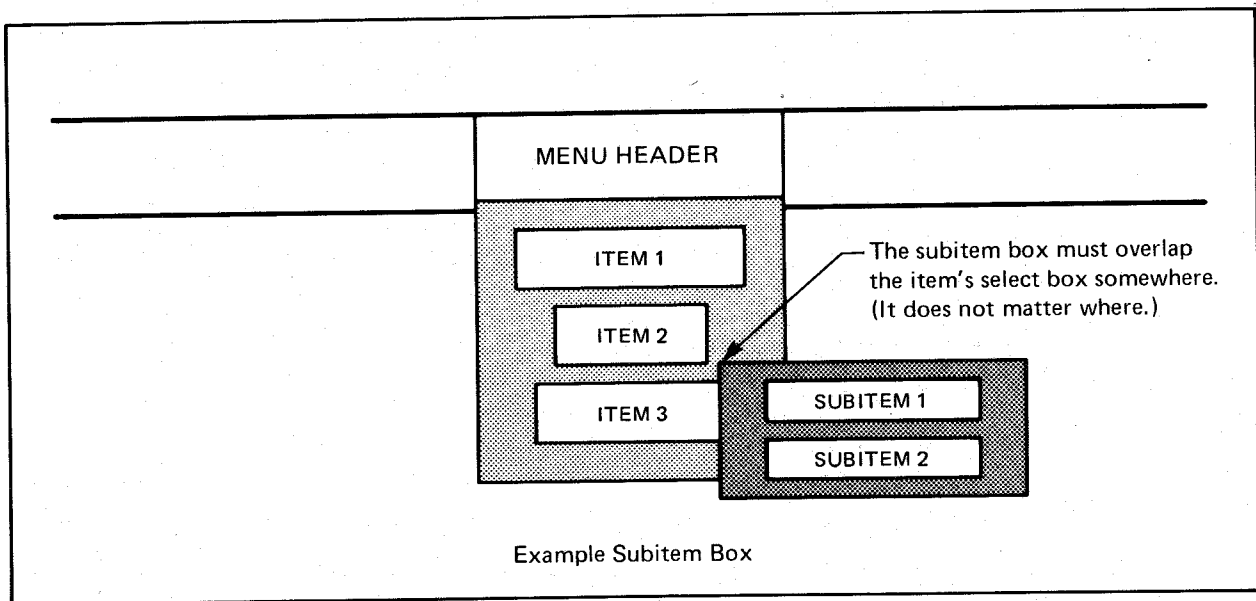


Figure 5-3: Example Subitem Box

## ACTION/ATTRIBUTE ITEMS AND THE CHECKMARK

Menu action items are selected and acted upon immediately. Action items can be selected repeatedly. Every time the user selects an action item, the selection is transmitted to your program.

Menu attribute items may be toggle-selected or mutually exclusive. A toggle-selected item is selected or deselected when the user accesses it. Accessing a mutually exclusive menu attribute item puts it in the selected state, where it remains until it is mutually excluded by the selection of some other attribute item. Intuition puts a checkmark beside any attribute item which has been selected. While the checkmark is the default symbol, you may have Intuition use a symbol of your own design.

You specify that a particular menu item is an attribute item by setting the **CHECKIT** flag in the **Flags** variable of the item's **MenuItem** structure.

You can initialize the state of an attribute item by presetting the item's **CHECKED** flag. If this flag is set when you submit your menu strip to Intuition, then the item is considered to be already selected and the checkmark will be drawn.

You can use the default Intuition checkmark ( ✓ ) or you can design your own and set a pointer to it in the **NewWindow** structure when you open a window. See the chapter on windows for details about supplying your own checkmark.

If your items are going to be checkmarked, you should leave sufficient blank space at the left edge of your select box for the checkmark imagery. If you are taking advantage of the default checkmarks, you should leave **CHECKWIDTH** amount of blank pixels on high-resolution screens and **LOWCHECKWIDTH** amount of blank pixels on low-resolution screens. These are defined constants describing the pixel width in high and low resolution.



They define the space required by the standard checkmarks (with a bit of space for aesthetic purposes). If you would normally place the **LeftEdge** of the image within the item's select box at 5, and you decide that you want a checkmark to appear with the item, then you should start the item at 5+CHECKWIDTH instead. You should also make your select box CHECKWIDTH wider than it would be without the checkmark.

## TOGGLE-SELECTION

You can make some of your attribute items toggle-select. Each time the user accesses such an item, it changes state, selected or unselected. To make an attribute item toggle-select, set both the CHECKIT and the MENUTOGGLE flags for that menu item. Of course, you may rpe-set the CHECKED flag to the desired initial state.

## MUTUAL EXCLUSION

You can choose to have some of your attribute items, when selected, cause other items to become unselected. This is known as *mutual exclusion*. For example, if you have a list of menu items describing the available type sizes for a particular font, the selection of any type size would mutually exclude all other type sizes. You use the **MutualExclude** variable in the **MenuItem** structure to specify other menu items to be excluded when the user selects an item. Exclusion also depends upon the CHECKED and CHECKIT flags of the **MenuItem**, as explained below.

- If CHECKED is *not* set, then this item is available to be selected. If the user selects this item, the CHECKED flag is set and the user cannot then reselect this item. If the item is selected, the CHECKED flag will be set, and the checkmark will be drawn to the left of the item.
- If the item selected has bits set in the **MutualExclude** field, the CHECKED flag is examined in the excluded items. If any item is currently CHECKED, its checkmark is erased, and its CHECKED flag is cleared.
- Mutual exclusion is an active event. It pertains only to items that have the CHECKIT flag set. Attempting to exclude items that do not have the CHECKIT flag set has no effect.

### NOTE

It is up to you to track internally which excluded items have been disabled and deselected.

In the **MutualExclude** field, bit 0 refers to the first item in the item list, bit 1 to the second, bit 2 to the third, and so on. In the adventure game example described earlier, in which carrying the heavy sword excludes carrying any other items, the **MutualExclude** fields of the four items would look like this:

Heavy sword	0xFFFFE
Stiletto	0x0001
Rope	0x0001
Canteen	0x0001

“Heavy Sword” is the first item on the list. You can see that it excludes all items except the first one. All of the other items exclude only the first item, so that carrying the rope excludes carrying the sword, but not the canteen.

## COMMAND-KEY SEQUENCES AND IMAGERY

A *command-key sequence* is an event generated when the user holds down one of the AMIGA keys (the ones with the fancy A) and presses one of the normal alphanumeric keys at the same time. You can associate a command-key sequence with a particular menu item. Menu command-key sequences are combinations of the right AMIGA key with any alphanumeric character. If the user presses a command-key sequence that is associated with one of your menu items, Intuition will send the program an event that will look like the user went through the entire process of selecting the menu item manually. This allows you to provide *shortcuts* to the user, because many people find it easy to memorize the command-key sequences for often-repeated menu selections. When accessing those often-repeated selections, most users would rather keep their hands on the keyboard than go to the mouse to make a menu selection.

You associate a command-key sequence with a menu item by setting the COMMSEQ flag in the **Flags** variable of the **MenuItem** structure and by putting the ASCII character (upper or lower case) that you want associated with the sequence into the **Command** variable of the **MenuItem** structure. Intuition ignores case when checking for command-key equivalents.

When items have alternate key sequences, the menu boxes show a special AMIGA key icon rendered about one character span plus a few pixels from the right edge of the menu select box and the command-key used with the AMIGA key rendered immediately to the right of the AMIGA key image, at the rightmost edge of the menu select box (see the figure).

If you want to show a command-key sequence for an item, you should make sure that you leave blank space at the right edge of your select box and imagery. You should leave COMMWIDTH amount of blank space on high-resolution screens, and LOWCOMMWIDTH amount of space on low-resolution screens.

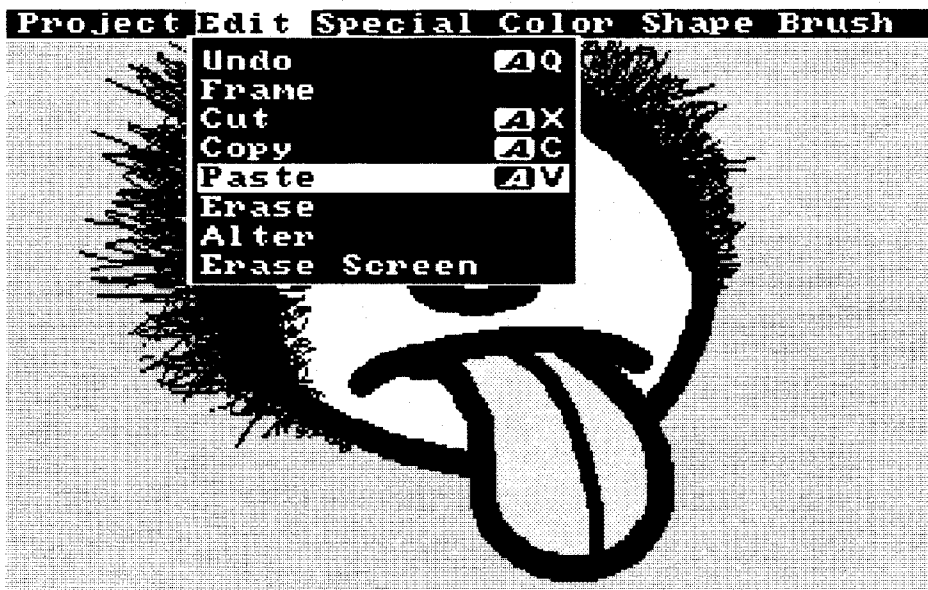


Figure 5-4: Menu Items with Command Key Shortcuts

## ENABLING AND DISABLING MENUS AND MENU ITEMS

Disabling menu items makes them unavailable for selection by the user. Disabled menus and menu items are displayed in a “ghosted” fashion; that is, the imagery is overlaid with a faint pattern of dots, making it less distinct. Enabling or disabling a menu or menu item is always a safe procedure, whether or not the user is currently using the menus. A problem arises only if the program disables a menu item that the user has already selected with extended select. The program will receive a MENU PICK message for that item, even though it thinks it has already disabled it. The program will have to ignore items that it knows are already disabled.

You use the routines `OnMenu()` and `OffMenu()` to enable and disable individual subitems, items or whole menus. These routines check if the user is using the menus and whether the menus need to be redrawn to reflect the new states.

## CHANGING MENU STRIPS

If you want to make changes to the menu strip you previously attached to your window, you *must* first call `ClearMenuStrip()`. You may alter the menu strip only after it has been removed from the window.

To add a new menu strip to your window, you *must* call `ClearMenuStrip()` before you call `SetMenuStrip()` with the new menus.

## MENU NUMBERS AND MENU SELECTION MESSAGES

An input event is generated every time the user activates the menu system by pressing the mouse menu button (or entering an appropriate command-key sequence). Your program receives a message of type MENU PICK telling which menu item has been selected. If one of your items has a subitem list, the menu number your program receives for that item includes some subitem selection.

Even if the user presses and releases the menu button without selecting any of the menu items, an event is generated. If the user presses and releases the menu button without selecting one of the menu items, the program receives a message with the menu number equal to MENUNULL. In this way, the program can always find out when the user has simply clicked the menu button rather than making a menu selection.

The user can select multiple menu items with one of the extended selection procedures (pressing the mouse select button without releasing the menu button or drag-selecting). Your program finds out whether or not multiple items have been chosen by examining the field called `NextSelect` in the `MenuItem` data structure. After taking the appropriate action for the item selected by the user, the program should check the `NextSelect` field. If the number there is equal to the constant MENUNULL, there is no next selection. However, if it is not equal to MENUNULL, the user has selected another option after this one. The program should process the next item as well, by checking its `NextSelect` field, until it finds a `NextSelect` equal to MENUNULL.

The following code fragment shows the correct way to process a menu event:

```

while (MenuNumber != MENUNULL)
{
    Item = ItemAddress(MenuStrip, MenuNumber);
    /* process this item */
    MenuNumber = Item->NextSelect;
}

```

When the user performs multiple selection, you will receive only *one* message of class **MENUPICK**. For your program to behave correctly, you must pay attention to the **NextSelect** field of the **MenuItem**, which will lead you to the other menu selections.

The number given in the **MENUPICK** message describes the ordinal position of the **Menu** in your linked list, the ordinal position of the **MenuItem** beneath that **Menu**, and (if applicable) the ordinal position of the subitem beneath that **MenuItem**. Ordinal means the successive number of the linked items, starting from 0. To discover the **Menus** and **MenuItems** that were selected, you should use the following macros:

Use **MENUNUM(num)** to extract the ordinal menu number from the value.  
 Use **ITEMNUM(num)** to extract the ordinal item number from the value.  
 Use **SUBNUM(num)** to extract the ordinal subitem number from the value.  
**MENUNULL** is the constant describing "no menu selection made."  
 Likewise, **NOMENU**, **NOITEM**, and **NOSUB** are the null states of the parts.

For example:

```

if (number == MENUNULL) then no menu selection was made, else
MenuNumber = MENUNUM(number);
ItemNumber = ITEMNUM(number);
SubNumber = SUBNUM(number);
if there were no subitems attached to that item, SubNumber will equal NOSUB.

```

The menu number received by the program describes either **MENUNULL** or a valid menu selection. If it is a valid selection, it will always have at least a menu number and a menu item number. Users can never "select" the menu text itself, but they always select at least an item within a menu. Therefore, the program always gets one menu specifier and one menu item specifier. If a given menu item has a subitem, a subitem specifier will also be received. Just as it is not possible to select a menu, it is not possible to select a menu item that has a list of subitems. The user must select one of the options in the subitem list before the program ever hears about it as a valid selection.

If the user enters a command-key sequence, Intuition checks to see if the sequence is associated with a current menu item. If so, Intuition sends the menu item number to the program with the active window just as if the user had made the selection using the mouse buttons.

The function **ItemAddress()** translates a menu number into an item address.

## HOW MENU NUMBERS REALLY WORK

The following is a description of how menu numbers really work. It should illuminate why there are certain numeric restrictions on the number of menu components Intuition allows. You should not use the information given here to access the menu number information directly. This discussion is included only for completeness. To assure upward compatibility, always use the macros supplied. To extract the item number from the variable **MenuNumber**, for example, call **ITEMNUM(MenuNumber)**. See the previous section, "Menu Numbers and Menu Selection Messages," for a complete description of the menu number macros.

Menu numbers are 16-bit numbers with 5 bits used for the menu number, 6 bits used for the menu item number, and 5 bits used for the subitem number. Everything is specified by its ordinal position in a list of same-level pieces, as shown below.

```

c c c c c b b b b b b a a a a
|         |         |
|         |         | > These bits are for the menu number.
|         |         |
|         |         | > These bits are for the menu items within the menu.
|         |         |
> These bits are for the subitems within the menu items.

```

Thus, for each level of menu item and subitem, up to 31 pieces can be specified. There are 63 item pieces that you can build under each menu, which is a lot, especially with 31 subitems per item. You can have 31 menu choices across the menu bar (it would be a tight squeeze, but in 80-column mode you could do it), and each of those menus can exercise up to 1,953 items. You should not need any more choices than that.

The value "all bits on" means that no selection of this particular component was made. MENUNULL actually equals "no selection of any of the components was made" so MENUNULL always equals "all bits of all components on."

Here's an example. Say that your program gets back the menu number (in hexadecimal) 0x0CA0. In binary that equals:

```

0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0
|          |          |
|          |          | > Menu number 0
|          |          |
|          |          | > Menu item number 0x25 = 37
|          |          |
> Subitem number 1

```

Again, it is never safe to examine these numbers directly. Use the macros described above if you want to design sanely and assure upward compatibility.

## INTERCEPTING NORMAL MENU OPERATIONS

You have two convenient ways to intercept the normal menu operations that take place when the user presses the right mouse button. The first, `MENUVERIFY`, gives your program the opportunity to react before menu operations take place and, optionally, to cancel menu operations. The second, `RMBTRAP`, allows the program to trap right mouse button events for its own use.

## A Warning on the MENUSTATE Flag

The `MENUSTATE` flag is set by `Intuition` in `Window.Flags` when the menus of that window are in use. Beware: in typical event-driven programming, such a state variable is not on the same timetable as your input message handling, and should not be used to draw profound conclusions in your program. To synchronize yourself with the menu handling, use `MENUVERIFY`.

## Menu-verify

Menu-verify is one of the Intuition verification functions. These functions allow you to make sure that your program is prepared for some event before it takes place. Using menu-verify, Intuition allows all windows in a screen to verify that they are prepared for menu operations before the operations begin. In general, you use this if the program is doing something special to the display of a custom screen, and you want to make sure it has completed before menus are rendered.

Any window can access the menu-verify feature by setting the `MENUVERIFY` flag in the `NewWindow` structure when opening the window. When your program gets a message of class `MENUVERIFY`, menu operations will not proceed until the program replies to the message.

The active window gets special menu-verify treatment. It is allowed to see the menu-verify message before any other window and has the option of canceling menu operations altogether. You could use this, for instance, to examine where the user has positioned the mouse when the right button was pressed. If the pointer is in the menu bar area, then you can let normal menu operations proceed. If the pointer is below the menu bar, then you can use the right button event for some non-menu purpose.

Your program can tell whether or not it is in the active window by examining the code field of the `MENUVERIFY` message. If the code field is equal to `MENUWAITING`, your window is not the active one and Intuition is simply waiting for you to verify that menu operations may continue. However, if the code field is equal to `MENUHOT`, your window *is* the active one and it gets to decide whether or not menu operations should proceed. If the program does *not* want them to proceed, it should change the code field of the message to `MENUCANCEL` before replying to the message. This will cause Intuition to cancel the menu operations.

## Shortcuts and MENUVERIFY

The idea behind `MENUVERIFY` (and to some degree `SIZEVERIFY` and `REQVERIFY`) is to synchronize your program with Intuition's menu handling sessions. The motive was to allow your program to arbitrate access to your screen's bitmap, so that Intuition doesn't put menus in the way of your drawing.

Some programs use `MENUVERIFY` to permit them to intercept the right mouse button for their own purposes. Other's use it to suspend menu operations while they recover from Wild Phenomena before menu operations proceed. These phenomena may be illegible colors of the screen or double buffering and related ViewPort operations.

In any case, it is vital to know when menu operations terminate. This is typically detected by watching for the `MENUPICK IDCMP` message. If you intercepted (`MENUCANCEL`) the menu operations, you will instead receive a `MOUSEBUTTONS` message with code equal to `MENUUP`. Menu shortcut keystrokes, for compatibility, also respect `MENUVERIFY`. They are always paired with a `MENUPICK` message so that your program knows the menu operation is over.

You may call `ModifyIDCMP()` to turn `MENUVERIFY` and the other `VERIFY IDCMP` options off. It is important that you do so if you ever do anything that directly or indirectly has you waiting for Intuition (since Intuition may be waiting for you).

You cannot wait for a gadget or mouse event without checking also for any `MENUVERIFY` event messages that may require your response. The most common problem area is System Requesters (`AutoRequest()`). Before `AutoRequest()` returns control to your program, Intuition must be free to run and accept a response from the user. If the user presses the menu button, Intuition will wait for you to `MENUVERIFY` and a deadlock results. Therefore, it

is extremely important to use `ModifyIDCMP()` to turn off all verify messages before you call `AutoRequest()` or, directly or indirectly, AmigaDOS, since many error conditions in the DOS require user input in the form of Autorequests. Indirect DOS calls include `OpenLibrary()`, `OpenDevice()`, and `OpenDiskFont()`.

### Intuition's Use of Your RastPort

Intuition has many rendering chores: screen and window titles and borders, gadgets, menus, and so on. Intuition uses a copy of the RastPort of the screen in which the rendering is to take place. This copy determines the bitmaps the rendering will end up in, and often the font and similar modal information.

One thing Intuition sets each time is the mask value of the RastPort. It is set to all ones (0xFFFF). If you wish to restrict Intuition's rendering to all bitplanes of your screen, you may change the Depth and Planes values in `Screen.RastPort.BitMap`. This will only affect rendering into the screen itself, which consists of the Screen title and gadgets, and menus. Window gadgets are not fooled, since they use the mask in the window's layer's rastport, which you should not be changing.

### No Menu Operations — Right Mouse Button Trap

By setting the `RMBTRAP` flag in the `NewWindow` structure when you open your window, you indicate that you do not want any menu operations at all for your window. Whenever the user presses the right button while your program's window is active, the program will receive right button events as normal `MOUSEBUTTON` events.

## REQUESTERS AS MENUS

You may, in some cases, want to use a requester instead of a menu. A requester can function as a "super-menu" because you can attach a requester to the double-click of the mouse menu button. This allows users to bring up the requester on demand. With a requester, however, the user must make some response before resuming input to the window. See the chapter entitled "Intuition: Requesters and Alerts," for more information.

## MENU STRUCTURES

The specifications for the menu structures are given below. **Menus** are the headers that show in the menu bar, and **MenuItems** are the items and subitems that can be chosen by the user.

### Menu Structure

Here is the specification for a Menu structure:

```
struct Menu
{
    struct Menu *NextMenu;
    SHORT LeftEdge, TopEdge, Width, Height;
    USHORT Flags;
    BYTE *MenuName;
    struct MenuItem *FirstItem;
};
```

The variables in the **Menu** structure have the following meanings:

#### **NextMenu**

This variable points to the next **Menu** header in the list. The last **Menu** in the list should have a **NextMenu** value of **NULL**.

#### **LeftEdge, TopEdge, Width, Height**

These fields describe the select box of the header. Currently, any values you may supply for **TopEdge** and **Height** are ignored by Intuition, which uses instead the screen's **TopBorder** for the **TopEdge** and the height of the screen's title bar for the **Height**. **LeftEdge** is relative to the **LeftEdge** of the screen plus the screen's left border width, so if you say **LeftEdge** is 0, Intuition puts this header at the leftmost allowable position.

#### **Flags**

The flag space is shared by your program and Intuition. The flags are:

##### **MENUENABLED**

This flag indicates whether or not this **Menu** is currently enabled. You set this flag before you submit the menu strip to Intuition. If this flag is not set, the menu header and all menu items below it will be disabled, and the user will be able to view, but not select any of the items. After you submit the strip to Intuition, you can change whether your menu is enabled or disabled by calling **OnMenu()** or **OffMenu()**.

##### **MIDRAWN**

This flag indicates whether or not this **Menu**'s items are currently displayed to the user.

#### **MenuName**

This is a pointer to a null-terminated character string that is printed on the screen's title bar starting at the **LeftEdge** of this **Menu**'s select box and at the **TopEdge** just below the screen title bar's top border.

#### **FirstItem**

This points to the first item in the linked list of this **Menu**'s items (**MenuItem** structures).

#### **MenuItem Structure**

Here is the specification for a **MenuItem** structure (used for both items and subitems):

```
struct MenuItem
{
    struct MenuItem *NextItem;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    LONG MutualExclude;
    APTR ItemFill;
    APTR SelectFill;
    BYTE Command;
    struct MenuItem *SubItem;
    USHORT NextSelect;
};
```

The fields have the following meanings:

#### **NextItem**

This field is a pointer to the next item in the list. The last item in the list should have a **NextItem** value of **NULL**.



**LeftEdge, TopEdge, Width, Height**

These fields describe the select box of the **MenuItem**. The **LeftEdge** is relative to the **LeftEdge** of the **Menu**. The **TopEdge** is relative to the topmost position Intuition allows. **TopEdge** is based on the way the user has the system configured — which font, which resolution, and so on. Use 0 for the topmost position.

**Flags**

The flag space is shared by your program and Intuition. See “MenuItem Flags” below for a description of the flag bits.

**MutualExclude**

This LONG word refers to the items that may be on the same “plane” as this one (maximum of 32 items). You use these bits to describe which if any of the other items are mutually excluded by this one. This does not mean that you cannot have more than 32 items in any given plane, just that only the first 32 can be mutually excluded.

**ItemFill**

This points to the data used in rendering this **MenuItem**. It can point to either an instance of an **IntuiText** structure with text for this **MenuItem** or an instance of an **Image** structure with image data. Your program tells Intuition what sort of data is pointed to by this variable by either setting or clearing the **MenuItem** flag bit **ITEMTEXT**. See “MenuItem Flags” below for more information about **ITEMTEXT**.

**SelectFill**

If you select the **MenuItem** highlighting mode **HIGHIMAGE** (in the **Flags** variable), Intuition substitutes this alternate image or text for the original rendering described by **ItemFill**. **SelectFill** can point to either an **Image** or an **IntuiText**, and the flag **ITEMTEXT** describes which.

**Command**

This variable is storage for a single alphanumeric character. If the **Flag** **COMMSEQ** is set, the user can hold down the right AMIGA key on the keyboard (to mimic using the right mouse menu button) and press the key for this character as a shortcut for using the mouse to select this item. If the user does this, Intuition transmits the menu number for this item to your program. It will look to your program exactly as if the user had selected a menu item using menus and the pointer.

**SubItem**

If this item has a subitem list, this variable should point to the first subitem in the list.

**NOTE**

A subitem cannot have a subitem attached to it. If this item is *not* an item, this variable is ignored.

**NextSelect**

This field is filled in by Intuition when this item is selected by the user. If this item is selected by the user, your program should process the request and then check the **NextSelect** field. If the **NextSelect** field is equal to **MENUNULL**, no other items were selected; otherwise, there is another item to process. See “Menu Numbers and Menu Selection Messages” above for more information about user selections.

**MenuItem Flags**

Here are the flags that you can set in the **Flags** field of the **MenuItem** structure:

**CHECKIT**

You set this flag to inform Intuition that this item is an attribute item and you want a checkmark to precede this item if the flag **CHECKED** is set. See the section “Action/Attribute Items and the CheckMark” above for full

details.

#### CHECKED

For an item with the CHECKIT flag set, set this bit to specify that this item has a checkmark. When you first submit the menu strip to Intuition, it maintains this bit based on effects from other items' mutual exclusions, or for MENUTOGGLE items, when the user accesses this item.

#### ITEMTEXT

You set this flag if the representation of this item (pointed to by the **ItemFill** field and possibly by **SelectFill**) is text and points to an **IntuiText**; you clear it if the item is graphic and points to an **Image**.

#### COMMSEQ

If this flag is set, this item has an equivalent command-key sequence (see the **Command** field above).

#### MENUTOGGLE

Set this flag for a CHECKIT menu item or subitem, and the item can be selected to turn the checkmark off, as well as on.

#### ITEMENABLED

This flag describes whether or not this item is currently enabled. If an item is not enabled, its image will be ghosted and the user will not be able to select it. Set this flag before you submit the menu strip to Intuition. Once you have submitted your menu strip to Intuition, you enable or disable items only by using **OnMenu()** or **OffMenu()**. If this item has subitems, all of the subitems are disabled when you disable this item.

#### HIGHFLAGS

An item can be highlighted when the user positions the pointer over the item. These bits describe what type of highlighting you want, if any. You must set one of the following bits according to the type of highlighting you want:

##### HIGHCOMP

This complements all of the bits contained by this item's select box.

##### HIGHBOX

This draws a box outside this item's select box.

##### HIGHIMAGE

This displays the alternate imagery in **SelectFill** (textual or image). For alternate text, make sure that **ITEMTEXT** is set, and that the **SelectFill** field points to an **IntuiText** structure.

##### HIGHNONE

This specifies no highlighting.

The following two flags are used by Intuition:

#### ISDRAWN

Intuition sets this flag when this item's subitems are currently displayed to the user and clears it when they are not.

#### HIGHITEM

Intuition sets this flag when this item is highlighted and clears it when the item is not highlighted.

## MENU FUNCTIONS

There are menu functions for attaching and clearing menu strips, for enabling and disabling menus or menu items, and for finding a menu number.

### Attaching and Removing a Menu Strip

The following functions attempt to attach a menu strip to a window or clear a menu strip from a window:

- **SetMenuStrip(Window, Menu)**

Menu is a pointer to the first menu in the menu strip. This procedure sets the menu strip into the window.

- **ClearMenuStrip(Window)**

This procedure clears any menu strip from the window.

### Enabling and Disabling Menus and Items

You can use the following functions to enable and disable items after a menu strip has been attached to the window. If the item component referenced by **MenuNumber** equals **NOITEM**, the entire menu will be disabled or enabled. If the item component equates to an actual component number, then that item will be disabled or enabled.

You can enable or disable whole menus, just the menu items, or just single subitems.

- To enable or disable a *whole menu*, set the item component of the menu number to **NOITEM**. This will disable all items and any subitems.
- To enable or disable a *single item* and all subitems attached to that item, set the item component of the menu number to your item's ordinal number. If your item has a subitem list, set the subitem component of the menu number to **NOSUB**. If your item has no subitem list, the subitem component of the menu number is ignored.
- To enable or disable a *single subitem*, set the item and subitem components appropriately.
- **OnMenu(Window, MenuNumber)**

This function enables the given menu or menu item.

- **OffMenu(Window, MenuNumber)**

This function disables the given menu or menu item.

## Getting an Item Address

This function finds the address of a menu item when given the item number:

- **ItemAddress(MenuStrip, MenuNumber)**

MenuStrip is a pointer to the first menu in the menu strip.

## Example

This example shows how to implement menus. If you look at the sample program for IDCMP's in "Intuition: Input and Output Methods", you'll see that the menu code is simply part of the processing for Intuition messages.

The example implements extended selection for menus, adaptation to fonts of different sizes, mutual exclusion, and checkmarks.

```
/* Menus.h -- All the structures needed to make the menus */

#define IWIDTH 96
#define IHEIGHT 8

/* Topaz 8, just in case we can't handle the default font */
struct TextAttr TOPAZ80 =
    {(STRPTR)"topaz.font", TOPAZ_EIGHTY, 0, 0};

/* Preferences Item IntuiText */
struct IntuiText PrefText[] =
{
    {2, 1, JAM2, CHECKWIDTH, 1, NULL, " Sound...", NULL},
    {2, 1, JAM2, CHECKWIDTH, 1, NULL, " Auto Save", NULL},
    {2, 1, JAM2, CHECKWIDTH, 1, NULL, " Have Your Cake", NULL},
    {2, 1, JAM2, CHECKWIDTH, 1, NULL, " Eat It Too", NULL}
};

struct MenuItem PrefItem[] =
{
    /* "Sound..." */
    {&PrefItem[1], 0, 0, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&PrefText[0], NULL, NULL, NULL, MENUNULL
    },
    /* "Auto Save" (toggle-select, initially selected) */
    {&PrefItem[2], 0, 10, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT|MENUTOGGLE|CHECKED, 0,
     (APTR)&PrefText[1], NULL, NULL, NULL, MENUNULL
    },
    /* "Have Your Cake" (initially selected, excludes "Eat It Too") */
    {&PrefItem[3], 0, 0, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT|CHECKED, 8,
     (APTR)&PrefText[2], NULL, NULL, NULL, MENUNULL
    },
    /* "Eat It Too" (excludes "Have Your Cake") */
    {NULL, 0, 0, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT, 4,
     (APTR)&PrefText[3], NULL, NULL, NULL, MENUNULL
    }
};

/* Edit Menu Item IntuiText */
```

```

struct IntuiText EditText[] =
{
    {2,1,JAM2,2,1, NULL, "Undo", NULL},
    {2,1,JAM2,2,1, NULL, "Cut", NULL},
    {2,1,JAM2,2,1, NULL, "Copy", NULL},
    {2,1,JAM2,2,1, NULL, "Paste", NULL},
    {2,1,JAM2,2,1, NULL, "Erase All", NULL}
};

/* Edit Menu Items */
struct MenuItem EditItem[] =
{
    /* "Undo" MenuItem (key-equivalent: 'Z') */
    {&EditItem[1], 0, 0, IWIDTH, IHEIGHT,
    ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&EditText[0], NULL, 'Z', NULL, MENUNULL
    },
    /* "Cut" (key-equivalent: 'X') */
    {&EditItem[2], 0, 10, IWIDTH, IHEIGHT,
    ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&EditText[1], NULL, 'X', NULL, MENUNULL
    },
    /* "Copy" (key-equivalent: 'C') */
    {&EditItem[3], 0, 20, IWIDTH, IHEIGHT,
    ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&EditText[2], NULL, 'C', NULL, MENUNULL
    },
    /* "Paste" (key-equivalent: 'V') */
    {&EditItem[4], 0, 30, IWIDTH, IHEIGHT,
    ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&EditText[3], NULL, 'V', NULL, MENUNULL
    },
    /* "Erase All" (disabled) */
    {NULL, 0, 40, IWIDTH, IHEIGHT,
    ITEMTEXT|HIGHCOMP, 0,
    (APTR)&EditText[4], NULL, NULL, NULL, MENUNULL
    }
};

/* IntuiText for the Print Sub-Items */
struct IntuiText PrtText[] =
{
    {2, 1, JAM2,2,1, NULL, "NLQ", NULL},
    {2, 1, JAM2,2,1, NULL, "Draft", NULL}
};

/* Print Sub-Items */
struct MenuItem PrtItem[] =
{
    /* "NLQ" */
    {&PrtItem[1], 61,-1, IWIDTH, IHEIGHT, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&PrtText[0], NULL, NULL, NULL, MENUNULL
    },
    /* "Draft" */
    {NULL, 61, 9, IWIDTH, IHEIGHT, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
    (APTR)&PrtText[1], NULL, NULL, NULL, MENUNULL
    }
};

/* Uses the >> character to indicate a sub-menu
 * 273 Octal, 0xBB Hex or ALT-0 from the Keyboard
 */

/* Project Menu Item IntuiText */
struct IntuiText ProjText[] =
{
    {2, 1, JAM2,2,1, NULL, " New", NULL},
    {2, 1, JAM2,2,1, NULL, " Open...", NULL},
    {2, 1, JAM2,2,1, NULL, " Save", NULL},
    {2, 1, JAM2,2,1, NULL, " Save As...", NULL},
    {2, 1, JAM2,2,1, NULL, "273 Print", NULL},
    {2, 1, JAM2,2,1, NULL, " About...", NULL},
};

```

```

        {2, 1, JAM2, 2, 1, NULL, " Quit", NULL}
    };

/* Project Menu Items */
struct MenuItem ProjItem[] =
{
    /* "New" */
    {&ProjItem[1], 0, 0, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[0], NULL, NULL, NULL, MENUNULL
    },
    /* "Open..." (key-equivalent: 'O') */
    {&ProjItem[2], 0, 10, IWIDTH, IHEIGHT,
     ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[1], NULL, 'O', NULL, MENUNULL
    },
    /* "Save" (key-equivalent: 'S') */
    {&ProjItem[3], 0, 20, IWIDTH, IHEIGHT,
     ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[2], NULL, 'S', NULL, MENUNULL
    },
    /* "Save As..." */
    {&ProjItem[4], 0, 30, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[3], NULL, NULL, NULL, MENUNULL
    },
    /* "Print" (has sub-menu) */
    {&ProjItem[5], 0, 40, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[4], NULL, NULL, &PrtItem[0], MENUNULL
    },
    /* "About..." */
    {&ProjItem[6], 0, 50, IWIDTH, IHEIGHT,
     ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[5], NULL, NULL, NULL, MENUNULL
    },
    /* "Quit" (key-equivalent: 'Q') */
    {NULL, 0, 60, IWIDTH, IHEIGHT,
     ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
     (APTR)&ProjText[6], NULL, 'Q', NULL, MENUNULL
    }
};

/* Menu Titles */
struct Menu Menus[] =
{
    {&Menus[1], 0, 0, 63, 0, MENUENABLED, "Project", &ProjItem[0]},
    {&Menus[2], 70, 0, 39, 0, MENUENABLED, "Edit", &EditItem[0]},
    {NULL, 120, 0, 88, 0, MENUENABLED, "Preferences", &PrefItem[0]},
};

/* A pointer to the first menu for easy reference */
struct Menu *FirstMenu = &Menus[0];

/* Window Text for Explanation of Program */
struct IntuiText WinText[] =
{
    {3, 0, JAM2, 54, 28, &TOPAZ80, "How to do a Menu", NULL},
    {3, 0, JAM2, 70, 38, &TOPAZ80, "(with Style)", &WinText[0]}
};

/* NewWindow structure for our example window */
struct NewWindow NewWindow =
{
    202, 66, 234, 66, 2, 1, MENUPICK|CLOSEWINDOW,
    WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|ACTIVATE|NOCAREREFRESH,
    NULL, NULL, "Menus", NULL, NULL, 0, 0, -1, -1, WBENCHSCREEN
};

/* End of Menus.h */

/* Menus.c */

```

```

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <libraries/dos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifdef LATTICE
#include <proto/all.h>
int CXBRK(void) {return(0);}
#endif

#include "Menus.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define LIB_REV 33

/* prototypes */
UBYTE handleIDCMP(struct Window *);
VOID OpenAll(VOID);
VOID cleanExit(int);

/* prototypes for functions used to determine menu sizing */
BOOL AdjustMenus(struct Menu *, struct TextAttr *);
VOID AdjustItems(struct RastPort *, struct MenuItem *, struct TextAttr *,
    USHORT, USHORT, USHORT, USHORT);
VOID AdjustText(struct IntuiText *text, struct TextAttr *attr);
USHORT MaxLength(struct RastPort *, struct MenuItem *, USHORT);

/* Globals */
struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Window *window = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;

    OpenAll();

    /* Set up the signals that you want to hear about ... */
    signalmask = 1L << window->UserPort->mp_SigBit;

    /* And wait to hear from your signals */
    while( !done )
    {
        signals = Wait(signalmask);
        if(signals & signalmask)
            done = handleIDCMP(window);
    };

    /* Exit the program */
    cleanExit(RETURN_OK);
}

/* Handle the IDCMP messages */
UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    USHORT code, selection, flags;
    struct IntuiMessage *message = NULL;
    ULONG class, menuNum, itemNum, subNum;

    /* Examine pending messages */
    while(message = (struct IntuiMessage *)GetMsg(win->UserPort))
    {
        class = message->Class;
        code = message->Code;
    }
}

```

```

/* When we're through with a message, reply */
ReplyMsg((struct Message *)message);

/* See what events occurred */
switch( class )
{
    case CLOSEWINDOW:
        flag = 1;
        break;
    case MENUPIK:
        selection = code;
        while(selection != MENUNULL)
        {
            menuNum = MENUNUM(selection);
            itemNum = ITEMNUM(selection);
            subNum = SUBNUM(selection);
            flags = ((struct MenuItem *)
                ItemAddress(FirstMenu, (LONG)selection))->Flags;
            printf("Selected ");
            if(flags&CHECKED)
                printf("(Checked) ");
            switch( menuNum )
            {
                case 0: /* Project Menu */
                    switch(itemNum)
                    {
                        case 0:
                            printf("New0);
                            break;
                        case 1:
                            printf("Open0);
                            break;
                        case 2:
                            printf("Save0);
                            break;
                        case 3:
                            printf("Save As0);
                            break;
                        case 4:
                            printf("Print ");
                            switch(subNum)
                            {
                                case 0:
                                    printf("Draft0);
                                    break;
                                case 1:
                                    printf("NLQ0);
                                    break;
                            }
                            break;
                        case 5:
                            printf("About0);
                            break;
                        case 6:
                            printf("Quit0);
                            flag = 1;
                            break;
                        default:
                            break;
                    } /* end switch */
                    break;
                case 1: /* Edit Menu */
                    switch(itemNum)
                    {
                        case 0:
                            printf("Undo0);
                            break;
                        case 1:
                            printf("Cut0);
                            break;
                        case 2:
                            printf("Copy0);

```



```

        break;
    case 3:
        printf("Paste0);
        break;
    case 4:
        printf("Erase All0);
        break;
    default:
        break;
    } /* end switch */
    break;
case 2: /* Preferences Menu */
    switch(itemNum)
    {
        case 0:
            printf("Sound0);
            break;
        case 1:
            printf("Auto Save0);
            break;
        case 2:
            printf("Have Your Cake0);
            break;
        case 3:
            printf("Eat It Too0);
            break;
        default:
            break;
    }
    break;
default:
    break;
    } /* end switch */
    selection = ((struct MenuItem *)ItemAddress
        (FirstMenu, (LONG)selection))->NextSelect;
    } /* end while */
    break; /* case of MENU PICK */
default:
    break;
    } /* end switch */
} /* end while */
return(flag);
}

```

```

/* Open the needed libraries, windows, etc. */
VOID OpenAll(VOID)
{
    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", LIB_REV);
    if(IntuitionBase == NULL)
        cleanExit(RETURN_WARN);

    /* Open the Graphics Library */
    GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library", LIB_REV);
    if(GfxBase == NULL)
        cleanExit(RETURN_WARN);

    /* Open the window */
    window = OpenWindow(&NewWindow);
    if(window == NULL)
        cleanExit(RETURN_WARN);

    /* Give a brief explanation of the program */
    PrintIText(window->RPort, &WinText[1], 0, 0);

    /* Adjust the menu to conform to the font (TextAttr) */
    AdjustMenus(FirstMenu, window->WScreen->Font);

    /* attach the menu to the window */
}

```

```

SetMenuStrip(window, FirstMenu);
}

/* Free up all the resources that we where using */
VOID cleanExit(int returnValue)
{
if(window)
{
/* If there is a menu strip, then remove it */
if(window->MenuStrip)
ClearMenuStrip(window);

/* Close the window */
CloseWindow(window);
}

/* Close the library, and then exit */
if(GfxBase)
CloseLibrary((struct Library *)GfxBase);

if(IntuitionBase)
CloseLibrary((struct Library *)IntuitionBase);
exit(returnValue);
}

/* -----
* The following routines adjust an entire menu system to conform to
* the specified fonts' width and height. Allows for Proportional Fonts.
* This is necessary for a clean look regardless of what the users
* preference in Fonts may be. Using these routines, you don't need to
* specify TopEdge, LeftEdge, Width or Height in the MenuItem structures.
*
* This set of routines does NOT check/correct if the menu runs off the
* screen due to large fonts, too many items, lo-res screen.
*/

BOOL AdjustMenus(struct Menu *firstmenu, struct TextAttr *attr)
{
struct RastPort textrp = {0}; /* Temporary RastPort */
struct Menu *menu;
struct TextFont *font; /* Font to use */
USHORT start, width, height, space;
BOOL retval = FALSE;

/* open the font */
if((font = OpenFont(attr)))
{
SetFont(&textrp, font); /* Put font into temporary RastPort */

width = font->tf_XSize; /* Get the Width of the Font */

/* To prevent crowding of the Amiga key when using COMMSEQ,
* don't allow the items to be less than 8 pixels high.
*/
height = (font->tf_YSize < 8) ? 8 : font->tf_YSize;
height++;

start = 2; /* Set Starting Pixel */

/* Step thru the menu structure and adjust it */
menu = firstmenu;
while(menu)
{
menu->LeftEdge = start;
menu->Width = space =
TextLength(&textrp, menu->MenuName,
(LONG)strlen(menu->MenuName)) + width;
AdjustItems(&textrp, menu->FirstItem, attr, width, height, 0, 0);
menu = menu->NextMenu;
start += (space + (width * 2));
}
CloseFont(font); /* Close the Font */
}
}

```

```

        retval = TRUE;
    }
    return(retval);
}

/* Adjust the MenuItems and SubItems */
VOID
AdjustItems(struct RastPort *txtrp, struct MenuItem *fi,
            struct TextAttr *atr, USHORT width, USHORT hght,
            USHORT lvl, USHORT edge)
{
    struct MenuItem *item = fi;
    register USHORT num;
    USHORT strip_width, sub_edge;

    if(fi==NULL) return;
    strip_width = MaxLength(txtrp, item, width);
    num = 0;
    while(item)
    {
        item->TopEdge = (num * hght) - lvl;
        item->LeftEdge = edge;
        item->Width = strip_width;
        item->Height = hght;
        sub_edge = strip_width - width;
        AdjustText((struct IntuiText *)item->ItemFill, atr);
        AdjustItems(txtrp, item->SubItem, atr, width, hght, 1, sub_edge);
        item = item->NextItem;
        num++;
    }
}

/* Steps thru each item to determine the maximum width of the strip */
USHORT MaxLength(struct RastPort *txtrp, struct MenuItem *fi, USHORT width)
{
    USHORT maxval = 0, textlen;
    struct MenuItem *item = fi;
    struct IntuiText *itext;

    while(item)
    {
        if(item->Flags&COMMSEQ)
        {
            width += (width + COMMWIDTH);
            break;
        }
        item = item->NextItem;
    }
    item = fi;
    while(item)
    {
        itext = (struct IntuiText *)item->ItemFill;
        textlen = itext->LeftEdge +
            TextLength(txtrp, itext->IText,
                (LONG)strlen(itext->IText)) + width;
        /* returns the greater of the two */
        maxval = (textlen < maxval) ? maxval : textlen;
        item = item->NextItem;
    }
    return(maxval);
}

/* Adjust the MenuItems font attribute */
VOID AdjustText(struct IntuiText *text, struct TextAttr *attr)
{
    struct IntuiText *nt;
    nt = text;
    while(nt)
    {

```

```
nt->ITextFont = attr;  
nt = nt->NextText;  
}  
)
```

## Chapter 6

### Intuition: Requesters and Alerts

Requesters are information exchange boxes that can be displayed in windows by the system or by application programs. There are also requesters that the user can bring up on demand. They are called requesters because the user has to “satisfy the request” before continuing input through the window. Alerts are similar to requesters but are reserved for emergency messages.

#### About Requesters

Requesters (see figure) are like menus in that both menus and requesters offer options to the user. Requesters, however, go beyond menus. They become “super menus” because you can place them anywhere in the window, design them to look however you want, and bring them up in the window whenever your program needs to elicit a response from the user—and they come replete with any kind of gadget you care to use. The most fundamental differences between requesters and menus are that requesters *require* a response from the user and that while the requester is in the window, the window locks out all user input. (See the NOISYREQ flag under “Requester Structure” for an exception.) The requirement of a user response is virtually the only restriction placed on your program’s use of requesters.

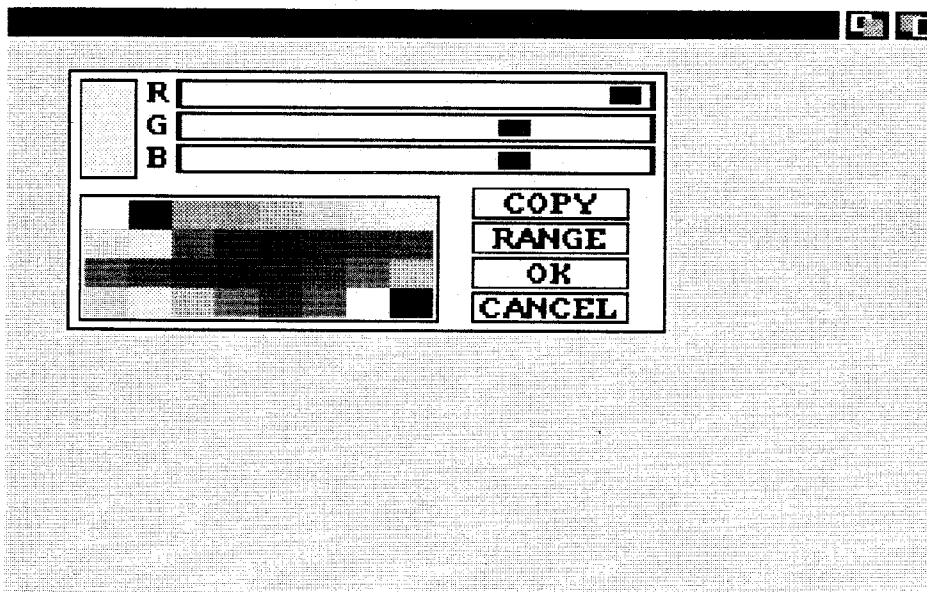


Figure 6-1: Requester Deluxe

## Requester Display

Requesters can be brought up in a window in three different ways.

- System requesters are invoked by the operating system; your program has no control over these. For example, someone using a text editor might try to save a file to disk when there is no disk in the drive. The system requester comes up and makes sure the user understands the situation and requests a response from the user.
- Your program can bring up regular application requesters whenever it needs input from the user.
- You can attach a requester to a double-click of the mouse menu button. Users can bring up this “double-menu request” whenever they need the particular option supplied by the requester.

Once a requester is brought up in a window, all further input to the program from that window is blocked (unless that requester is of type `NOISYREQ`). This is true even if the user brought up the requester. The requester remains in the window and input remains blocked until the user satisfies the request by choosing one of the requester gadgets. You decide which of your gadgets meets this criterion. While the requester is in the window, the only input the program receives from that window is made up of broadcasts when the user selects a requester gadget. Even though the window containing the requester is locked for input, the user can work in another application or even in a different window of your application and respond to the requester later.

A window with an unsatisfied requester is *not* blocked for program output. Nothing prevents your program from writing to the window. Be aware, however, that the requester obscures part of the display. This may hinder the user. Fortunately, there are several ways to monitor the comings and goings of requesters, which your program can use to ensure that it can safely bring up an application requester. (See “IDCMP Requester Features” below.)

In displaying any kind of requester (except the super-simple yes-or-no kind created with `AutoRequest()`), you can specify the location in one of two ways. You can select either a constant location that is an offset from the top left corner of the window or a location relative to the current location of the pointer. Displaying the requester relative to

the pointer can get the user's attention immediately and closely associates the requester with whatever the user was doing just before the requester came up in the window.

You can nest several application requesters in the same window, and the system may present requesters of its own that become nested with the application requesters. These are all satisfied in reverse sequence; the last requester to be displayed must be satisfied first.

## Application Requesters

In adding requesters to your program, you have several options. You can supply a minimum of information and let Intuition do the work of rendering the requester or you can design a completely custom requester, drawing the background, borders, and gadgets yourself and submitting the requester to Intuition for display.

You can select that a requester be rendered by Intuition in one of two ways. If the requester is complex and you want to attach gadgets and have some custom features, you initialize a requester for general usage. In the requester structure, you supply the gadget list, borders, text, and size of the rectangle that encloses the requester. Intuition will allocate the buffers, construct a bit-map that lasts for the duration of the display, and render the requester in the window on demand from your program or the user. Alternatively, if the requester requires only a simple yes or no answer from the user, you can use the special **AutoRequest()** function that builds the requester, displays it, and waits for the user's response.

On the other hand, you can design your own custom requester with your own hand-drawn image for the background, gadgets, borders, and text. You get your own bit-map with a custom requester, so you can design the imagery pixel by pixel if you wish, using any of the Amiga art creation tools. When you have completed the design, you submit it to Intuition for display as usual. Consistency and style are the only restrictions imposed on designing your own requester. The gadgets should look like gadgets and the gadget list should correspond to your images (particularly the gadget select-boxes, to avoid confusing the user).

You should always provide a safe way for the user to back out of a requester without taking any action that affects the user's work. This is *very important*.

A user's action or response to a requester can be as simple as telling the requester to go away. Because the user's action consists of choosing a requester gadget, there must be one or more gadgets that terminate the requester.

## Another Option

As an option to bringing up a requester, you can flash your screen in a complementary color (binary complement, that is—see the "Intuition: Images, Line Drawing, and Text" chapter for an explanation). This is handy if you want to notify the user of an event that is not serious enough to warrant a requester and to which the user does not really need to respond. For instance, the user might be trying to choose an unavailable function from a menu or trying to use an incorrect command-key sequence. If the event is a little more serious, you can flash all the screens simultaneously. See the description of **DisplayBeep()** in the "Intuition: Other Features" chapter.

## RENDERING REQUESTERS

There are two ways of having complex requesters rendered—you can supply Intuition with enough information to do the rendering for you, or you can supply your own completely customized bit-map image. You fill in the **Requester** structure differently according to which rendering method you have chosen.

If you want Intuition to render the requester for you, you need to supply regular gadgets, a pen-color for filling the requester background, and one or more text structures and border structures.

For custom bit-map requesters, you draw the gadgets yourself, so you supply a valid list of gadgets, but the text and image information in the gadget structures can be set to NULL, because it will be ignored. Other gadget information—select-box dimensions, highlighting, and gadget type—is still relevant. The select-box information is especially important since the select-box must have a well-defined correspondence with the gadget imagery that you supply. The basic idea here is to make sure that the user understands your requester imagery and gadgetry. The fields that define borders, text, and pen color are ignored and can be set to NULL.

## REQUESTER DISPLAY POSITION

You can have Intuition display the requester in a position relative to the position of the pointer or as an offset from the upper left corner of the window.

To display the requester relative to the current mouse pointer position, specify **POINTREL** in the **Flag** field and initialize the **RelLeft** and **RelTop** variables, which describe the offset of the upper left corner of the requester from the pointer position. These values can be either negative or positive. The values you supply are only advisory; the actual position will be restricted such that the requester is entirely contained in its window, if possible. The actual top and left positions are stored in the **TopEdge** and **LeftEdge** variables. Then install the requester as a double-menu requester using **SetDMRequest()** (later use **ClearDMRequest()** to remove it). Positioning relative to the mouse pointer is possible only with double-menu requesters.

To display the requester as an offset from the upper left corner of the window, initialize the **TopEdge** and **LeftEdge** variables. These should be positive values.

System requests appear on the Workbench screen by default. They can be made to appear on custom screens by changing the **WindowPtr** field of **Process** structure to point to a **Window** on a custom screen. The original value of **WindowPtr** should be cached and restored before the window is closed. When a system request is posted (using **AutoRequest()** or **BuildSysRequest()**) it will move the screen it appears on to the front, if it is not already there. After satisfying a request, the user may have to rearrange the screens since the order of the screens is not restored to their original state.

## DOUBLE-MENU REQUESTERS

A double-menu requester is exactly like other requesters with one exception: it is displayed only when the user double-clicks the mouse menu button. You give the user the ability to bring up a double-menu requester by calling **SetDMRequest()**. After the user brings up one of these requesters, window input is blocked as if your program or Intuition had brought up the requester. A message stating that a requester has been brought up in its window is entered into the input stream. If you want to stop the user from bringing up a double-menu requester (for instance, if you want to modify it or simply throw it away), you can unlink it from the window by calling **ClearDMRequest()**.

## GADGETS IN REQUESTERS

Each requester gadget should have the **REQGADGET** flag set in its **GadgetType** variable.

Each requester must have at least one gadget that satisfies the request and allows input to begin again. For each gadget that ends the interaction and removes the requester, you set the **ENDGADGET** flag in the gadget **Activation**



field. Every time one of the requester gadgets is selected, Intuition examines the **ENDGADGET** flag; if the flag is set, the requester is erased from the screen and unlinked from the window's active-requester list.

Algorithmic (Intuition-rendered) and custom bit-map requesters differ in how their gadgets are rendered. In algorithmic requesters, you supply regular gadgets just like the application gadgets in windows. In custom bit-map requesters, the gadgets are part of the bit-map that you supply for display. Even in custom bit-map requesters, however, you must supply a list of gadgets, because you must still define the select-box, highlighting, and gadget type for each gadget even though the gadget image information is ignored.

## **IDCMP REQUESTER FEATURES**

If you are using the IDCMP for input, the following IDCMP flags add refinements to the use of requesters:

### **REQVERIFY**

With this flag set, you can make sure that your program is ready to allow a requester to appear in the window. When the program receives a **REQVERIFY** message, it must reply to that message to allow the requester to be rendered.

### **REQSET**

With this flag set, your program will receive a message whenever a requester opens in its window.

### **REQCLEAR**

With this flag set, your program will receive a message whenever a requester is cleared from its window.

You set these flags when you call **ModifyIDCMP()** or create a **NewWindow** structure. See the chapter entitled "Intuition: Input and Output Methods," for further information about these IDCMP flags.

## **A SIMPLE, AUTOMATIC REQUESTER**

For a simple requester that prompts the user for a positive or negative response, you can use the **AutoRequest()** function (see figure). You supply some explanatory text for the body of the requester, negative and positive text to prompt the user's response, the width and height of the requester, and some optional flags for the IDCMP. The positive text is the text you want associated with the user's choice of "Yes," "True," "Retry," and similar responses. Likewise, the negative text is associated with the user's choice of "No," "False," "Cancel," and so on. The positive text is automatically rendered in a gadget in the lower left of the requester, and the negative text is rendered in a gadget in the lower right of the requester. The positive text pointer can be set to **NULL**, specifying that there is no positive choice for the user to make. The IDCMP flags allow either positive or negative external events to satisfy the request. For instance, the positive external event of the user putting a disk in the drive could satisfy the request.

When you call the function, Intuition will build the requester, display it, and wait for a response from the user. If possible, the requester is displayed in the window supplied as an argument to the routine. If not, Intuition opens a window to display the requester.

Requests generated with **AutoRequest()** and **BuildSysRequest()** can be satisfied by the user via the keyboard. The key strokes **left-AMIGA-V** and **left-AMIGA-B** correspond to clicking (with the mouse) on the left and right system

request gadgets, respectively.

### IMPORTANT

Use the function **ModifyIDCMP()** to turn off all verify messages (such as **MENUVERIFY**) before calling **AutoRequest()**. Neglecting to do so can cause situations where Intuition is waiting for the return of a message which the application program is unable to receive because its input is shut off while the requester is up.

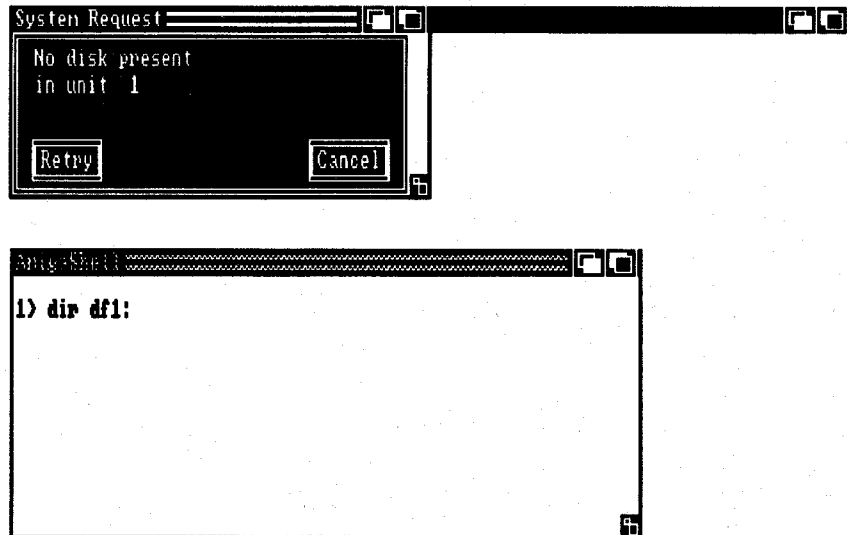


Figure 6-2: A Simple Requester Made with **AutoRequest()**

The **AutoRequest()** function calls **BuildSysRequest()** to construct the simple requester. Your program can call **BuildSysRequest()** directly if you want the program to use the simple requester and to monitor the requester itself. All gadgets created by **BuildSysRequest()** have the following gadget flags set:

**BOOLGADGET**

It is a boolean TRUE or FALSE gadget.

**RELVERIFY**

The program receives a broadcast if this gadget is activated.

**REQGADGET**

This flag specifies that this is a requester gadget.

**TOGGLESELECT**

This flag specifies that this is a toggle-select type of gadget.

### User Rendering

A requester appears in a Layer. You may render to this layer through its **RastPort**, which can be found as **Requester.ReqLayer.rp**. The requester layer is of type "smart", so that your rendering is preserved, but if the window is sized it may damage your work, so that you will need to refresh your rendering. If you specify **NOISYREQ** (explained later) in the requester's **Flags** field, your program can receive **NEWSIZE** messages to let you know when to refresh the requester.

## REQUESTER STRUCTURE

To create a **Requester** structure, follow these steps:

1. Dynamically allocate memory for a **Requester** structure and initialize it with **InitRequester()**. After calling **InitRequester()**, you need fill in only those requester values that your program needs.

*or*

Statically define a **Requester** structure containing the values your program needs. Do *not* call **InitRequester()** with a statically allocated **Requester** structure!

2. Set up a gadget list.
3. Supply a **BitMap** structure if this is a custom requester.

The specification for a **Requester** structure follows.

```
struct Requester
{
    struct Requester *OlderRequest;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT RelLeft, RelTop;
    struct Gadget *ReqGadget;
    struct Border *ReqBorder;
    struct IntuiText *ReqText;
    USHORT Flags;
    UBYTE BackFill;
    struct Layer *ReqLayer;
    UBYTE ReqPad1[32];
    struct BitMap *ImageBMap;
    struct Window *RWindow;
    UBYTE ReqPad2[36];
};
```

Here are the meanings of the fields in the **Requester** structure:

### NOTE

See "Intuition Rendering" and "Custom Bit-Map Rendering" below for information about how the initialization of the structure differs according to how the requester is rendered.

#### OlderRequest

This is a link maintained by Intuition, which points to requesters that were rendered before this one.

#### LeftEdge, TopEdge

Initialize these if the requester is to appear relative to the upper left corner of the window (as contrasted to the POINTREL method, where the requester is rendered relative to the pointer).

#### Width, Height

These fields describe the size of the entire requester rectangle, containing all the text and gadgets.

#### RelLeft, RelTop

Initialize these if the requester is to appear relative to the current position of the pointer. Also, specify POINTREL in the requester's **Flags** field.

**ReqGadget**

This field is a pointer to the first in a linked list of gadget structures.

There must be at least one gadget with the **ENDGADGET** flag set to terminate the requester.

**ReqBorder**

This field is a pointer to an optional **Border** structure for drawing lines around and within your requester.

**ReqText**

This field is a pointer to an **IntuiText** structure containing text for the requester.

**Flags**

You can specify these flags:

**POINTREL**

Specify **POINTREL** if you want the requester to appear relative to the pointer (rather than offset from the upper left corner of your window).

**PREDRAWN**

Specify **PREDRAWN** if you are supplying a custom **BitMap** structure for the requester and **ImageBMap** points to the structure.

**NOISYREQ**

Specify **NOISYREQ** if you do not want the presence of a requester to inhibit input to the window the requester appears in.

Intuition uses these flags:

**REQOFFWINDOW**

Set by Intuition if the requester is currently active but is positioned off-window.

**REACTIVE**

This flag is set or cleared by Intuition as your requesters are posted and removed. The active requester has always been indicated by the value of **Window.FirstRequest**.

**SYSREQUEST**

This flag is set by Intuition if this is a system-generated requester.

**BackFill**

Pen number for filling the requester rectangle before anything is drawn into the rectangle.

**ReqLayer**

This contains the address of the **Layer** structure used in rendering the requester.

**ImageBMap**

This flag is a pointer to the custom bit-map for this requester. If you are not supplying a custom bit-map for this requester, Intuition ignores this variable.

If you are supplying a custom bit-map, you must specify **PREDRAWN** in the requester's **Flags** field.

**RWindow**

This is a system variable.

## ReqPad1, ReqPad2

These are reserved for system use.

The following sections describe the differences in the **Requester** structure between requesters rendered by Intuition and custom-bit-map requesters.

### Requesters in Low-Memory situations

In low-memory situations, system requests such as **AutoRequest()** will change into recoverable alerts (discussed in "Alerts" below).

### Requesters Rendered by Intuition

The following notes apply to requesters rendered by Intuition.

- **ReqGadget** is a pointer to the first in a list of regular gadgets to be rendered in the requester box. Take care not to specify gadgets that extend beyond the Requester rectangle that you describe in the **Width** and **Height** fields, for Intuition does no boundary checking. **REQGADGET** must be specified in the **Gadget's GadgetTypes** field.
- **ReqBorder** is a pointer to a **Border** structure for your requester. The lines specified in this structure can go anywhere in the requester; they are not confined to the perimeter of the requester.
- **ReqText** is a pointer to an **IntuiText** structure. This is for general text in the requester.
- **BackFill** is the pen number to be used to fill the rectangle of your requester before any drawing takes place.

For example, the following **Requester** structure allows Intuition to do the rendering.

```
struct Requester MyRequest =
{
    NULL,                /* OlderRequester maintained by Intuition */
    20, 20, 200, 100,    /* LeftEdge, TopEdge, Width, Height */
    0, 0,                /* RelLeft, RelTop */
    &BoolGadget,          /* First gadget */
    NULL,                /* ReqBorder */
    &MyText               /* ReqText */
    NULL,                /* Flags */
    2,                   /* BackFill */
    NULL,                /* ReqLayer */
    {NULL},              /* pad */
    NULL,                /* BitMap */
    NULL,                /* RWindow */
    {NULL},              /* pad */
};
```

### Custom Bit-Map Rendering

These notes apply to custom bit-map requesters.

- **ReqGadget** points to a valid list of gadgets, which are real gadgets in every way except that the gadget text and imagery information are ignored (and can be **NULL**). The select-box, highlighting, and gadget

type data is still pertinent. The user may get confused unless there is a well-defined correspondence between the gadgets' select-boxes and the requester imagery that you supply.

#### NOTE

Under Amiga system software versions 1.2 and 1.3, Intuition will not render string gadget text in a predrawn requester.

- The **ReqBorder**, **ReqText**, and **BackFill** variables are ignored and can be set to **NULL**.
- The **ImageBMap** pointer points to your own **BitMap** of imagery for this requester.
- You should set the flag **PREDRAWN**.

### THE VERY EASY REQUESTER

Here are the arguments you supply to **AutoRequest()** for the automatic, simple boolean requester that Intuition will build for you:

#### **Window**

This is a pointer to the window in which the requester is to appear.

#### **BodyText**

This is a pointer to an **IntuiText** structure that explains the purpose of the requester.

#### **PositiveText**

This is a pointer to the **IntuiText** structure containing the positive response text.

This field can be **NULL** if there is no positive response.

#### **NegativeText**

This is a pointer to the **IntuiText** structure containing the negative response text.

#### **PositiveFlags**

These are IDCMP flags for positive external events that will satisfy the request.

#### **NegativeFlags**

These are IDCMP flags for negative external events that will satisfy the request.

#### **Width, Height**

These specify the size of the rectangle enclosing the requester.

### REQUESTER FUNCTIONS

A brief rundown of the requester functions follows.

## **The Easy Yes-or-No Requester**

The following function automatically builds, displays, and gets a negative or positive response from a requester:

- **AutoRequest (Window, BodyText, PositiveText, NegativeText, PositiveFlags, NegativeFlags, Width, Height)**

This function builds a requester from the arguments supplied, displays the Requester, and returns TRUE or FALSE.

## **Submitting a Requester for Display**

The following function submits regular requesters to Intuition for display:

- **Request(Requester, Window)**

This function displays a requester in the specified window.

## **Removing a Requester from the Display**

- **EndRequest(Requester, Window)**

This function erases a requester invoked by the user or application and resets the window. It removes only the one requester named. If a requester has one or more gadgets which will satisfy the request, and the user selects one of them, the requester will be removed by the system. If the program needs to cancel the request early, or cancel it only after some specific manipulation of the gadgets, **EndRequest()** should be used.

## **Double-Menu Requesters**

The following functions affect double-menu requesters:

- **SetDMRequest(Window, Requester)**

This function attaches a requester to the double click of the mouse menu button.

- **ClearDMRequest(Window, Requester)**

This function unlinks the requester from the window and disables the ability to bring it up.

## Alerts

Alerts are for emergency messages. There are two types: system alerts and application alerts.

System and application alerts display absolutely essential messages and should be reserved for critical communications in situations that require the user to take some immediate action; for instance, when an application has experienced a fatal error or the system has or is about to crash. System alerts are managed entirely by Intuition (see figure).

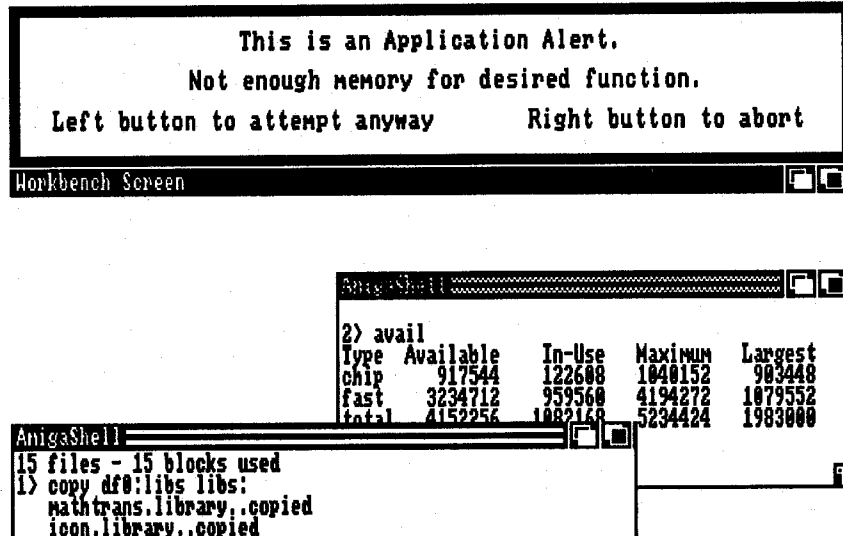


Figure 6-3: The "Out of Memory" Alert

The sudden display of an alert is a jarring experience for the user, and the system stops and holds its breath while the alert is displayed. For these reasons, you should use alerts only when there is no recourse. If you can, use requesters with warning messages instead.

The alert display has a black background and red border, a 640-pixel resolution, and can be as tall as needed to display your text. The alert appears at the top of the video display. If the rest of the display is still healthy, it is pushed down low enough to show the alert. If this is a fatal alert and the system is going down, the alert takes over the entire display.

There are two levels of severity for alerts: `RECOVERY_ALERT`, and `DEADEND_ALERT`.

- `RECOVERY_ALERT` displays your text and flashes the alert's border outline while waiting for the user to respond. This alert is optimistic and presumes that the system can continue operations after the alert is satisfied. It returns `TRUE` if the user presses the left mouse button in response to your message. Otherwise it returns `FALSE`.
- `DEADEND_ALERT` prints your text and returns `FALSE` immediately.



The boolean function **DisplayAlert()** creates and displays an alert message. Your message will most likely get out to the screen regardless of the state of the machine (with the exception of catastrophic hardware failures). If the user presses one of the mouse buttons, the display returns to its original state, if possible. **DisplayAlert()** also displays the Amiga system alert messages. If a recoverable alert cannot be displayed (because memory is low), **DisplayAlert()** will return FALSE, as if the user had selected CANCEL.

**DisplayAlert()** needs three arguments: an **AlertNumber**, a pointer to a string, and a number describing the required display height.

- **AlertNumber** is a LONG value, specifying whether this is a RECOVERY\_ALERT or a DEADEND\_ALERT. (See the *intuition/intuition.h* include file.)
- The **String** argument points to an **AlertMessage** string that is made up of one or more substrings. Each substring contains the following:
  - The first component is a 16-bit x-coordinate and an 8-bit y-coordinate describing where on the alert display you want the string to appear. The units are in pixels. The y-coordinate describes the location of the text baseline.
  - The second component is the text itself. The string must be null-terminated (it ends with a zero byte).
  - The last component is the continuation byte. If this byte is zero, this is the last substring in the message. If this byte is non-zero, there is another substring in this alert message.
- The last argument, **Height**, tells Intuition how many display lines are required for your alert display.

## Examples

### AUTOREQUEST EXAMPLE

The following program shows how to implement an AutoRequester.

```
/* autorequester.c */
/* This program implements an AutoRequester. */
/* Inserting a disk will make the Requester go away. */
/* The user must still click on the close gadget to end the program. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. */
#include "hires.h"
#include "graniteWindow.h"

struct IntuiText reqtext[] = {
    { 1,2,JAM2,20,5,NULL,"An Autorequester",NULL },
    { 0,1,JAM2,5,4,NULL,"YES",NULL },
    { 0,1,JAM2,6,4,NULL,"NOT YET", NULL }
```

```

};

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33

extern VOID cleanExit( struct Screen *, struct Window *, int );
extern UBYTE handleIDCMP( struct Window *);

struct IntuitionBase *IntuitionBase = NULL;

VOID main(VOID)
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;
    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Open the screen */
    screen1 = OpenScreen(&fullHires);
    if (screen1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Make the assignments that were postponed above */

    /* Attach the window to the open screen ... */
    graniteWindow.Screen = screen1;

    /* ... and open the window */
    window1 = OpenWindow(&graniteWindow);
    if (window1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Set up the signals that you want to hear about ... */
    signalmask = 1L << window1->UserPort->mp_SigBit;

    /* Call the functions that do the main processing */
    /* Call the autorequester */
    AutoRequest( window1, &reqtext[0], &reqtext[1], &reqtext[2],
        DISKINSERTED, 0, 200, 50 );

    /* And wait to hear from your signals */
    while( !done ) {
        signals = Wait(signalmask);
        if (signals & signalmask)
            done = handleIDCMP(window1);
    };

    /* Exit the program */
    cleanExit(screen1, window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) ) {
        class = message->Class;

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);
    }
}

```

```

        /* See what events occurred */
        switch( class ) {
            case CLOSEWINDOW:
                flag = 1;
                break;
            default:
                break;
        }
    }
    return(flag);
}

VOID cleanExit( scrn, wind, returnValue )
struct Screen *scrn;
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Close the window and the screen */
    if (wind) CloseWindow( wind );
    if (scrn) CloseScreen( scrn );

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}
/* End of autorequester.c */

```

## DISPLAY ALERT EXAMPLE

The next program shows a display alert. Read the explanation of positioning values for display alert strings in the comment that precedes the `AlertMessage` string. The information there complements that given above.

```

/* displayalert.c */
/* This program implements a recoverable alert */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. */
#include "hires.h"
#include "graniteWindow.h"

/* Each string requires its own positioning information, as explained */
/* in the manual. We use octal notation to specify the positions we */
/* want. Octal numbers start with a backslash and must be three digits */
/* long. For the first line, x = 00360 (two bytes, for 16 bits) */
/* and y = 20 (for one byte, eight bits), and the second line has */
/* x = 00240 and y = 40. */

UBYTE alertMsg[ ] =
{
    " 00360 20OH NO, NOT AGAIN! ",
    " 00240 40PRESS LEFT MOUSE BUTTON TO CONTINUE. 00"
};

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33
#define PAUSE( seconds ) (Delay((seconds) * TICKS_PER_SECOND))

```

```

extern VOID cleanExit( struct Screen *, struct Window *, int );
extern UBYTE handleIDCMP( struct Window *);

struct IntuitionBase *IntuitionBase = NULL;

VOID main(VOID)
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;
    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Open any other required libraries */

    /* Make the assignments that were postponed above */
    alertMsg[21] = NULL;
    alertMsg[22] = 0x01;

    /* Open the screen */
    screen1 = OpenScreen(&fullHires);
    if (screen1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Make the window non-draggable and non-sizable. */
    graniteWindow.Flags &= ~(WINDOWDRAG | WINDOWSIZING);

    /* Attach the window to the open screen ... */
    graniteWindow.Screen = screen1;

    /* ... and open the window */
    window1 = OpenWindow(&graniteWindow);
    if (window1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Set up the signals that you want to hear about ... */
    signalmask = 1L << window1->UserPort->mp_SigBit;

    /* Call the functions that do the main processing */
    /* Delay a bit, so that the Alert does not seem to appear */
    /* at the same time as the window */
    PAUSE( 3L );

    /* Mount the Alert on the display */
    DisplayAlert( RECOVERY_ALERT, alertMsg, 52 );

    /* And wait to hear from your signals */
    while( !done ) {
        signals = Wait(signalmask);
        if (signals & signalmask)
            done = handleIDCMP(window1);
    };

    /* Exit the program */
    cleanExit(screen1, window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */

```

```

while( message = (struct IntuiMessage *)GetMsg(win->UserPort) ) {
    class = message->Class;

    /* When we're through with a message, reply */
    ReplyMsg( (struct Message *)message);

    /* See what events occurred */
    switch( class ) {
        case CLOSEWINDOW:
            flag = 1;
            break;
        default:
            break;
    }
}
return(flag);
}

VOID cleanExit( scrn, wind, returnValue )
struct Screen *scrn;
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Close the window and the screen */
    if (wind) CloseWindow( wind );
    if (scrn) CloseScreen( scrn );

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}
/* End of displayalert.c */

```

## DOUBLE MENU REQUEST EXAMPLE

Here we show how to implement a double-menu requester that appears relative to the mouse pointer. Run the program, and double-click on the menu button. Click on one of the gadgets in the requester that appears, and the requester goes away. The gadgets don't do anything else, but in your programs, you would normally act on IDCMP events triggered by user interaction with your requester.

```

/* dblmenureq.h -- The structures for the Requester and its gadgets. */

#define BACKPEN 0

/* Force text to use Topaz 8 font */
struct TextAttr TOPAZ80 =
    {(STRPTR)"topaz.font", TOPAZ_EIGHTY, 0, 0};

/* Tell something about the program */
struct IntuiText WinText[] =
{
    {3,0,JAM2,95,66,&TOPAZ80,"to activate", NULL},
    {3,0,JAM2,48,56,&TOPAZ80,"Double-Click Menu Button",&WinText[0]}
};

/* Text for the Requester and its gadgets */
struct IntuiText ReqTxt[] =
{
    {3,BACKPEN,JAM2,23,3,&TOPAZ80,"Control Panel",NULL},
    {2,0,JAM1,22,1,&TOPAZ80,"Exit", NULL},
    {2,0,JAM1,22,1,&TOPAZ80,"Fast", NULL},
    {2,0,JAM1,21,1,&TOPAZ80,"Slow", NULL}
};

/* image and mask data for cool buttons */

```

```

USHORT chip MaskData[] =
{
    0x07FF,0xFFFF,0xFFFF,0xFFFF,0xF000,0x3FFF,0xFFFF,0xFFFF,
    0xFFFF,0xFE00,0x7FFF,0xFFFF,0xFFFF,0xFFFF,0xFF00,0xFFFF,
    0xFFFF,0xFFFF,0xFFFF,0xFF80,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
    0xFF80,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFF80,0xFFFF,0xFFFF,
    0xFFFF,0xFFFF,0xFF80,0x7FFF,0xFFFF,0xFFFF,0xFFFF,0xFF00,
    0x3FFF,0xFFFF,0xFFFF,0xFFFF,0xFE00,0x07FF,0xFFFF,0xFFFF,
    0xFFFF,0xF000
};

struct Image Button =
    {0,0,73,10,1,MaskData,0x0001,0x0000,NULL};

/* Mask information for gadgets */
struct BoolInfo ButtonMask = {BOOLMASK,MaskData,0};

/*The buttons for the requester use REQGADGET
 *to indicate that they are gadgets for a
 *requester. ENDGADGET indicates that the
 *requester ends when this button is released.
 */

struct Gadget ReqGad[] =
{
    {NULL, 35,60,73,10,GADGHCOMP|GADGIMAGE,
    RELVERIFY|GADGIMMEDIATE|ENDGADGET|BOOLEXTEND,
    BOOLGADGET+REQGADGET, (APTR)&Button,NULL,&ReqTxt[1],NULL,
    (APTR)&ButtonMask,100,NULL
    },
    {&ReqGad[0],35,40,73,10,GADGHCOMP|GADGIMAGE,
    RELVERIFY|GADGIMMEDIATE|ENDGADGET|BOOLEXTEND,
    BOOLGADGET+REQGADGET, (APTR)&Button,NULL,&ReqTxt[2],NULL,
    (APTR)&ButtonMask,101,NULL
    },
    {&ReqGad[1],35,20,73,10,GADGHCOMP|GADGIMAGE,
    RELVERIFY|GADGIMMEDIATE|ENDGADGET|BOOLEXTEND,
    BOOLGADGET+REQGADGET, (APTR)&Button,NULL,&ReqTxt[3],NULL,
    (APTR)&ButtonMask,102,NULL
    }
};

/* Draw a pretty border around the requester */
SHORT BorderVectors[] =
{
    0, 0,
    148, 0,
    148,84,
    0,84,
    0, 0
};

struct Border ReqBorder = {0,0,1,0,JAM1,5,BorderVectors,NULL};

struct Requester DMRequester =
{
    NULL,
    79,14, /* LeftEdge and TopEdge */
    149,85, /* Width and Height */
    -75,-43,
    &ReqGad[2], /* Gadgets used */
    &ReqBorder, /* Border */
    &ReqTxt[0], /* Text to render within requester */
    POINTREL,
    BACKPEN, /* Color to use as the background */
    NULL,
    NULL,
    NULL
};

/* End of dblmenureq.h */

```

```

/* dblmenureq.c -- This program illustrates a Double-Menu Requester. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
int CXBRK(void) {return(0);}
#endif
/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. */
#include "graniteWindow.h"
#include "dblmenureq.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33

extern VOID cleanExit( struct Window *, int );
extern UBYTE handleIDCMP( struct Window *);

struct IntuitionBase *IntuitionBase = NULL;

VOID main(VOID)
{
    /* Declare variables here */
    ULONG signalmask, signals;
    UBYTE done = 0;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library",INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit(window1, RETURN_WARN);

    /* Make the assignments that were postponed above */
    graniteWindow.Type = WBENCHSCREEN;

    /* Open the window */
    window1 = OpenWindow(&graniteWindow);
    if (window1 == NULL)
        cleanExit(window1, RETURN_WARN);

    /* Display the information about the program */
    PrintIText(window1->RPort,&WinText[1],56,0);

    /* Attach a Double-Menu Requester to this window */
    SetDMRequest(window1, &DMRequester);

    /* Set up the signals that you want to hear about ... */
    signalmask = 1L << window1->UserPort->mp_SigBit;

    /* And wait to hear from your signals */
    while( !done ) {
        signals = Wait(signalmask);
        if (signals & signalmask)
            done = handleIDCMP(window1);
    };

    /* Exit the program */
    cleanExit(window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{
    UBYTE flag = 0;
    struct IntuiMessage *message = NULL;
    ULONG class;

```

```

/* Examine pending messages */
while( message = (struct IntuiMessage *)GetMsg(win->UserPort) ) {
    class = message->Class;

    /* When we're through with a message, reply */
    ReplyMsg( (struct Message *)message);

    /* See what events occurred */
    switch( class ) {
        case CLOSEWINDOW:
            flag = 1;
            break;
        default:
            DisplayBeep(NULL);
            break;
    }
}
return(flag);
}

VOID cleanExit( wind, returnValue )
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Close the window and the screen */
    if (wind) CloseWindow( wind );

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

    exit(returnValue);
}
/* End of dblmenureq.c */

```



## **Chapter 7**

### **Intuition: Input and Output Methods**

#### **An Overview of Input and Output**

From the Intuition point of view, information flows through the system in the following steps (see the following figure):

- Information originates from somewhere in the user's cranial area.
- From there, it flows through biological output devices such as fingers and into electro-mechanical input devices such as keyboards, mice, graphics tablets, and light pens. These input devices create input signals that enter the Amiga through several different ports.
- Inside, these input signals are merged into a coherent stream of input events.
- This input stream is examined and manipulated by several entities, including Intuition. Intuition gazes deeply into the essence of every event it sees. Sometimes it consumes events, other times it adds to the stream, and often it sits lazily by, watching the stream flow through its fourth dimension.
- Finally, application programs receive the input stream and take action based on the data contained therein. The result of the action often involves creating output, which is presented to the user via a video monitor.

- The user's eye input devices detect the information being displayed on the video output device. The eyes, and some still-mysterious merge mechanism, translate the data into signals that are transmitted to the brain, thus completing the cycle.

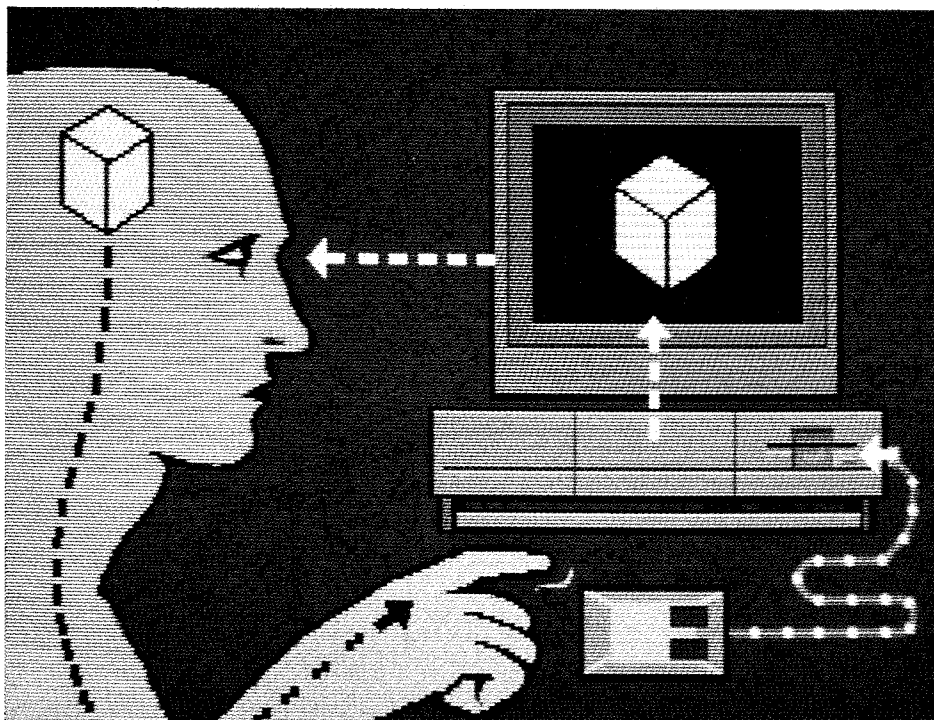


Figure 7-1: Watching the Stream

## About Input and Output

The Amiga has an input device to monitor all input activity, which nominally includes keyboard and mouse activity, but which can be extended to include many different types of input signals. Whenever the user moves the mouse, presses one of the mouse buttons, or types on the keyboard, the input device detects it and constructs an **InputEvent** (a message describing what just occurred). Other devices and programs can ask the input device to construct an input message using their own data (for instance, AmigaDOS is able to generate an input event whenever a disk is inserted or removed, and an application-installed music-keyboard device can add note events to the stream). All of these events are merged into the *input stream*. The input device then broadcasts this input event stream through special message ports so that any interested party can monitor the events, intercept some of the events, and even add new ones to the stream. Intuition is one of the interested parties.

Some of the events, such as “mouse-button pressed,” may have great meaning to Intuition. If they do, Intuition *consumes* them, which is to say that Intuition extracts those events from the input stream. Other events, such as the “disk inserted” event, may be of interest to more than one user of Intuition, so Intuition translates these into a separate message for each application. Still other events, such as most of the keyboard events, mean nothing to Intuition, and Intuition merely passes them along.

A typical application decides what to do from moment to moment by responding to the events in the input stream. Although many applications may be waiting for input simultaneously, in most cases, only the application that Intuition regards as active for input will receive these input stream events. Usually, as described in the "Intuition: Windows," chapter, the user selects which application is active for input by using the Intuition pointer to select that application's window. If your program is the active one, you get to see the input stream events after Intuition has examined them. Your program receives the input stream either directly from Intuition or via another mechanism known as the console device.

Intuition provides two paths for your program to receive messages from the input stream. One is immediate and involves no preprocessing of the data. The other can supply you with standard terminal input functions, buffers, and data representations. The paths are explained below:

- Intuition's Direct Communications Message Ports system (IDCMP) makes standard Amiga Exec message communications easily available for you and gives you input data in its most raw (untranslated) form. This also supplies the only mechanism you have for communicating to Intuition.
- The console device gives you "cooked" input data, including key-code conversions to ASCII and conversions to ANSI escape sequences (Intuition-generated events, such as CLOSEWINDOW, will be translated into escape sequences).

When you want your program to present visual information to the user via your window or screen, you can choose from three methods. The one you choose depends on your particular needs. These three methods are:

- Creating imagery by sending your output directly to the graphics, text, and animation primitives of the Amiga ROM kernel. You can use these for rendering functions like line drawing, area fill, specialized animation, and output of unformatted text. This is the most elementary method.
- Using the Intuition-supplied support functions for rendering text, graphical imagery, and line drawing. These provide many of the same functions as the deeper ROM routines, but these routines do the clerical work of saving, initializing, and restoring states. Also, the image functions provide a new method of object-oriented rendering.
- Outputting text via the console device, which formats text with special text primitives such as `ClearEndOfLine()` and text functions such as automatic line-wrapping and scrolling. For string output, if you want to do anything more than the simplest text rendering, you should use the console device. This gives you nicely formatted text with little fuss.

#### NOTE

The console device is mentioned both as a source for input and as a mechanism for output. It is convenient to do both input and output via the console device only. In particular, text-only programs can open the console and do all their I/O there without ever learning anything about windows, bit-maps, or message ports. Use of the console device for most text-only applications is encouraged, since it requires less work on your part and simplifies the I/O logic of your programs.

If you do not need the console device or are willing to forego its features, it may be better for you to open the IDCMP for input and do your graphics rendering directly through the Intuition and graphics primitives. Under some conditions (for instance, when you have a complex program doing lots of different things), you might want to open both the console device and the IDCMP for input. There is no rule for deciding which mechanism you should use. After you read this chapter, you'll be able to decide for yourself.

The following description of how I/O flow works with (and around) your program is actually a super-simplified model of how system-wide I/O really works, but it is a true representation of I/O at the microcosmic level of your program.

In the illustrations that follow, the input device is found at the top of the diagram. In this device mouse, keyboard, and other input events are merged into a single stream of input events, which is then submitted to Intuition for further processing.

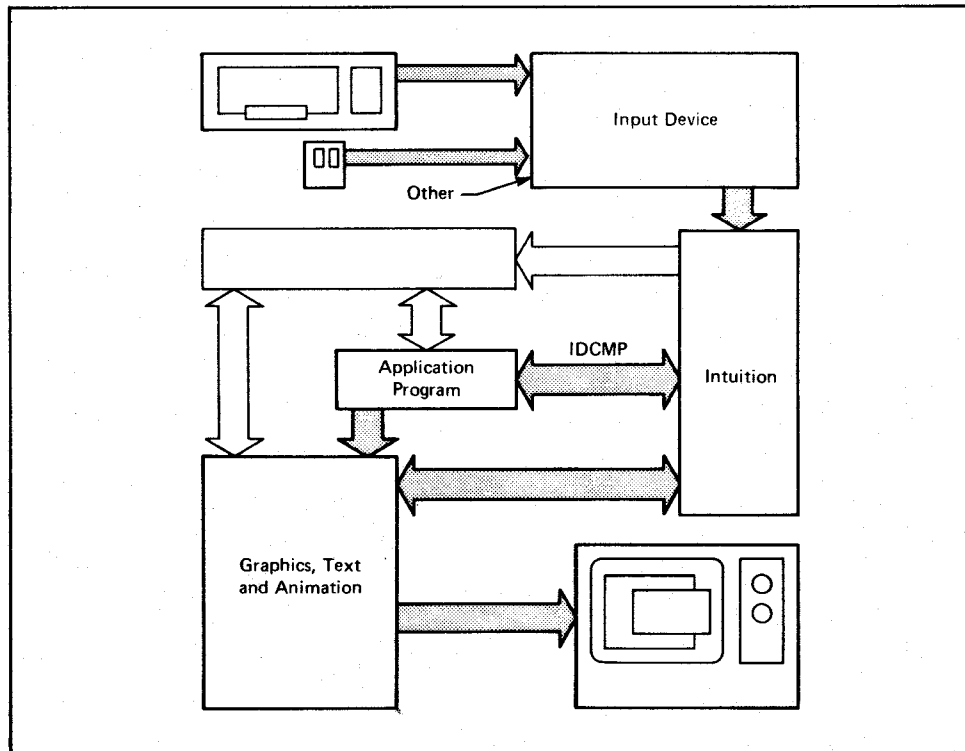


Figure 7-2: Input from the IDCMP, Output through the Graphics Primitives

The above figure shows an example of a program after it has opened the IDCMP. This will be the typical configuration for a CAD package, or other applications that are willing to process input data themselves. The IDCMP allows you to receive only the events that are important to you. Your program can, for instance, learn about gadget events and hear when the user selects the close gadget, but the program may not want to learn about other mouse or keyboard events. If you set up the program to learn about raw keyboard events through the IDCMP, the key codes received come straight from the keyboard to the program. These keycodes are as raw as they get, although the IDCMP also provides the special **Qualifier** field to assist your translations. Alternatively, you can receive keyboard events translated into ASCII (or some other standard). Messages sent via the IDCMP are instances of the structure **IntuiMessage**. When you open the IDCMP, you must monitor the message port supplied by Intuition.

The following figure illustrates the flow of information when the only the console is opened. This will be the typical configuration for text-only applications and applications that want the simplest I/O possible. Refer to the console device chapter for details on opening a console device and performing I/O through it.

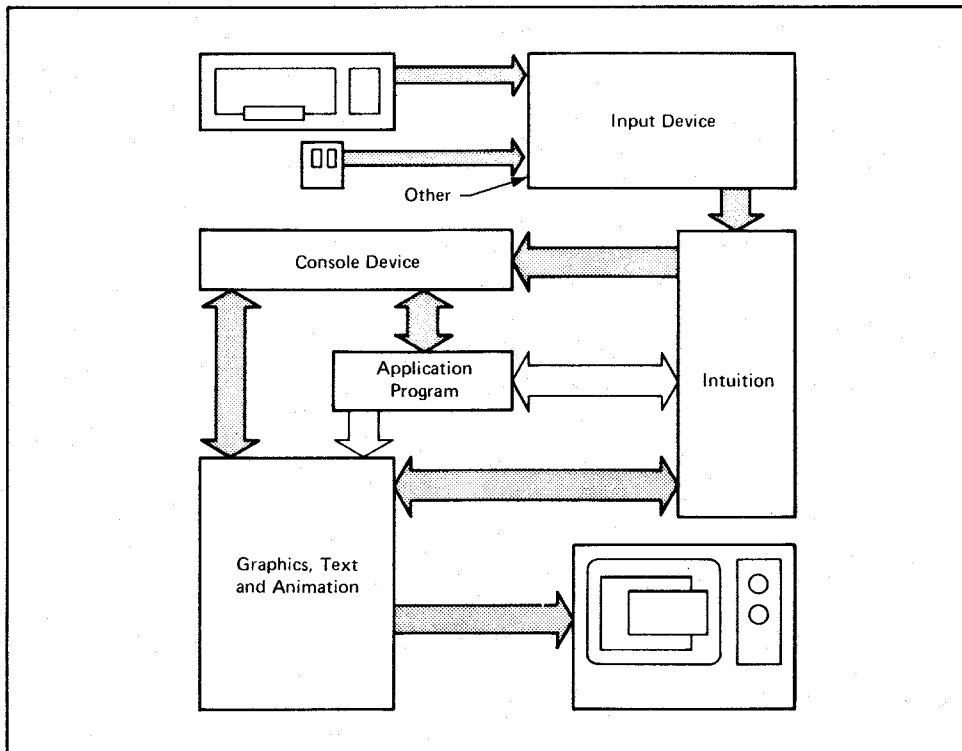


Figure 7-3: Input and Output through the Console Device

The following figure shows a complex program that needs the features of both the console device and the IDCMP. An example might be a program that needs ASCII input and formatted output and the IDCMP verification functions (for example, to verify that it has finished writing to the window before the user can bring up a requester).

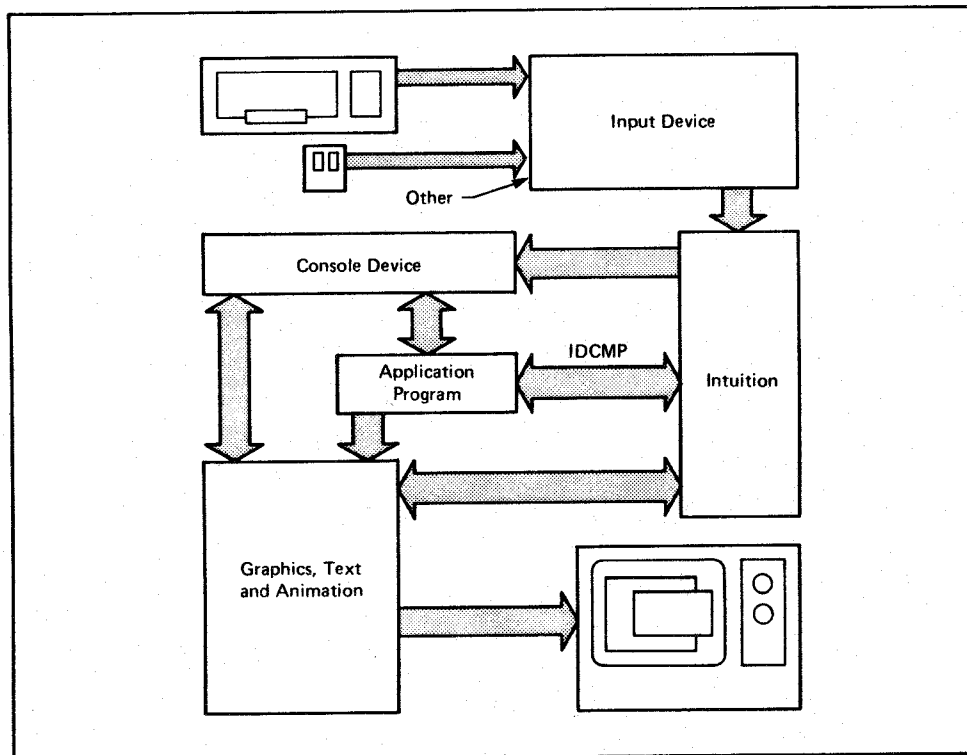


Figure 7-4: Full-system Input and Output (a Busy Program)

The following figure shows an application that has opened a window with neither a console nor an IDCMP. This window gets no input, and the application can write to the window only via the graphics primitives. You might want to do this if your program has opened other windows that do I/O and you want special graphics-only windows (for instance, to monitor RAM usage or watch the clock) that you will close later. If the user selects a window that has no console or IDCMP, further input is discarded until a different window is selected.

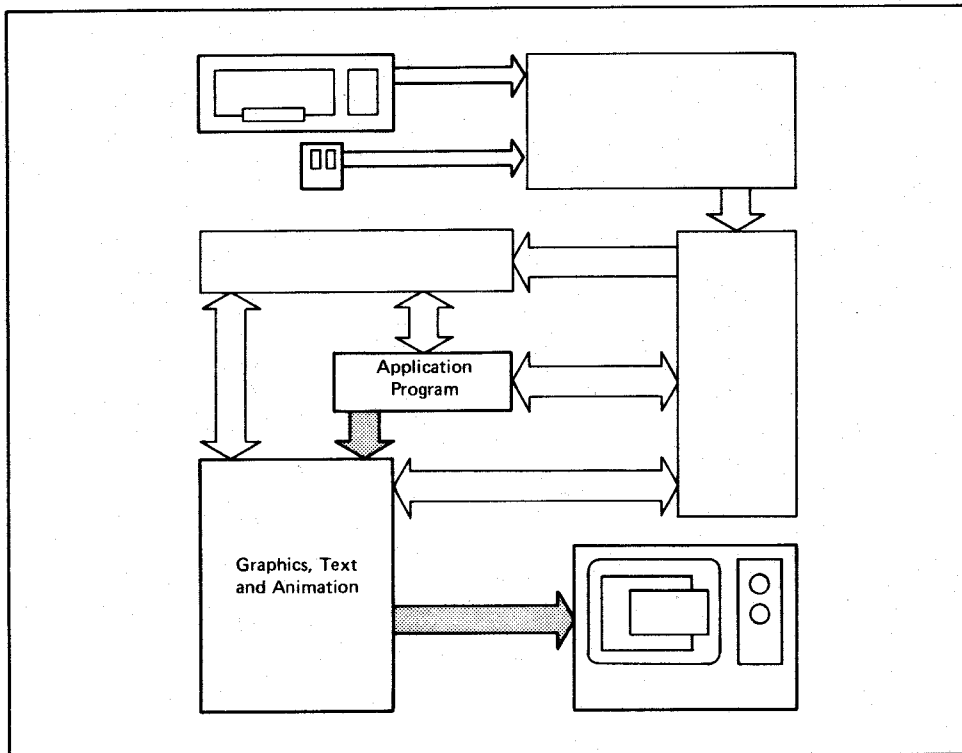


Figure 7-5: Output Only

## Using the IDCMP

The IDCMP ports allow your application and Intuition to talk directly to each other. You can use the IDCMP to learn about mouse, keyboard, and Intuition events without going through the console device. Also, certain useful Intuition features, most notably the verification functions (described under “IDCMP Flags” below), require that the IDCMP be opened, as this is the only mechanism available for communicating to Intuition.

The IDCMP consists of a pair of *message ports*, which may be allocated and initialized by Intuition at your request: one port supplied by you and one port supplied by Intuition. These are standard Exec message ports, used to allow interprocess communications in the Amiga multitasking environment. To open these ports automatically, you set IDCMP flags in the `NewWindow` structure. To open or close them later, you call `ModifyIDCMP()`, which allocates or deallocates message ports or changes which events will be broadcast to your program through the IDCMP. Once the IDCMP is opened, you can receive many different flavors of information directly from Intuition, based on which flags you have set. As with much of Intuition, all of the “grunt work” with message ports is done for you, leaving you free to concentrate on more global issues.

If you have a message port that you have already created, you can have Intuition use that port to communicate with

you. This is described below.

### CAUTION

If you attempt to close the IDCMP, either by calling **ModifyIDCMP()** or by closing the window, without first having **Reply()**'d to all of the messages sent out by Intuition, Intuition will reclaim and deallocate those messages without waiting for a **Reply()** from you. If you attempt to **Reply()** after the close, you will get to watch the Amiga FIREWORKS\_DISPLAY mode.

To learn more about message ports and message passing, see that chapter elsewhere in this manual.

## INTUIMESSAGES

The **IntuiMessage** data type is an **Exec Message** that has been extended to include Intuition-specific information. The **ExecMessage** field in the **IntuiMessage** is used by Exec to manage the transmission of the message. The Intuition extensions of the **IntuiMessage** are used to transmit all sorts of information to your program. Here is what the **IntuiMessage** looks like:

```
struct IntuiMessage
{
    struct Message ExecMessage;
    ULONG Class;
    USHORT Code;
    USHORT Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
};
```

**IntuiMessages** contain the following components:

### ExecMessage

The data in this field is maintained by Exec. It is used for linking the message into the system and broadcasting it to a message port.

### Class

This is a ULONG variable whose bits correspond directly with the IDCMP flags.

### Code

This is a USHORT variable whose bits contain special values, such as menu numbers or special code values, set by Intuition. The meaning of this field is directly tied to the **Class** (above) of this message. Often, there is no special meaning for the code field, and it is merely a copy of the code of the **InputEvent** initially sent to Intuition by the input device. When this message is of class RAWKEY, this field has the raw key code generated by the keyboard device. When this message is of class VANILLAKEY, this field has the translated character.

### Qualifier

This contains a copy of the **ie\_Qualifier** field that is transmitted to Intuition by the input device. This field is useful if your program handles raw key codes, since the **Qualifier** tells the program, for instance, whether or not the SHIFT key or CTRL key is currently pressed. This is a faithful copy of the **ie\_Qualifier** field. Check the **inputevent.h/i** file for the definitions of the qualifier bits.



### **MouseX and MouseY**

Every **IntuiMessage** you receive will have the mouse coordinates in these variables. The coordinates can be either relative to the upper left corner of your window, or expressed as deltas (amount of change since the last reported positions).

### **Seconds and Micros**

These **ULONG** values are copies of the current system clock time in seconds and microseconds. Microseconds range from zero up to one million minus one. The 32 bits allocated to the **Seconds** variable means that the Amiga clock can run for 139 years before wrapping around to zero again.

### **IAddress**

This has the address of some Intuition object, such as a gadget or a screen, when the message concerns, for example, a gadget selection or screen operation.

### **IDCMPWindow**

This contains the address of the window to which this message pertains.

### **SpecialLink**

This is for system use only.

## **IDCMP FLAGS**

You specify the information you want Intuition to send you via the IDCMP by setting the IDCMP flags. You can set them either in the **NewWindow** structure when opening a window or when calling **ModifyIDCMP()** to change the IDCMP specifications. The following is a specification of the IDCMP functions and flags.

### **Mouse flags:**

#### **MOUSEBUTTONS**

This flag causes reports about mouse-button up and down events to be sent to you, if these transitions do not mean something to Intuition. When your program receives a **MOUSEBUTTONS** class of event, it can examine the **Code** field to discover which button was pressed or released. The **Code** field will be equal to **SELECTDOWN**, **SELECTUP**, **MENUDOWN**, or **MENUUP**.

#### **NOTE**

If the user clicks the mouse button over a gadget, Intuition deals with it and your program does not hear about it. Also, the only way your program can learn about menu button events in this way is by setting the **RMBTRAP** flag in the window. See the chapter entitled "Intuition: Windows," for more information.

#### **MOUSEMOVE**

Reports about mouse movements are sent in the form of x and y coordinates. Don't ask for **MOUSEMOVE** unless you are prepared to keep up with a *large* volume of IDCMP messages. If you cannot keep up with, and **ReplyMsg()** to, the volume of messages, Intuition will allocate additional message blocks for your window. Intuition cannot reuse these message blocks until you **ReplyMsg()** them. Therefore, in the extreme case, not keeping up with the **MOUSEMOVE** messages could use up all memory in the system. None would be left over for system housekeeping. Kablooeey! The system crashes.

**ReportMouse()** can be used to toggle on and off the reports of **MOUSEMOVE** events. Also, other

IDCMP messages contain a mouse x and y position.

#### NOTE

This works only if the REPORTMOUSE flag is set in the NewWindow structure or if some gadget is selected with the FOLLOWMOUSE flag set.

Your program will not be sent MOUSEMOVE messages while Intuition has the layers of your screen locked (during menu operations and window sizing/dragging). This avoids problems of messages accumulating while your program is blocked trying to render to a layer which Intuition has locked.

#### DELTAMOVE

When this flag is set, mouse movements are reported as deltas (amount of change from the last position) rather than as absolute positions. This flag works in conjunction with the MOUSEMOVE flag.

#### NOTE

Delta mouse movements are reported even after the Intuition pointer has reached the limits of the display.

If you have this IDCMP flag set, your MOUSEBUTTONS messages will also have relative values, instead of the absolute window position of the mouse.

#### Gadget flags:

##### GADGETDOWN

Your program will receive a message of this class. when the user selects a gadget that was created with the GADGIMMEDIATE flag set.

##### GADGETUP

When the user releases a gadget that was created with the flag RELVERIFY set, your program will receive a message of this class.

##### CLOSEWINDOW

If the user has selected your window's close gadget, the message telling the program about it will be of this class.

#### Menu flags:

##### MENUPICK

This flag indicates that the user has pressed the menu button. If a menu item was selected, the menu number of the menu item can be found in the Code field of the IntuiMessage. If no item was selected, the Code field will be equal to MENUNULL.

##### MENUVERIFY

This is a special verification mode which, like the others, allows your program to confirm that it has finished drawing to your window before Intuition allows the users to start menu operations. This is a special kind of verification, however, in that *any* window in the entire screen that has this flag set will have to respond so that menu operations may proceed. Also, the active window of the screen is allowed to cancel the menu operation. This is unique to MENUVERIFY. Please refer to the "Intuition: Menus" chapter for a complete description.

See the "Verification Functions" section below for some things to consider when using this flag.

#### Requester flags:

##### REQSET

Set this flag to receive a message when the first requester opens in a window.

##### REQCLEAR

Set this flag to receive a message when the last requester is cleared from the window.

##### REQVERIFY

Set this flag if you want your application to make sure that other rendering to its window has ceased before a requester is rendered in the window. This includes requiring the system to get your approval before opening a system requester in your window. With this flag set, Intuition sends the application a message that a requester is pending, and then **Wait()**s for the application to **Reply()** before drawing the requester in the window.

If several requesters open in the window, Intuition asks the application to verify only the first one. After that, Intuition assumes that all output is being held off until all the requesters are gone. You can set the **REQCLEAR** flag to find out when *all* requesters are removed from the window. Once the application receives a message of the type **REQCLEAR**, it is safe to write to the window until another **REQVERIFY** is received. You can also check the **INREQUEST** flag of the window, although this is not as safe a method because of the asynchronous nature of any multitasking environment.

See the “Verification Functions” section below for some things to consider when using this flag.

#### Window flags:

##### NEWSIZE

Intuition sends your program a message after the user has resized the window. After receiving this, the program can examine the size variables in the window structure to discover the new size of the window. The message is sent, even if the size of the window did not actually change.

##### REFRESHWINDOW

A message is sent to the application whenever your window needs refreshing. This flag makes sense only with windows for which the **SIMPLE\_REFRESH** or **SMART\_REFRESH** type of refresh has been selected.

##### SIZEVERIFY

You set this flag if your program is drawing to the window in such a way that the drawing must be finished before the user sizes the window. If the user tries to size the window, a message is sent to the application and Intuition will **Wait()** until the program replies. See the “Verification Functions” section below for some things to consider when using this flag.

##### ACTIVIEWINDOW and INACTIVIEWINDOW

Set these flags to discover when your window becomes activated or deactivated.

#### Other flags:

##### VANILLAKEY

This is the raw keycode **RAWKEY** event translated into the current default character keymap of the console device. In the USA, the default keymap is ASCII characters. When you set this flag, you will get **IntuiMessages** with the **Code** field containing a character representing the key struck on the keyboard. An **IDCMP** message is sent only if the translation results in a single byte, therefore you cannot read such keys as **HELP** or the function keys with **VANILLAKEY**.

Most programs will prefer to use the RAWKEY IDCMP class, and perform their own **RawKeyConvert()**. See also, the **DeadKeyConvert()** example in the chapter entitled, "Intuition: Mouse and Keyboard."

#### RAWKEY

Keycodes from the keyboard are sent in the **Code** field. They are raw keycodes, so you may want the program to process them.

The **Qualifier** field contains the information generated by the input device about this key.

#### NEWPREFS

When the user changes the system Preferences by using the Preferences tool, or when some other routine causes the system Preferences to change, you can make sure your program finds out about it by setting this flag.

When your program gets a message of class NEWPREFS, it can call the procedure **GetPrefs()** to get the new Preferences.

#### NOTE

Everyone who sets this flag will learn about these events, not just the active window.

#### DISKINSERTED and DISKREMOVED

When the user inserts or ejects any disk with any drive, the program will be told about the event if either or both of these flags are set.

#### NOTE

Everyone who sets these flags will learn about these events, not just the active window.

#### INTUITICKS

This gives you simple timer events from Intuition when your window is the active one; it may help you avoid opening and managing the timer device. With this flag set, you will get only one queued-up INTUITICKS message at a time. If Intuition notices that you've been sent an INTUITICKS message and haven't replied to it, another message will *not* be sent. In other words, the INTUITICKS messages are paced: until you reply to one, no subsequent one will be sent to you.

Intuition receives timer events roughly ten times a second. These events are to be used as "prods", and not as time counters.

#### Verification Functions

SIZEVERIFY, REQVERIFY, and MENUVERIFY are exceptional in that Intuition sends an **IntuiMessage** and then waits, by calling the Exec function **Wait()**, for the application to reply that it is all right to proceed. The application replies by calling the Exec message passing function **ReplyMsg()**. The discussion in the "Intuition: Menus" chapter on the MENUVERIFY IDCMP flag also applies to REQVERIFY and SIZEVERIFY.

#### NOTE

A bug in the input handling stream through Intuition may confuse window sizing by the user if your program takes too much time responding to the SIZEVERIFY message. Please respond quickly.

The implication is that the user requested some operation but the operation will not happen immediately and, in fact, will not happen at all until your application says it is safe. Because this delay can be frustrating and intimidating, you should strive to make the delay as short as possible. Your program should always reply to a verification

message as soon as possible.

You can overcome these problems by setting up a separate task to monitor the IDCMP and respond to incoming **IntuiMessages** immediately. This is recommended whenever you are planning heavy traffic through the IDCMP, which occurs when you have set many IDCMP flags.

It is not safe to leave any of the **VERIFY** functions enabled, at a time when your task may not be able to respond for a long period.

It is *not* safe to call AmigaDOS directly ( with **Open()**, for example ), or indirectly ( with **OpenLibrary()**, for a disk-based library, for example ), when a **VERIFY** function is active. If AmigaDOS needs to put up a disk requester for you, Intuition may end up waiting for you to reply to the **VERIFY** message, while your program waits for the AmigaDOS call to finish. This deadlock will freeze the Amiga. *Use **ModifyIDCMP()** to turn off all **VERIFY** messages before calling AmigaDOS.*

If you do set up a separate task to monitor the IDCMP, and you call AmigaDOS functions using some other task, and if the monitor task will always be able to reply to the **VERIFY** message without any help from the other task, then the above warning does not apply.

## SETTING UP YOUR OWN IDCMP MONITOR TASK AND USER PORT

IDCMP communication takes place through a pair of Exec message ports, the **UserPort** (your application's input port, where you wait for messages), and the **WindowPort** (Intuition's window port).

In the simplest case, Intuition allocates (and deallocates) both of these ports when you open a window with non-NULL IDCMP flags or call **ModifyIDCMP()**. If the **WindowPort** is not already opened when one of these functions is called, it will be allocated and initialized. The **UserPort** is checked separately to see whether it is already opened. Intuition will send messages to your program via the **UserPort** and will receive replies via the **WindowPort**. The port variables point to a valid message port if they are opened, and are NULL if not opened.

When Intuition initializes the **UserPort** for you, Intuition calls **AllocSignal()** to get a signal bit. Since your task called **OpenWindow()**, this allocation of a signal is valid for your task. The address of your task is saved into the **SigTask** variable of the message port.

You can choose to supply your own port. You might do this in an environment in which your program is going to open several windows and you want the program to monitor input from all of the windows using only one message port. To supply your own port, do the following:

1. Create a port for your IDCMP by calling **CreatePort(NULL,0)**, which returns a pointer to a port.
2. Open your windows with no IDCMP flags set.
3. Set the window **UserPort** field to the newly created port.
4. Call **ModifyIDCMP(window,FlagsIReallyWant)**. Intuition will use the port you supplied.

**Warning** - When you are sharing an IDCMP among several windows, you must be very careful not to call **ModifyIDCMP(window,NULL)** for any windows that are using the shared port, because Intuition will free the port and the signal bit.

5. When you're through with them, close windows that share an IDCMP by using `CloseWindowSafely()`, as illustrated in the example below. Note - it assumes there is a `UserPort`.

Since at least two windows are sharing a single IDCMP, there can be messages pending for any of the windows, when you decide to close any one of them. It is essential that messages destined for a given window be removed and replied to, before that window is closed. `CloseWindowSafely()` takes care of this for you. Also, it sets the window's `UserPort` to `NULL` so that Intuition knows not to delete the port that you created.

6. Delete the port that you created in step 1, by calling `DeletePort()`.

## Examples

1. This example shows how to receive Intuition events. It reports on a variety of events: a window resizing, a disk insertion and removal, the Select button up and down, and the Menu button up and down.

```
/* quartzWindow.h -- This file implements a rather small window that */
/*                  appears in the right half of the screen.          */

#include "sysgads.h"

#define QUAR_LEFTEDGE 300
#define QUAR_TOPEDGE 50
#define QUAR_WIDTH 200
#define QUAR_HEIGHT 75

struct NewWindow quartzWindow =
{
    QUAR_LEFTEDGE,
    QUAR_TOPEDGE,
    QUAR_WIDTH,
    QUAR_HEIGHT,
    0,1, /* Plain vanilla DetailPen and BlockPen. */
    CLOSEWINDOW, /* Tell program when close gadget has been hit */
    WINDOWCLOSE | SMART_REFRESH | ACTIVATE | WINDOWDRAG |
    WINDOWDEPTH | WINDOWIZING | NOCAREREFRESH,
    NULL, /* Pointer to the first gadget -- */
    /* may be initialized later. */
    NULL, /* No checkmark. */
    "quartzWindow", /* A silly title. */
    NULL, /* Attach a screen later. */
    NULL, /* No bitmap. */
    SYSGADSWIDTH, /* Minimum width. */
    SYSGADSHEIGHT, /* Minimum height. */
    0xFFFF, /* Maximum width. */
    0xFFFF, /* Maximum height. */
    CUSTOMSCREEN /* A screen of our own. */
};

/* End of quartzWindow.h */

/* IDCMPDemo.c -- Tests the IDCMP by printing IDCMP classes */
/*                  to the console. */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
```

```

int CXBRK(void) {return(0);}
#endif

/* Include other required vendor- or Commodore-Amiga-supplied header */
/* files here. */

/* Include user-written header files here. */
#include "quartzWindow.h"

/* Use lowest non-obsolete version that supplies the functions you need. */
#define INTUITION_REV 33

extern VOID cleanExit( struct Window *, int );
extern UBYTE handleIDCMP( struct Window * );

struct IntuitionBase *IntuitionBase = NULL;

VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, signals, moreFlags;
    UBYTE done = 0;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library",INTUITION_REV );

    if (IntuitionBase == NULL)
        cleanExit(window1, RETURN_WARN);

    /* Open any other required libraries */

    /* Make the assignments that were postponed above */
    /* We need a couple more flags in the window */

    quartzWindow.Flags |= REPORTMOUSE | RMBTRAP;
    quartzWindow.Type = WBENCHSCREEN;

    /* Open the window */
    window1 = OpenWindow(&quartzWindow);
    if (window1 == NULL)
        cleanExit(window1, RETURN_WARN);

    /* QuartzWindow has only the CLOSEWINDOW IDCMP flag set. */
    /* We could have set these flags in quartzWindow.IDCMP, */
    /* but instead, we'll set all of them in quartzWindow */
    /* by using the ModifyIDCMP() function. */
    moreFlags = CLOSEWINDOW | NEWSIZE | DISKINSERTED | DISKREMOVED | MOUSEBUTTONS;

    ModifyIDCMP( window1, moreFlags );

    /* Set up the signals that you want to hear about ... */
    signalmask = 1L << window1->UserPort->mp_SigBit;

    /* Call the functions that do the main processing */
    /* None to call in this example */

    /* And wait to hear from your signals */
    while( !done )
    {
        signals = Wait(signalmask);
        if (signals & signalmask)
            done = handleIDCMP(window1);
    };

    /* Exit the program */
    cleanExit(window1, RETURN_OK);
}

UBYTE handleIDCMP( struct Window *win )
{

```

```

UBYTE flag = 0;
struct IntuiMessage *message = NULL;
USHORT code;
SHORT mousex, mousey;
ULONG class;

/* Examine pending messages */
while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
{
    class = message->Class;
    code = message->Code;
    mousex = message->MouseX;
    mousey = message->MouseY;

    /* Always reply to messages as soon as possible. */
    /* We make copies of the interesting fields of */
    /* the message, since we no longer have access */
    /* to them after replying. */
    ReplyMsg( (struct Message *)message);

    /* See what events occurred */
    switch( class )
    {
        case CLOSEWINDOW:
            flag = 1;
            break;
        case NEWSIZE:
            printf( "NEWSIZE0 );
            break;
        case DISKINSERTED:
            printf( "DISKINSERTED0 );
            break;
        case DISKREMOVED:
            printf( "DISKREMOVED0 );
            break;
        case MOUSEBUTTONS:
            switch( code )
            {
                case SELECTUP:
                    printf( "SELECTUP at %d,%d0,mousex,mousey );
                    break;
                case SELECTDOWN:
                    printf( "SELECTDOWN at %d,%d0,mousex,mousey );
                    break;
                case MENUUP:
                    printf( "MENUUP0 );
                    break;
                case MENUDOWN:
                    printf( "MENUDOWN0 );
                    break;
                default:
                    printf( "UNKNOWN CODE0 );
                    break;
            }; /* end of switch on code */
            break;
        default:
            printf( "Unknown IDCMP message0 );
            break;
    } /* End switch */
} /* End while */

return(flag);
}

VOID cleanExit( wind, returnValue )
struct Window *wind;
int returnValue;
{
    /* Close things in the reverse order of opening */

    /* Close the windows */
    if (wind) CloseWindow( wind );

```



```

/* Close the library, and then exit */
if (IntuitionBase) CloseLibrary( (struct Library *)IntuitionBase );

exit(returnValue);
}

```

## 2. This is the CloseWindowSafely() example promised in the last section.

```

/** CloseWindowSafely.c */

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/ports.h>
#include <intuition/intuition.h>

/* this function closes an intuition window that shares a port with
 * other intuition windows.
 *
 * It is careful to set the UserPort to null before closing, and to
 * free any messages that might have been sent.
 */

CloseWindowSafely(win)
struct Window *win;
{
    Forbid();

    /* Send back any unprocessed messages for this window */
    StripIntuiMessages(win->UserPort, win);

    /* Null UserPort so Intuition won't free it */
    win->UserPort = NULL;

    /* Tell Intuition to stop sending more messages */
    ModifyIDCMP(win, 0);

    /* Turn tasking back on */
    Permit();

    /* And really close the window */
    CloseWindow(win);
}

StripIntuiMessages(mp, win)
struct MsgPort *mp;
struct Window *win;
{
    struct IntuiMessage *msg, *succ;

    msg = mp->mp_MsgList.lh_Head;

    while (succ = msg->ExecMessage.mn_Node.ln_Succ)
    {
        if (msg->IDCMPWindow == win)
        {
            /* Intuition is about to rudely free this message.
             * Make sure that we have politely sent it back.
             */
            Remove(msg);
            ReplyMsg(msg);
        }
        msg = succ;
    }
}

```

## Chapter 8

# Intuition: Images, Line Drawing, And Text

Intuition provides two approaches to producing graphics images, lines, and text in displays. For quick and easy rendering, you can use Intuition's high-level data structures and functions. You are also free to use all of the lower-level Amiga graphics, animation, and text primitives.

This chapter shows you how to use the Intuition structures and functions, but the Amiga primitives are a large topic in themselves and the discussion here can only point the way. You will find instructions for using the primitives in the "Graphics Primitives" chapter.

## Using Intuition Graphics

**Images**, **Borders**, and **IntuiText** are the general-purpose Intuition structures for rendering graphics and text into your display. They are called *illustration data types*.

- **Images** are graphic objects of any size and complexity.
- **Borders** are connected lines of any length and number, drawn at any angle, and defining any arbitrary shape.

- **IntuiText** strings can be written in the default font or in a custom font of your own design.

The illustration data types are easy to design and economical to use. They are easy to design because their definitions are brief and flexible. Even though each structure defines a different data type, the data types share a consistency of features and capabilities, so once you have learned one you have pretty much learned them all. This decreases the amount of energy spent in learning new things, and you can reuse the same structures in many places. It also reduces the number of Intuition-internal routines, so we all win.

Each of these illustration data types is located with respect to a *display element*, or *containing element*, which can be any of the primary Intuition components: a window, screen, menu, gadget, or requester. The starting location of an image, border, or text string is defined as an offset relative to some particular pixel, usually the top left corner of the element. Any of the illustration data types can be rendered in any of the display elements. In fact, you can display the same structure in more than one of the elements at the same time.

There are two methods of rendering images, borders, and text into display elements:

- In menus, gadgets, and requesters, you use a pointer field provided in the menu, gadget, or requester structure. Then, as Intuition handles those structures, the illustrations are drawn for you.
- In windows or screens, you draw the illustration types directly into the display element by using one of the functions **DrawImage()**, **DrawBorder()**, or **PrintIText()**.

In the definitions of all three of these general-purpose structures, you supply a top left location that is a relative offset from the top left of the display element that will contain the illustration. These relative offsets allow you to use the underlying data arrays across limitless instances of **Image**, **Border**, or **IntuiText** structures. For example, if you have numerous gadgets of the same size, you can use the same **Border** coordinate pairs to draw a line around each gadget.

An important fact about the illustration elements is that each can point to another of its own kind. You can link many of them together and have them all drawn with just one procedure call.

## DISPLAYING BORDERS, INTUITEXT, AND IMAGES

Requester, gadget, and menu structures contain one or more fields for rendering borders, text, and images. These fields each contain a pointer to an instance of a **Border**, **IntuiText**, or **Image** structure. For drawing the illustration types directly into screens and windows, however, you use the Intuition functions **DrawBorder()**, **DrawImage()**, and **PrintIText()**. You supply a **Border**, **Image**, or **IntuiText** structure as an argument to the function.

These three functions have x and y offsets as arguments which are *added* to the offsets in the graphics structures. Sometimes this extra level of offset can come in handy, especially when positioning as a group a linked list of illustration structures.

For drawing into screens and windows, you also need a pointer into the window or screen **RastPort**. See the "Using the Graphics Primitives" section below.

## CREATING BORDERS

Although this data structure is called a **Border**, it is actually a general-purpose structure for drawing connected lines at any angle and rendering any arbitrary shape made up of groups of connected lines. It is called a border because that is how it started out.

To define a **Border**, you specify the following:

- A set of x and y offsets to the beginning point of the line.
- A set of coordinate pairs for each vertex.
- A color for the lines.
- One of several drawing modes.
- An optional pointer to another instance of **Border**.

### **Border Coordinates**

Intuition draws lines between points that you specify as sets of x,y coordinates. The **Border** variables **LeftEdge** and **TopEdge** contain the offsets of the starting origin of the border with respect to the upper left of the containing element. The **XY** field contains a pointer to an array of coordinate pairs. All of these coordinates are offsets from the starting origin of the border. Thus, you can define one line and use it in different display elements or use it many times in the same element. The first coordinate pair describes the starting point of the first line. Every coordinate pair after the first describes the ending point of the current line and, if there is another coordinate pair, the starting point of the next line.

Here is an example. Consider a gadget whose select box is 140 pixels wide and 80 pixels high. The top left corner of the gadget's select box is located in a window at position (10,5). If the border's (**LeftEdge**, **TopEdge**) coordinates are (10,10), this results in an absolute base position of (10+10,5+10), or (20,15), as shown in figure 9-1.

If the first set of coordinates in the array of coordinates is (0,5), the starting point of the first line will be at (20+0,15+5), or (20,20). If the next coordinate pair is (15,5), the end point of the first line will be at (20+15,15+5), or (35,20). A line will be drawn from absolute position (20,20) to absolute position (35,20). If there is one last coordinate pair, (15,0), the next point is at (20+15,15+0), or (35,15). A second line segment is drawn from (35,20) to (35,15).

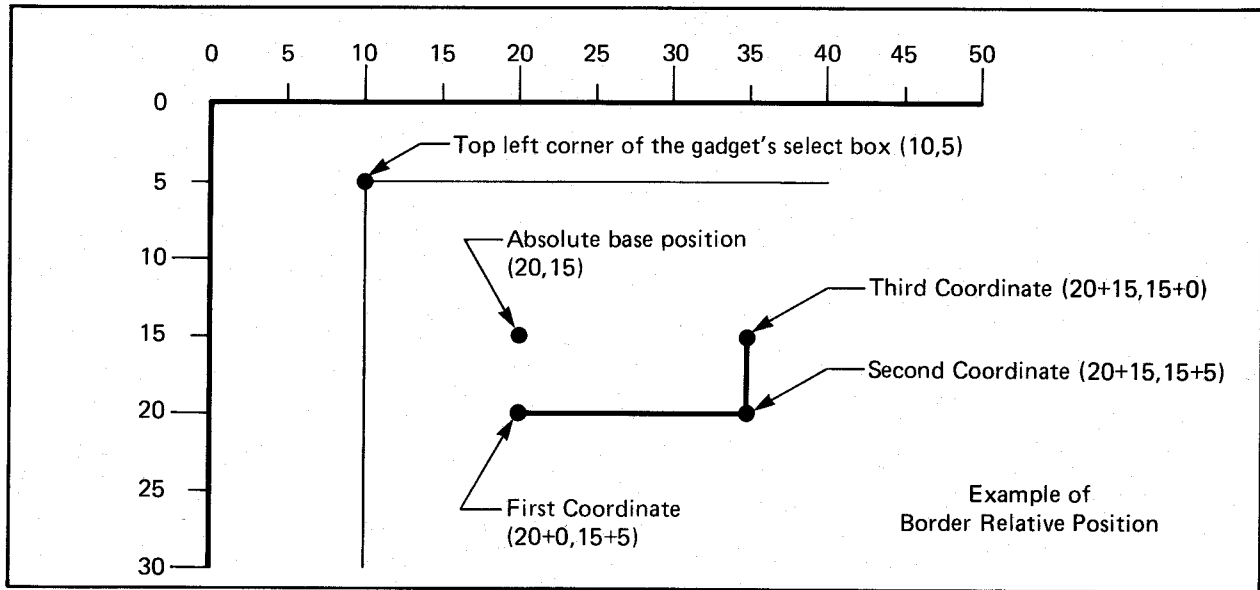


Figure 8-1: Example of Border Relative Position

For a border that is outside the select box of a gadget, you can specify negative offsets. For example, starting position  $(-1, -1)$  for a gadget border is just outside the gadget select box.

### Border Colors and Drawing Modes

Intuition uses the current set of colors in the color register to draw the border and, optionally, to draw its background. As usual, the available colors depend upon the number of bit-planes used in the screen. For instance, if the screen has five bit-planes, then you can select from the colors in color registers 0 through 31. The lines are always drawn in the color in the **FrontPen** field.

Two drawing modes pertain to border lines: **JAM1**, and **COMPLEMENT**. To draw the line in your choice of color, use **JAM1**. You can choose to have the line “invert” the color of the pixels over which it is drawn by selecting the **COMPLEMENT** drawing mode. If you use **COMPLEMENT** mode, for every pixel the line is drawn over, the data bits of the pixel are changed to their binary complement. The complement is formed by reversing all the 0 bits and 1 bits in the binary representation of the color register number. In a three-bit-plane display, for example, color 6 is 110 in binary. If a pixel is color 6, it will be changed to the complement of 001 (binary), which is color 1.

### Linking Borders Together

The **NextBorder** field can point to another instance of a **Border** structure. This allows you to link borders together to describe complex line-drawn shapes. Having multiple borders allows you to draw multiple, distinct groups of lines, each with its own set of line segments and its own color and draw mode. For example, you may want a double border to make a requester stand out more from the surrounding display. You can define the inner border in a second **Border** structure and link it to the first structure by using this field.

## Border Structure Definition

Here is the specification for a **Border** structure:

```
struct Border
{
    SHORT LeftEdge, TopEdge;
    UBYTE FrontPen, BackPen, DrawMode;
    BYTE Count;
    SHORT *XY;
    struct Border *NextBorder
};
```

The meanings of the fields in the **Border** structure are:

### LeftEdge, TopEdge

This field gives the starting origin for the border as an offset from the top left of the containing element. **LeftEdge** is the x coordinate and **TopEdge** is the y coordinate for the top left bit of the image.

### LeftEdge

This field contains the number of pixels from the left edge of the containing element.

### TopEdge

This field specifies the number of lines from the top line of the containing element.

### FrontPen, BackPen, DrawMode

**FrontPen** is the color used to draw the line. The pen color fields contain color registers numbers. **BackPen** is currently unused.

You set the **DrawMode** field to one of the following:

### JAM1

This specification uses **FrontPen** to draw the line and makes no change in the background.

### COMPLEMENT

This specification changes the background beneath the line to its binary complement.

### NextBorder

This field is a pointer to another instance of a **Border** structure. Set this field to NULL if there is no other **Border** structure or if this is the last **Border** structure in the linked list.

**XY** This field is a pointer to an array of coordinate pairs, one pair for each line. These are measured relative to the starting origin for the border.

### Count

This field specifies the number of pairs in the array of coordinate pairs.

## CREATING TEXT

The **IntuiText** structure provides a simple way of writing text strings anywhere in your display. For example, an array of **IntuiText** strings is handy in creating menus.

To define and display **IntuiText**, you specify the following:

- Colors for the text and, optionally, for the text's background.
- One of three drawing modes.
- The starting location for the text.
- The default font or your own special font.
- A pointer to another instance of **IntuiText** (if any).

### Text Colors and Drawing Modes

As with border colors, Intuition uses the current set of colors in the color register to write the text and, optionally, to draw its background. As usual, the available colors depend upon the number of bit-planes used in the screen. For instance, if the screen has five bit-planes, you can select from the colors in color registers 0 through 31. The text is usually drawn in the color in the **FrontPen** field.

Text characters in general are made of two areas: the character image itself and the background area surrounding the character image.

In addition to the two drawing modes for borders, JAM1 and COMPLEMENT, you also have JAM2 and the flag INVERSVID. These modes are described in the following paragraphs.

If you select JAM1 drawing mode, the text character images, but not the character background areas, will be drawn. The character image is drawn in **FrontPen** color. Because the background of a character is not drawn, the pixels of the destination memory around the character image are not disturbed. This is called *overstrike*.

If you select JAM2 drawing mode, the character image is drawn in **FrontPen** and the character background is drawn in the color in the **BackPen** field. Using this mode, you completely cover any graphics that previously appeared beneath the letters.

If the drawing mode is COMPLEMENT, the character is drawn in the binary complement of the colors at its destination. The destination is the display memory where the text is drawn. **FrontPen** and **BackPen** are ignored. To form the complement, you reverse all the 0 bits and 1 bits in the binary representation of the color register number. In a three-bit-plane display, for example, color 6 is 110 in binary. The complement is 001 (binary), which is color 1.

The INVERSVID flag inverses the video for the drawing modes. Where the character image would be nothing is drawn, but the character background is drawn in the color in the **FrontPen** field.

## Linking Text Strings

The **NextText** field can point to another instance of an **IntuiText** structure. Using this field, you can create several distinct groups of characters with one stroke; each group has its own color, font, location, and drawing mode.

## Starting Location

The starting **TopEdge** for a text string is the topmost pixels of the tallest characters. Note that this is different from the baseline of the text. The baseline is the horizontal line on which the characters and punctuation marks rest. The system default fonts are designed to be both above and below the baseline. The descenders of letters (the part of certain letters that is usually below the writing line, like the tail on the lower-case "y") are rendered below the baseline. Therefore, you need to allow for this in drawing text in the display. For more information about text imagery, refer to the *Text* chapter in this manual.

## Fonts

You can use the default font, as set by Preferences, or you can have your own custom font in a **TextAttr** structure and use the **TextAttr** field to point to the custom font. For more information about custom fonts, see the *ext* chapter in this manual.

## IntuiText Structure

Here is the specification for an **IntuiText** structure:

```
struct IntuiText
{
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    SHORT LeftEdge;
    SHORT TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
}
```

The meanings of the fields in the **IntuiText** structure are as follows.

### FrontPen, BackPen

**FrontPen** is the color used to draw the text. **BackPen** is the color used to draw the background for the text, if JAM2 drawing mode is specified.

These fields contain color register numbers.

### DrawMode

This field specifies one of four drawing modes:



#### JAM1

**FrontPen** is used to draw the text; background color is unchanged.

#### JAM2

**FrontPen** is used to draw the text; background color is changed to the color in **BackPen**.

#### COMPLEMENT

The characters are drawn in the complement of the colors that were in the background.

#### INVERSVID

The character is untouched while the background is filled with the color of the **FrontPen**.

#### LeftEdge

This field specifies the starting position for the text as an offset, in pixels, from the left corner of the containing element.

#### TopEdge

This field specifies the starting position for the text as an offset, in pixels, from the top line of the display element.

#### TextAttr

This field is a pointer to a **TextAttr** structure containing your own font description. Set this to NULL if you want the default font.

#### IText

This field is a pointer to the null-terminated text string to be displayed.

#### NextText

This field is a pointer to another instance of **IntuiText**, if this text is part of a linked list of **IntuiTexts**. Set this field to NULL if this text is not part of a list or if it is the last structure in the list.

### CREATING IMAGES

With an **Image** structure you can create graphics objects quickly and easily and display them almost anywhere. Images have an additional attribute that makes them even more economical—with one minor change in the structure, you can display the same image in different colors within the same display element.

To define and display an image, you specify the following:

- The location of the image within the containing element.
- The width and height of the image and the data to create it.
- The depth of the image that is, how many bit-planes are used to define it.
- The bit-planes in the display element that are used to display the image. This determines the colors in the image.

## Image Location

You specify a location for the image that places its top left corner as an offset from the top left corner of the element that contains the image.

## Defining Image Data

To create the data for your image, you write 1s and 0s into a block of 16-bit memory words, which are located at sequentially increasing addresses. When the image is displayed, this sequential series of memory words is organized into a rectangular area, called a bit-plane. The bit-planes in an image are drawn together when the image is displayed.

The color of each pixel in the image is directly related to the value in one or more memory bits, depending upon how many bit-planes there are in the image data and in which bit-planes of the screen or window you choose to display your image.

The color of a given pixel is determined by one or more data bits. Each bit in the pixel is taken from the same position in each of the bit-planes used to define the image. For each pixel, the system combines all the bits in the same position to create a binary value that corresponds to one of the system color registers. This method of determining pixel color is called color indirection, because the actual color value is not in the display memory. Instead, it is in color registers that are located somewhere else in memory.

If an image consists of only one bit-plane and is displayed in a one-bit-plane display, then:

- Wherever there is a 0 bit in the image data, the color in color register 0 is displayed.
- Wherever there is a 1 bit, the color in color register 1 is displayed.

In an image composed of two bit-planes, the color of each pixel is obtained from a binary number formed by the values in two bits, one from bit-plane 0 and one from bit-plane 1. If bit-plane 0 contains all 1s and bit-plane 1 contains 0s and 1s, the pixels derive their colors from register 1 (binary 01) and register 3 (binary 11).

### Note

The actual image data (but not the **Image** structure itself) must be located in chip memory (MEMF\_CHIP). Refer to the "Setting up a Custom Pointer" section in the windows chapter for more information on this.

You create your image data by giving Intuition a series of data words. Before specifying these numbers, you may find it helpful to lay out your image on graph paper, or to use one of the Amiga art tools to assist you. The figure below shows the layout for the system sizing gadget, which is a one-bit-plane image.

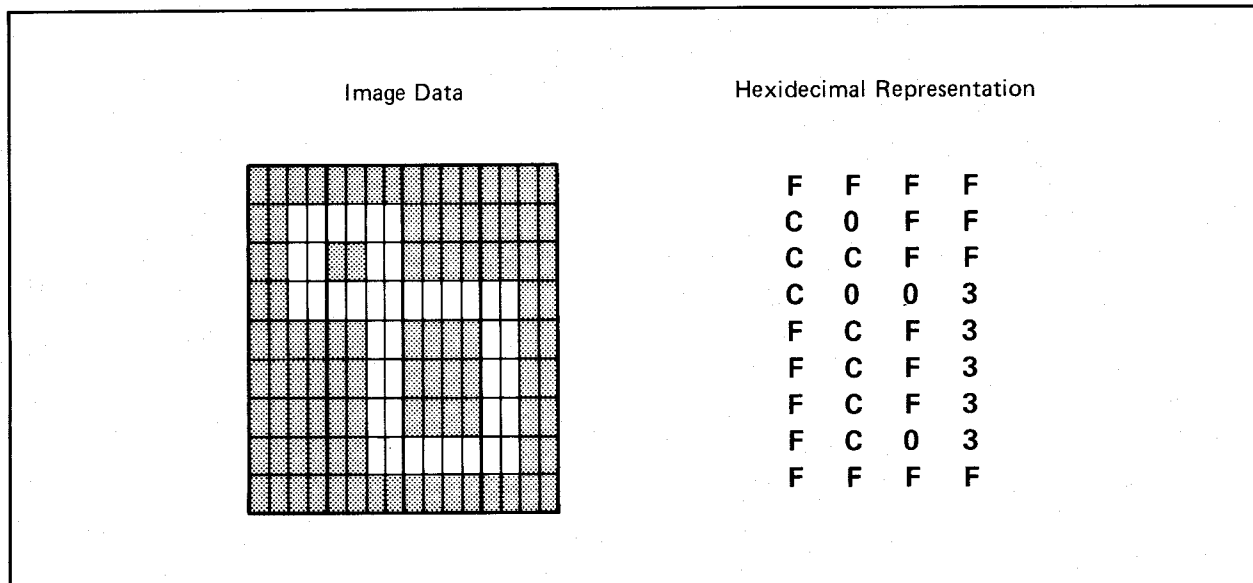


Figure 8-2: Intuition's High-resolution Sizing Gadget Image

In hex notation, the data words of the sizing gadget image are defined as follows:

```
USHORT chip SizeData[] =
{
    0xFFFF,
    0xC0FF,
    0xCCFF,
    0xC003,
    0xFCF3,
    0xFCF3,
    0xFCF3,
    0xFC03,
    0xFFFF,
};
```

In the image data, you need to specify enough whole words to contain the image width. For example, an image 7 bits wide requires one word per line, whereas an image 17 bits wide requires two words per line. In the **Width** field of the **Image** structure, you specify the actual width in pixels of the widest part of the image, not how many pixels are contained in the words that define the image. The **Height** field contains the height of the image in pixels.

Here is the actual **Image** structure of the system-sizing gadget. The last two fields in the structure, **PlanePick** and **PlaneOnOff**, are explained in the next section.

```

struct Image SizeImage =
{
    0, 0,          /* left top */
    16, 9, 1,      /* width, height, depth */
    &SizeData[0],  /* Address
    0x1, 0x0,      /* PlanePick, PlaneOnOff */
    NULL,         /* NextImage */
};

```

## Picking Bit-Planes for Image Display

An image may actually contain fewer bit-planes than the display element it is rendered in. This gives you great flexibility in using **Image** structures. You can:

- Draw an image into a screen or window of any depth (if you have designed it properly).
- Make one image and display it in different colors.
- Minimize the amount of memory needed to define a simple image that is destined for a display of multiple bit-planes.

**PlanePick** “picks” the bit-planes of the containing window or screen **RastPort** that will receive the bit-planes of the image. **PlaneOnOff** specifies what to do with the window or screen bit-planes that are not picked to receive image data. For each display element plane that is “picked” to receive data, the next successive plane of image data is drawn there. For every bit-plane not picked to receive image data, you tell Intuition to fill the plane with 0s or 1s. For both variables, the binary form of the number you supply has a direct correspondence to the bit-planes of the window or screen containing the image. The lowest bit position corresponds to the lowest-numbered bit-plane. For example, for a window or screen with three bit-planes (consisting of Planes 0, 1, and 2), all the possible values for **PlanePick** or **PlaneOnOff** and the planes picked are as follows.

PlanePick or PlaneOnOff	Planes Picked
000	No planes
001	Plane 0
010	Plane 1
011	Planes 0 and 1
100	Plane 2
101	Planes 0 and 2
110	Planes 1 and 2
111	Planes 0, 1, and 2

The system sizing gadget shown above has only one bit-plane of data. To display this gadget in plane 0 of a four-bit-plane window using color 1 for the image and color 0 for its background, you set **PlanePick** to 0001 (binary) and **PlaneOnOff** to 0000 (binary). These settings give Intuition the following instructions:

- Display the data that describes the image in plane 0 of the destination **RastPort**.

- For all of the other planes in the **RastPort**, set the bits in the area where the image is displayed to 0.

The following figure illustrates the discussion in the preceding paragraphs.

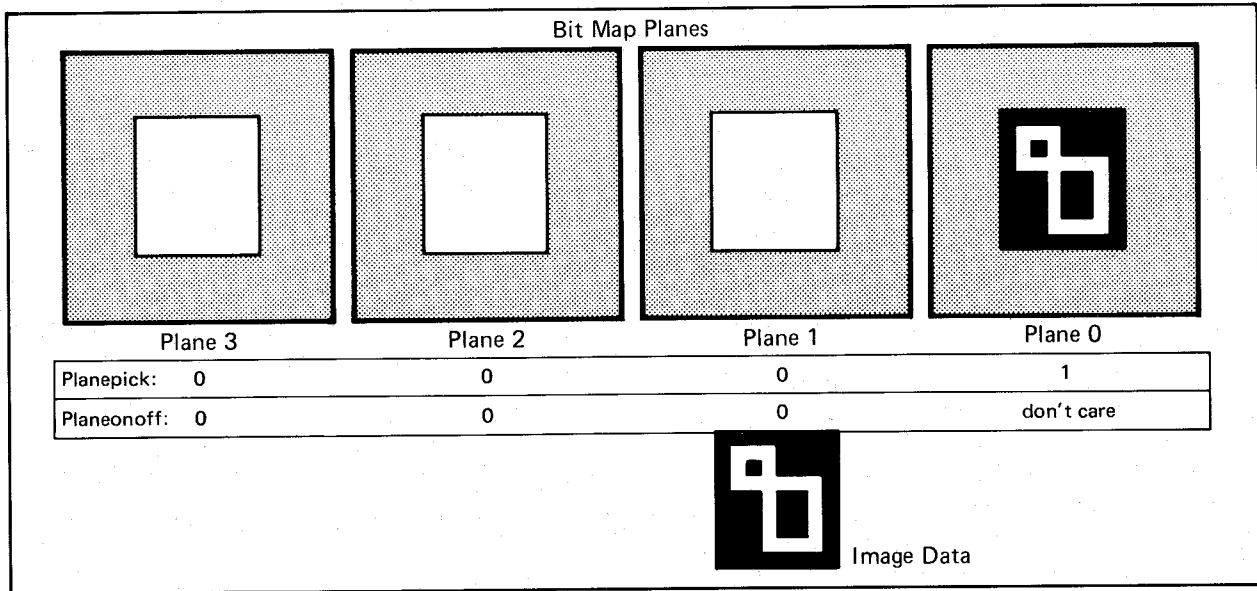


Figure 8-3: Example of PlanePick and PlaneOnOff

If you want the sizing gadget to be drawn in color 2 and its background drawn in color 0, you need to define pixels whose values are 0010 and 0000. To do this, simply change **PlanePick** to 0010.

If you want color 3 for the sizing gadget and color 1 for its background, you need to define pixels with values 0011 and 0001. Therefore, plane 1 defines the image and plane 0 has to be all 1s. You can achieve this by setting **PlanePick** to 0010 and **PlaneOnOff** to 0001.

If you want an image that is simply a filled rectangle, you need not supply any image data at all. You specify a **Depth** of zero, set **Width** and **Height** to any size you like, and set **PlanePick** to 0000 since there are no planes of image data to pick. Then, set **PlaneOnOff** to the color you want for the rectangle. To see how a gadget like this looks, refer to the “Requester Deluxe” illustration in the “Intuition: Requesters and Alerts” chapter.

## Image Structure

Here is the specification for an **Image** structure:

```

struct Image
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height, Depth;
    USHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
};

```

The meanings of the fields in the **Image** structure are:

#### **LeftEdge, TopEdge**

These are offsets from the top left of the display element.

#### **LeftEdge**

This field contains the number of pixels from the left edge of the display element.

#### **TopEdge**

This field contains the number of lines from the top line of the display element.

#### **Width**

This field contains the width of the actual image in pixels.

#### **Height, Depth**

These fields specify the height of the image in pixels and the number of bit-planes needed to define the image.

#### **ImageData**

This field is a pointer to the actual bits defining the image.

#### **PlanePick, PlaneOnOff**

**PlanePick** tells which planes of the containing element you pick to receive planes of image data.  
**PlaneOnOff** tells what to do about the planes that are not picked.

These fields are a bit-wise representation of bit-plane numbers.

### **Image Example**

A more complex example of an image is presented below. The image shown in figure 9-4 belongs to one of the system depth-arrangement gadgets (the front gadget, which brings a window or screen to the front of the display).

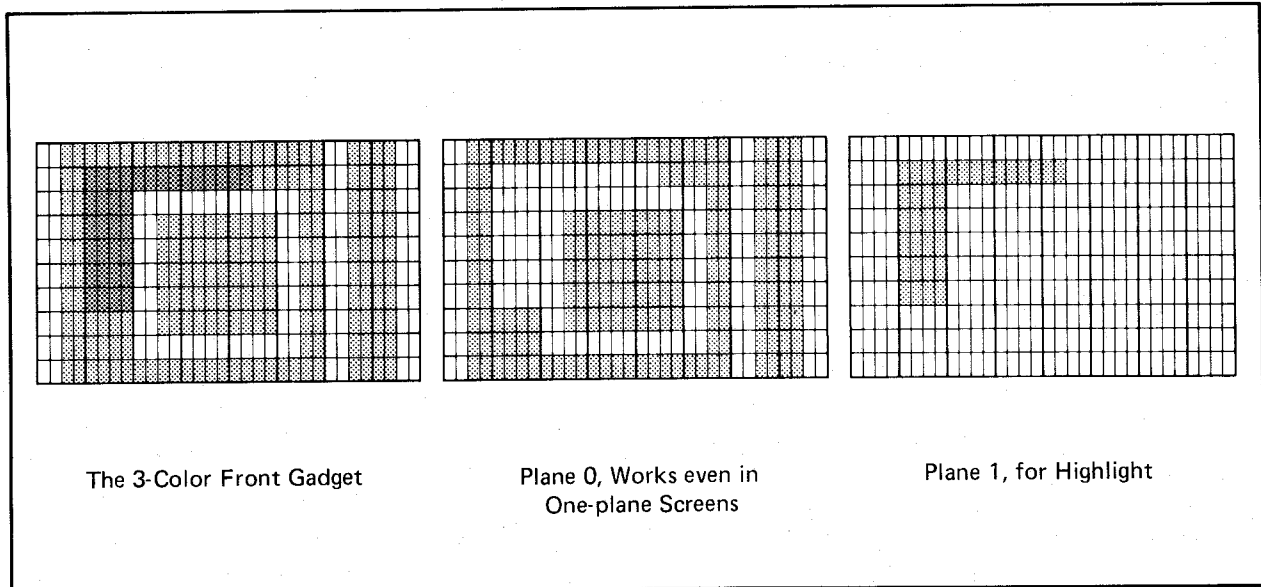


Figure 8-4: Example Image — the Front Gadget

Its data structure and data definition look like this:

```
USHORT chip UpFrontData[] =
{
    0x3FFF, 0xFF3C,
    0x3000, 0x3F3C,
    0x3000, 0x033C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x303F, 0xF33C,
    0x3F3F, 0xF33C,
    0x3F00, 0x033C,
    0x3FFF, 0xFF3C,
    /**/
    0x0000, 0x0000,
    0x0FFF, 0xC000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0F00, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
};
```

```

struct Image UpFImage =
{
    0, 0,                /* left top */
    29, 10, 2,          /* width, height, depth */
    &UpFrontData[0],    /* image data */
    0x3, 0x0,           /* PlanePick, PlaneOnOff */
    NULL,               /* NextImage */
};

```

This gadget was designed to look good in a window or screen of any depth. **PlanePick 0x3 (000011)** picks planes 0 and 1 of the destination **RastPort** for planes 0 and 1 of the gadget. If this gadget is displayed in a window or screen of depth 1, only plane 0 of its data is displayed. Color 0 is used for the background and color 1 for the imagery.

If this gadget is displayed in a window or screen of depth 2 or more, both planes are displayed. The resulting colors are 0 for the background and 1 and 2 for the imagery.

## Image Memory

Just as for sprite data, the image data has to be in chip memory. Refer to the “Setting up a Custom Pointer” section of the windows chapter for more information on this.

## INTUITION GRAPHICS FUNCTIONS

The following are brief descriptions of the Intuition functions that relate to the use of the Intuition illustration data types and the Amiga graphics primitives.

### Drawing Images, Lines, or Text in a Window or Screen

- **DrawImage (RPort, Image, LeftOffset, TopOffset)**

This function renders the **Image** data into the **RastPort** of the screen or window.

- **DrawBorder (RPort, Border, LeftOffset, TopOffset)**

This function draws the vectors of the **Border** into the window or screen **RastPort**.

- **PrintIText (RPort, IText, LeftOffset, TopOffset)**

This function prints **IntuiText** into the window or screen **RastPort**.

### Obtaining the Width of a Text String

- **IntuiTextLength (IText)**

This function returns the width of an **IntuiText** in pixels.



## Obtaining the Address of a View or ViewPort

- **ViewAddress()**

This function returns the address of the Intuition **View** structure for any graphics, text, or animation primitive that requires a pointer to a **View**.

- **ViewPortAddress (window)**

This function returns the address of the screen **ViewPort** associated with the specified window for any graphics, text, or animation primitive that requires a pointer to a **ViewPort**.

## Chapter 9

# Intuition: Mouse and Keyboard

In the Intuition system, the mouse is the normal method of making selections. This section describes how users employ the mouse to interact with the system and your programs and how you can arrange for your program to use the mouse in other ways. It also describes the use of the keyboard as an alternate means of input.

### About the Mouse

A mouse is a small, hand-held input device connected to the Amiga by a flexible cable. By rolling the mouse around on a smooth surface, the user can input horizontal and vertical position coordinates to the computer. The mouse also provides a pair of input keys, called *mouse buttons*, for the user to input further information to the computer.

Most of the things the user does with the mouse are meaningful to Intuition. Because of this, Intuition monitors mouse activity closely. As the user moves the mouse, Intuition follows the motion by changing the position of the Intuition *pointer*. The Intuition pointer is an image (using hardware sprite 0) that can move around the entire video display, mimicking the user's movement of the mouse. The user can use the mouse and pointer to point at some object and then have some action performed on that object. Typically, users specify an action by manipulating either or both mouse buttons. Users can also position the mouse while the buttons are activated.

The basic mouse activities are shown in the table below.

Table 9-1: Mouse Activities

Action	Explanation
Pressing a button	Positioning the pointer while holding down a button. The action specified by the position of the pointer can continue to occur until the button is released, or alternatively may not occur at all until the button is released.
Clicking a button	Positioning the pointer and quickly pressing and releasing one of the mouse buttons.
Double-clicking a button	Positioning the pointer and pressing and releasing a mouse button twice.
Dragging	Positioning the pointer over some object, pressing a button, moving the mouse to a new location, and releasing the button.

The left mouse button is most often used for *selection*. The right mouse button is most often used for *information transfer*. The terms selection and information are intentionally left open to some interpretation, as it is impossible to imagine all the uses you will find for the mouse buttons. The selection/information paradigm can be crafted to cover most interaction between the user and your program. You are encouraged, when designing mouse usage, to emphasize this model. It will help the user to understand and remember the elements of everyone's design.

When the user presses the left button, Intuition examines the state of the system and the position of the pointer. Intuition uses this information to decide whether or not the user is trying to select some object, operation, or option. For example, the user positions the pointer over a gadget and then presses the left button to select that gadget. Alternatively, the user may position the pointer over a window and press the select button to activate the window. If the user moves the mouse while holding down the select button, this sometimes means that the user wants to select everything that the pointer moves over while the button is still pressed.

The right mouse button is used to initiate and control information-gathering processes. Intuition uses this button most often for menu operations. Pressing the right button usually displays the active window's menu bar over the screen title bar. Moving the mouse while holding down the right button sometimes means that the user wishes to browse through all available information; for example, browsing through the menus. Double-clicking the right mouse button can bring up a special requester for extended exchange of information. This requester is called the double-menu requester, because of the double-click of the menu button required to reveal it, and because this requester is like a super menu through which a complex exchange of information can take place. Because the requester is used for the transfer of information, it is appropriate that this mechanism is called up by using the right button.

Your program can receive mouse button and mouse movement events directly. If you are planning to handle mouse button events yourself, you should continue the selection/information model used by Intuition.

You can combine mouse button activations and mouse movement to create compound instructions. Here is an example of how Intuition combines multiple mouse events. While the right button is pressed to reveal the menu items of the active window, the user can press the left button several times to select more than one option from the menus. Also, you can allow the user to move objects or select multiple objects by moving the mouse while holding

down the buttons. As another example, consider the Workbench tool. To move an object on the Workbench screen, the user places the pointer within the object's icon, presses the left button, and moves the pointer. When the icon is in the desired location, the user releases the button.

Dragging can have different effects, depending on the object being dragged. To move a window to another area of the screen, the user positions the pointer within the window's drag gadget and drags the window to a new position. To change the size of a window, the user positions the pointer within the size gadget and drags the window to some smaller or larger size. In drag selection, the user can hold down both buttons while in menu mode and move the pointer across the menu display, making multiple selections with one stroke.

## Mouse Messages

Mouse events are broadcast to your program via the IDCMP or the console device. See the "Intuition: Input and Output Methods" chapter for information on how to receive communications.

Simple mouse button activity not associated with any Intuition function will be reported in **IntuiMessages** as the event class **MOUSEBUTTONS**, with the codes **SELECTDOWN**, **SELECTUP**, **MENUDOWN**, and **MENUUP** to specify changes in the state of the left and right buttons, respectively. Mouse button activity over your gadgets is reported with a class of **GADGETDOWN** or **GADGETUP**, and the **IAddress** field (or **EventAddress** field of **InputEvents**) has the address of the selected gadget. Menu selections appear with a class of **MENUPICK**, with the menu number in the **Code** field.

Your program receives mouse position changes in the event class **MOUSEMOVE**. The **MouseX** and **MouseY** position coordinates describe the position of the mouse relative to the upper left corner of your window. These coordinates are always in the resolution of the screen you are using, and may represent any pixel position in your screen, even though the hardware sprites can be positioned only on the even-numbered pixels of a high-resolution screen and on the even-numbered rows of an interlaced screen.

To get mouse movement reported as deltas (amount of change from the last position) instead of as absolute positions, you can use the IDCMP flag, **DELTAMOVE**.

## About the Keyboard

A program can receive keyboard data through an IDCMP port by setting the **RAWKEY** or **VANILLAKEY** flags. **VANILLAKEY** events provide for simple ASCII text and standard control keys like space, return and backspace. **RAWKEY** events provide a more complex input stream, which the program must process to generate ASCII data. **RAWKEY** returns all keycodes, both key-up and key-down, including function keys.

### NOTE

Keystrokes do not always come in key-down/key-up pairs. For example, repeating keys appear as a sequence of key-down messages.

The example at the end of this chapter uses `RawKeyConvert()` to convert the RAWKEY input stream into an ANSI input stream. See the “Console Device” chapter for more information on `RawKeyConvert()` and the data it returns.

#### NOTE

If `Intuition()` responds to any input events, then your program will not see them. This happens for system shortcuts (Left-AMIGA + key), and menu shortcuts (Right-AMIGA + key) if the menu shortcut is defined for the active window.

The Amiga keyboard has several special command keys, which are listed in the following table. These keys are used to modify the meaning of other keys. When you receive a RAWKEY or VANILLAKEY event (or, in fact, any other event) through the IDCMP, the input message’s **Qualifier** field contains the status of all of the keyboard special keys. Since all messages contain the qualifier bits, the program does not have to track the state of the special command keys. For any message, a program can examine the **Qualifier** field to quickly determine the state of these keys.

These special command keys (and their flags) are shown in the table.

Table 9-2: Special Command Keys

Key	Label	Explanation
<i>control</i>	CTRL	The associated <b>Qualifier</b> flag is the <b>CONTROL</b> flag.
<i>alternate</i>	ALT	NOTE: that there are two separate ALT keys, one on each side of the space bar. These can be treated distinctly. Your program can detect which one was pressed by examining the <b>LALT</b> and <b>RALT</b> commands for the Left ALT and Right ALT keys respectively
<i>escape</i>	ESC	When this key is struck, its keycode is entered into the input stream as an actual keystroke.
<i>function</i>	F1 to F10	Shortcut methods for entering command-key sequences starting with the ESC key.
<i>AMIGA</i>	Fancy A	There are two Amiga keys, one on each side of the space bar. These, like the ALT keys, are distinctly identifiable. The Left AMIGA key is recognized by the <b>Qualifier</b> flag <b>LCOMMAND</b> , and the Right AMIGA key by <b>RCOMMAND</b> .

Certain command-key sequences starting with one of the AMIGA keys have special meaning to Intuition. Most notably, these involve shortcuts and alternatives to using the mouse, as described in the following section.

## Using the Keyboard as an Alternate to the Mouse

All Intuition mouse activities can be emulated using the keyboard, by combining the Amiga command keys with other keystrokes.

The pointer can be moved by pressing down either AMIGA key along with one of the four cursor keys (the ones with the arrows). The longer these keys are held down, the faster the mouse will move. Also, you can hold down either SHIFT key to make the pointer leap greater distances.

To emulate the left mouse button, users can press the left ALT key and the left AMIGA key simultaneously. To emulate the right mouse button, users can press the right ALT key and the right AMIGA key simultaneously. These key combinations permit users to make gadget selections and perform menu operations using the keyboard alone. This will be a boon for mouse-haters.

There are a number of special shortcut functions supported by Intuition. These involve holding down the Left AMIGA key and simultaneously pressing a another key. These functions allow the user to do such things as move the Workbench Screen to the front using the keyboard. See the "Intuition Style" chapter for more information.

### NOTE

These functions emulate left mouse button and mouse movement operations. Also note that Intuition always consumes these two command-key sequences for its own use. That is, it always detects these events and removes them from the input stream. Your program will no see these events.

You can pair up menu items with command-key sequences to associate certain letters with specific menu item selections. This gives the user a shortcut method to select often-used menu operations, such as UNDO, CUT, and PASTE. Whenever the user presses the right AMIGA key with some alphanumeric key, the menu strip of the active window is scanned to see if there are any command-key sequences in the list that match the sequence entered by the user. If there is a match, Intuition translates the key combination into the appropriate menu item number and transmits the menu number to the application program.

### NOTE

It looks to the application as if the user had selected a given menu item with the mouse. Your program will receive a menu event, not a key event. For more information on menu item selection, see the "Intuition: Menus" chapter.

If Intuition sees a command-key sequence that means nothing to it, the key sequence is broadcast to your program as usual. See the "Intuition: Input and Output Methods" section for how this works.

It is recommended that you abide by certain command-key standards to provide a consistent interface for Amiga users. The "Intuition Style" section contains a complete list of the recommended standards.

## MOUSE AND KEYBOARD EXAMPLE

The example program below shows the use of RAWKEYS, MOUSEBUTTONS, MOUSEMOVE, RawKeyConvert() and DoubleClick().

```

/* MouseKeys.c */
#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <devices/inputevent.h>
#include <libraries/dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef LATTICE
#include <proto/all.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

VOID cleanExit(int);
VOID OpenAll(VOID);
VOID DoKeys(struct IntuiMessage *);
LONG DeadKeyConvert(struct IntuiMessage *, UBYTE *, LONG, struct KeyMap *);
VOID DoMouseMove(struct RastPort *, struct IntuiMessage *);
VOID DoButtons(struct IntuiMessage *);

#define BUFSIZE 15
#define RP window->RPort
#define WIDTH 320
#define HEIGHT 50
#define SHIFTED (IEQUALIFIER_LSHIFT|IEQUALIFIER_RSHIFT)

struct IntuitionBase *IntuitionBase=NULL;
struct GfxBase *GfxBase=NULL;
struct Window *window=NULL;
struct ConsoleDevice *ConsoleDevice=NULL;
struct IOStdReq ioreq;

struct NewWindow NewWindow =
{
    0,0,WIDTH,HEIGHT,
    -1,-1,
    CLOSEWINDOW|RAWKEY|MOUSEMOVE|MOUSEBUTTONS,
    WINDOWDRAG|WINDOWCLOSE|ACTIVATE|REPORTMOUSE|RMBTRAP,
    NULL,NULL,
    "Mouse & KeyBoard",
    NULL,NULL,
    0,0,0,0,
    WBENCHSCREEN
};

/*
The Main Loop
*/
VOID main(VOID)
{
    USHORT keep=TRUE; /* loop control */
    struct IntuiMessage *msg; /* for the Intuition message */
    ULONG class; /* event class */

    OpenAll();
    printf("Monitors the Mouse AND Keyboard\n");
    printf("Try DoubleClicking and Special Keys\n");
    while(keep)
    {
        Wait((1<<window->UserPort->mp_SigBit));
        while(msg=(struct IntuiMessage *)GetMsg(window->UserPort))
        {
            class=msg->Class; /* Get the event class */
            switch(class) /* handle our events */
            {
                case CLOSEWINDOW: keep=FALSE; break;
                case RAWKEY: DoKeys(msg); break;
                case MOUSEMOVE: DoMouseMove(RP,msg); break;
                case MOUSEBUTTONS: DoButtons(msg); break;
            }
        }
    }
}

```

```

        }
        ReplyMsg((struct Message *)msg);
    }
}
cleanExit(RETURN_OK);
}

/*
Show what mouse buttons where pushed
*/
VOID DoButtons(struct IntuiMessage *msg)
{
    static ULONG lsecs = 0L, lmics = 0L; /* For detecting DoubleClick */
    USHORT code;
    USHORT qual;
    ULONG secs, mics;

    code = msg->Code;
    qual = msg->Qualifier;
    secs = msg->Seconds; /* get the current time for */
    mics = msg->Micros; /* DoubleClick() */

    /* Yes, qualifiers can apply to the mouse also. That is how
    * we get the shift select on the WorkBench. This shows how
    * to see if a specific bit is set within the qualifier
    */

    if(qual & SHIFTED) printf("Shift ");
    switch(code)
    {
        case SELECTDOWN:
            printf("Left Button Down");
            if(DoubleClick(lsecs, lmics, secs, mics))
            {
                printf(" DoubleClick!");
            }
            else
            {
                lsecs = secs;
                lmics = mics;
            }
            break;
        case SELECTUP:
            printf("Left Button up");
            break;
        case MENUDOWN:
            printf("Right Button down");
            break;
        case MENUUP:
            printf("Right Button up");
            break;
    }
    printf("\n");
}

/* Show the current position of the mouse relative to the
* upper left-hand corner of our window
*/
VOID DoMouseMove(
    struct RastPort *rp, /* RastPort to write coordinates into */
    struct IntuiMessage *msg) /* IntuiMessage containing mouse coords */
{
    UBYTE coords[12];

    sprintf(coords, "X%4d Y%4d", msg->MouseX, msg->MouseY);
    Move(rp, ((WIDTH-88)/2), ((HEIGHT-8)/2)+10); /* ASSUMES 8 Pixel-High Font */
    Text(rp, &coords[0], 11);
}

/*
Show what keys where pressed...
*/

```



```

VOID DoKeys(struct IntuiMessage *msg)
{
    register LONG i;
    LONG    numchars;
    UBYTE    buffer[BUFSIZE]; /* buffer is large enough for normal keymaps */

    strcpy(buffer, "                "); /* clear the buffer area */

    numchars=DeadKeyConvert(msg, &buffer[0], BUFSIZE, 0L);

    /* numchars now contains the number of characters placed
     * within the buffer. It returns zero when it is either a
     * 'dead' key or key release. Special keys (like HELP,
     * the cursor keys, FKeys, etc) return multiple characters that
     * have to then be parsed. If there wasn't enough room in the
     * buffer (indicated by a return of -1), then you should re-call
     * the same function with a larger buffer using the SAME
     * IntuiMessage.
     */
    if(numchars>0L)
    {
        printf("key %d maps to %ld character(s)\n", msg->Code, numchars);
        for(i=0L; i<numchars; i++)
        {
            printf("  %3d = %c\n", buffer[i], buffer[i]);
        }
    }
}

/*
Open up all the resources that we need
*/
VOID OpenAll(VOID)
{
    if(!(IntuitionBase=(struct IntuitionBase *)
        OpenLibrary("intuition.library",33)))
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);
    if(!(GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",33)))
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);

    /* must have the console.device opened to use RawKeyConvert()
     */

    if(OpenDevice("console.device",-1L,(struct IORequest *)&ioreq,0L))
        cleanExit(ERROR_DEVICE_NOT_MOUNTED);
    ConsoleDevice=(struct ConsoleDevice *)ioreq.io_Device;

    /* center the window that we are going to use to display
     * the current mouse position in.
     */

    NewWindow.LeftEdge = (GfxBase->NormalDisplayColumns - WIDTH) / 2;
    NewWindow.TopEdge = (GfxBase->NormalDisplayRows - HEIGHT) / 2;

    if(!(window=(struct Window *)OpenWindow(&NewWindow)))
        cleanExit(ERROR_NO_FREE_STORE);

    /* initialize the drawing variables used for rendering the
     * current mouse position
     */
    SetAPen(RP,1);
    SetBPen(RP,0);
    SetDrMd(RP,JAM2);
}

/*
Free up all the resources that we used
*/
VOID cleanExit(int retval)
{
    if(window)        CloseWindow(window);
    if(ConsoleDevice) CloseDevice((struct IORequest *)&ioreq);
}

```

```

    if(GfxBase)      CloseLibrary((struct Library *)GfxBase);
    if(IntuitionBase) CloseLibrary((struct Library *)IntuitionBase);
    exit(retval);
}

/*
Convert RAWKEYS into VANILLAKEYS, also shows
special keys like HELP, Cursor Keys, FKeys, etc.

See the Included and Autodocs Manual for information on
the RawKeyConvert() function. It returns places an ANSI
character stream into the buffer. This ANSI sequence is
described in the Console Device Chapter.

Returns:
-2 if not a RAWKEY event
-1 if not enough room in the buffer
the number of characters placed in the buffer
*/
LONG DeadKeyConvert( struct IntuiMessage *msg,
                     UBYTE               *kbuffer,
                     LONG                 ksize,
                     struct KeyMap        *kmap)
{
    static struct InputEvent ievent = {NULL, IECLASS_RAWKEY, 0, 0, 0};

    if(msg->Class != RAWKEY)
        return(-2);
    ievent.ie_Code = msg->Code;
    ievent.ie_Qualifier = msg->Qualifier;
    ievent.ie_position.ie_addr = *((APTR*)msg->IAddress);

    return(RawKeyConvert (&ievent, kbuffer, ksize, kmap));
}

```

# Chapter 10

## Intuition: Other Features

### Introduction

There are several Intuition topics which, while not large enough to fill chapters of their own, nonetheless deserve to be discussed. The subjects discussed here include locking **IntuitionBase**, the Intuition memory functions **AllocRemember()** and **FreeRemember()**, Preferences, ViewPorts, and sprites.

### Locking IntuitionBase

It is sometimes necessary to examine the **IntuitionBase** structure. Items such as the address of the active screen and window, current mouse coordinates and more can be found there. It is never a good idea to simply “peek” at these fields, as they are prone to sudden change. It is necessary to inform Intuition that you are about to examine **IntuitionBase** so that it will remain static during this time. The call **LockIBase()** will lock the state of **IntuitionBase** so that it may be examined. **LockIBase()** is passed a ULONG indicating the Intuition lock desired. For all foreseeable uses of this call this value should be 0. **LockIBase()** returns a ULONG, which must be passed to **UnlockIBase()** to allow **IntuitionBase** to change once again. **UnlockIBase()** has no return value. During the time that you have **IntuitionBase** locked, all Intuition input processing is frozen. Make every effort to examine

**IntuitionBase** and release the lock as quickly as possible.

#### IMPORTANT

This function should not be called while holding any other system locks such as **Layer** and **LayerInfo** locks.

#### VERY IMPORTANT

There are fields in **IntuitionBase** which are considered “private”. (Refer to the *intuition/intuitionbase.h* include file.) Application programs cannot depend on (and should not use) the contents of these fields; their usage is subject to change in future revisions of Intuition.

#### EXTREMELY IMPORTANT

Never, *ever*, modify *any* of the fields in **IntuitionBase** directly.

## Easy Memory Allocation and Deallocation

Intuition has a pair of routines that enable one to make multiple memory allocations which are easily deallocated with a single call. The routines are **AllocRemember()** and **FreeRemember()**; they both rely upon a linked list of **Remember** structures to keep track of allocations.

### INTUITION HELPS YOU REMEMBER

The **AllocRemember()** routine actually calls the Exec **AllocMem()** function to do the memory allocation. It also allocates memory for a **Remember** structure and uses it as a link node to save the specifics of the allocation in a linked list. When **FreeRemember()** is called it uses the information in this linked list to free all of the previous memory allocations. **AllocRemember()** returns a NULL if its allocation fails.

#### **AllocRemember(&RememberKey, Size, Flags)**

##### **&RememberKey**

is the address of a pointer to a **Remember** structure.

**Size** is the size in bytes of the requested allocation.

##### **Flags**

gives the specifications for the memory allocation. These are the same as the specifications for the Exec **AllocMem()** function, a description of which can be found in the memory allocation chapter.

The **FreeRemember()** function gives the option of freeing memory in either of two ways: freeing both the link nodes that **AllocRemember()** created *and* the memory blocks to which they correspond, or freeing only the link nodes, leaving the memory blocks for further use (and later deallocation via Exec’s **FreeMem()** function).

#### **FreeRemember(&RememberKey, ReallyForget)**

**&RememberKey**

is the address of a pointer to a **Remember** structure.

**ReallyForget**

indicates whether both the link nodes *and* the memory blocks should be freed (TRUE), or only the link nodes should be freed (FALSE).

These routines have two primary uses. The most general use of these routines is to do all of a program's memory allocations using **AllocRemember()**. The advantage of this is that a linked list of all your memory allocations is created for you, so that when you want to free all the memory, a single call to **FreeRemember()** does the job.

The other use is to do a series of memory allocations and abandon it in midstream easily, if you must. Say that you're doing a long series of allocations in a procedure (for example, the Intuition **OpenWindow()** procedure), and you detect some error condition, such as out-of-memory. When aborting, you must free any memory that you have already managed to allocate. These procedures allow you to free that memory easily, without being required to keep track of how many allocations you have already done, the sizes of the allocations, and where the memory was allocated.

**HOW TO REMEMBER**

You create the "anchor" for the linked list by declaring a variable that is a pointer to a **Remember** structure and initializing that pointer to NULL. This variable is called the **RememberKey**. Whenever you call **AllocRemember()**, the routine actually does two memory allocations, one for the memory you want and the other for a **Remember** structure. The **Remember** structure is filled in with data describing your memory allocation, and it is linked into the list to which the **RememberKey** points. Then, to free any memory that has been allocated, all you have to do is call **FreeRemember()** with that **RememberKey**.

**THE REMEMBER STRUCTURE**

The **Remember** structure is defined in *intuition/intuition.h* as follows:

```
struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};
```

The contents of the **Remember** structure are handled by the system, but are explained here for completeness.

**NextRemember**

is the link to the next **Remember** node.

**RememberSize**

is the size of the memory remembered by this node.

**Memory**

is a pointer to the memory remembered by this node.

## AN EXAMPLE OF REMEMBERING

```
/* RememberTest
   Illustrates the use of AllocRemember() and FreeRemember().
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <proto/all.h>
#include <stdlib.h>

struct IntuitionBase *IntuitionBase;

#define SIZE_A 100L
#define SIZE_B 200L

#define FLAGS_A (MEMF_CLEAR | MEMF_PUBLIC)
#define FLAGS_B MEMF_PUBLIC

VOID methodOne(VOID), methodTwo(VOID);

VOID main(VOID)
{
    int exitVal = RETURN_OK;

    /* Open Intuition */
    IntuitionBase = (struct Intuition *) OpenLibrary("intuition.library", 33L);
    if (IntuitionBase)
    {
        methodOne();

        methodTwo();

        CloseLibrary((struct Library *)IntuitionBase);
    }
    else
        exitVal = RETURN_FAIL;

    exit(exitVal);
}

/* MethodOne
   Illustrates using AllocRemember() to allocate all memory and
   FreeRemember() to free it all.
*/
VOID methodOne(VOID)
{
    CPTR memBlockA = NULL, memBlockB = NULL;
    struct Remember *rememberKey = NULL;

    memBlockA = AllocRemember(&rememberKey, SIZE_A, FLAGS_A);
    if (memBlockA)
    {
        /* The memBlockA allocation succeeded; try for memBlockB. */
        memBlockB = AllocRemember(&rememberKey, SIZE_B, FLAGS_B);
        if (memBlockB)
        {
            /* Both memory allocations succeeded.
               The program may now use this memory.
            */
        }
    }
}
```

```

/*
    It is not necessary to keep track of the status of each allocation.
    Intuition has kept track of all successful allocations by updating its
    linked list of Remember nodes. The following call to FreeRemember() will
    deallocate any and all of the memory that was successfully allocated.
    The memory blocks as well as the link nodes will be deallocated because
    the "ReallyForget" parameter is TRUE.
*/

FreeRemember(&rememberKey, TRUE);

/*
    It is possible to have reached the above call to FreeRemember()
    in one of three states. Here they are, along with their results.

    1. Both memory allocations failed.
        RememberKey is still NULL. FreeRemember() will do nothing.
    2. The memBlockA allocation succeeded but the memBlockB allocation failed.
        FreeRemember() will free the memory block pointed to by memBlockA.
    3. Both memory allocations were successful.
        FreeRemember() will free the memory blocks pointed to by
        memBlockA and memBlockB.
*/

}

/* MethodTwo
    Illustrates using AllocRemember() to allocate all memory,
    FreeRemember() to free the link nodes, and FreeMem() to
    free the actual memory blocks.
*/
VOID methodTwo(VOID)
{
    CPTR memBlockA = NULL, memBlockB = NULL;
    struct Remember *rememberKey = NULL;

    memBlockA = AllocRemember(&rememberKey, SIZE_A, FLAGS_A);
    if (memBlockA)
    {
        /* The memBlockA allocation succeeded; try for memBlockB. */
        memBlockB = AllocRemember(&rememberKey, SIZE_B, FLAGS_B);
        if (memBlockB)
        {
            /* Both memory allocations succeeded. */

            /* For the purpose of illustration, FreeRemember()
               is called at this point, but only to free the
               link nodes. The memory pointed to by memBlockA
               and memBlockB is retained.
            */
            FreeRemember(&rememberKey, FALSE);

            /* Pretend that memBlockA was needed only temporarily.
               It can now be freed. The Exec FreeMem() call must
               be used, as the link nodes are no longer available.
            */
            FreeMem((VOID *)memBlockA, SIZE_A);

            /* The memory pointed to by memBlockB is used by the program. */

            /* All memory blocks allocated with AllocRemember() must be
               freed individually, now that the link nodes are gone.
            */
            FreeMem((VOID *)memBlockB, SIZE_B);
        }
    }

    FreeRemember(&rememberKey, TRUE);
}

```

```

/*
It is possible to have reached the above call to FreeRemember()
in one of three states. Here they are, along with their results.

1. Both memory allocations failed.
   RememberKey is still NULL. FreeRemember() will do nothing.
2. The memBlockA allocation succeeded but the memBlockB allocation failed.
   FreeRemember() will free the memory block pointed to by memBlockA.
3. Both memory allocations were successful.
   If this is the case, the program has already freed the link nodes
   with FreeRemember() and the memory blocks with FreeMem().
   When FreeRemember() freed the link nodes, it reset RememberKey
   to NULL. This (second) call to FreeRemember() will do nothing.
*/
}

```

## Preferences

Preferences is a program that lets the user see and change many system-wide parameters on the Amiga. Users can also edit the standard Intuition pointer image and colors.

The user invokes Preferences to make settings and your program can call `GetPrefs()` to find out what settings the user has made. In a system in which the user does not use Preferences, you can call `GetDefPrefs()` to find out the Intuition default Preference settings. If you are using the IDCMP for input, you can set the IDCMP flag `NEWPREFS`. With this flag set, your program will receive an `IntuiMessage` telling it that there is a new set of Preferences for it to examine. To get the new settings, the program then calls `GetPrefs()`.

Programs that use system printer drivers should always call `GetPrefs()` just before every print job, because the user may run Preferences to modify the printer settings or change to a different printer.

When Intuition is initialized (when the system is reset), you can call `GetDefPrefs()` to find the default Preferences settings that Intuition uses when it is first opened. Then, under AmigaDOS, Intuition is configured according to the set of Preferences that are saved on the start-up disk.

Upon invoking the Preferences tool, the user is shown a screen full of gadgets and can change settings by selecting and playing with the gadgets. In some cases, a requester appears after the user selects a gadget.

One of the arguments to `GetPrefs()` and `GetDefPrefs()` is the size of the buffer you are supplying to receive the Preferences data. If you are interested only in the first few bits of data, you do not have to allocate a buffer large enough to hold the entire Preferences structure. For this reason, the most commonly used data has been grouped near the beginning of the structure.

Preferences allows the user to change the following:

- Date and time of day. These are automatically saved in the battery-backed clock, if one is present.
- Key repeat speed — the speed at which a key repeats when held down.
- Key repeat delay — the amount of delay before the key begins repeating.
- Mouse speed — how far the pointer moves when the user moves the mouse.



- Double-click delay — maximum time allowed between the two clicks of a mouse double-click. For information about how to test for double-click timeout, see the description of the `DoubleClick()` function in *The Amiga ROM Kernel Reference Manual: Includes and Autodocs*.
- Text size — size of the default font characters. The user can choose 64-column mode (64 characters on a line in high-resolution mode and 32 characters in low-resolution mode) or 80-column mode (80 characters on a line in high-resolution mode and 40 characters in low-resolution mode). The first variable in the Preferences structure is `FontHeight`, which is the height of the characters in display lines. If this is equal to the constant `TOPAZ_EIGHTY`, the user has chosen the 80-column version. If it is equal to `TOPAZ_SIXTY`, the user has chosen the 64-column version.
- Display centering — allows the user to center the image on the video display.
- Baud rate — the user can change the rate of data transmission to accommodate whatever device is attached to the serial connector.
- Workbench colors — the user can change any of the four colors in the Workbench display by adjusting the amounts of red, green, and blue in each color.
- Printer — the user can select from a number of printers supported by Amiga. The user can also indicate whether the printer is connected to the serial connector or the parallel connector.
- Print characteristics — the user can select paper size, right and left margin, continuous feed or single sheets, draft or letter quality, pitch, and line spacing. For graphics dumps, the user can indicate how he wishes the dump to appear on paper by setting the density and scaling method, selecting vertical or horizontal dumps, etc.

The Preferences settings can be written to the `devs:system-configuration` file, to be used for the next work session. See the Amiga user's manual for more information about Preferences from the user's standpoint.

## PREFERENCES STRUCTURE

Here is the Preferences data structure:

```
struct Preferences
{
    BYTE    FontHeight;
    UBYTE   PrinterPort;
    USHORT  BaudRate;
    struct  timeval KeyRptSpeed;
    struct  timeval KeyRptDelay;
    struct  timeval DoubleClick;
    USHORT  PointerMatrix[POINTERSIZE];
    BYTE    XOffset;
    BYTE    YOffset;
    USHORT  color17;
    USHORT  color18;
    USHORT  color19;
    USHORT  PointerTicks;
    USHORT  color0;
    USHORT  color1;
    USHORT  color2;
    USHORT  color3;
    BYTE    ViewXOffset;
    BYTE    ViewYOffset;
    WORD    ViewInitX, ViewInitY;
    BOOL    EnableCLI;
    USHORT  PrinterType;
    UBYTE   PrinterFilename[FILENAME_SIZE];
}
```

```

USHORT PrintPitch;
USHORT PrintQuality;
USHORT PrintSpacing;
UWORD PrintLeftMargin;
UWORD PrintRightMargin;
USHORT PrintImage;
USHORT PrintAspect;
USHORT PrintShade;
WORD PrintThreshold;
USHORT PaperSize;
UWORD PaperLength;
USHORT PaperType;
UBYTE SerRWBits;
UBYTE SerStopBuf;
UBYTE SerParShk;
UBYTE LaceWB;
UBYTE WorkName[FILENAME_SIZE];
BYTE RowSizeChange;
BYTE ColumnSizeChange;
UWORD PrintFlags;
UWORD PrintMaxWidth;
UWORD PrintMaxHeight;
UBYTE PrintDensity;
UBYTE PrintXOffset;
UWORD wb_Width;
UWORD wb_Height;
UBYTE wb_Depth;
UBYTE ext_size;
};

```

The meanings of the fields in the **Preferences** structure are as follows:

#### **FontHeight**

This variable will contain one of two constants: **TOPAZ\_SIXTY** or **TOPAZ\_EIGHTY**. These are the font heights required to cause the default Topaz font to be rendered in either 64- or 80-column mode wherever the default font is requested.

#### **PrinterPort**

This is set to either **PARALLEL\_PRINTER** or **SERIAL\_PRINTER** to describe which type of printer is attached to the printer port.

#### **BaudRate**

This can be set to any of these default baud rates: **BAUD\_110**, **BAUD\_300**, **BAUD\_1200**, **BAUD\_2400**, **BAUD\_4800**, **BAUD\_9600**, **BAUD\_19200** or **BAUD\_MIDI**.

#### **KeyRptSpeed, KeyRptDelay**

These are **timeval** structures, which have two components, seconds and microseconds. **KeyRptDelay** describes how long the system hesitates before the input device starts repeating the keys. **KeyRptSpeed** describes the time between repeats of the key.

#### **DoubleClick**

This is a **timeval** structure that describes the maximum time allowable between clicks of the mouse button for the operation to be considered a double-click operation. See the chapter "Intuition: Keyboard and Mouse," for details about double-clicking.

#### **PointerMatrix[POINTERSIZE]**

This contains the sprite data for the Intuition pointer.

#### **XOffset, YOffset**

This describes the offsets from the upper left corner of the pointer image to the pointer's active spot.

**color17, color18, color19**

These are the colors of the Intuition pointer.

**PointerTicks**

This describes how many ticks are required for the mouse to move one increment. It should always be a power of two. The Preferences tool allows it to be set to 1, 2, or 4. Setting it to greater than 4 is not advised. For instance, if **PointerTicks** is set to 32768, to move the pointer from the bottom to the top of the screen the user would have to move the mouse more than a mile.

**color0, color1, color2, color3**

These are the Workbench colors.

**ViewXOffset, ViewYOffset**

These describe the offset of the **View** from its initial start-up position. This configurable offset allows the user to position the display on his monitor.

**ViewInitX, ViewInitY**

These have copies of the initial **View** values, as created by the graphics library.

**EnableCLI**

This field is obsolete, but not yet free for other uses.

**PrinterType**

These are the definitions of the available printer types. See *The Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a complete list of the definitions you might find in this variable.

**PrinterFilename[FILENAME\_SIZE]**

The default name for the disk-based printer configuration file is kept in this buffer.

**PrintPitch, PrintQuality, PrintSpacing**

These describe the pitch, print quality, and page spacing for printer drivers.

**PrintLeftMargin, PrintRightMargin**

The character spacing of the print margins are described by these variables.

**PrintImage, PrintAspect, PrintShade**

The values of these variables tell printer drivers about the desired type of page imagery.

**PrintThreshold**

For simple black/white printer dumps, this describes the intensity threshold required to trigger a print of a pixel.

**PaperSize, PaperLength, PaperType**

These describe the user's choice of printer paper.

**SerRWBits**

The upper nibble = (8 - number of read bits), and the lower nibble = (8 - number of write bits).

**SerStopBuf**

The upper nibble = (number of stop bits - 1), and the lower nibble = (table value for BufSize).

**SerParShk**

The upper nibble = (value for Parity setting), and the lower nibble = (value for Handshake mode).

**LaceWB**

If workbench is to be interlaced.

**WorkName[FILENAME\_SIZE]**

Obsolete.

**RowSizeChange**

System reserved.

**ColumnSizeChange**

System reserved.

**PrintFlags**

The user preference flags, such as center, user indications of how the print is to appear on paper.

**PrintMaxWidth**

Maximum width of the printed picture in 10ths of an inch.

**PrintMaxHeight**

Maximum height of the printed picture in 10ths of an inch.

**PrintDensity**

The print density.

**PrintXOffset**

The offset of the printed picture in 10ths of an inch.

**wb\_Width**

Override default Workbench width.

**wb\_Height**

Override default Workbench height.

**wb\_Depth**

Override default Workbench depth.

**ext\_size**

Reserved.

**PREFERENCES FUNCTIONS**

Your program can use the following functions to check the current Preferences settings.

- **GetPrefs((struct Preferences \*)PrefBuffer, Size)**

Gets a copy of the current Preferences data.

**PrefBuffer** - pointer to the memory buffer to receive the Preferences data

**Size** - number of bytes to copy to the buffer. You should use `sizeof(struct Preferences)`.

- **GetDefPrefs(PrefBuffer, Size)**

Gets a copy of the default Preferences data.

**PrefBuffer** - pointer to the memory buffer to receive the Preferences data

**Size** - number of bytes to copy to the buffer

## Remaking the ViewPorts

This section is for advanced programmers who are interested in controlling their custom screens directly and want to control the entire Intuition display.

There are two functions that operate on the entire display—**RethinkDisplay()** and **RemakeDisplay()**. The **MakeScreen()** function works only with the Copper lists of your custom screen.

**RethinkDisplay()** reworks Intuition's internal state data, rethinks the relationship of all of the screen **ViewPorts** to one another and reconstructs the entire Intuition display by calling the graphics primitives **MrgCop()** and **LoadView()**. This includes all the screens in the display, not just the ones controlled by your program. It is especially handy if you are creating custom screens and want to make up your own lists of Copper instructions for handling the display. For more information about the Copper, see the *Amiga Hardware Reference Manual*.

**RethinkDisplay()** makes calls to the graphics primitives **MrgCop()** and **LoadView()**, which causes the display of Intuition's screens to be reconstructed. **MrgCop()** merges all the various Copper instructions for different **ViewPorts** of the display into a single instruction stream. This creates a complete set of instructions for each *display field* (complete scanning of the video beam from top to bottom of the video display). **LoadView()** uses this merged Copper instruction list to create the display. Before calling **RethinkDisplay()**, you may wish to call **MakeScreen()** to create the Copper instruction list for your own custom screens.

Note that **RethinkDisplay()** can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multitasking Executive, so do not use this routine lightly.

The function **RemakeDisplay** reconstructs the entire Intuition display. It calls **MakeScreen()** for every screen in the system and then calls **RethinkDisplay()**. As with **RethinkDisplay()**, **RemakeDisplay()** can take several milliseconds to run, and it locks out all other tasks while it runs. This can seriously degrade the performance of the multitasking Executive, so do not use this routine lightly.

To remake the Copper lists of your custom screen, call **MakeScreen()**. The only difference between **MakeScreen()** and the graphics library routine **MakeVPort()** is that Intuition synchronizes your call to **MakeVPort()** with any calls that it needs to make.

## Current Time Values

The function **CurrentTime()** gets the current time values. To use this function, you first declare the variables **Seconds** and **Micros**. Then, when you call the function, the current time is copied into the argument pointers. The synopsis of this function is:

```
ULONG Seconds, Micros;  
CurrentTime(&Seconds, &Micros);
```

## Flashing the Display

Because the Amiga has no internal bell or beeper, the screen-flashing function is supplied to notify the user of some event that is not serious enough to require a requester. For example, Intuition uses this function when the user types an invalid character into an integer gadget. This function flashes the background color of the screen. If the argument to the function is NULL, every screen in the display is flashed. The synopsis of this function is:

**DisplayBeep(Screen);**

Screen is a pointer to your screen or NULL

## Using Sprites in Intuition Windows and Screens

Sprite functionality has limitations under Intuition. The hardware and graphics library sprite systems manage sprites independently of the Intuition display. In particular:

- Sprites cannot be attached to any particular screen. Instead, they always appear in front of every screen.
- When a screen is moved, the sprites do not automatically move with it. The sprites move to their correct locations only when the appropriate function is called (either **DrawGList()** or **MoveSprite()**).

Hardware sprites are of limited use under the Intuition paradigm. They travel out of windows and out of screens, unlike all other Intuition mechanisms (except the Intuition pointer, which is meant to be global).

Remember that sprite data must be in CHIP memory to be accessible to the custom chips. This may be done with your compiler's *chip* keyword, if available. Otherwise, chip memory can be allocated with the **Exec AllocMem()** function or the **Intuition AllocRemember()** function, setting the memory requirement flag to **MEMF\_CHIP**. You may then copy the data to CHIP memory using a function like **Exec CopyMem()**.

# **Chapter 11**

## **Intuition: Style**

**This chapter describes some important aspects of Intuition style. If you adhere to these style notes, you will help to ensure that Intuition applications present a consistent interface to the user. Try to exercise all of the suggestions in this chapter.**

## Menu Style

Always make sure that you use **OffMenu()** when an item becomes meaningless or non-functional. Do not ever let the user select something and then have the application do nothing in response. Always take away the user's ability to select that item.

You should always inform the user that a submenu is present. This prevents the user from being surprised when the submenu appears. For instance:

Drop
Pick Up >>
Fumble

This method uses the single character ">>" symbol from the ECMA Latin-1 character set (\$BB). The >> character may appear to the left or the right of the menu item.

You should also inform the user which menu selections will bring up a requester. Again, the goal is to let the user know what's going on. For instance:

Open...
Save
Save As...

This method uses the three characters "..." to inform about the requester.

Submenus should be positioned to the right of the menu (if at all possible). Submenu choices should abut to avoid flashing problems.

Your program should always support extended selection for menus. This is especially useful for selections with checkmarks.

The pens you set when you open a window are used to render the menu bar and the items. If you are opening multiple windows, you might consider color-coding the window frames and menus. Please use subdued colors as they are easier on the eyes.



## PROJECT MENUS

If you are going to allow the user to select which project to work with, you should create a Project-menu. For consistency, it is suggested that menu strips be created with the Project menu as the leftmost menu. This menu should contain the items shown in the following table. If possible, the items should be in the order shown.

Table 11-1: Project Menus

Menu Item	Function
NEW	Creates a project
OPEN...	Gets back a project previously saved
SAVE	Saves the current project to the disk
SAVE AS...	Saves the current project using a different name
PRINT	Prints the entire project
PRINT AS...	Prints part of a project or selects other than the default printer settings
ABOUT...	Displays information about the program.
QUIT	Stops the program (If the project was modified, ask if the user wants to save the work.)

## EDIT MENUS

If your application can perform edit-like functions, it is suggested that you create an Edit-menu, which should appear to the right of the Project menu. It should contain the items shown in the following table. If possible, the items should be in the order shown in the table.

Table 11-2: Edit Menus

Menu Item	Function
UNDO	Undoes the previous operation (if possible; if not, disable this option!)
CUT	Removes the selected portion of the project and puts it in the Clipboard
COPY	Puts a copy of the selected bit of the project in the Clipboard
PASTE	Puts a copy of the Clipboard into the project
ERASE	Removes the selected bit without putting it into the Clipboard

## Gadget Style

When creating a list of gadgets, in a requester or perhaps a window, be sure to design bolder, more eye-catching imagery for the obvious or safe choice. For example, note how the CANCEL choice is highlighted in the requester example in this chapter.

Overlapping the select boxes of gadgets is in general not a good thing to do. This is especially true when it is not obvious to users which gadget they are selecting. Unless you are very careful, all sorts of weird things can happen and the gadgets will behave in unusual ways.

As with menus, use `OffGadget()` to remove a gadget when it becomes meaningless or nonfunctional.

User feedback from proportional gadgets should be quick and snappy. You should use the `FOLLOWMOUSE` gadget activation flag and the `IDCMP MOUSEMOVE` flag when displaying anything that can be quickly redrawn (like a list of names). You can avoid unneeded redrawing by checking to see if the gadget has moved far enough to require changing the image.

When you use proportional gadgets to control a list it is suggested that you base your `HorizBody` or `VertBody` value on one less than the number of things in the list. This way, if the user scrolls through the list by clicking in the container of the scroll gadget, one of the old lines will still be visible in the new view, letting the user know just where they are.

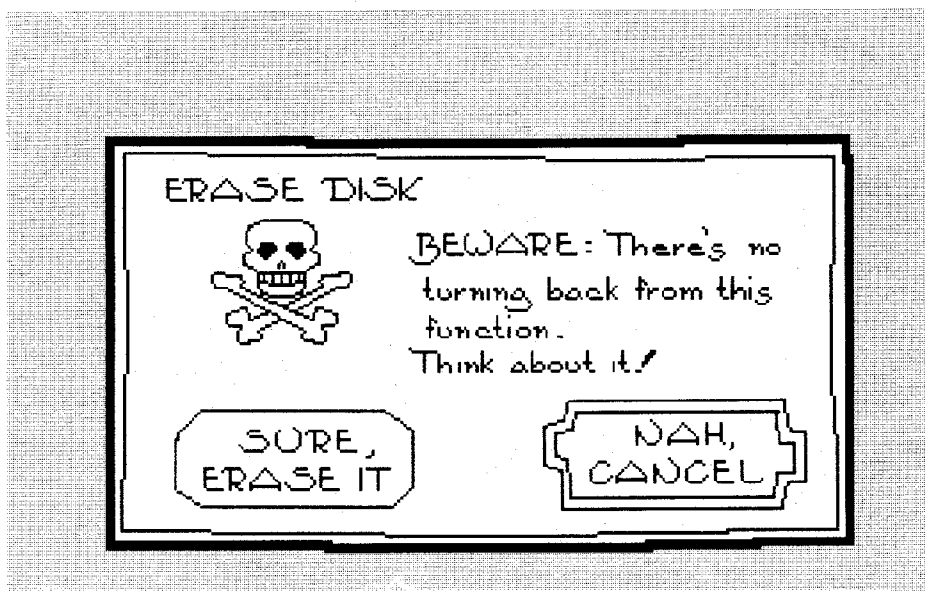


Figure 11-1: The Dreaded Erase-Disk Requester

## Requester Style

This is easily the *most important* rule about requesters: always make sure that there is a safe way to exit from any requester. In the above figure, notice that the dreaded “ERASE DISK” requester can be canceled; in fact, the “run away” option is rendered in bolder imagery. If, for instance, the user accidentally selected the ERASE DISK option from a menu, the CANCEL option saves the day. **This is extremely important. We cannot emphasize this point strongly enough.**

When you design a requester with your own **BitMap** imagery, make sure that the imagery works well with the select boxes of the gadget list that you supply.

Take special care when you design requesters that have string gadgets. Make sure that the string gadget is active when the requester appears. If you make a requester with more than one string gadget, pressing RETURN should activate the next string gadget.

### The Sides of Good and Bad

Whenever the user is presented with a pair of choices that could be characterized as positive/negative options, the positive option should always appear on the left and the negative option on the right. For example, if you are designing a requester with “OK” and “CANCEL” options, “OK” should appear on the left and “CANCEL” on the right. If your options are “RETRY” and “ABORT,” you should render “RETRY” on the left and “ABORT” on the right.

The Intuition **AutoRequest()** requester and **DisplayAlert()** alert both use this scheme. If all programs adopt this design, the user will come to feel secure in knowing that the right gadget can always be used to abort some dire sequence of events while the left gadget selects the normal, placid continuation of events.

Do not create requesters with two identical buttons. **AutoRequest()** will create requesters with just a right hand button. (CONTINUE is a good choice for text for this button.)

## Command Key Style

Treat the AMIGA keys like SHIFT keys. To enter a shortcut, users should be able to hold down the AMIGA key with the little finger of one hand, and press one of the keys they would normally press with the other hand. This will help touch typists as well as prevent that clumsy feeling that everyone experiences.

The **right** AMIGA key combinations are for use by your programs as shortcuts. The **left** AMIGA key combinations are reserved for use by Intuition itself, except for the six selection exceptions listed in the table below.

The table shows recommendations for standard selection shortcuts (using the **left** AMIGA key to emulate usage of the left button of the mouse):

Table 11-3: Selection Shortcuts

Key Pressed with Left AMIGA Key	Function
N	Bring the Workbench to the front (this is automatically handled by Intuition)
M	Send the Workbench to the back (this is automatically handled by Intuition)
B	Give a POSITIVE response to an Intuition AUTOREQUEST
V	Give a NEGATIVE response to an Intuition AUTOREQUEST
I	"Select a small piece to the right of the cursor," such as the next word
O	"Select a bigger piece to the right of the cursor," such as the next sentence
P	"Select an even bigger piece to the right of the cursor," such as the next paragraph
J	"Select a small piece to the left of the cursor," such as the previous word
K	"Select a bigger piece to the left of the cursor," such as the previous sentence
L	"Select an even bigger piece to the left of the cursor," such as the previous paragraph

The next table shows recommendations for standard information (menu) shortcuts (using the **right** AMIGA key to emulate usage of the right button of the mouse). Programs benefit from an extensive set of menu shortcuts. Users quickly learn the keyboard equivalents of common menu operations. Having a large set means that the expert user will feel more comfortable with your software package, while the novice user will use the menus.

Table 11-4: Information (Menu) Shortcuts

Key Pressed with Right AMIGA Key	Function
X	Cut
C	Copy
V	Paste
I	Change font type to italic
B	Change font type to bold
U	Change font mode to underline
P	Reset font characteristics to plain defaults
Z	Undo (cancel)
O	Open
S	Save
Q	Quit

### The HELP key

Use the HELP key. If your application has built-in help, the key marked 'Help' is a good way for the user to get to it. Stamp out that helpless feeling users have when they hit the HELP key and nothing happens.

### Cursor Key Style

Cursor keys are a convenient way to control the movement of the cursor inside an application. Here are our recommendations for standard cursor key definitions:

Table 11-5: Cursor Keys

Press a cursor key with:	To:
	move one unit in the indicated direction (backwards a character, for instance).
ALT	move the next largest unit in the indicated direction (forward a word, or down one line less than a full page, for instance).
SHIFT	move the largest unit in the indicated direction (to the end of a line, or the top of the file, for instance).
CTRL	special or unassigned

## Mouse Style

Intuition uses the left mouse button for *selection* and the right mouse button for *information transfer*. To help the user understand and remember the elements of every application, you are encouraged to follow this model.

When the user presses the left button, Intuition examines the state of the system and the position of the pointer. It uses this information to decide if the user is trying to select some object, operation, or option. For example, the user positions the pointer over a gadget and then presses the left button to select that gadget. If the user moves the mouse while holding down the select button, this sometimes means that the user wants to select everything that the pointer moves over while the button is still pressed.

Intuition usually uses the right button for menu operations. Pressing the right button usually displays the active window's menu bar over the screen title bar. Moving the mouse while holding down the right button usually means that the user wishes to browse through all available information—for example, browsing through the menus. Double-clicking the right mouse button can bring up a special requester for extended exchange of information. Because this requester is used for the transfer of information, it is appropriate to use the right mouse button.

If you are planning to handle mouse button events directly, you should follow the selection/information model described above.

## Window and Screen Style

Always give the user a way to get to the screen title bar. (This can be as simple as opening your windows a single pixel down from the top of the screen.) Avoid opening full size windows that cannot be moved or resized. Your program may not be the only one the user is running, so be considerate.

Let the maximum window size you allow be as large as possible. If your program has no limit set `MaxWidth/MaxHeight` to -1. The user may have a screen larger than 640 by 200.

### Important

Whenever you open a custom screen, we strongly suggest you redirect AmigaDOS requesters to your screen. (An example of this may be found in the Intuition "Windows" chapter.) Otherwise Intuition will bring the Workbench screen to the front when it needs to tell the user something. You *must* reset the redirection pointer before closing the window.

## Miscellaneous Style Notes

Programs should always make a call to **OpenWorkBench()** when they exit, even if you did not call **CloseWorkBench()**. Workbench should be open as much as possible. If Workbench was closed and your departure has freed up enough memory for Workbench to reopen, it is preferable that it be reopened. **OpenWorkBench()** will not necessarily work (if there is no memory for the display, it will not open). But if every program calls **OpenWorkBench()**, then Workbench will open if it can. By using this mechanism, you can help give the user a consistent environment. Intuition always checks to see if Workbench *must* open whenever any screen is closed.

As much as possible, allow the user to configure the parameters of your program. For instance, if you have opened a custom screen, let the user change the colors. If your program makes sound, give the user the ability to adjust the tone and volume. Do not make the configuration a requirement, however, and always give the user an avenue for restoring the defaults.

If your program is following the mouse position (like a paint package), try to provide a safe place for the user to click on to activate your window. If your program has no safe place (and no window title bar) we recommend that your program ignore the mouse click that activates your window. This will stop the user from ruining a picture with an ill-placed dot.

Here are the color assignments used for the Intuition pointer sprite data:

- Color 0 is transparent.
- Color 1 of the sprite (hardware color register 17) is the color with medium intensity.
- Color 2 of the sprite (hardware color register 18) is low intensity.
- Color 3 of the sprite (hardware color register 19) is high intensity.

Your pointer should be framed by either color 1 or color 3.

Since the Intuition pointer is always hardware sprite zero, you can set the colors of the pointer by calling the **SetRGB4()** function on the **ViewPort** of any screen. An example of this follows:

```
struct Screen *MyScreen;

SetRGB4(&MyScreen->ViewPort, 17, Red17, Green17, Blue17);
SetRGB4(&MyScreen->ViewPort, 18, Red18, Green18, Blue18);
SetRGB4(&MyScreen->ViewPort, 19, Red19, Green19, Blue19);
```

### Icons

Icons should be designed with the light source coming from the top left corner of the screen. Your icons should be visible in one bitplane as well as two bit planes. We suggest edging your icons with black, so they show up, no matter what the background. Finally, make your icons as beautiful as you can, make them reflect your program, but most of all, make them as small as you can. Users are annoyed by huge icons. They take up space on the screen and on the disk. Try for a minimalist approach when designing the perfect icon for your program.

### **A Final Note on Style From --RJ--**

“Design beautiful Gadgets, Menus, Requesters. Think simplicity and elegance. Always remember the fourth grader, the sophisticated user, and the poor soul who is terrified of breaking the machine.”

“Dare to be gorgeous and unique. But don’t ever be cryptic or otherwise unfathomable. Make it unforgettably great.”

### **A Final Note on Style From Jim Mackraz**

“A program’s user interface, ideally, would be completely unnoticed. Remember that the user is generally trying to get something done, and the value of your program depends only on whether he or she is successful. Don’t indulge in user-interface cleverness, unless it directly contributes to making the user’s interaction with the computer more understandable, efficient, or familiar.”



## Chapter 12

### Intuition: Functions

<b>ActivateGadget</b>	Activates a string gadget
<b>ActivateWindow</b>	Activates an Intuition Window
<b>AddGadget</b>	Adds a Gadget to the Gadget list of a Window
<b>AddGList</b>	Add a linked list of gadgets to a Window or Requester
<b>AllocRemember</b>	AllocMem and create a link node to make FreeMem easy
<b>AutoRequest</b>	Automatically builds and gets response from a Requester
<b>BeginRefresh</b>	Sets up a Window for optimized refreshing
<b>BuildSysRequest</b>	Builds and displays a system Requester
<b>ClearDMRequest</b>	Clears the DMRequest of the Window
<b>ClearMenuStrip</b>	Clears the Menu strip from the Window
<b>ClearPointer</b>	Restores the default Intuition pointer in a Window
<b>CloseScreen</b>	Closes an Intuition Screen
<b>CloseWindow</b>	Closes an Intuition Window
<b>CloseWorkBench</b>	Attempts to close the WorkBench Screen
<b>CurrentTime</b>	Gets the current time values

<b>DisplayAlert</b>	Creates a display of an Alert message
<b>DisplayBeep</b>	“Beeps” the video display
<b>DoubleClick</b>	Tests two time values for double-click timing
<b>DrawBorder</b>	Draws the specified border into the RastPort
<b>DrawImage</b>	Draws the specified Image into the RastPort
<b>EndRefresh</b>	Ends the optimized refresh state of the Window
<b>EndRequest</b>	Ends the Request and resets the Window
<b>FreeRemember</b>	Frees memory allocated by calls to AllocRemember()
<b>FreeSysRequest</b>	Frees up memory used by a call to BuildSysRequest()
<b>GetDefPrefs</b>	Gets a copy of the the Intuition default Preferences
<b>GetPrefs</b>	Gets the current setting of the Intuition Preferences
<b>GetScreenData</b>	Get copy of a screen data structure
<b>InitRequester</b>	Initializes a Requester structure
<b>IntuiTextLength</b>	Returns the length (pixel-width) of an IntuiText
<b>ItemAddress</b>	Returns the address of the specified MenuItem
<b>LockIBase</b>	Get an Intuition lock in order to access IntuitionBase safely
<b>MakeScreen</b>	Does an Intuition-integrated MakeVPort() of a custom screen
<b>ModifyIDCMP</b>	Modifies the state of the Window’s IDCMP
<b>ModifyProp</b>	Modifies the current parameters of a proportional Gadget
<b>MoveScreen</b>	Attempts to move the Screen by the delta amounts
<b>MoveWindow</b>	Asks Intuition to move a Window
<b>NewModifyProp</b>	Same as ModifyProp(), but with selective refresh
<b>OffGadget</b>	Disables the specified Gadget
<b>OffMenu</b>	Disables the given menu or menu item
<b>OnGadget</b>	Enables the specified Gadget
<b>OnMenu</b>	Enables the given menu or menu item
<b>OpenScreen</b>	Opens an Intuition Screen
<b>OpenWindow</b>	Opens an Intuition Window
<b>OpenWorkBench</b>	Opens the WorkBench Screen
<b>PrintIText</b>	Draws the specified IntuiText into the RastPort
<b>RefreshGadgets</b>	Refreshes (redraws) the Gadget display
<b>RefreshGList</b>	Refresh (redraw) a chosen number of gadgets
<b>RefreshWindowFrame</b>	Ask Intuition to redraw your window border
<b>RemakeDisplay</b>	Remakes the entire Intuition display
<b>RemoveGadget</b>	Removes a Gadget from a Window
<b>RemoveGList</b>	Removes a sublist of Gadgets from a Window
<b>ReportMouse</b>	Tells Intuition whether or not to report mouse movement
<b>Request</b>	Activates a Requester
<b>RethinkDisplay</b>	The grand manipulator of the entire Intuition display
<b>ScreenToBack</b>	Sends the specified Screen to the back of the display
<b>ScreenToFront</b>	Brings the specified Screen to the front of the display

<b>SetDMRequest</b>	Sets the DMRequest of the Window
<b>SetMenuStrip</b>	Attaches the Menu strip to the Window
<b>SetPointer</b>	Sets a Window with its own Pointer
<b>SetPrefs</b>	Sets new Preferences values
<b>SetWindowTitles</b>	Sets the Window's titles for both Window and Screen
<b>ShowTitle</b>	Sets the Screen title bar display mode
<b>SizeWindow</b>	Asks Intuition to size a Window
<b>UnlockIBase</b>	Releases the Intuition lock obtained by LockIBase
<b>ViewAddress</b>	Returns the address of the Intuition View structure
<b>ViewPortAddress</b>	Returns the address of a Window's ViewPort structure
<b>WBenchToBack</b>	Sends the WorkBench Screen in back of all Screens
<b>WBenchToFront</b>	Brings the WorkBench Screen in front of all Screens
<b>WindowLimits</b>	Sets the minimum and maximum size limits of the Window
<b>WindowToBack</b>	Asks Intuition to send this Window to the back
<b>WindowToFront</b>	Asks Intuition to bring this Window to the front

## Assembly Language Conventions

In all Intuition routines, the arguments always follow the same order: addresses first, data second. The registers are allocated in ascending order from register 0 (always). Thus, you can look at any routine, start from register A0 if the routine's arguments start with an address, and start from D0 when the routine's arguments become data values. As an added mnemonic, even the register names are in alphabetical order—A0 precedes D0. The register names for each function are given in the *Includes and Autodocs Manual*.

Unfortunately for assembly programmers, many of you will have to use assemblers that do not give you macros to declare and reference structure elements. If this is the case, you should use the include file called *intuition.i*, in which every Intuition structure variable has a unique name, found in assembler format.

# Chapter 13

## Intuition: Internal Procedures

This chapter discusses the more esoteric and internal Intuition functions. You have more leeway when using these functions in a machine environment in which you have taken complete control of the Amiga and do not intend to allow other tasks to coexist with yours.

### NOTE

These functions are for internal system use only. They are not fully documented, and their use can lead to unpredictable results.

These functions are covered in this appendix:

#### **SetPrefs()**

This routine allows you to set Intuition's internal state of the Preferences.

#### **AlohaWorkbench()**

This routine allows the Workbench tool to make its presence and departure known to Intuition.

#### **Intuition()**

This is the main entry point into Intuition, where input events arrive and are dispatched.

## SetPrefs()

This routine configures Intuition's internal data states according to the specified Preferences structure. Normally, this routine is called only by:

- The Preferences program itself after the user has changed the Preferences. The Preferences program also saves the user's Preferences data into a disk file named /fidevs:system-configuration/fp.
- AmigaDOS when the system is being booted up. AmigaDOS opens the devs:system-configuration file and passes the information found there to the SetPrefs() routine. This way, the user can create an environment and have that environment restored every time the system is booted.

### NOTE

The intended use for the SetPrefs() call is entirely to serve the user. You should never use this routine to make your programming or design job easier at the cost of yanking the rug out from beneath the user.

The synopsis of this function is:

- **SetPrefs(Preferences, Size, RealThing)**

**Preferences** - a pointer to a Preferences structure

**Size** - the number of bytes contained in your Preferences structure. Typically, you will use "sizeof(struct Preferences)" for this argument.

**RealThing** - a Boolean TRUE or FALSE designating whether or not this is an intermediate or final version of the Preferences. The difference is that final changes to Intuition's preferences causes a global broadcast of NEWPREFS events to every application that is listening for this event. Intermediate changes may be used, for instance, to update the screen colors while the user is playing with the color gadgets.

Refer to the chapter "Other Features," for information about the Preferences structure and the standard Preferences procedure calls.

## AlohaWorkbench()

In Hawaiian, "aloha" means both hello and goodbye. The AlohaWorkbench() routine allows the Workbench program to inform Intuition that it has become active and that it is shutting down.

If the Workbench program is active, Intuition is able to tell it to open and close its windows when someone uses the Intuition OpenWorkBench() and CloseWorkBench() functions to open or close the Workbench screen. If the Workbench program is not active, presumably it has no opened windows, so there is no need for this communication.

This routine is called with one of two kinds of arguments—either a pointer to an initialized message port (which designates that Workbench is active and communications can take place), or NULL to designate that the Workbench tool is shutting down.

When the message port is active, Intuition will send **IntuiMessages** to it. The messages will have the **Class** field set to **WBENCHMESSAGE**. The **Code** field will equal either **WBENCHOPEN** or **WBENCHCLOSE**, depending on whether the Workbench application should open or close its windows. Intuition assumes that Workbench will comply, so as soon as the message is replied to, Intuition proceeds with the expectation that the windows have been opened or closed accordingly.

The procedure synopsis is:

- **AlohaWorkbench(WBPort)**

**WBPort** - a pointer to an initialized **MsgPort** structure in which the special communications are to take place.

## Intuition()

This is Intuition's main entry point. All of Intuition's I/O operations originate here. The input stream flows into Intuition at this portal.

This routine accepts a single argument: a pointer to a linked list of **InputEvent** structures. These events have all the real-time state information that Intuition needs to create its art. Refer to the *Amiga ROM Kernel Manual* for more information about **InputEvent** structure and the operation of the input device.

When **Intuition()** exits, it returns a pointer to a linked list of **InputEvent** structures. This list of **InputEvents** has no dependable correspondence to the list that was initially submitted to **Intuition()**. Intuition may add events to the list and extract events from the list. This list of events is normally intended for the console device.

If you are considering feeding false input events to Intuition, please think again. If you are running in an environment in which you have taken over the machine, it is probably safe to fool Intuition in a controlled way. If you are running in a multitasking environment, however, especially one in which the input device is still feeding input events directly into the stream, you can easily cause more harm than good. You may not be able to anticipate the things that could go wrong when other programs try to exist in an environment that you are modifying.

If you are determined to feed false input events to Intuition, it is much safer to add an input handler to the system than to call **Intuition()**. Add the input handler to the system at a priority higher than Intuition's; the input queue priority of Intuition is 50, so a priority of 51 will suffice. This will allow your program to see *all* input events before Intuition sees them. You can filter the input events, allowing Intuition to see only those events that you want it to see. Also, you can add synthesized events to the input event stream. This allows you to fool Intuition in an honest, system-integrated way.

For example, say that you want to position the pointer yourself but you want to let the user interact with the rest of the system as usual. If you see mouse movement events, you can filter them out and not let Intuition see them. At the same time, you can create mouse movement events of your own. On the other hand, if you see keyboard events you can leave them undisturbed. See the *Amiga ROM Kernel Manual* for details about the input-handler queue.

### NOTE

**Intuition()** is sometimes required to call the **Exec Wait()** function. Normally, **Intuition()** is called from within the input device's task, so the input device enters the wait state when these situations arise. If you call **Intuition()** directly, your task may have to wait. The obvious problem here is the classic lockout problem—your task cannot create the required response because your task has forced itself to wait, which will cause the system to freeze. The best way to get around this is to have a separate task that calls **Intuition()** and does nothing more.

The synopsis of this function is:

- **Intuition(InputEvent)**

**InputEvent** - a pointer to the first in a linked list of **InputEvent** structures.

# Chapter 14

## Exec: Libraries

The Amiga system software comprises many separate subsystems known as *libraries* and *devices*. A library provides a group of related functions which may be accessed by multiple applications. Through the library interface, it is possible to call any of the system routines without knowing its location in the system. This chapter discusses how libraries are designed and used.

### What Is a Library?

A library is a group of related functions that are accessed by system software and applications through a library base jump table. A library consists of reentrant function code and a library base. The library base contains a **Library** node structure, which is preceded by a table of 6-byte vectors to the library functions and followed by library-specific data structures. Some libraries reside in the system ROM, and others are loaded from disk when they are needed. Each library may be opened and closed individually. When a library is open, any of its functions may be called. When all openers of a library have closed it, the library becomes a candidate for purging from the system memory. The jump table and base of a library are generally built dynamically when the library is initialized.



## How To Access a Library

You must perform two steps to access a library. First, you must use the Exec **OpenLibrary()** function to get the base address of the library you wish to access. Since a library's base is dynamically built when the library is initialized, the base address may be anywhere in RAM.

The following diagram illustrates the full structure of a library base in memory:

Lower memory addresses

etc...	etc...
jump to function 6 (LVO -36)	user func
jump to function 5 (LVO -30)	user func
jump to function 4 (LVO -24)	RESERVED
jump to function 3 (LVO -18)	EXPUNGE
jump to function 2 (LVO -12)	CLOSE
jump to function 1 (LVO -6)	OPEN
Library node structure (at base address)	
Library-specific Data Segment	

Higher memory addresses

The base address of a library is a pointer to the beginning of its **Library** node structure. Using the base address of a library, you can call the library's functions by using defined negative offsets from the base known as LVO's or Library Vector Offsets. As shown in the diagram, the 6-byte library function vectors precede the **Library** node, so the vector offset to the first function is at offset -6 from the library base, the next at -12, and so on. By using positive offsets defined in a library's base structure include file, you can also reference the base data in the library's node structure and data segment. This form of indirection allows you to develop code that is not dependent on the absolute locations of the system routines or libraries. This is important because the system routines themselves and the dynamically built library bases appear at different addresses on different systems, and can even appear at different addresses on the same system. Therefore, accessing the system routines through library calls is necessary to assure that your software can work on different machines.

### OPENING A LIBRARY

You prepare a library for use by calling the routine **OpenLibrary()**. This call takes the form

```
LibPtr = OpenLibrary(LibName, Version)
D0                      A1          D0
```

where

#### **LibPtr**

is a non-zero pointer to the library's base (Library node) if the requested library has been located. Be sure to check that the returned value is non-zero *before* attempting to use it. If it is zero, the open failed.

**LibName**

is a pointer to a string variable (null-terminated) that contains the name of the library that you wish to open.

**Version**

is the version number of the library that you require. Any library version equal to or greater (more recent) than the your requested version will satisfy your **OpenLibrary()** call. However, if you specify a newer version than is present, the open will fail.

You can use the value 0 if any version of the named library contains the functions you need. Libraries of the same name will include the functions of previous versions. If you require functions which were added to library in a certain release (as specified in the autodoc or function description file for that library), you should specify the lowest library version number that contains the functions you need. Do not specify a higher version than you need, and do not use the **LIBRARY\_VERSION** constant defined in the include files (use of the constant would cause your code to stop working on previous versions of the operating system if re-compiled or re-assembled with newer include files).

The version numbers to be used with **OpenLibrary()** are:

- 0 = Any version
- 30 = Kickstart V1.0 (obsolete)
- 31 = Kickstart V1.1 NTSC only (obsolete)
- 32 = Kickstart V1.1 PAL only (obsolete)
- 33 = Kickstart V1.2 (oldest revision still in use)
- 34 = Kickstart V1.3 (1.2 plus autoboot expansion.library)  
(should only be specified by autoboot device drivers)
- 35 = Special RAM-loaded release for A2024 monitor  
(should not be specified unless opening an A2024 screen)

From the above table, you can see that except for autoboot driver or A2024 code, software should currently be specifying 0 or 33 to **OpenLibrary()**.

The function **OpenLibrary()** causes the system to search for a library of that name within the system library list. If such an entry is found, the library's open-entry function is called. If the library is not currently RAM-resident, AmigaDOS will search the directory currently assigned to **LIBS:**. If that library is present, it will be loaded, initialized, and added to the system library list. If the library allows you access, the library pointer will be returned in **LibPtr**. As long as you have the library open, it will not be expunged from the system and the **LibPtr** will remain valid.

Assembly language programmers may cache the returned library base pointer wherever it is convenient, since they will be using and passing the pointer directly in the A6 register when making a library call. C programmers, however, must generally store the returned base address in an externally visible (above main) long variable of a specific name so that their compiler and linker can find the correct variable from which to read the library base pointer value when interfacing C function calls to system library routines:

The names of the libraries that are currently part of the Amiga software and associated library base pointer names are as follows:

Library Name	Library Base Pointer Name
diskfont.library	DiskfontBase
dos.library	DOSBase† ‡
exec.library	SysBase†
expansion.library	ExpansionBase
graphics.library	GfxBase
icon.library	IconBase
intuition.library	IntuitionBase
layers.library	LayersBase
mathffp.library	MathBase
mathieeedoubbas.library	MathIeeeDoubBasBase
mathieeedoubtrans.library	MathIeeeDoubTransBase
mathtrans.library	MathTransBase
ramlib.library	(system private)
romboot.library	(V1.3 system private)
translator.library	TranslatorBase
version.library	(system private)
† Automatically opened by the standard C startup module	
‡ dos.library is documented in the <i>AmigaDOS Manual</i>	

## CALLING A LIBRARY FUNCTION

After successfully opening a library and storing the base pointer in the correctly named library base variable, C programmers can call the library's functions as though they were C functions:

```
result = FunctionName(arguments);
```

Your C compiler or linker library should supply the necessary interface code between your C call and the system function. The results and arguments for each library function are documented in the *Includes and Autodocs* manual.

The following code fragment demonstrates calling a system library routine from assembler. The `_LVO<routineName>` label (for example `_LVOOpenLibrary`) will be resolved to the correct negative library vector offset for the function when you link your code with `amiga.lib`. Alternatively, you can look up the correct value in an LVO offset table (see the *Includes & Autodocs* manual).

### NOTE

The save/restore of A6 is necessary only if A6 does not already contain the correct value. If you need to preserve D0, D1, A0, or A1, you will have to save/restore these also since these are defined as scratch registers for system routines. All other registers will be preserved.

\* Register arguments for each function are defined in the autodocs.

\* (see the Addison-Wesley "Includes and Autodocs" manual)

\* You would set up the register arguments for the function, then

```
move.l  A6, -(SP)           ;save current contents of A6
move.l  <libptr>, A6        ;move library pointer into A6
jsr     _LVO<routineName>(A6) ;jsr through library vector table
move.l  (SP)+, A6          ;restore A6 to original value
                        ;D0 contains result, if any
```

The example above is the actual assembly code generated by the use of a machine language macro named LINKLIB:

```
LINKLIB functionOffset,libraryBase
```

where

**functionOffset**

is “\_LVO” followed by the name of the routine as called from C.

**libraryBase**

is the register or variable that contains the address of the base of the library.

For example, if you had stored the base of intuition.library in a variable called IntuitionBase (as C programmers do)

```
LINKLIB _LVODisplayBeep,IntuitionBase
```

will produce the same code sequence as shown above. This macro is located in the file *exec/libraries.i*. Notice that it handles *only* the linkage to the routine. It does not save any registers or preload any registers for passing values to the routine.

By convention, A6 *must* contain the library pointer when a library routine is called. This allows any library routine to locate the library and access its data or any of its other entry points. Registers A0, A1, D0, and D1 may be used as scratch registers by any routine. All other registers, both address and data, if used in a routine, will be saved and restored before exit.

## USING A LIBRARY TO REFERENCE DATA

You can use the LibPtr to reference a data segment associated with a library by specifying a positive offset from LibPtr, such as:

```
move.l <libptr>,A1          ; Move library base
move.l <offset>(A1),D0       ; Retrieve data located at <offset>
```

OR in C

```
value = LibBase->offset
```

where offset is a label defined in the library base include file.

Library data is not usually accessed directly from outside of a library, but rather is accessed by the routines that are part of the library itself. The sample code retrieves data specifically associated with that library.

## NOTE

Different languages have different interface requirements. This example shows only a typical assembly language interface. When you design your own libraries, you may decide how the associated data segment is to be used. The system itself places no restrictions on its use.

## CACHING LIBRARY POINTERS

To make your library calls more efficient, you may cache various pointers. These pointers are the libPtr itself (because the library node, while it is open, may not be moved) and the address within the library at which a jump instruction is located (because offsets from the libPtr do not change). You must not, however, cache the jump vector from within the library.

## CLOSING A LIBRARY

When your task has finished using a specific library, your program should call the routine **CloseLibrary()**. This call takes the form:

```
CloseLibrary(libPtr)
           A1
```

where **libPtr** is the value returned to you by the call to **OpenLibrary()**.

You close a library to tell the library manager that there is one fewer task currently using that library. If there are no tasks using a library, it is possible for the system, on request, to purge that library and free up the memory resources it is currently using. Each successful open should be matched by exactly one close. Do not attempt to use a library pointer after you have closed that library.

## Adding a Library

Exec provides several methods for adding your own libraries to the system library list. It is possible to call **LoadSeg()** (a dos.library function) to load your library and then use the Exec **MakeLibrary()** and **AddLibrary()** functions to initialize your library and add it to the system. **MakeLibrary()** allocates space for the code vectors and data area, initializes the library node, and initializes the data area according to your specifications, returning to you a library base pointer. The base pointer may then be passed to **AddLibrary()** to add your library to the system.

However, the more common method of initializing and adding a library or device is the automatic method provided through the use of a **Resident** structure or *romtag* (defined in *exec/resident.h* and *.i*). Use of a *romtag* allows you to simply place your library or device in a directory (default LIBS: for libraries) and have it automatically loaded and initialized when it is opened by an application.

## RESIDENT (ROMTAG) STRUCTURE

A *romtag*'ed library should start with **MOVEQ #-1,D0** (to safely return an error if a user tries to execute the file), followed by a Resident structure:

```
STRUCTURE RT,0
  UWORD RT_MATCHWORD      * romtag identifier (== $4AFC)
  APTR  RT_MATCHTAG       * pointer to the above UWORD (RT_MATCHWORD)
  APTR  RT_ENDSKIP        * usually ptr to end of your code
  UBYTE RT_FLAGS          * usually RTF_AUTOINIT
  UBYTE RT_VERSION        * release version number (example: 33)
  UBYTE RT_TYPE           * type of module (NT_LIBRARY)
  BYTE  RT_PRI            * initialization priority (example: 0)
  APTR  RT_NAME            * pointer to node name ("my.library")
  APTR  RT_IDSTRING        * pointer to id string ("name ver.rev (date)")
  APTR  RT_INIT            * pointer to init code or AUTOINIT tables
  LABEL RT_SIZE
```

If you wish to perform **MakeLibrary()** and **AddLibrary()** yourself, then your **RT\_FLAGS** will not include **RTF\_AUTOINIT**, and **RT\_INIT** will be simply be a pointer to your own initialization code. To have Exec automatically perform these functions for you, set the **RTF\_AUTOINIT** flag in your Resident structure, and point

RT\_INIT to a set four longwords containing the following:

**dataSize**

This is the size of your library data area, i.e., the combined size of the standard **Library** node structure plus your own library-specific data.

**vectors**

This is a pointer to a table of pointers to your library's functions, terminated with a -1. If the first word of the table is -1, then the table is interpreted as a table of words specifying the relative displacement of each function entry point from the start of the table. Otherwise it is treated as a table of longword address pointers to the functions. **vectors** must specify a valid table address.

**structure**

This parameter points to the base of an **InitStruct()** data region. That is, it points to the first location within a table that the **InitStruct()** routine can use to initialize your **Library** node structure, library-specific data, and other memory areas. **InitStruct()** will typically be used to initialize the data segment of the library, perhaps forming data tables, task control blocks, I/O control blocks, etc. If this entry is a 0, then **InitStruct()** is not called.

**initFunction**

This parameter points to a routine that is to be executed after the library node has been allocated and the code and data areas have been initialized. When this routine is called, the **libAddr** (address of this library) is placed into data register D0. If **initFunction** is zero, no init routine is called.

Complete source code for an RT\_AUTOINIT library may be found in the Addison-Wesley *Includes and Autodocs* manual.

## MINIMUM SUBSET OF LIBRARY CODE VECTORS

The first four code vectors of a library must be the following entries: **OPEN**, **CLOSE**, **EXPUNGE**, and one reserved entry.

**OPEN** is the entry point called when you use the command **OpenLibrary()**. In most libraries, **OPEN** increments the library variable **lib\_OpenCnt**. This variable is also used by **CLOSE** and **EXPUNGE**.

**CLOSE** is the entry point called when you use the command **CloseLibrary()**. It decrements the library variable **lib\_OpenCnt** and may do a delayed **EXPUNGE**.

**EXPUNGE** prepares the library for removal from the system. This often includes deallocating memory resources that were reserved during initialization. **EXPUNGE** not only frees the memory allocated for data structures, but also the areas reserved for the library node itself.

**RESERVED**

is a fourth function vector reserved for future use. It must always return zero.

## STRUCTURE OF A LIBRARY NODE

A library node contains all the information that the system needs to manage a library. Here is the library structure as it appears in the *exec/libraries.h* include file:

```
struct Library
{
    struct    Node lib_Node;
    UBYTE    lib_Flags;
    UBYTE    lib_pad;
    UWORD    lib_NegSize;           /* number of bytes before library */
    UWORD    lib_PosSize;          /* number of bytes after library */
    UWORD    lib_Version;
    UWORD    lib_Revision;
    APTR     lib_IdString;
    ULONG    lib_Sum;              /* the checksum itself */
    UWORD    lib_OpenCnt;          /* number of current opens */
};

/* meaning of the flag bits: */

/* a task is currently running a
** checksum on this library
** (system maintains this flag)
*/
#define LIBF_SUMMING (1 << 0)

/* one or more entries have been changed
** in the library code vectors used by
** SumLibrary (system maintained flag)
*/
#define LIBF_CHANGED (1 << 1)

/* a checksum fault should cause
** a system panic (library flag)
*/
#define LIBF_SUMUSED (1 << 2)

/* a user has requested expunge but
** another user still has the library open.
** (maintained by library)
*/
#define LIBF_DELEXP (1 << 3)
```

## CHANGING THE CONTENTS OF A LIBRARY

After a library has been constructed and linked to the system library list, you can use the routine **SetFunction()** either to add or to replace the contents of one of the library vectors. The format of this routine is as follows:

```
SetFunction( Library, FuncOffset, FuncEntry)
            A1      A0      D0
```

where

**Library**

is a pointer to the library in which a function entry is to be changed.

**FuncOffset**

is the offset (negative) at which the entry to be changed is located.

**FuncEntry**

is a longword value that is the absolute address of the routine that is to be inserted at the selected position in the library code vectors.

When you use **SetFunction()** to modify a function entry in a library, it automatically recalculates the checksum of the library.

**WARNING**

**SetFunction()** is for advanced users only. It is very difficult to remove a **SetFunction()** because other tasks may be executing your code, and also because additional **SetFunction()**'s may have occurred on the same function. Also note that certain libraries (for example **dos.library**) and some individual library function vectors are of non-standard format and can not be replaced via **SetFunction()**.

## Relationship of Libraries to Devices

A device is an interface specification and an internal data structure based on the **Library** structure. The interface specification defines a means of device control. The structures of libraries and devices are so similar that the routine **MakeLibrary()** is used to construct both libraries and devices. Devices require the same basic four code vectors but have additional code vectors that must be located in specific positions in the code vector table. The functions that devices are expected to perform, at minimum, are shown in the Exec "Input/Output" chapter. Complete sample device source code is provided in the *Includes and Autodocs* manual.



# Chapter 15

## Exec: Memory Allocation

### Introduction

Exec manages all of the random access memory (RAM) in the system. When your application needs memory space, it provides the size and requirements to Exec. Exec will assign a range of memory for your exclusive use. Your application is responsible for freeing this memory before exit.

Areas of free memory are maintained as a special linked list of free regions. Each memory allocation function returns the starting address of a block of memory at least as large as the size that you requested to be allocated. The allocated memory is not tagged or initialized in any way unless you have specified, for example, **MEMF\_CLEAR**.

You must return allocated memory to the system when your task completes. As noted above, the system only keeps track of available system memory and has no idea which task may have allocated memory and not returned it to the system free list. If your program does not return allocated memory when its task exits, that memory is unavailable until the system is powered down or reset.

When you ask for memory to be allocated, the system always allocates blocks of memory in even multiples of eight bytes. If you request more or less than eight bytes, your request is always rounded up to the nearest multiple of eight. In addition, the address at which the memory deallocation is made is always rounded down to the nearest even multiple of eight bytes.

## WARNING

Do not depend on this size! Future revisions of the system may require a different size to guarantee alignment of the requested area to a specific boundary. You *can* depend upon allocation being aligned to at least a longword boundary.

# Using Memory Allocation Routines

## MEMORY REQUIREMENTS

You must tell the system about your memory requirements when requesting a chunk of memory. There are four memory requirement possibilities. Three of these tell where within the hardware address map memory is to be allocated. The fourth, **MEMF\_CLEAR**, tells the allocator that this memory space is to be zeroed before the allocator returns the starting address of that space.

The memory requirements that you can specify are listed below:

### **MEMF\_CHIP**

This indicates a memory block within the address range of the Amiga custom chips. Unless this flag is set properly, your code will fail on machines with expanded memory. Chip memory is required for any data that will be accessed by custom chip DMA. This includes floppy disk buffers, screen memory, images that will be blitted, sprite data, copper lists, and audio data.

### **MEMF\_FAST**

This indicates a memory block outside of the range that the special purpose chips can access. "FAST" means that the special-purpose chips do not have access to the memory and thus cannot cause processor bus contention, therefore processor access will likely be faster. Since the flag specifies memory that the custom chips cannot access, this flag is mutually exclusive with the **MEMF\_CHIP** flag.

### **MEMF\_PUBLIC**

This indicates that the memory requested is to be used for different tasks or interrupt code, such as task control blocks, messages, ports, and so on. The designation **MEMF\_PUBLIC** should be used to assure compatibility with future versions of the system.

### **MEMF\_CLEAR**

This indicates that memory is to be initialized with zeros before returning.

If no preferences are specified, **MEMF\_FAST** is assumed first, then **MEMF\_CHIP**.

## WARNING

Always check the result of any memory allocate to be sure the type and amount of memory requested is available. Failure to do so will lead to trying to use an non-valid pointer.

## SAMPLE CALLS FOR ALLOCATING SYSTEM MEMORY

The following examples show how to allocate memory.

```
APTR mypointer, anotherptr;

mypointer = AllocMem(100, 0);
if(!mypointer)
{
    /* COULDN'T GET MEMORY, EXIT */
}
```

**AllocMem()** returns the address of the first byte of a memory block that is at least 100 bytes in size or null if there is not that much free memory. Because the requirement field is specified as 0, memory will be allocated from any one of the system-managed memory regions.

```
anotherptr = (APTR)AllocMem(1000, MEMF_CHIP | MEMF_CLEAR);
if (!anotherptr)
{
    /* COULDN'T GET MEMORY, EXIT */
}
```

Memory is allocated only out of chip-accessible memory; zeroes are filled into memory space before the address is returned. If the system free-list does not contain enough contiguous memory bytes in an area matching your requirements and of the size you have requested, **AllocMem()** returns a zero. You *must* check for this condition.

### NOTE

Do not attempt to allocate or deallocate system memory from within interrupt code. The “Interrupts” chapter explains that an interrupt may occur at any time, even during a memory allocation process. As a result, system data structures may not necessarily be internally consistent.

## SAMPLE FUNCTION CALLS FOR FREEING SYSTEM MEMORY

The following examples free the memory chunks shown in the earlier call to the system allocation routines.

```
FreeMem(mypointer, 100);

FreeMem(anotherptr, 1000);
```

### NOTE

**FreeMem()** rounds down the size of the request to free memory in the same way as **AllocMem()** rounds up the size, thereby maintaining a consistent memory free-list.

The routine **FreeMem()** returns no status. However, if you attempt to free a memory block in the middle of a chunk that the system believes is already free, you will cause a system crash.

## Using Memory Information Routines

To determine the amount of memory available and the type of a particular block of memory, the memory information routines **AvailMem()** and **TypeOfMem()** are provided.

### MEMORY REQUIREMENTS

For the memory information routines the same memory type flags are valid as for the allocation routines. In addition **MEMF\_LARGEST** can be added to the requirement argument in the **AvailMem()** routine to find out what the largest available memory block of a particular type is.

### SAMPLE CALLS INFORMATION ROUTINES

The following example shows how to find out how much memory of a particular type is available in the system.

```
ULONG size;

size = AvailMem(MEMF_CHIP | MEMF_LARGEST);
```

**AvailMem()** returns the size of the largest chunk of available chip memory.

#### NOTE

Because of the effect of multitasking the returned value doesn't necessarily represent the amount of memory available at that moment.

The following example shows how to determine the type of memory of a specified memory address.

```
UWORD memtype;

memtype = TypeOfMem(0x090000);
if ((memtype & MEMF_CHIP) == MEMF_CHIP)
{
    /* It's chip memory */
}
```

**TypeOfMem()** returns the attribute of the memory address. If no valid memory address is specified, a zero will be returned. This routine is normally used to determine if a particular chunk of memory is in chip memory.

## Using Memory Copy Routines

For memory block copies, the **CopyMem()** and **CopyMemQuick()** functions can be used.

## SAMPLE CALLS FOR COPYING SYSTEM MEMORY

The following samples show how to use the copying routines.

```
APTR source, target;

source = AllocMem(1000, MEMF_CLEAR);
target = AllocMem(1000, MEMF_CHIP);

CopyMem(source, target, 1000);
```

**CopyMem()** copies the specified number of bytes from the source data region to the target data region. The pointers to the regions can be on arbitrary alignments. An attempt is made to optimize large copies with more efficient copies. Byte copies are being used for small copies, parts of larger copies, or the entire copy if the source and target regions are misaligned with respect to each other. Very small copies are better done with in-line code.

```
CopyMemQuick(source, target, 1000);
```

**CopyMemQuick()** performs an optimized copy of the specified number of bytes from the source data region to the target data region. The source and target pointers must be longword aligned and the size must be an integral number of longwords.

### NOTE

Neither routine supports arbitrary overlapping copying.

## SUMMARY OF SYSTEM CONTROLLED MEMORY HANDLING ROUTINES

### **AllocMem() and FreeMem()**

These are system-wide memory allocation and deallocation routines. They use a memory free-list owned and managed by the system.

### **AvailMem()**

This routine returns the number of free bytes in a specified type of memory.

### **TypeOfMem()**

This routine returns the memory attributes of a specified memory address.

### **CopyMem() and CopyMemQuick()**

**CopyMem()** is a general purpose memory copy routine. **CopyMemQuick()** is an optimized version of **CopyMemQuick()**, but has restrictions on the size and alignment of the arguments.

## Allocating Multiple Memory Blocks

Exec provides the routines **AllocEntry()** and **FreeEntry()** to allocate multiple memory blocks in a single call. **AllocEntry()** accepts a data structure called a **MemList**, which contains the information about the size of the memory blocks to be allocated and the requirements, if any, that you have regarding the allocation. The **MemList**

structure is found in the include file *exec/memory.h* and is defined as follows:

```
struct MemList
{
    struct Node    ml_Node;
    UWORD          ml_NumEntries;    /* number of MemEntrys */
    struct MemEntry ml_ME[1];        /* where the MemEntrys begin*/
};
```

where:

#### **Node**

allows you to link together multiple **MemLists**. However, the node is ignored by the routines **AllocEntry()** and **FreeEntry()**.

#### **ml\_NumEntries**

tells the system how many **MemEntry** sets are contained in this **MemList**. Notice that a **MemList** is a variable-length structure and can contain as many sets of entries as you wish.

The **MemEntry** structure looks like this:

```
struct MemEntry
{
    union
    {
        ULONG    meu_Reqs;    /* the AllocMem requirements */
        APTR      meu_Addr;    /* address of your memory */
    } me_Un;
    ULONG    me_Length;    /* the size of this request */
};
```

### **Sample Code for Allocating Multiple Memory Blocks**

```
#include <exec/types.h>
#include <exec/memory.h>

#ifdef LATTICE
#include <proto/all.h>
#include <stdio.h>
#include <stdlib.h>
#endif

#define ALLOCERROR 0x80000000

struct MemList *mymemlist;    /* pointer to a MemList */

/* define a new structure because C cannot initialize unions */
struct MyNeeds
{
    struct MemList mn_head;    /* one entry in the header */
    struct MemEntry mn_body[3]; /* additional entries follow directly as */
                                /* part of the same data structure */
} myneeds;

VOID main (VOID);

VOID main(VOID)
{
    myneeds.mn_head.ml_NumEntries = 4; /* 4! Since the MemEntry starts at 1! */

    myneeds.mn_body[0].me_Reqs = MEMF_CHIP | MEMF_CLEAR;
    myneeds.mn_body[0].me_Length = 100000;
```

```

myneeds.mn_body[1].me_Reqs  = MEMF_FAST | MEMF_CLEAR;
myneeds.mn_body[1].me_Length = 200000;

myneeds.mn_body[2].me_Reqs  = MEMF_PUBLIC;
myneeds.mn_body[2].me_Length = 25000;

/* saying 'struct MemEntry mn_body[3]' is simply a way of adding
 * extra MemEntry structures contiguously at the end of the first
 * such structure at the end of the MemList. Thus members of the
 * MemList of type MemEntry can be referenced to in C as additional
 * members of the 'me[]' data structure.
 */

mymemlist = (struct MemList *)AllocEntry((struct MemList *)&myneeds);

if ((ULONG)mymemlist & ALLOCERROR)
{
    printf("AllocEntry FAILED\n"); /* 'error' bit 31 is set */
    exit(200);                    /* see below */
}

/* we got the memory we wanted. We can use FreeEntry() now */
FreeEntry(mymemlist);
}

```

**AllocEntry()** returns a pointer to a new **MemList** of the same size as the **MemList** that you passed to it. For example, ROM code can provide a **MemList** containing the requirements of a task and create a RAM-resident copy of the list containing the addresses of the allocated entries. The pointer to the **MemList** is used as the argument for **FreeEntry()** to free the memory blocks.

#### NOTE

The **MemList** structure used by assembly programmers is slightly different; it has no **MemEntry** structure.

#### Result of Allocating Multiple Memory Blocks

The **MemList** created by **AllocEntry()** contains **MemEntry** entries. **MemEntrys** are defined by a union statement, which allows one memory space to be defined in more than one way.

If **AllocEntry()** returns a value with bit 31 clear, then all of the **meu\_Addr** positions in the returned **MemList** will contain valid memory addresses meeting the requirements you have provided.

To use this memory area, you would use code similar to the following:

```

struct MemList *ml;
APTR    mydata, moredata;

if (((ml & (1L<<31)) < 0)
{
    mydata      = ml->ml_me[0].me_Addr;
    moremydata = ml->ml_me[1].me_Addr;
}
else
{
    exit(200); /* error during AllocEntry */
}

```

If **AllocEntry()** has problems while trying to allocate the memory you have requested, instead of the address of a new **MemList**, it will return the memory requirements value with which it had the problem. Bit 31 of the value returned will be set, and no memory will be allocated. Entries in the list that were already allocated will be freed.

## Memory Allocation and Tasks

If you want to take advantage of Exec's automatic cleanup, use the **MemList** and **AllocEntry()** facility to do your dynamic memory allocation.

In the task control block structure, there is a list header named **tc\_MemEntry**. This is the list header that you initialize to point to the **MemLists** that your task has created by call(s) to **AllocEntry()**. Here is a short program segment that handles task memory list header initialization only. It assumes that you have already run **AllocEntry()** as shown in the simple **AllocEntry()** example above.

```
struct Task *tc;
struct MemList *ml;

NewList(tc->tc_MemEntry); /* Initialize the task's memory list header. *
                          * Do this once only!                          */
AddTail(tc->tc_MemEntry, ml);
```

Assuming that you have only used the **AllocEntry()** method (or **AllocMem()**) and built your own custom **MemList**, the system now knows where to find the blocks of memory that your task has dynamically allocated. The **RemTask()** function automatically frees all memory found on **tc\_MemEntry**.

### NOTE

The **amiga.lib** "CreateTask()" function sets up and initializes a **MemList** for you.

## SUMMARY OF MULTIPLE MEMORY BLOCKS ALLOCATION ROUTINES

### **AllocEntry()** and **FreeEntry()**

These are routines for allocating and freeing multiple memory blocks with a single call.

### **InitStruct()**

This routine initializes memory from data and offset values in a table. Typically only assembly language programs benefit from using this routine. See the *ROM Kernel Manual: Includes & Autodocs* for more details.

## Managing Memory With **Allocate()** And **Deallocate()**

**Allocate()** and **Deallocate()** use a memory region header, called **MemHeader**, as part of the calling sequence. You can build your own local header to manage memory locally. This structure takes the form:



```

struct MemHeader
{
    struct Node      mh_Node;
    UWORD            mh_Attributes; /* characteristics of this region */
    struct MemChunk *mh_First;      /* first free region */
    APTR             mh_Lower;      /* lower memory bound */
    APTR             mh_Upper;      /* upper memory bound + 1 */
    ULONG            mh_Free;       /* total number of free bytes */
};

```

where

**mh\_Attributes**  
is ignored by **Allocate()** and **Deallocate()**.

**mh\_First**  
is the pointer to the first **MemChunk** structure.

**mh\_Lower**  
is the lowest address within the memory block. This must be a multiple of eight bytes.

**mh\_Upper**  
is the highest address within the memory block + 1. The highest address will itself be a multiple of eight if the block was allocated to you by **AllocMem()**.

**mh\_Free**  
is the total free space.

This structure is included in the include files *exec/memory.h* and *exec/memory.i*.

The following sample program fragment shows the correct initialization of a **MemHeader** structure. It assumes that you wish to allocate a block of memory from the global pool and thereafter manage it yourself using **Allocate()** and **Deallocate()**.

```

#include <exec/types.h>
#include <exec/memory.h>

#ifdef LATTICE
#include <proto/all.h>
#include <stdio.h>
#include <stdlib.h>
#endif

#define BLOCKSIZE 4000 /* or whatever you want */
VOID main(VOID);

VOID main(VOID)
{
    struct MemHeader *mh;
    struct MemChunk *mc;
    APTR block1;
    APTR block2;

    /* Get the MemHeader needed to keep track of our new block */
    mh = (struct MemHeader *)AllocMem((LONG)sizeof(struct MemHeader), MEMF_CLEAR);
    if (!mh)
        exit(10);

    /* Get the actual block the above MemHeader will manage */
    mc = (struct MemChunk *)AllocMem(BLOCKSIZE, 0);
    if (!mc)
    {
        FreeMem(mh, (LONG)sizeof(struct MemHeader)); exit(10);
    }
}

```

```

    }

    mh->mh_Node.ln_Type = NT_MEMORY;
    mh->mh_Node.ln_Name = "myname";
    mh->mh_First = mc;
    mh->mh_Lower = (APTR)mc;
    mh->mh_Upper = (APTR)(BLOCKSIZE + (ULONG)mc);
    mh->mh_Free = BLOCKSIZE;

    /* Set up first chunk in the freelist */
    mc->mc_Next = NULL;
    mc->mc_Bytes = BLOCKSIZE;

    block1 = (APTR)Allocate(mh, 20);
    block2 = (APTR)Allocate(mh, 314);

    printf("mh = %lx mc=%lx\n", mh, mc);
    printf("block1 = %lx block2 = %lx\n", block1, block2);

    FreeMem(mh, (LONG)sizeof(struct MemHeader));
    FreeMem(mc, (LONG)BLOCKSIZE);
}

```

### NOTE

Only free memory is “tagged” using a **MemChunk** linked list. Once memory is allocated, the system has no way of determining which task now has control of that memory.

If you allocate a large chunk from the system, use **tc\_MemEntry** or assure that in your **finalPC** routine (specified when you perform **AddTask()**) you deallocate this large chunk as your task exits. Thus, local memory allocation and deallocation from a single large block can perhaps save some bookkeeping—that which might have been required if you had extensively used **AllocMem()** and **FreeMem()** instead. This can most easily be done by recording the allocated block in your task’s **tc\_MemEntry** structure.

## Allocating Memory at an Absolute Address

For special advanced applications, **AllocAbs()** is provided. With the **AllocAbs()** routine a memory block starting at a specified absolute memory address can be allocated. If the memory is already allocated, or if there is not enough memory available for the request, **AllocAbs()** returns a zero. Here is an example call:

```

APTR absoluteptr;

absoluteptr = (APTR)AllocAbs(10000, 0x2F0000);

if (!(absoluteptr))
{
    /* Couldn't get memory, act accordingly */
}

/* After we're done using it, we can use FreeMem() to free the memory block */

FreeMem(absoluteptr);

```

# **Chapter 16**

## **Exec: Lists and Queues**

This chapter describes Exec lists and queues. A list is an unsorted chain of elements. A queue is a sorted list. A basic understanding of lists and queues is important to understanding Exec itself. Be sure to read the important note on shared lists at the end of this chapter.

### **Introduction**

The Amiga system software operates in a highly dynamic environment of data structures. An early design goal of Exec was to keep the system flexible and open-ended by eliminating artificial boundaries on the number of system structures used. Rather than using static system tables, Exec uses dynamically created structures that are attached to the system as needed. A list can be empty, but never full. This concept is central to the design of Exec.

Exec uses lists to maintain its internal database of system structures. Tasks, interrupts, libraries, devices, messages, I/O requests, and all other Exec data structures are supported and serviced through the consistent application of Exec's list mechanism. Lists have a common data structure, and a common set of functions is used for manipulating them. Because all of these structures are treated in a similar manner, only a small number of list handling functions need be supported by Exec.

## List Structure

A list is composed of a *header* and a doubly-linked chain of elements called *nodes*. The header contains memory pointers to the first and last nodes of the linked chain. The address of the header is used as the handle to the entire list. To manipulate a list, you must provide the address of its header.

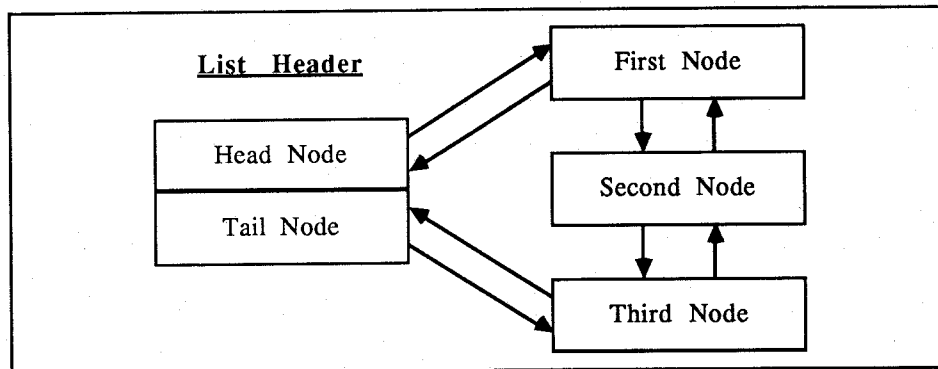


Figure 16-1: Simplified Overview of an Exec List

Nodes may be scattered anywhere in memory. Each node contains two pointers; a successor and a predecessor. As illustrated above, a list header contains two placeholder nodes that contain no data. In an empty list, the head and tail nodes point to each other.

## List Functions

Exec provides a number of symmetric functions for handling lists. There are functions for inserting and removing nodes, for adding and removing head and tail nodes, for inserting nodes in a priority order, and for searching for nodes with a particular name. In the following section, **header** represents a pointer to list header, and **node** represents pointer to a node.

### INSERTION AND REMOVAL

The **Insert()** function is used for inserting a new node into any position in a list. It always inserts the node following a specified node that is already part of the list. For example, **Insert(header,node,pred)** inserts the node **node** after the node **pred** in the specified list. If the **pred** node points to the list header or is **NULL**, the new node will be inserted at the head of the list. Similarly, if the **pred** node points to the **lh\_Tail** of the list, the new node will be inserted at the tail of the list. However, both of these actions can be better accomplished with the functions mentioned in the "Special Case Insertion" section below.

The **Remove()** function is used to remove a specified node from a list. For example, **Remove(node)** will remove the specified node from whatever list it is in. *Please note:* to be removed, a node must actually be in a list. If you attempt to remove a node that is not in a list, you will cause serious system problems.

## **SPECIAL CASE INSERTION**

Although the **Insert()** function allows new nodes to be inserted at the head and the tail of a list, the **AddHead()** and **AddTail()** functions will do so with higher efficiency. Adding to the head or tail of a list is common practice in queue type operations, as in first-in-first-out (FIFO) or last-in-first-out (LIFO or stack) operations. For example, **AddHead(header,node)** would insert the node at the head of the specified list.

## **SPECIAL CASE REMOVAL**

The two functions **RemHead()** and **RemTail()** are used in combination with **AddHead()** and **AddTail()** to create special list ordering. When you combine **AddTail()** and **RemHead()**, you produce a first-in-first-out (FIFO) list. When you combine **AddHead()** and **RemHead()** a last-in-first-out (LIFO or stack) list is produced. **RemTail()** exists for symmetry. Other combinations of these functions can also be used productively.

Both functions remove a node from the list, and return a pointer to the removed node. If the list is empty, the function return a NULL result.

## **MINLIST / MINNODE OPERATIONS**

All of the above functions and macros will work with long or short format node structures. A **MinNode** structure contains only linkage information. A full **Node** structure contains linkage information, as well as type, priority and name fields. The smaller **MinNode** is used where space and memory alignment issues are important. The larger **Node** is used for queues or lists that require a name tag for each node.

## **PRIORITIZED INSERTION (QUEUES)**

The list functions discussed so far do not make use of the priority field in a **Node**. The **Enqueue()** function is equivalent to **Insert()** for a priority sorted list. It performs an insert on a priority basis, keeping the higher-priority nodes towards the head of the queue. All nodes passed to this function must have their priority and name assigned prior to the call. **Enqueue(header,node)** inserts the node into the prioritized list after the last node of same or higher priority.

As mentioned earlier, the highest-priority node is at the head of the queue, and the lowest-priority node is at the tail of the queue. The **RemHead()** function will remove the highest-priority node, and **RemTail()** will remove the lowest-priority node.

### **NOTE**

If you add a node that has the same priority as another node in the queue, **Enqueue()** will use FIFO ordering. The new node is inserted following the last node of equal priority.

## SEARCHING BY NAME

Because many lists contain nodes with symbolic names attached (via the `ln_Name` field), it is possible to find a node by its name. This naming technique is used throughout Exec for such nodes as tasks, libraries, devices, and resources.

The `FindName()` function searches a list for the first node with a given name. For example, `FindName(header, "Furrbol")` returns a pointer to the first node named "Furrbol." If no such node exists, a NULL is returned. The case of the name characters is significant; "foo" is different from "Foo."

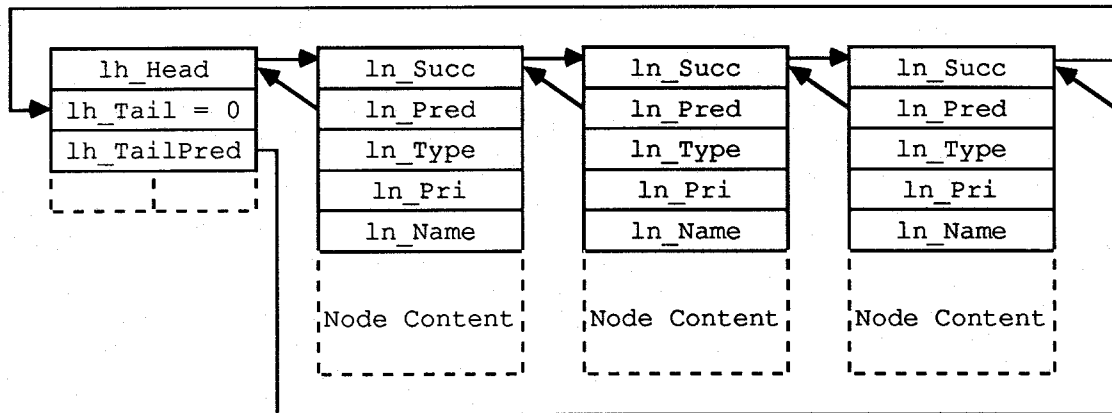


Figure 16-2: Complete Sample List Showing all Interconnections

## NODE STRUCTURE DEFINITION

A node structure is divided into three parts: linkage, information, and content. The linkage part contains memory pointers to the node's successor and predecessor nodes. The information part contains the node type, the priority, and a name pointer. The content part stores the actual data structure of interest. For nodes that require linkage only, a small **MinNode** structure is used:

```
struct MinNode
{
    struct MinNode *mln_Succ;
    struct MinNode *mln_Pred;
};
```

where

**mln\_Succ**  
points to the next node in the list (successor).

**mln\_Pred**  
points to the previous node in the list (predecessor).

When a type, priority, or name is required, a full-featured **Node** structure is used:

```
struct Node
{
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE      ln_Type;
    BYTE       ln_Pri;
    char       *ln_Name;
};
```

where the additional fields are used as follows:

**ln\_Type**  
defines the type of the node (see *exec/types.h* for a list).

**ln\_Pri**  
specifies the priority of the node (+127 (highest) to -128 (lowest)).

**ln\_Name**  
points to a printable name for the node (a NULL terminated C string).

The node structure is usually prepended to the content part. For example, the **Exec Interrupt** structure is defined as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR       is_Data;
    VOID       (*is_Code)();
};
```

Here the **is\_Data** and **is\_Code** fields represent the useful content of the node. Since the **Interrupt** structure begins with a **Node** structure, it may be passed to any of the list manipulation functions. Content may be appended to either a **MinNode** or a **Node** structure.

## NODE INITIALIZATION

Before linking a node into a list, certain fields may need initialization. Initialization consists of setting the **ln\_Type**, **ln\_Pri**, and **ln\_Name** fields to their appropriate values (If you are using **MinNode** structures, these fields do not exist). The successor and predecessor fields do not require initialization.

The **ln\_Type** field contains the data type of the node. This indicates to Exec (and other interested subsystems) the type, and hence the structure, of the content portion of the node. The standard system types are defined in the *exec/nodes.h* include file. Some examples of standard system types are **NT\_TASK**, **NT\_INTERRUPT**, **NT\_DEVICE**, and **NT\_MSGPORT**.

The **ln\_Pri** field uses a signed numerical value ranging from +127 to -128 to indicate the priority of the node (relative to other nodes in the same list). Higher-priority nodes have more positive values; for example, 127 is the highest priority, zero is nominal priority, and -128 is the lowest priority. Some Exec lists are kept sorted by priority order. In such lists, the highest-priority node is at the head of the list, and the lowest-priority node is at the tail of the list. For most Exec node types, priority is not used. In such cases it is a good practice to initialize the priority field to zero.

The **ln\_Name** field is a pointer to a NULL-terminated string of characters. Node names are used for identification, and to bind symbolic names to actual nodes. Names are also useful for debugging purposes, so it is a good idea to provide every node with a name. Take care to provide a valid name pointer; Exec does not copy name strings.

For example, a Library structure consists of a Node followed by library-specific data; the **ln\_Name** field of the Node contains the library name. Task names are stored in the **ln\_Name** of the task structure. Tasks in the Exec wait queue are sorted with **Enqueue()**.

Here is a C example showing how you might initialize a node called **myInt**, an instance of the interrupt structure defined above:

```
struct Interrupt myInt;

myInt.is_Node.ln_Type = NT_INTERRUPT;
myInt.is_Node.ln_Pri  = -10;
myInt.is_Node.ln_Name = "sample.interrupt";
```

## LIST HEADER STRUCTURE DEFINITION

As mentioned earlier, a list header maintains memory pointers to the first and last nodes of the linked chain of nodes. It also serves as a handle for referencing the entire list. The minimum list header ("**mlh\_**") and the full-featured list header ("**lh\_**") are generally interchangeable.

Here is the C-structure of a minimum list header:

```
struct MinList
{
    struct MinNode *mlh_Head;
    struct MinNode *mlh_Tail;
    struct MinNode *mlh_TailPred;
};
```



where:

**mlh\_Head** points to the first node in the list.

**mlh\_Tail** is always NULL.

**mlh\_TailPred** points to the last node in the list.

In a few limited cases a full-featured **List** structure will be required:

```
struct List
{
    struct Node *lh_Head;
    struct Node *lh_Tail;
    struct Node *lh_TailPred;
    UBYTE      lh_Type;
    UBYTE      lh_Pad;
};
```

where the additional fields are used as follows;

**lh\_Type** defines the type of nodes within the list (see *exec/types.h*).

**lh\_pad** is a structure alignment byte.

One subtlety here must be explained further. The list header is constructed in a efficient, but confusing manner. Think of the header as a structure containing the head and tail nodes for the list. The head and tail nodes are placeholders, and never carry data. The head and tail portions of the header actually overlap in memory. **lh\_Head** and **lh\_Tail** form the head node. **lh\_Tail** and **lh\_TailPred** form the tail node. This makes it very easy to find the start or end of the list, and eliminates any special cases for insertion or removal.

The **lh\_Head** and **lh\_Tail** fields of the list header act like the **ln\_Succ** and **lh\_Pred** fields of a node. The **lh\_Tail** field is set permanently to NULL, indicating that the head node is indeed the first on the list — that is, it has no predecessors. See figure 3 below.

Likewise, the **lh\_Tail** and **lh\_TailPred** fields of the list header act like the **ln\_Succ** and **lh\_Pred** fields of a node. Here the NULL **lh\_Tail** indicates that the tail node is indeed the last on the list — that is, it has no successors. See figure 3 below.

## HEADER INITIALIZATION

List headers must be properly initialized before use. It is not adequate to initialize the entire header to zero. The head and tail entries must have specific values. The header must be initialized as follows:

1. Set the **lh\_Head** field to the address of **lh\_Tail**.
2. Clear the **lh\_Tail** field.
3. Set the **lh\_TailPred** field to the address of **lh\_Head**.
4. Set **lh\_Type** to the same data type as the nodes to be kept the list. (Unless you are using a **MinList**).

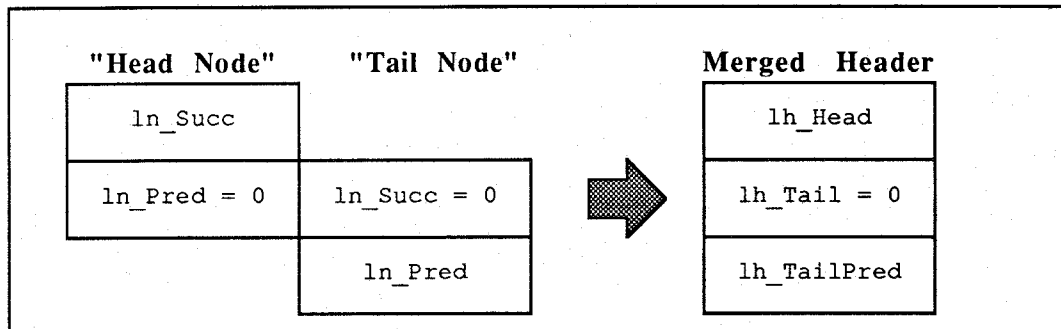


Figure 16-3: List Header Overlap

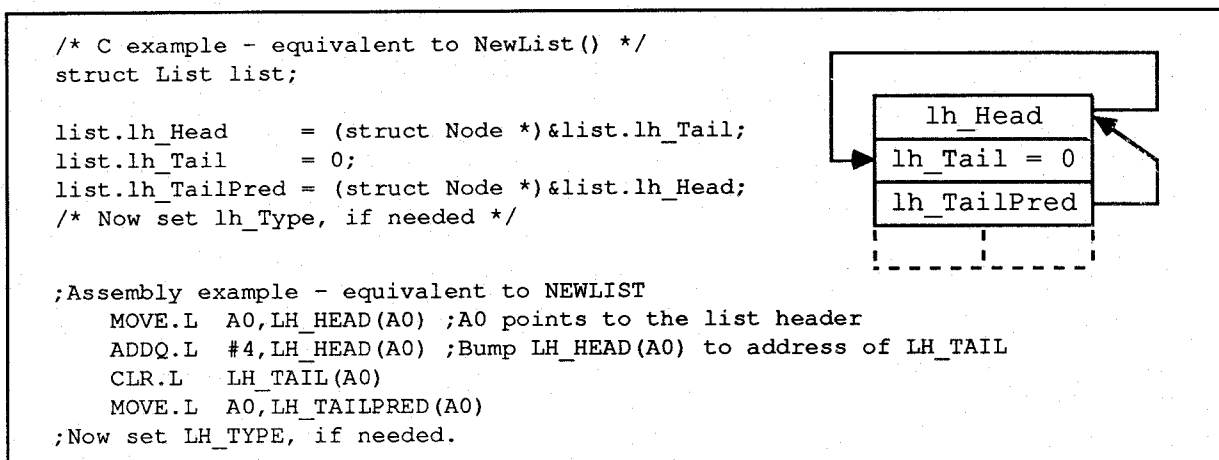


Figure 16-4: Initializing a List Header Structure

The sequence of assembly instructions in figure 4 is equivalent to the macro **NEWLIST**, contained in the include file *exec/lists.i*. Since the **MinList** structure is the same as the **List** structure except for the type and pad fields, this sequence of assembly language code will work for both structures. The sequence performs its function without destroying the pointer to the list header in A0 (which is why **ADDQ.L** is used). This function may also be accessed from C as a call to **NewList(header)**, where **header** is the address of a list header.

## MORE ON THE USE OF NAMED LISTS

To find multiple occurrences of nodes with identical names, the **FindName()** function is called multiple times. For example, if you want to find all the nodes with the name pointed to by **name**:

```
VOID DisplayName(struct List *list, UBYTE *name)
{
    struct Node *node;

    if (node = FindName(list, name))
        while (node)
        {
            printf("Found %s at location %lx\n", node->ln_Name, node);
            node = FindName((struct List *)node, name);
        }
    else printf("No node with name %s found.\n", name);
}
```

Notice that the second search uses the node found by the first search. The **FindName()** function *never* compares the specified name with that of the starting node. It always begins the search with the successor of the starting point.

## List Macros for Assembly Language Programmers

Assembly language programmers may want to optimize their code by using assembly code list macros. Because these macros actually embed the specified list operation into the code, they result in slightly faster operations. The file *exec/lists.i* contains the recommended set of macros. For example, the following instructions implement the **REMOVE** macro:

```
MOVE.L LN_SUCC(A1), A0      ; get successor
MOVE.L LN_PRED(A1), A1     ; get predecessor
MOVE.L A0, LN_SUCC(A1)     ; fix up predecessor's succ pointer
MOVE.L A1, LN_PRED(A0)    ; fix up successor's pred pointer
```

## Empty Lists

It is often important to determine if a list is empty. This can be done in many ways, but only two are worth mentioning. If either the **lh\_TailPred** field is pointing to the list header or the **ln\_Succ** field of the **lh\_Head** is **NULL**, then the list is empty.

In C, for example, these methods would be written as follows:

```
if (list->lh_TailPred == (struct Node *)list)
    printf("list is empty\n");
```

or

```
if (NULL == list->lh_Head->ln_Succ)
    printf("list is empty\n");
```

In assembly code, if A0 points to the list header, these methods would be written as follows:

```
CMP.L    LH_TAILPRED(A0),A0
BEQ      list_is_empty
```

or

```
MOVE.L    LH_HEAD(A0),A1
TST.L     LN_SUCC(A1)
BEQ      list_is_empty
```

Because LH\_HEAD and LN\_SUCC are both zero offsets, the second case may be simplified or optimized by your assembler.

## Scanning a List

Occasionally a program may need to scan a list to locate a particular node, find a node that has a field with a particular value, or just print the list. Because lists are linked in both the forward and backward directions, the list can be scanned from either the head or tail.

Here is a code fragment that uses a for loop to print the names of all nodes in a list:

```
struct List *list;
struct Node *node;

for (node = list->lh_Head ; node->ln_Succ ; node = node->ln_Succ)
    printf("%lx -> %s\n",node,node->ln_Name);
```

A common mistake is to process the head or tail nodes. Valid data nodes have non-NULL successor and predecessor pointers. The above loop exits when node->ln\_Succ is NULL. Another common mistake is to free a node from within a loop, then reference the free memory to obtain the next node pointer. An extra temporary pointer solves this second problem.

In assembly code, it is more efficient to use a look-ahead cache pointer when scanning a list. In this example the list is scanned until the first zero-priority node is reached:

```
scan:    MOVE.L    (A1),D1          ; first node
         MOVE.L    D1,A1
         MOVE.L    (A1),D1          ; lookahead to next
         BEQ.S     not_found        ; end of list...
         TST.B     LN_PRI(A1)
         BNE.S     scan
         ...
         ; found one

not_found:
```

## Full Example

```
/* Lists Example - Creates a list, adds some nodes, then displays them.
 * Compile with Lattice C 5.04: LC -Lt -catsfq
 */
#include <exec/types.h>
#include <exec/lists.h>
#include <exec/nodes.h>
#include <exec/memory.h>
#ifdef LATTICE
#include <proto/exec.h>
#include <string.h>
#include <stdio.h>
int CXBRK(VOID) { return(0); }
void main(void);
#endif

struct MyNodes {
    struct Node My_Node;      /* System Node structure */
    UBYTE data[62];          /* Node-specific data */
};

#define NAME_NODE_ID 100     /* The type of our "MyNodes"... */

/* Allocate a MyNode structure, copy the given name into the structure,
 * then add it the specified list. This example does not provide an
 * error return for the out of memory condition. */
void AddName(struct List *list, UBYTE *name)
{
    struct MyNodes *name_node;

    if (!(name_node = AllocMem(sizeof(struct MyNodes), MEMF_CLEAR) ))
        printf("Out of memory\n");
    else
    {
        strcpy(name_node->data, name);
        name_node->My_Node.ln_Name = name_node->data;
        name_node->My_Node.ln_Type = NAME_NODE_ID;
        name_node->My_Node.ln_Pri = 0;
        AddHead( (struct List *)list, (struct Node *)name_node );
    }
}

/*
 * Free entire list, including the header. The header is not updated
 * as the list is freed. This function demonstrates how to avoid
 * referencing freed memory when deallocating nodes. */
void FreeMyList(struct List *list)
{
    struct MyNodes *work_node;
    struct MyNodes *next_node;

    work_node = (struct MyNodes *) (list->lh_Head); /* First node */
    while (next_node = (struct MyNodes *) (work_node->My_Node.ln_Succ))
    {
        FreeMem(work_node, sizeof(struct MyNodes));
        work_node = next_node;
    }
    FreeMem(list, sizeof(struct List)); /* Free list header */
}

/*
 * Print the names of each node in a list. */
void DisplayList(struct List *list)
{
    struct Node *node;

    if (list->lh_TailPred == (struct Node *)list)
        printf("List is empty.\n");
```

```

    else
    {
        for (node = list->lh_Head ; node->ln_Succ ; node = node->ln_Succ)
            printf("%lx -> %s\n",node,node->ln_Name);
    }
}

/*
 * Print the location of all nodes with a specified name. */
void DisplayName(struct List *list, UBYTE *name)
{
    struct Node *node;

    if (node = FindName(list,name))
        while (node)
        {
            printf("Found %s at location %lx\n",node->ln_Name,node);
            node = FindName((struct List *)node,name);
        }
    else printf("No node with name %s found.\n",name);
}

void main()
{
    struct List *MyList; /* Note that a MinList would also work */

    if (!(MyList = AllocMem(sizeof(struct List), MEMF_CLEAR)) )
        printf("Out of memory\n");
    else
    {
        NewList(MyList); /* Important: prepare header for use */

        AddName(MyList, "Name7"); AddName(MyList, "Name6");
        AddName(MyList, "Name5"); AddName(MyList, "Name4");
        AddName(MyList, "Name2"); AddName(MyList, "Name0");

        AddName(MyList, "Name7"); AddName(MyList, "Name5");
        AddName(MyList, "Name3"); AddName(MyList, "Name1");

        DisplayName(MyList, "Name5");
        DisplayList(MyList);

        FreeMyList(MyList);
    }
}

```

## Important Note - Shared Lists

It is possible to collide with other tasks when manipulating a list that is shared by more than one task. *None* of the standard Exec list functions arbitrate for access to the list. For example, if some other task happens to be modifying a list while your task scans it, an inconsistent view of the list may be formed. This can result in a corrupted system.

Generally it is not permissible to read or write a shared list without first locking out access from other tasks. *All* users of a list must use the same arbitration method. Several arbitration techniques are used on the Amiga. Some lists may only be accessed only during **Forbid()** or **Disable()** (see the "Tasks" chapter for more information). Other lists are protected by a semaphore. The **ObtainSemaphore()** call grants ownership of the list (see the "Semaphores" chapter for more information). Some special lists require special arbitration. For example, you must use the Intuition **LockIBase(0)** call before accessing any Intuition lists.

Failure to lock a shared list before use *will* result in unreliable operation.

# Chapter 17

## Exec: Tasks

One of the most powerful features of the Amiga operating system is its ability to run and manage multiple independent program tasks, providing each task with processor time based on their priority and activity. These tasks include system device drivers, background utilities, and user interface environments, as well as normal application programs. This multitasking capability is provided by the Exec library's management of task creation, termination, scheduling, event signals, traps, exceptions, and mutual exclusion.

### Introduction

The Amiga Exec library provides a real-time message-based multitasking operating environment.

#### *Real-time*

means that the ROM routines can respond to events as they happen. It also means that the system optimizes routines for fast response.

#### *Multitasking*

means that many tasks can be operating simultaneously in the Amiga, with no task forced to be aware that another is present.

### *Message-based*

means that the entire system has been designed to operate on the basis of messages passed back and forth between tasks.

The system manages all Amiga programs as *tasks* that run along with other tasks on the system. As the system first powers up, there are several tasks operating simply waiting for activity or commands. Devices such as the keyboard (keyboard.device), the mouse (gameport.device), the timer (timer.device), and the disks (trackdisk.device) are each managed by tasks whose job is to sit and wait for instructions from other system and application tasks. While one task, such as trackdisk, may be moving the disk head to read a specific sector for one application, another task can perform part of its functions, using the time that might have been wasted while the head was being moved. One task may be reading the trackdisk, while others may be drawing screen graphics, printing documents, playing music, or telecommunicating.

## Tasks on the Amiga

Exec manages the sharing of the Amiga's 68000 family processor among all tasks running in the system, providing each with its own stack, its own exception handling, and full access to all processor registers. System tasks and user-started applications (which are generally higher level tasks known as *processes*) are all multitasked and managed by Exec in the same manner. *Processes* are extended tasks created by dos.library which are able to perform DOS interactions such as file IO, stdio, and opening of disk resident libraries, devices, and fonts. User programs started by Workbench or CLI are *processes*.

### SCHEDULING

Exec accomplishes multitasking by *multiplexing* the 68000 processor among a number of task contexts. Every task has an assigned priority, and tasks are scheduled to use the processor on a priority basis. The highest-priority *ready* task receives processor time until one of the following events occurs:

- a higher-priority task becomes active.
- the running task needs to wait for an external event.
- the running task exceeds a preset time period (a quantum) and there is another equal-priority task ready to run.

Task scheduling is normally preemptive in nature. The running task may lose the processor at nearly any moment by being displaced by another more urgent task. Later, when the preempted task regains the processor, it continues from where it left off.

It is also possible to run a task in a non-preemptive manner. This mode of execution is generally reserved for system data structure access. It is discussed in the "Exclusion" section toward the end of this chapter.

In addition to the prioritized scheduling of tasks, *time-slicing* also occurs for tasks with the same priority. In this scheme a task is allowed to execute for a quantum (a preset time period). If the task exceeds this period, the system will preempt it and give other tasks of the same priority a chance to run. This will result in a time-sequenced *round robin* scheduling of all equal-priority tasks.



## WAITING

Because of the prioritized nature of task scheduling, tasks must avoid performing the *busy wait* technique of polling. In this technique, a piece of code loops endlessly waiting for a change in state of some external condition. Tasks that use the busy wait technique waste all of the spare power of the processor. In most cases this prevents lower-priority tasks from receiving any processor time, and can waste as much as half of the processor time that would have been available for equal-priority tasks. Certain devices, such as the keyboard and the disk, depend on their associated tasks. Hence, using a busy wait at a high priority may defer important system services. Busy waiting can even cause system deadlocks. As an alternative to polling, Exec provides a number of functions which allow tasks to wait for external events without using processor time. The most basic and most flexible of these is the `Wait()` function which allows a task to wait for activity or input from one or more sources.

When there are no ready tasks, the processor is halted and only interrupts will be serviced. Because task multiplexing often occurs as a result of events triggered by system interrupts, this is not a problem. Halting the processor often helps improve the performance of other system bus devices.

## TASK STATES

For every task, Exec maintains state information to indicate its status. A normally operating task will exist in one of three states:

<i>running</i>	A task that is running is one that currently owns the processor. This usually means that the task is actually executing, but it is also possible that it has been temporarily displaced by a system interrupt.
<i>ready</i>	A task that is ready is one that is not currently executing but that is scheduled for the processor. The task will receive processor time based on its priority relative to the priorities of other running and ready tasks.
<i>waiting</i>	A task that is waiting is in a paused state waiting for an external event to occur. Such a task is not scheduled to use the processor. The task will be made ready only when one of its external events occurs (see the "Signals" section below).

A task may also exist in a few transient states:

<i>added</i>	A task in the added state has just been added to Exec and has not yet been scheduled for processing.
<i>removed</i>	A task in the removed state is being removed. Tasks in this state are effectively terminated and are usually undergoing clean-up operations.
<i>exception</i>	A task in the exception state is scheduled for special exception processing.

## TASK QUEUES

Tasks that are not in the running state are linked into one of two system queues. Tasks that are marked as ready to run but are awaiting an opportunity to do so are kept in the *ready queue*. This queue is always kept in a priority sorted order with the highest priority task at the head of the queue. A *waiting queue* accounts for tasks that are awaiting external events. Unlike the ready queue, the waiting queue is not kept sorted by priority. New entries are appended to the tail of the queue. A task will remain in the waiting queue until it is awakened by an event (at which time it is placed into the ready queue).

## PRIORITY

A task's priority indicates its importance relative to other tasks. Higher-priority tasks receive the processor before lower-priority tasks do. Task priority is stored as a signed number ranging from -128 to +127. Higher priorities are represented by more positive values; zero is considered the neutral priority. Normally, system tasks execute somewhere in the range of +20 to -20, and most application tasks execute at priority 0.

It is not wise to needlessly raise a task's priority. Sometimes it may be necessary to carefully select a priority so that the task can properly interact with various system tasks. The `SetTaskPri()` Exec function is provided for this purpose.

## STRUCTURE

Exec maintains task context and state information in a task-control data structure. Like most Exec structures, Task structures are dynamically linked onto various task queues through the use of an embedded Exec list **Node** structure (see the Lists chapter). Any task can find its own task structure by calling `FindTask(NULL)`. The C-language form of this structure is defined in the *exec/tasks.h* include file as follows:

```
struct Task
{
    struct Node tc_Node;
    UBYTE      tc_Flags;
    UBYTE      tc_State;
    BYTE       tc_IDNestCnt; /* intr disabled nesting */
    BYTE       tc_TDNestCnt; /* task disabled nesting */
    ULONG      tc_SigAlloc; /* sigs allocated */
    ULONG      tc_SigWait; /* sigs we are waiting for */
    ULONG      tc_SigRecvd; /* sigs we have received */
    ULONG      tc_SigExcept; /* sigs we will take excepts for */
    UWORD      tc_TrapAlloc; /* traps allocated */
    UWORD      tc_TrapAble; /* traps enabled */
    APTR       tc_ExceptData; /* points to except data */
    APTR       tc_ExceptCode; /* points to except code */
    APTR       tc_TrapData; /* points to trap code */
    APTR       tc_TrapCode; /* points to trap data */
    APTR       tc_SPReg; /* stack pointer */
    APTR       tc_SPLower; /* stack lower bound */
    APTR       tc_SPUpper; /* stack upper bound + 2 */
    VOID       (*tc_Switch)(); /* task losing CPU */
    VOID       (*tc_Launch)(); /* task getting CPU */
    struct List tc_MemEntry; /* allocated memory */
    APTR       tc_UserData; /* per task data */
};
```

A similar assembly code structure is available in the *exec/tasks.i* include file.

Most of these fields are not relevant for simple tasks; they are used by Exec for state and administrative purposes. A few fields, however, are provided for the advanced programs that support higher level environments (as in the case of *processes*) or require precise control (as in *devices*). The following sections explain these fields in more detail.

## Creation

To create a new task you must allocate a task structure, initialize its various fields, and then link it into Exec with a call to `AddTask()`. The task structure may be allocated by calling the `AllocMem()` function with the `MEMF_CLEAR` and `MEMF_PUBLIC` allocation attributes. These attributes indicate that the data structure is to be pre-initialized to zero and that the structure is shared.

The `Task` fields that require initialization depend on how you intend to use the task. For the simplest of tasks, only a few fields must be initialized:

**tc\_Node** The task list node structure. This includes the task's priority, its type, and its name (refer to the "Lists and Queues" chapter).

**tc\_SPLower**  
The lower memory bound of the task's stack.

**tc\_SPUpper**  
The upper memory bound of the task's stack.

**tc\_SPReg**  
The initial stack pointer. Because task stacks grow *downward* in memory, this field is usually set to the same value as `tc_SPUpper`.

Zeroing all other unused fields will cause Exec to supply the appropriate system default values. Allocating the structure with the `MEMF_CLEAR` attribute is an easy way to be sure that this happens.

Once the structure has been initialized, it must be linked to Exec. This is done with a call to `AddTask()` in which the following parameters are specified:

**task** A pointer to an initialized task structure.

**initialPC** The entry point of your task code. This is the address of the first instruction the new task will execute.

**finalPC** The finalization code for your task. This is a code fragment that will receive control if the `initialPC` routine ever performs a return (RTS). This exists to prevent your task from being launched into random memory upon an accidental return. The `finalPC` routine should usually perform various program-related clean-up duties and should then remove the task. If a zero is supplied as this parameter, Exec will use its default finalization code (which simply calls the `RemTask()` function).

## CREATETASK

A simpler method of creating a task is provided by the amiga.lib Exec support function `CreateTask()`, which can be accessed if your code is linked with the amiga.lib.

**CreateTask(name,priority,initialPC,stacksize)**

A task created with `CreateTask()` may be removed with the amiga.lib `DeleteTask()` function, or it may simply return when it is finished. `CreateTask()` adds a `MemList` to the `tc_MemEntry` of the task it creates, describing all memory it has allocated for the task, including the task stack and the Task structure itself. This memory will be deallocated by Exec when the task is either explicitly removed (`RemTask()` or `DeleteTask()`) or when it exits to Exec's default task removal code (`RemTask()`).

If your development language is not linkable with amiga.lib, it may provide an equivalent built-in function, or you can create your own based on the `CreateTask()` source code in the *Includes and Autodocs* manual.

Depending on the priority of a new task and the priorities of other tasks in the system, the newly added task may immediately begin execution.

Here is an example of simple manual task creation. In this example there is no coordination or communication between the main process and the simple task it has created. A more complex example might use named ports and messages to coordinate the activities and shutdown of two tasks. Because our task is very simple and never calls any system functions which could cause it to be signalled or awakened, we can safely remove the task at any time.

### NOTE

Because we are not using `CreateTask()`, the main process must clean up the memory it allocated for the task after the task is removed.

```
/* SimpleTask.c 09/89
 * Compiled with Lattice 5.02: LC -b1 -cfist -v -y
 * Linkage: c.o,simpletask.o library LC.lib,amiga.lib
 */
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/tasks.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

#define STACK_SIZE 1000L

/* Task name, pointers for allocated task struct and stack */
APTR stack = NULL;
struct Task *tc = NULL;
char *simpletaskname = "MySimpleTask";

ULONG sharedvar;

void simpletask(void);
void cleanup(void);
void cleanexit(UBYTE *,LONG);

void main(argc,argv)
int argc;
```

```

char **argv;
{
    if((stack = AllocMem(STACK_SIZE, MEMF_CLEAR)) == NULL)
        cleanexit("Not enough memory for task stack\n",RETURN_FAIL);

    if ((tc = (struct Task *)
        AllocMem(sizeof(struct Task),MEMF_CLEAR | MEMF_PUBLIC)) == NULL)
        cleanexit("Not enough memory for task structure\n",RETURN_FAIL);

    /* Initialize necessary fields, others were cleared by MEMF_CLEAR */
    tc->tc_Node.ln_Type = NT_TASK;
    tc->tc_Node.ln_Name = simpletaskname;
    tc->tc_SPLower = (APTR)stack;
    tc->tc_SPUpper = (APTR)(STACK_SIZE + (ULONG)stack);
    tc->tc_SPReg = tc->tc_SPUpper;

    sharedvar = 0L;
    AddTask(tc, simpletask, 0L);

    printf("This program initialized a variable to zero, then started a\n");
    printf("separate task which is incrementing that variable right now,\n");
    printf("while this program waits for you to press RETURN.\n");
    printf("Press RETURN now: ");
    getchar();

    printf("The shared variable now equals %ld\n",sharedvar);
    /* We can simply remove the task we added because our simpletask
     * does not make any system calls which could cause it to be awakened
     * or signalled later.
     */
    RemTask(tc);

    cleanup();
    exit(RETURN_OK);
}

void simpletask()
{
    while(sharedvar < 0x8000000) sharedvar++;
    /* Wait forever because main() is going to RemTask() us */
    Wait(0L);
}

void cleanexit(s,e)
UBYTE *s;
LONG e;
{
    if(*s) printf(s);
    cleanup();
    exit(e);
}

void cleanup()
{
    if(tc) FreeMem(tc,sizeof(struct Task));
    if(stack) FreeMem(stack,STACK_SIZE);
}

```

## STACK

Every task requires a stack. All task stacks are *user mode* stacks (in the language of the 68000) and are addressed through the A7 CPU register. All normal code execution occurs on this task stack. Special modes of execution (processor traps and system interrupts for example) execute on a single *supervisor mode* stack and do not directly affect task stacks.

Task stacks are normally used to store local variables, subroutine return addresses, and saved register values. Additionally, when a task loses the processor, all of its current registers are preserved on this stack (with the exception of the stack pointer itself, which must be saved in the task structure).

The amount of stack used by a task can vary widely. The theoretical minimum stack size is 72 bytes, which is the number required to save 17 CPU registers and a single return address. Of course, a stack of this size would not give you adequate space to perform any subroutine calls (because the return address occupies stack space). On the other hand, a stack size of 1K would suffice to call most system functions but would not allow much in the way of local variable storage. Processes that call dos.library functions need an additional 1500 bytes of stack.

Because stack-bounds checking is not provided as a service of Exec, it is important to provide enough space for your task stack. Stack overflows are always difficult to debug and may result not only in the erratic failure of your task but also in the mysterious malfunction of other Amiga subsystems. Some compilers provide a stack-checking option.

#### NOTE

Such stack-checking options generally can not be used if part of your code will be running on the system stack (interrupts, exceptions, handlers, servers), or on a different task's stack (libraries, devices, created tasks).

When choosing your stack size, do not cut it too close. Remember that any recursive routines in your code may use varying amounts of stack, and that future versions of system routines may use additional stack variables. By dynamically allocating buffers and arrays, most application programs can be designed to function comfortably within the default process stack size of 4000 bytes.

## Termination

Task termination may occur as the result of a number of situations:

1. A program returning from its **initialPC** routine and dropping into its **finalPC** routine or the system default finalizer.
2. A task trap that is too serious for a recovery action. This includes traps like processor bus error, odd address access errors, etc.
3. A trap that is not handled by the task. For example, the task might be terminated if your code happened to encounter a processor **TRAP** instruction and you did not provide a trap handling routine.
4. An explicit call to the Exec **RemTask()** function.

Task termination involves the deallocation of system resources and the removal of the task structure from Exec. The most important part of task termination is the deallocation of system resources. A task must return all memory that it allocated for its private use, it must terminate any outstanding I/O commands, and it must close access to any system libraries or devices that it has opened.

It is wise to adopt a strategy for task clean-up responsibility. You should decide whether resource allocation and deallocation is the duty of the creator task or the newly created task. Sometimes it is easier and safer for the creator to handle the necessary resource allocation and deallocation on behalf of its offspring. In such cases it is important to make sure that a child task is in a safe state before it is removed and is not using the allocated resources or waiting

for any condition or signal that might still occur.

#### NOTE

Certain resources, such as signals and created ports, must be allocated and deallocated by the same task that will wait on them. Also note that if your subtask code is part of your loaded program, you must not allow your program to exit before its subtasks have cleaned up their allocations, and have been either deleted or placed in a safe state such as `Wait(0L)`.

## Signals

Tasks often need to coordinate with other concurrent system activities (other tasks and interrupts). Such coordination is achieved through the synchronized exchange of specific event indicators called *signals*. This is the primary mechanism responsible for all intertask communication and synchronization on the Amiga.

The signal mechanism operates at a low level and is designed for high performance. Signals often remain hidden from the user program. The message system, for instance, may use signals to indicate the arrival of a new message. The message system is described in more detail in the “Messages and Ports” chapter.

The signal system is designed to support independent simultaneous events. Signals may be thought of as occurring in parallel. Each task has up to 32 independent signals. These signals are stored as single bits in the task control structure. One or more signals can occur at the same time.

All of these signals are considered *task relative*: a task may assign its own significance to a particular signal. Signals are not broadcast to all tasks; they are directed only to individual tasks. A signal has meaning to the task that defined it and to those tasks that have been informed of its meaning. For example, signal bit 12 may indicate a timeout event to one task, but to another task it may indicate a message arrival event.

## ALLOCATION

As mentioned above, a task assigns its own meaning to a particular signal. Because certain system libraries may occasionally require the use of a signal, there is a convention for signal allocation. It is unwise ever to make assumptions about which signals are actually in use.

Before a signal can be used, it must be allocated with the `AllocSignal()` function. This marks the signal as being in use and prevents the accidental use of the same signal for more than one event. You may ask for either a specific signal number or the next free signal. The state of the newly allocated signal is cleared (ready for use). Generally it is best to let the system assign you the next free signal. Of the 32 available signals, the lower 16 are usually reserved for system use. This leaves the upper 16 signals free for the user. Other subsystems that you may call depend on `AllocSignal()`.

The following C example asks for the next free signal to be allocated for its use:

```
if (-1 == (signal = AllocSignal(-1)))
{
    printf("no signal bits available\n");
}
else
{
    printf("allocated signal number %ld\n", signal);
}
```

## NOTE

The value returned by **AllocSignal()** is a signal bit number. This value cannot be used directly in calls to signal-related functions without first being converted to a mask:

```
mask = 1L << signal;
```

When a signal is no longer needed, it should be freed for reuse with **FreeSignal(signal)**.

It is important to realize that signal bit allocation is relevant only to the running task. You cannot allocate a signal from another task.

## WAITING FOR A SIGNAL

Signals are most often used to wake up a task upon the occurrence of some external event. This happens when a task is in its wait state and another task (or a system interrupt) causes a signal. The **Wait()** function specifies the set of signals that will wake up the task and then puts the task to sleep (into the waiting state). Any one signal or any combination of signals from this set are sufficient to awake the task. **Wait()** returns a mask indicating which signals from this set satisfied the wait. The **Wait()** function implicitly clears those signals that satisfied the wait. This effectively resets those signals for reuse.

Because tasks (and interrupts) normally execute asynchronously, it is often possible to receive a particular signal before a task actually waits for it. In such cases the **Wait()** will be immediately satisfied, and the task will not be put to sleep.

A task may wait for a combination of signal bits and be awakened when any of the signals occur. When the task returns from the wait, a signal mask is returned specifying which signal or signals were received. Usually the program must check the returned mask for each signal it was waiting on, and take the appropriate action for each that occurred. The order in which these bits are checked is often important. Here is a hypothetical example of a process that is using the console and timer devices, and is waiting for a message from either device:

```
consoleSig = 1L << myConsolePort->mp_SigBit;
timerSig   = 1L << myTimerPort->mp_SigBit;

signals = Wait(consoleSig | timerSig);

if (signals & consoleSig)
{
    printf ("new character\n");
    timeout = 10;
}
if (signals & timerSig)
{
    timeout--;
    if(timeout==0L) printf ("timeout\n");
}
```

This will put the task to sleep, waiting for a new character, or the expiration of a time period. Notice that this code checks for an incoming character signal before checking for a timeout. Although a program can check for the occurrence of a particular event by checking whether its signal has occurred, this may lead to busy wait polling. Such polling is wasteful of the processor and is usually detrimental to the proper function of the system. However, if a program needs to do constant processing and also check signals (a compiler for example) **SetSignal(0,0)** may be used to get a copy of your task's current signals.



## GENERATING A SIGNAL

Signals may be generated from both tasks and system interrupts with the `Signal()` function. For example `Signal(tc,mask)` would signal the task with the mask signals. More than one signal can be specified in the mask.

## Exclusion

From time to time the advanced system program may find it necessary to access global system data structures. Because these structures are shared by the system and by other tasks that execute asynchronously to your task, it is wise for you to exclude simultaneous access to these structures. This can be accomplished by *forbidding* or *disabling*, or with the use of *semaphores*. A section of code that requires the use of any of these mechanisms to lock out access by others is termed a *critical section*. Use of these functions is discouraged. For arbitrating between your tasks, semaphores are a superior solution. (See the Exec "Semaphores" chapter)

## FORBIDDING

Forbidding is used when a task is accessing shared structures that might also be accessed at the same time from another task. It effectively eliminates the possibility of simultaneous access by imposing *nonpreemptive* task scheduling. This has the net effect of disabling multitasking for as long as your task remains in its running state. While forbidden, your task will continue running until it performs a call to `Wait()` or exits from the forbidden state. Interrupts will occur normally, but no new tasks will be dispatched, *regardless of their priorities*.

When a task running in the forbidden state calls the `Wait()` function, directly or indirectly, it implies a temporary exit from its forbidden state. Since almost all stdio, device IO, and file IO functions must `Wait()` for IO completion, performing such calls will cause your task to `Wait()`, temporarily breaking the forbid. While the task is waiting, the system will perform normally. When the task receives one of the signals it is waiting for, it will again reenter the forbidden state. To become forbidden, a task calls the `Forbid()` function. To escape, the `Permit()` function is used. The use of these functions may be nested with the expected affects; you will not exit the forbidden mode until you call the outermost `Permit()`.

As an example, Exec library and device lists should be accessed only when forbidden. To access these lists without forbidding jeopardizes the integrity of the entire system. Therefore, all printing of information about the memory list must be performed after the scan of the list is completed.

```
/* LibList.c 09/89
 * Compiled with Lattice 5.02: LC -bl -cfist -v -y
 * Linkage: c.o,liblist.o library LC.lib,amiga.lib
 */
#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/libraries.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

#define ARRAYSIZE 64L
extern struct ExecBase *SysBase;
```

```

void main(argc,argv)
int  argc;
char **argv;
{
    struct Library *lib;
    ULONG count = 0L, k;
    char *names[ARRAYSIZE];

    Forbid();
    /* Note - printing within Forbid() would break the forbidden state */
    for ( lib = (struct Library *)SysBase->LibList.lh_Head;
        NULL != lib->lib_Node.ln_Succ;
        lib = (struct Library *)lib->lib_Node.ln_Succ)
    {
        if (count < ARRAYSIZE) names[count++] = lib->lib_Node.ln_Name;
    }
    Permit();

    printf("Libraries currently in system:\n");
    for (k=0; k<count; k++) printf(" %s\n",names[k]);
    if (count == ARRAYSIZE) printf("Error: array overflow\n");
}

```

As this program traverses down the library list, it remains forbidden to prevent the list from changing as it is being accessed. There is still a possibility that a library could be expunged before we print its name, invalidating our pointer to its name string. Copying the name strings, rather than their pointers, would prevent this.

## DISABLING

Disabling is similar to forbidding, but it also prevents interrupts from occurring during a critical section. Disabling is required when a task accesses structures that are shared by interrupt code. It eliminates the possibility of an interrupt accessing shared structures by preventing interrupts from occurring. Use of disabling is strongly discouraged.

To disable interrupts you can call the **Disable()** function. To enable interrupts again, use the **Enable()** function. Although assembler **DISABLE** and **ENABLE** macros are provided, assembler programmers should use the system functions rather than the macros for upwards compatibility, ease of debugging, and smaller code size.

Like forbidden sections, disabled sections can be nested. To restore normal interrupt processing, an **Enable()** call must be made for every **Disable()**. Also like forbidden sections, any direct or indirect call to the **Wait()** function will enable interrupts until the task regains the processor.

**WARNING:** It is important to realize that there is a danger in using disabled sections. Because the software on the Amiga depends heavily on its interrupts occurring in nearly real time, you cannot disable for more than a very brief instant. Disabling interrupts for more than 250 microseconds can impede the normal operation of vital system functions, especially serial IO.

**WARNING:** Masking interrupts by changing the 68000 processor interrupt priority levels with the **MOVE SR** instruction can also be dangerous and is very strongly discouraged. The disable- and enable-related functions control interrupts through the 4703 custom chip and *not* through the 68000 priority level. In addition, the processor priority level can be altered only from supervisor mode (which means this process is much less efficient).

It is never necessary to both disable and forbid. Because disable prevents interrupts, it also prevents preemptive task scheduling. When disable is used within an interrupt, it will have the effect of locking out all higher level interrupts (lower level interrupts are automatically disabled by the CPU). Many Exec lists can only be accessed while disabled. Suppose you want to print the names of all system tasks. You would need to access both the **TaskReady** and **TaskWait** lists from within a single disabled section. In addition, you must avoid calling system functions that would break a disable by waiting (**printf()** for example). In this example, the names are gathered into a name array

while interrupts are disabled. Then interrupts are enabled and the names are printed.

```

/* TaskList.c 09/89
 * Compiled with Lattice 5.02: LC -b1 -cfist -v -y
 * Linkage: c.o,tasklist.o library LC.lib,amiga.lib
 */
#include "exec/types.h"
#include "exec/execbase.h"
#include "exec/tasks.h"
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

#define ARRAYSIZE 128L
extern struct ExecBase *SysBase;

void main(argc,argv)
int argc;
char **argv;
{
    struct Task *task;
    LONG count=0, i=0, readi=0;
    char *names[ARRAYSIZE];

    Disable();
    for ( task = (struct Task *)SysBase->TaskWait.lh_Head;
          (NULL != task->tc_Node.ln_Succ) && (count < ARRAYSIZE);
          task = (struct Task *)task->tc_Node.ln_Succ)
    {
        names[count++] = task->tc_Node.ln_Name;
    }
    readi=count;
    for ( task = (struct Task *)SysBase->TaskReady.lh_Head;
          (NULL != task->tc_Node.ln_Succ) && (count < ARRAYSIZE);
          task = (struct Task *)task->tc_Node.ln_Succ)
    {
        names[count++] = task->tc_Node.ln_Name;
    }
    Enable();

    if (count == ARRAYSIZE) names[count-1]="error: array overflow";

    printf("WAITING Tasks:\n");
    for (i = 0; i < count; i++)
    {
        if(i==readi) printf("READY tasks:\n");
        printf(" %s\n", names[i]);
    }
    printf("THIS Task:\n %s\n",SysBase->ThisTask->tc_Node.ln_Name);
}

```

Of course, the code in this example will have problems if a task is removed before its name is printed. If this were to happen, the name-string pointer would no longer be valid. To avoid such problems it is a good programming practice to copy the entire name string into a temporary buffer.

## SEMAPHORES

Semaphores can be used for the purposes of mutual exclusion. With this method of locking, all tasks agree on a locking convention before accessing shared data structures. Tasks that do not require access are not affected and will run normally, so this type of exclusion is considered preferable to forbidding and disabling. This form of exclusion is explained in more detail in the Exec "Semaphores" chapter.

# Exceptions

## NOTE

This is a topic for advanced programmers. Exceptions are difficult to use safely. A task exception can interrupt a task that is executing a critical section of code within a system function, or one that has locked some system resource such as a disk unit or the blitter (note that even simple text output uses the blitter.) This possibility makes it extremely dangerous to use most system functions within an exception unless you are certain that your interrupted task was performing only local non-critical operations.

Tasks can specify that certain asynchronous events cause *exceptions*, which are *task-private* interrupts that redirect a task's flow of control without affecting other tasks or interrupts in the system. The task essentially suspends what it is doing and enters a special routine to process its exceptional event.

Exceptions are driven by the task signal mechanism described earlier in this chapter. Instead of waiting for a signal to occur, you indicate that it is an exception signal with the `SetExcept()` function. When the signal occurs, the task will be "interrupted" from its normal execution and placed in a special exception handler.

The `tc_ExceptCode` and `tc_ExceptData` task fields are used to establish the exception handler. The field `tc_ExceptCode` points to the routine that will handle the initial processing of all exceptions. If this field is zero, Exec will ignore all exceptions. The `tc_ExceptData` field can be used to provide a pointer to related data structure.

On entry to the exception code, the system passes certain parameters in the processor registers. `D0` contains a signal mask indicating which exception has just occurred, and `A1` points to the related exception data (from `tc_ExceptData`). In addition, the previous task context is pushed onto the task's stack. This includes the previous `PC`, `SR`, `D0-D7`, and `A0-A6` registers. You can think of an exception as a subtask outside of your normal task. Because task exception code executes in *user* mode, however, the task stack must be large enough to supply the extra space consumed during an exception.

While processing a given exception, Exec prevents that exception from occurring recursively. At exit from your exception-processing code, you should return the same value in `D0` to re-enable that exception signal. When the task executes the `RTS` at the end of the handler, the system restores the previous contents of all of the task registers and resumes the task at the point where it was interrupted by the exception signal. When two or more exception codes occur simultaneously, the exception-processing code determines the order in which they are handled by the order in which the signal bits are examined.

## Traps

Task *traps* are synchronous exceptions to the normal flow of program control. They are always generated as a direct result of an operation performed by your program's code. Whether they are accidental or purposely generated, they will result in your program being forced into a special condition in which it must immediately handle the trap. Address error, privilege violation, zero divide, and trap instructions all result in task traps. They may be generated directly by the 68000 processor (Motorola calls them "exceptions") or simulated by software.

A task that incurs a trap has no choice but to respond immediately. The task must have a module of code to properly handle the trap. Your task may be aborted if a trap occurs and no means of handling it has been provided. Default trap handling code (`tc_TrapCode`) is provided by the operating system.

You may instead choose to do your own processing of traps. The `tc_TrapCode` field is the address of the handler that you have designed to process the trap. The `tc_TrapData` field is the address of the data area for use by the trap handler.

The 68000 traps of particular interest are:

Table 17-1: Traps (68000 Exception Vector Numbers)

2	Bus error	access of nonexistent memory
3	Address error	long/word access of odd address (68000)
4	Illegal instruction	illegal opcode (other than Axxx or Fxxx)
5	Zero divide	processor division by zero
6	CHK instruction	register bounds error trap by CHK
7	TRAPV instruction	overflow error trap by TRAPV
8	Privilege violation	user execution of supervisor opcode
9	Trace	status register TRACE bit trap
10	Line 1010 emulator	execution of opcode beginning with \$A
11	Line 1111 emulator	execution of opcode beginning with \$F
32-47	Trap instructions	TRAP N instruction where N = 0 to 15

The actual stack frames generated for these traps are processor-dependent. The 68010, 68020, and 68030 processors will generate a different type of stack frame than the 68000. If you plan on having your program handle its own traps, you should not make assumptions about the format of the supervisor stack frame. Check the flags in the `AttnFlags` field of the `ExecBase` structure for the type of processor in use and process the stack frame accordingly.

## HANDLERS

For compatibility with the 68000, Exec performs trap handling in supervisor mode. This means that all task switching is disabled during trap handling. At entry to the task's trap handler, the system stack contains a processor-dependent trap frame as defined in the 68000/10/20/30 manuals. A longword exception number is added to this frame. That is, when a handler gains control, the top of stack contains the exception number and the trap frame immediately follows.

To return from trap processing, remove the exception number from the stack (note that this is the supervisor stack, not the user stack) and then perform a return from exception (RTE).

Because trap processing takes place in supervisor mode, with task dispatching disabled, it is strongly urged that you keep trap processing as short as possible or switch back to user mode from within your trap handler. If a trap handler already exists when you add your own trap handler, it is smart to propagate any traps that you do not handle down to the previous handler. This can be done by saving the previous address from `tc_TrapCode` and having your handler pass control to that address if the trap which occurred is not one you wish to handle.

The following example installs a simple trap handler which intercepts processor divide-by-zero traps, and passes on all other traps to the previous default trap code. The example has two code modules which are linked together. The trap handler code is in assembler. The C module installs the handler, demonstrates its effectiveness, then restores the previous `tc_TrapCode`.

```

/* Trap_c.c 09/89 - C module of sample integer divide-by-zero trap
 * Compiled with Lattice 5.02: LC -bl -cfist -v -y
 * Linkage: c.o,trap_c.o,trap_a.o library LC.lib,amiga.lib
 */
#include <exec/types.h>
#include <exec/tasks.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

/* assembler trap code in trap_a.asm */
extern ULONG trapa();

APTR oldTrapCode;
ULONG countdiv0;

void main(argc,argv)
int  argc;
char **argv;
{
    struct Task *thistask;
    ULONG k,j;

    thistask = FindTask(NULL);

    /* Save our task's current trap code pointer */
    oldTrapCode = thistask->tc_TrapCode;

    /* Point task to our assembler trap handler code
     * Ours will just count divide-by-zero traps, and
     * pass other traps on to the normal TrapCode
     */
    thistask->tc_TrapCode = (APTR)trapa;

    countdiv0 = 0L;
    /* Let's divide by zero a few times */
    for(k=0; k<4; k++)
    {
        printf("dividing %ld by zero... ",k);
        j = k/0L;
        printf("did it\n");
    }
    printf("\nDivide by zero happened %ld times\n",countdiv0);

    /* Restore old trap code */
    thistask->tc_TrapCode = oldTrapCode;
}

*
* trap_a.asm - Example trap handling code (leaves D0 intact)
*
* Entered in supervisor mode with the following on the supervisor stack:
* 0(sp).l = trap#
* 4(sp) Processor dependent exception frame

INCLUDE "exec/types.i"
INCLUDE "libraries/dos.i"

XDEF _trapa

XREF _countdiv0
XREF _oldTrapCode

CODE

_trapa:                                ; our trap handler entry

```

```

        CMPI.L   #5,(SP)           ; is this a divide by zero ?
        BNE.S    notdiv0          ; no
        ADD.L    #1,_countdiv0    ; yes, increment our div0 count
endtrap:
        ADDQ     #4,SP            ; remove exception number from SSP
        RTE                      ; return from exception

notdiv0:
        TST.L    _oldTrapCode     ; is there another trap handler ?
        BEQ.S    endtrap          ; no, so we'll exit
        MOVE.L    _oldTrapCode,-(SP) ; yes, go on to old TrapCode
        RTS                      ; jumps to old TrapCode

        END

```

## TRAP INSTRUCTIONS

The **TRAP** instructions in the 68000 generate traps 32-47. Because many independent pieces of system code may desire to use these traps, the **AllocTrap()** and **FreeTrap()** functions are provided. These work in a fashion similar to that used by **AllocSignal()** and **FreeSignal()**, mentioned above.

Allocating traps is simply a bookkeeping job within a task. It does not affect how the system calls the trap handler; it helps coordinate who owns what traps. Exec does nothing to determine whether or not the task is prepared to handle this particular trap. It simply calls your code. It is up to your program to handle the trap.

To allocate any trap, you can use the following code:

```

if (-1 == (trap = AllocTrap(-1)))
{
    printf("all trap instructions are in use\n");
}

```

or you can select a specific trap using this code:

```

if (-1 == (trap = AllocTrap(3)))
{
    printf("trap #3 is in use\n");
}

```

To free a trap you use **FreeTrap()**.

# Chapter 18

## Exec: Messages and Ports

### Introduction

For intersystem communication, Exec provides a consistent, high-performance mechanism of messages and ports. This mechanism is used to pass message structures of arbitrary sizes from task to task, interrupt to task, or task to software interrupt. In addition, messages are often used to coordinate operations between cooperating tasks.

A *message* data structure has two parts: system linkage and message body. The system linkage is used by Exec to attach a given message to its destination. The message body contains the actual data of interest. The message body is any arbitrary data block less than 64K bytes in size.

Messages are always sent to a predetermined destination *port*. At a port, incoming messages are queued in a first-in-first-out (FIFO) order. There are no system restrictions on the number of ports or the number of messages that may be queued to a port (other than the amount of available system memory).

Messages are always queued by *reference*. For performance reasons message copying is not performed. In essence, a message between two tasks is a temporary license for the receiving task to use a portion of the memory space of the sending task—that portion being the message itself. This means that if task A sends a message to task B, the message is still part of the task A context. Task A, however, should not access the message until it has been *replied*—that is, until task B has sent the message back, using the `ReplyMsg()` function. This technique of message exchange imposes important restrictions on message access.



# Ports

Ports are rendezvous points at which messages are collected. A port may contain any number of outstanding messages from many different originators. When a message arrives at a port, the message is appended to the end of the list of messages for that port, and a prespecified arrival action is invoked. This action may do nothing, or it may cause a predefined task signal or software interrupt (see the “Interrupts” chapter).

Like many Exec structures, ports may be given a symbolic name. Such names are particularly useful for tasks that must rendezvous with dynamically created ports. They are also useful for debugging purposes.

## STRUCTURE

A message port consists of a **MsgPort** structure as defined in the *exec/ports.h* and *exec/ports.i* include files. The C structure for a port is as follows:

```
struct MsgPort
{
    struct Node    mp_Node;
    UBYTE         mp_Flags;
    UBYTE         mp_SigBit;
    struct Task   *mp_SigTask;
    struct List    mp_MsgList;
};
```

where

### **mp\_Node**

is a standard **Node** structure. This is useful for tasks that might want to rendezvous with a particular message port by name.

### **mp\_Flags**

are used to indicate message arrival actions. See the explanation below.

### **mp\_SigBit**

is the signal bit *number* when a port is used with the task signal arrival action.

### **mp\_SigTask**

is a pointer to the task to be signaled. If a software interrupt arrival action is specified, this is a pointer to the interrupt structure.

### **mp\_MsgList**

is the list header for all messages queued to this port. (See the “Lists and Queues” chapter).

The **mp\_Flags** field contains a subfield indicated by the **PF\_ACTION** mask. This sub-field specifies the message arrival action that occurs when a port receives a new message. The possibilities are as follows:

### **PA\_SIGNAL**

This subfield tells the program to signal the specified task on the arrival of a new message. Every time a message is put to the port another signal will occur regardless of how many messages have been queued to the port.

## PA\_SOFTINT

This subfield causes the specified software interrupt. Like PA\_SIGNAL, PA\_SOFTINT will cause the software interrupt to be posted every time a message is received.

## PA\_IGNORE

This subfield tells the program to perform no operation other than queuing the message. This action is often used to stop signaling or software interrupts without disturbing the contents of the mp\_SigTask field.

It is important to realize that a port's arrival action will occur for each new message queued, and that there is not a one-to-one correspondence between messages and signals. Task signals are only single-bit flags so there is no record of how many times a particular signal occurred. There may be many messages queued and only a single task signal; sometimes however there may be a signal, but no messages. All of this has certain implications when designing code that deals with these actions. Your code should not depend on receiving a signal for every message at your port. All of this is also true for software interrupts.

## CREATION

To create a new message port, you must allocate and initialize a MsgPort structure. If you want to make the port *public*, you will also need to call the **AddPort()** function. Port structure initialization involves setting up a Node structure, establishing the message arrival action with its parameters, and initializing the list header. The following example of port creation is equivalent to the **CreatePort()** function as supplied in *amiga.lib*:

```
struct MsgPort *CreatePort(name, pri)
UBYTE *name;
LONG pri;
{
    int sigBit;
    struct MsgPort *mp;

    if ((sigBit = AllocSignal(-1L)) == -1)
        return(NULL);

    mp = (struct MsgPort *)
        AllocMem((ULONG)sizeof(struct MsgPort), (ULONG)MEMF_CLEAR | MEMF_PUBLIC);

    if (!mp)
    {
        FreeSignal(sigBit);
        return(NULL);
    }

    mp->mp_Node.ln_Name = name;
    mp->mp_Node.ln_Pri = pri;
    mp->mp_Node.ln_Type = NT_MSGPORT;

    mp->mp_Flags = PA_SIGNAL;
    mp->mp_SigBit = sigBit;
    mp->mp_SigTask = (struct Task *)FindTask(0L); /* find THIS task */

    if (name)
        AddPort(mp);
    else
        NewList(&(mp->mp_MsgList)); /* init message list */

    return(mp);
}
```

## DELETION

Before a message port is deleted, all outstanding messages from other tasks must be returned. This is done by replying to each message until the message queue is empty. Of course, there is no need to reply to messages owned by the current task (the task performing the port deletion). Public ports attached to the system with **AddPort()** must be removed from the system with **RemPort()** before deallocation. The following example of port deletion is equivalent to the **DeletePort()** function as supplied in *amiga.lib*:

```
void DeletePort(mp)
struct MsgPort *mp;
{
    if ( mp->mp_Node.ln_Name ) /* if it was public... */
        RemPort(mp);

    /* Make it difficult to re-use the port */
    mp->mp_SigTask    = (struct Task *) -1;
    mp->mp_MsgList.lh_Head = (struct Node *) -1;

    FreeSignal( mp->mp_SigBit );

    FreeMem( mp, (ULONG)sizeof(struct MsgPort) );
}
```

## RENDEZVOUS

The **FindPort()** function provides a means of finding the address of a public port given its symbolic name. For example, **FindPort("Spyder")** will return either the address of the message port or a zero indicating that no such public port exists. Since no arbitration is done, the usage of **FindPort()** must be protected with **Forbid()/Permit()**. Names should be unique to prevent collisions among multiple applications. It is a good idea to use your application name as a prefix for your port name. **FindPort()** does not arbitrate for access to the port list. The owner of a port might remove it at any time. For these reasons a **ForBid()/Permit()** pair is required for the use of **FindPort()**. The port address can no longer be regarded as being valid after **Permit()**.

The following is an example of how to safely put a message to a specific port:

```
#include <exec/types.h>
#include <exec/ports.h>

BOOL MsgPort SafePutToPort(message, portname)
struct Message *message;
STRPTR        portname;
{
    struct MsgPort *port;

    Forbid();
    port = FindPort(portname);
    if (port)
        PutMsg(port,message);
    Permit();
    return((BOOL)port); /* If zero, the port has gone away */

    /* Once we've done a Permit(), the port might go away and leave us with
     * an invalid port address. So we return just a BOOL to indicate whether
     * the message has been sent or not.
     */
}
```

# Messages

As mentioned earlier, a message contains both system header information and the actual message content. The system header is of the **Message** form defined in *exec/ports.h* and *exec/ports.i*. In C this structure is as follows:

```
struct Message
{
    struct Node    mn_Node;
    struct MsgPort *mn_ReplyPort;
    UWORD          mn_Length;
};
```

where

**mn\_Node**  
is a standard **Node** structure used for port linkage.

**mn\_ReplyPort**  
is used to indicate a port to which this message will be returned when a reply is necessary.

**mn\_Length**  
indicates the total length of the message, including the **Message** structure itself.

This structure is always attached to the head of all messages. For example, if you want a message structure that contains the x and y coordinates of a point on the screen, you could define it as follows:

```
struct XYMessage
{
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
}
```

For this structure, the **mn\_Length** field should be set to `sizeof(struct XYMessage)`.

## PUTTING A MESSAGE

A message is delivered to a given destination port with the **PutMsg()** function. The message is queued to the port, and that port's arrival action is invoked. If the action specifies a task signal or a software interrupt, the originating task may temporarily lose the processor while the destination processes the message. If a reply to the message is required, the **mn\_ReplyPort** field must be set up prior to the call to **PutMsg()**.

Here is a simple program for putting a message to a public port. The complete program is printed at the end of the chapter.

```

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <libraries/dos.h>

VOID main(VOID);
BOOL SafePutToPort(struct Message *, STRPTR);

struct XYMessage
{
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
};

VOID main(VOID)
{
    struct MsgPort *xyp, *xyreplyp;
    struct XYMessage *xymsg, *msg;
    BOOL foundport;

    xymsg = (struct XYMessage *)AllocMem(sizeof(struct XYMessage), MEMF_PUBLIC);
    if (xymsg == 0)
    {
        printf("Not enough memory for message structure\n");
        exit(30);
    }

    /* The replyport we'll use to get response */

    xyreplyp = CreatePort("xyreplyport", 0);
    if (xyreplyp == 0)
    {
        printf("Couldn't create xyreplyport\n");
        DeletePort((struct MsgPort *)xyp);
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(31);
    }

    xymsg->xy_Msg.mn_Node.ln_Type = NT_MESSAGE;
    xymsg->xy_Msg.mn_Length = sizeof(struct XYMessage);
    xymsg->xy_Msg.mn_ReplyPort = xyreplyp;
    xymsg->xy_X = 10;
    xymsg->xy_Y = 20;

    foundport = SafePutToPort((struct Message *)xymsg, "xyport");
    if (foundport == 0)
    {
        /* couldn't find port */
        DeletePort((struct MsgPort *)xyreplyp);
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(32);
    }

    /* Now lets wait 'till someone responds... */
    .
    .
    .

    DeletePort((struct MsgPort *)xyp);
    FreeMem(xymsg, sizeof(struct XYMessage));
}

```

## WAITING FOR A MESSAGE

A task may go to sleep waiting for a message to arrive at one or more ports. This technique is widely used on the Amiga as a general form of event notification. For example, it is used extensively by tasks for I/O request completion.

The `mp_SigTask` field contains the address of the task to be signaled and `mp_SigBit` contains a preallocated signal number (as described in the "Tasks" chapter).

You can call the `WaitPort()` function to wait for a message to arrive at a port. This function will return the first message (it may not be the only) queued to a port. If the port is empty, your task will go to sleep waiting for the first message. If the port is not empty, your task will not go to sleep. It is possible to receive a signal for a port without a message being present yet. The code processing the messages should be able to handle this. The following code illustrates this.

```
struct XYMessage *xy_msg;
struct MsgPort *xyp;

xyp = CreatePort("xyport", 0);
if (xyp == 0)
{
    printf("Couldn't create xyport\n");
    exit(31);
}

xy_msg = WaitPort(xyp);    /* go to sleep until message arrives */
```

A more general form of waiting for a message involves the use of the `Wait()` function (see the "Tasks" chapter). This function waits for task event signals directly. If the signal assigned to the message port occurs, the task will awaken. Using the `Wait()` function is more general because you can wait for more than one signal. By combining the signal bits from each port into one mask for to the `Wait()` function, a loop can be set up to process all messages at all ports.

Here's an example using `Wait()`:

```
struct XYMessage *xy_msg;
struct MsgPort *xyp;

ULONG usersig, portsig;
BOOL ABORT = FALSE;

xyp = CreatePort("xyport", 0);
if (xyp == 0)
{
    printf("Couldn't create xyport\n");
    exit(31);
}

portsig = 1 << xyp->mp_SigBit;

usersig = SIGBREAKF_CTRL_F;    /* User can break with CTRL-F */
/* we could wait for more signals */

FOREVER
{
    signal = Wait(portsig | usersig);    /* sleep 'till someone signals */

    if (signal & portsig)
    {
        /* got a signal at the msgport */
        /* Someone send a message */
    }
    if (signal & usersig)
    {
        /* got a signal from the user */
        /* Time to clean up */
        ABORT = TRUE;
    }

    if (ABORT)
    {
        DeletePort((struct MsgPort *)xyp);
        exit(0);
    }
}
```

## NOTE

**WaitPort()** only returns a pointer to the first message in a port. It does not actually remove the message from the port queue.

## GETTING A MESSAGE

Messages are usually removed from ports with the **GetMsg()** function. This function removes the next message at the head of the port queue and returns a pointer to it. If there are no messages in a port, this function returns a zero.

The example below illustrates the use of **GetMsg()** to print the contents of all messages in a port:

```
while (msg = GetMsg(mp))
{
    printf("x=%ld y=%ld\n", msg->xy_X, msg->xy_Y);
}
```

Certain messages may be more important than others. Because ports impose FIFO ordering, these important messages may get queued behind other messages regardless of their priority. If it is necessary to recognize more important messages, it is easiest to create another port for these special messages.

## REPLYING

When the operations associated with receiving a new message are finished, it is usually necessary to send the message back to the originator. The receiver replies the message by returning it to the originator using the **ReplyMsg()** function. This is important because it notifies the originator that the message can be reused or deallocated. The **ReplyMsg()** function serves this purpose. It returns the message to the port specified in the **mn\_ReplyPort** field of the message. If this field is zero, no reply is returned.

The previous example can be enhanced to reply to each of its messages:

```
while (msg = GetMsg(mp))
{
    printf("x=%ld y=%ld\n", msg->xy_X, msg->xy_Y);
    ReplyMsg(msg);
}
```

Notice that the reply does not occur until *after* the message values have been used.

Often the operations associated with receiving a message involve returning *results* to the originator. Typically this is done within the message itself. The receiver places the results in fields defined (or perhaps reused) within the message body before replying the message back to the originator. Receipt of the replied message at the originator's reply port indicates it is once again safe for the originator to use or change the values found within the message.

Here is a complete example of waiting for and replying to messages:

```
/* port.c 10/89
 * Compiled with Lattice C 5.04: LC -bl -cfist -v -y
 * LINK c.o+port.o library lib:lc.lib+lib:amiga.lib
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <libraries/dos.h>
```

```

#ifdef LATTICE
#include <stdio.h>
#include <stdlib.h>
#include <proto/all.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

VOID main(VOID);

BOOL SafePutToPort(struct Message *, STRPTR);

struct XYMessage
{
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
};

VOID main(VOID)
{
    struct MsgPort *xymp, *xyreplymp;
    struct XYMessage *xymsg, *msg;
    ULONG portsig, usersig, signal;
    BOOL foundport, ABORT = FALSE;

    xymsg = (struct XYMessage *)AllocMem(sizeof(struct XYMessage), MEMF_PUBLIC);
    if (xymsg == 0)
    {
        printf("Not enough memory for message structure\n");
        exit(30);
    }

    /* In this example we set up the public XY port ourselves */

    xymp = CreatePort("xyport", 0);
    if (xymp == 0)
    {
        printf("Couldn't create xyport\n"); /* so much for the example */
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(30);
    }

    /* The replyport we'll use to get response */

    xyreplymp = CreatePort("xyreplyport", 0);
    if (xyreplymp == 0)
    {
        printf("Couldn't create xyreplyport\n");
        DeletePort((struct MsgPort *)xymp);
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(31);
    }

    xymsg->xy_Msg.mn_Node.ln_Type = NT_MESSAGE;
    xymsg->xy_Msg.mn_Length = sizeof(struct XYMessage);
    xymsg->xy_Msg.mn_ReplyPort = xyreplymp;
    xymsg->xy_X = 10;
    xymsg->xy_Y = 20;

    foundport = SafePutToPort((struct Message *)xymsg, "xyport");
    if (!foundport)
    {
        /* couldn't find port */
        DeletePort((struct MsgPort *)xyreplymp);
        DeletePort((struct MsgPort *)xymp);
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(32);
    }

    /* we just found the port we created ourselves and put a message to it! WOW*/
    /* let's pretend we're the other guy now */

    printf("Exit with CTRL-F\n"); /* User can abort with CTRL-F */
}

```



```

msg = (struct XYMessage *)WaitPort((struct MsgPort *)xymp);
/* doesn't wait since there HAS to be a message */

while (msg = (struct XYMessage *)GetMsg((struct MsgPort *)xymp))
{
    printf("x = %d y = %d\n", msg->xy_X, msg->xy_Y);
    msg->xy_X += 50;
    msg->xy_Y += 50;
    ReplyMsg((struct Message *)msg);    /* reply to the sender */
}

/* We put on the other hat again & wait for the reply to our message sent */
/* We'll use Wait() for this one */

portsig = 1 << xyreplymp->mp_SigBit;
usersig = SIGBREAKF_CTRL_F;    /* User can break with CTRL-F */

FOREVER
{
    signal = Wait(portsig | usersig);    /* sleep 'till someone signals */

    if (signal & portsig)
    {
        /* got a signal at the msgport */
        while (msg = (struct XYMessage *)GetMsg((struct MsgPort *)xyreplymp))
        {
            printf("X = %d Y = %d\n", msg->xy_X, msg->xy_Y);
            /* no need to reply. WE have sent the message */
        }
    }

    if (signal & usersig)
    {
        /* user abort */
        while (msg = (struct XYMessage *)GetMsg((struct MsgPort *)xyreplymp)); /* just in case */
        ABORT = TRUE;
    }

    if (ABORT)
    {
        /* this about sums it up. let's clean up */
        DeletePort((struct MsgPort *)xyreplymp);
        DeletePort((struct MsgPort *)xymp);    /* the 'other guy's' port */
        FreeMem(xymsg, sizeof(struct XYMessage));
        exit(0);
    }
}

}

BOOL SafePutToPort(message, portname)
struct Message *message;
STRPTR    portname;
{
    struct MsgPort *port;

    Forbid();
    port = FindPort(portname);
    if (port)
        PutMsg(port, message);
    Permit();
    return((BOOL)port); /* If zero, the port has gone away */

    /* Once we've done a Permit(), the port might go away and leave us with
    * an invalid port address. So we return just a BOOL to indicate whether
    * the message has been sent or not.
    */
}

```

# Chapter 19

## Exec: Input/Output

### Introduction

One of the primary purposes of Exec is to provide a standard form for all device input/output (I/O). This consists of the definition of a standard device interface, the format for I/O requests, and the establishment of rules for devices and tasks to interact. In addition, the guidelines for nonstandard device I/O are also defined. In the design of the Amiga I/O system, great care has been taken to avoid dictating the form of implementation or the internal operational characteristics of a device.

In its purest sense, a *device* is an abstraction that represents a set of well-defined interactions with some form of physical media. This abstraction is supported by a standard Exec data structure and an independent system code module. The data structure provides the external interface and maintains the current device state. The code module supplies the operations necessary to make the device functional. (In many operating systems, this code module is referred to as a device *driver*. See *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for the source assembly language code for a disk-resident device driver with its own task for handling I/O requests. To see how a device is used, refer to any of the examples located within the specific device chapters themselves.)

A device *unit* is an instance of a device. It shares the same device data structure and code module with all other units of the same device; however, it operates in an independent fashion. Often units correspond to separate physical subsystems of the same general device class. For example, each Amiga floppy disk drive is an independent unit of the same device. There is only one device data structure and one code module to support all these units.

Exec I/O is performed using the message system described in the Exec “Messages and Ports” chapter. Most aspects of message passing are concealed within the Exec I/O support routines. However, it is important to realize that I/O request blocks, once issued, must *not* be modified or reused until they are returned to the control of your program by Exec.

## Request Structure

An I/O *request* is always directed to a device unit. This request is organized as a control block and contains a *command* to be performed on a specified unit. It is passed through a standard device interface function, where it is processed and executed by the code module of the device. All request parameters are included in the request control block, and I/O request results are returned in the same control block.

Every device unit responds to a standard set of commands, and may optionally provide a nonstandard set of commands as well. The standard commands are explained later in this chapter. Nonstandard commands are discussed in the documentation pertaining to the particular device involved. The nonstandard commands vary from device to device. Similar devices may have similar nonstandard commands for similar operations, but this is by no means a hard and fast rule.

An I/O request always includes at least an **IORequest** data structure. This is a standard header used for all I/O requests. It is defined in the *exec/io.h* include files as follows:

```
struct IORequest
{
    struct Message *io_Message;
    struct Device *io_Device;
    struct Unit *io_Unit;
    UWORD io_Command;
    UBYTE io_Flags;
    BYTE io_Error;
};
```

where

- |                   |  |
|-------------------|--|
| <b>io_Message</b> | is a message header (see the “Messages and Ports” chapter). This header is used by the device to return I/O requests upon completion. It is also used by devices internally for I/O request queuing. This header must be properly initialized for I/O to work correctly. |
| <b>io_Device</b>  | is a pointer to the device data structure node. This field is automatically set up by an Exec function when the device is opened, and must not be modified directly by your program.   |
| <b>io_Unit</b>    | Unit pointer. This is a device <i>private</i> field and must not be accessed by your program. The format of this field is device dependent and is set up by the device during the open sequence.   |
| <b>io_Command</b> | is the command requested. This may be either one of the system standard commands or a device-specific command.   |
| <b>io_Flags</b>   | is used to indicate special request options and state. This field is divided into two subfields of four bits each. The lower four bits are for use by Exec and the upper four bits are available to the device for device dependent options.                             |
| <b>io_Error</b>   | is an error or warning value valid on request completion. A non zero value indicates an error.   |

Most devices use an expanded form of the standard **IORequest** structure:

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

where the additional fields are used as follows:

**io\_Actual**

indicates the actual number of bytes transferred. This field is valid only upon completion.

**io\_Length**

is the requested number of bytes to transfer. This field must be set up prior to the request. Some devices allow variable length data transfers. This is indicated by a special value in **io\_Length**. See the documentation on the specific device for additional information.

**io\_Data**

is a pointer to the transfer data buffer.

**io\_Offset**

indicates a byte offset (for structured devices). For block-structured devices (such as a floppy disk device) this number must be a multiple of the block size.

The **io\_Device** and **io\_Unit** fields are private to the device and should be left unchanged by your program between I/O calls. The **io\_Command** field of the I/O request and the ReplyPort (which the device will use to communicate with your program) are not changed by the servicing of the request, which permits repeated I/O using the same request.

The values contained in the **io\_Data**, **io\_Length**, and **io\_Offset** fields may be modified by the device. This is device dependent. Some devices may guarantee to leave additional fields in the I/O request unchanged. See the documentation on the specific device for additional information.

Devices with nonstandard commands may add their own special fields to the I/O request structure as needed. Such extensions are device specific.

## Interface Functions

Five Exec functions are responsible for interfacing I/O requests to actual device drivers. These functions operate independently of the particular device command requested. They deal with the request block as a whole, ignoring its command and its command parameters.

### **DoIO()**

is the most commonly used I/O function. It initiates an I/O request and waits for its completion. This is a *synchronous* form of device I/O; control is not returned to the caller until completion. If the I/O request never completes, this call never returns. **DoIO()** automatically handles all the details, including QUICKIO, waiting, and removing the reply message.

### **SendIO()**

is used to initiate an I/O request without waiting for completion. This is an *asynchronous* form of device I/O; control is returned even if the request has not yet completed.

### **WaitIO()**

is used to wait for the completion of a previously initiated asynchronous I/O request. This function will not return control until the request has completed (successfully or unsuccessfully). **WaitIO()** automatically removes the message from the message port, and clears the signal bit. Your program should not **WaitIO()** on an I/O request that has not actually been sent.

### **CheckIO()**

is used to see if an asynchronous I/O request has completed. Your program should not **CheckIO()** on an I/O request that has not actually been sent.

### **AbortIO()**

attempts to cancel a previous I/O request made to an Exec device. This function is accessed as an assembly code macro **ABORTIO** or through the C library Exec function **AbortIO()**. Either results in a call to the **ABORTIO** vector of the device with the I/O request to be aborted.

The **AbortIO()** command may fail; if it succeeds, the device will stop processing the **IORequest**, and reply to it earlier than it may have otherwise done. **AbortIO()** does not remove the **IORequest** from your replyport, and does not wait for it to complete. Your program must wait for the reply message before actually reusing the **IORequest**. If a request is already completed when **AbortIO()** is called, no action is taken. Your program must never attempt to **AbortIO()** a request that has not actually been sent, or fireworks may result.

The following is the usual way to use **AbortIO()**.

```
AbortIO(ior);  
WaitIO(ior);
```

In addition to the above Exec functions, there is an I/O related function that is actually a direct entry into the device driver itself. This function is part of the actual device driver interface to the system and should be used with care. It incurs slightly less overhead but requires more knowledge of the I/O system internals (you must know how QUICK I/O works, for instance):

### **BeginIO()**

initiates an IO request. The request will be synchronous or asynchronous depending on the command and on the device driver.

### **NOTE**

**SendIO()** and **BeginIO()** are actually equivalent, except that **SendIO()** clears the **io\_Flags** field of the I/O request. **BeginIO()** is defined in **/flamiga.lib/fp** as a direct call to the **BEGINIO** entry point of a device.

## Standard Device Commands

There are eight standard commands to which all devices are expected to respond. If the device is not capable of performing one of these commands, it will at least return an error indication that the command is not supported. The command is send in the `io_Command` field of the I/O Request These commands are defined in the `exec/io.h` include files.

### **CMD\_RESET**

This command resets the device unit. It completely initializes the device unit, returning it to its default configuration, aborting all of its pending I/O, cleaning up any internal data structures, and resetting any related hardware.

### **CMD\_READ**

This command reads a specified number of bytes from a device unit into the data buffer. The number of bytes to be read is specified in the `io_Length` field. The number of bytes *actually* read is returned in the `io_Actual` field.

### **CMD\_WRITE**

This command writes a specified number of bytes to a device unit from a data buffer. The number of bytes to be written is specified in the `io_Length` field. The number of bytes *actually* written is returned in the `io_Actual` field.

### **CMD\_UPDATE**

This command forces out all internal buffers, causing device internal memory buffers to be written out to the physical device unit. A device will transparently perform this operation when necessary, but this command allows you to request explicitly that such an action take place. It is useful for devices that maintain internal caches, such as the floppy disk device.

### **CMD\_CLEAR**

This command clears all internal read buffers. It deletes the entire contents of a device unit's internal read buffers. No update is performed; all data is lost.

### **CMD\_STOP**

This command stops the device unit immediately (at the first opportunity). All I/O requests continue to queue, but the device unit stops servicing them. This command is useful for devices that may require user intervention (printers, plotters, data networks, etc.).

### **CMD\_START**

This command causes the device unit to continue after a previous `CMD_STOP` command. The device resumes from where it was stopped.

### **CMD\_FLUSH**

This command aborts all I/O requests in both the read and write request queues of the device. All pending I/O requests are returned with an error message. `CMD_FLUSH` does not effect active requests.

### **CMD\_NONSTD**

Any nonstandard commands begin here. Non standard commands are designated as `CMD_NONSTD+0`, `CMD_NONSTD+1`, and so on.

## **CMD\_INVALID**

This is a command to which the device replies with an error `IOERR_NOCMD` as defined in *exec/errors.h* indicating the command is not supported.

## **Performing I/O**

Exec I/O is always performed using I/O request blocks. Before a device is sent an I/O request block, that block must be properly initialized by both the system and the user. Once this has been done, normal I/O may commence.

### **PREPARATION FOR EXEC I/O**

Devices are identified within the system by name (a null-terminated character string). Device units are usually identified by number. Before a device can be used, your program must make an `OpenDevice()` call with the I/O request to be used. The I/O request must already contain a pointer to the initialized message port that the device will use to communicate with your program. The `io_Message` field is set up in the same manner as a message. This is described in the "Messages and Ports" chapter. The `OpenDevice()` function maps the device name to an actual device and then calls the device to perform its initialization. The device will map the unit number into an internal form for later use. Both Exec and the device driver will initialize fields the I/O request passed to `OpenDevice()`.

For example, `OpenDevice("trackdisk.device",1,ior,0)` will attempt to open unit one of the floppy disk device, mapping its symbolic name ("trackdisk.device") into the address of a device data structure. It also sets up the private fields of the request, such as `io_Unit` for example. `OpenDevice()` will return a zero if it was successful and a nonzero error number if it was not.

### **SYNCHRONOUS REQUESTS**

Synchronous I/O requests are initiated with the `DoIO()` function mentioned earlier. `DoIO()` will not return control until the request has completed. Because the device may respond to a request immediately or queue it for later action, an undetermined amount of time may pass before control is returned. With this type of I/O, only one request is serviced at a time.

To perform synchronous I/O, the I/O request block must be prepared as described in the previous section. In addition, the `io_Command` (as well as other fields depending on the command and the device) must be initialized.

The `io_Command` field is set to the desired command. For example, to perform an update command use:

```
ior->io_Command = CMD_UPDATE;  
DoIO(ior);
```

More involved commands require other fields to be initialized. For example, the commands to read a sector from a disk might look something like the following:

```

ior->io_Command = CMD_READ;
ior->io_Length  = TD_SECTOR;
ior->io_Offset  = 20 * TD_SECTOR;
ior->io_Data    = buffer;
DoIO(ior);

```

When the request has completed, the request block is returned with the command results. If an error occurred, **DoIO()** will return the error number. The error number is also indicated in the **io\_Error** field of the request.

## ASYNCHRONOUS REQUESTS

More efficient programs can take advantage of the multitasking characteristics of the I/O system by using asynchronous I/O, which allows many requests to be performed at the same time. This type of I/O is supported by the **SendIO()**, **WaitIO()**, **CheckIO()**, and **AbortIO()** functions. Asynchronous I/O requests will return almost immediately to your program regardless of whether the request has actually completed. This lets the user maintain control while the I/O is being performed. Multiple I/O requests can be posted in this fashion.

In the disk read example above, asynchronous I/O could be performed by changing the **DoIO()** call to a **SendIO()**:

```

ior->io_Command = CMD_READ;
ior->io_Length  = TD_SECTOR;
ior->io_Offset  = 20 * TD_SECTOR;
ior->io_Data    = buffer;
SendIO(ior);

```

From the time the I/O has been initiated to the time it completes, the request block must never be directly accessed by your program. The device can be said to “own” the request block, as well as any data buffers pointed to by the request block. Only after the request has completed or successfully aborted should your program access the request block or the data buffers.

When the I/O completes, the device will return the I/O request block to the reply port specified in its **io\_Message** field. After this has happened, you know that the device has finished the I/O. The reply port used to receive the returned request can be set up to cause a task signal when the reply arrives. This technique lets a task sleep until the the request is complete. The **WaitIO()** function can be called to wait for the completion of a previously initiated request.

**WaitIO()** will handle all of the interaction with the message reply port automatically. If you are using just the **Wait()** function, do not forget to remove the I/O request from your reply port with **GetMsg()**. Once this is done, the request may be reused.

The **CheckIO()** function is handy to determine if a particular I/O request has been satisfied. This function deals with some of the subtleties of I/O in the proper manner.

If you wish to queue several I/O requests to a device, you must issue multiple **SendIO()** requests, each with its own separately-opened request structure. This type of I/O is supported by most devices. A task can also request I/O from a number of devices and then check later for their completion.

Exec also allows for certain types of optimization in device communication. One form of optimization, in which you call the device driver directly, is called quick I/O. This concept is discussed later in this chapter.



## COMPLETING THE USE OF A DEVICE

When a program has completed all its I/O, access to the device should be concluded with **CloseDevice()**. This function will inform the device that no further I/O is to be performed with this request. For every **OpenDevice()** there must be one corresponding **CloseDevice()**.

## QUICK I/O

For some types of I/O, the normal internal mechanisms of I/O may present a large amount of overhead compared to the amount of data transferred. This is often true for character-oriented I/O, in which each character might be transferred with a separate I/O request. The overhead for such requests could significantly overload the I/O system, resulting in an efficiency loss for the entire system.

To allow devices to optimize their I/O handling, a mechanism called quick I/O was created. In the **IORequest** data structure, one of the **io\_flags** is reserved for quick I/O. When set prior to an I/O request, this flag indicates that the device is allowed to handle the I/O in a special manner. This enables some devices to take certain "short-cuts" when it comes to performing and completing the request.

The quick I/O bit (**IOB\_QUICK**) allows the device to avoid returning the I/O request to the user via the message system (for example, via **ReplyMsg()**) if it can complete the request immediately. If the **IOB\_QUICK** bit is still set at the end of the **BeginIO()** call, the request has already completed and your program will not find the I/O request on the reply port (so should not look).

The **DoIO()** function normally requests the quick I/O option, whereas the **SendIO()** function does not. Complete control over the mode for quick I/O is possible by calling a device's **BeginIO()** entry directly.

It is up to the device to determine whether it can handle a request marked as quick I/O. If the quick I/O flag is still set when the request has completed, the I/O was performed quickly. This means that no message reply occurred, so the message was not queued to the reply port.

```
ior->io_Flags= IOF_QUICK;    /* request QUICK IO */
BeginIO(ior);

/* asynchronous program activity goes here */

if(ior->io_Flags & IOF_QUICK)
{
    /* the command finished QUICK, no need to wait */
}
else
{
    /* wait for I/O completion in the normal manner, */
    /* or possibly do more asynchronous program activity. */
    /* for the purposes of this code fragment, a simple */
    /* WaitIO() is used. In general, you will just add the */
    /* signal bit from this ior to the list of signals you */
    /* are already planning to Wait(). */
    WaitIO(ior);    /* WaitIO will remove the message from the */
    /* reply port if QUICK IO was not possible */
}
```

## Example of Device Use

The following simple example demonstrates the use of an Exec device (specifically, the trackdisk) and shows opening the device, sending device commands, and closing the device.

```
/* A complete example of using the trackdisk.device. Requires use of
 * ANSI function prototypes.
 * Compile with Lattice C 5.04: LC -L -cq
 */

#include <exec/types.h>
#include <devices/trackdisk.h>
#include <libraries/dos.h>

#ifdef LATTICE
#include <proto/exec.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
#endif

struct IOExtTD *trackIO; /* global pointer to trackdisk IORequest */
short          openererror; /* global flag */

void cleanExit(returncode)
int returncode;
{
    if(!openererror)CloseDevice(trackIO);

    if(trackIO)
    {
        /* extract Port address from I/O Request */
        DeletePort(trackIO->iotd_Req.io_Message.mn_ReplyPort);
        DeleteExtIO(trackIO);
    }
    exit(returncode);
}

void main()
{
    /* CreatePort() and pass the result to CreatreExtIO in one step */
    trackIO=(struct IOExtTD *)
        CreateExtIO( CreatePort(0,0),sizeof(struct IOExtTD) );
    if(!trackIO)cleanExit(RETURN_FAIL+1);

    if(openererror=OpenDevice("trackdisk.device",0L,trackIO,0L))
        cleanExit(RETURN_FAIL+2);

    trackIO->iotd_Req.io_Command=TD_SEEK; /* command */

    trackIO->iotd_Req.io_Offset =0L; /* out */
    DoIO(trackIO); printf("1\n");

    trackIO->iotd_Req.io_Offset =79*11*2*512L; /* in */
    DoIO(trackIO); printf("2\n");

    trackIO->iotd_Req.io_Offset =0L; /* out */
    DoIO(trackIO); printf("3\n");

    trackIO->iotd_Req.io_Offset =79*11*2*512L; /* in */
    DoIO(trackIO); printf("4\n");

    cleanExit(RETURN_OK);
}
```

## Standard Devices

The following standard system devices are normally available when the Amiga starts up (either in Kickstart or located on the Workbench disk).

### Audio

The audio device is provided to control the use of the audio channels.

### Clipboard

The clipboard device provides a means of “cutting” data from and “pasting” data into applications.

### Console

The console device receives its input from the input device. The input portion of the console device is simply a handler for input events filtered by Intuition. It provides what might be called the “traditional” video display terminal user interface.

### Gameport

Gameport handles raw information from the mouse or a joystick device. Gameport events are queued so that no movements will be missed. You can tell the system what type of device is connected and how often to check and report the current status of the device.

### Input

The input device combines requests from both the keyboard and the gameport device. Input events from both are merged into a single input event stream on a first-in-first-out basis.

### Keyboard

The keyboard device handles raw information from the keyboard and converts it into input events that can be retrieved and interpreted. Keyboard input events are queued so that no keystrokes will be missed.

### Narrator

The narrator device is loaded from disk and uses the audio device to produce humanlike synthesized speech.

### Parallel

The parallel device is loaded from disk and initialized on being loaded. It controls parallel communications, and is most often used by the printer device.

### Printer

The printer device driver is loaded from disk. It converts escape codes from the standard set of Amiga printer escape sequences to the code sequences understood by the individual printers.

### Serial

The serial device is loaded from disk and initialized on being loaded. It controls serial communications buffering of the input/output, baud rate, and so on.

### Timer

Provides a flexible way of causing task signals or interrupts at second and microsecond intervals.

### Trackdisk

Trackdisk provides direct access to the 3 1/2-inch and 5 1/4-inch floppy disk drives. Among the functions provided are format, seek, read, and write. Normally, trackdisk is used only by AmigaDOS.

# Chapter 20

## Exec: Semaphores

Semaphores are a feature of Exec which provide a general method for tasks to arbitrate for the use of memory or other system resources they may be sharing. This chapter describes the structure of Exec semaphores and the various support functions provided for their use. Since the semaphore system uses Exec lists and signals, some familiarity with these concepts is helpful for understanding semaphores.

### Introduction

In any multi-tasking or multi-processing system there is a need to share data among independently executing tasks. If the data is static (that is, it never changes), then there is no problem. However, if the data is variable, then there must be some way for a task that is about to make a change to keep other tasks from looking at the data.

For example, to add a node to a linked list of data, a task would normally just add the node. However, if the list is shared with other tasks, this could be dangerous. Another task could be walking down the list while the change is being made and pick up an incorrect pointer. The problem is worse if two tasks attempt to add an item to the list at the same time. Exec semaphores provide a way to prevent such problems.

A semaphore is much like getting a key to a locked data item. When you have the key (semaphore), you can access the data item without worrying about other tasks causing problems. Any other tasks that try to obtain the semaphore will be put to sleep until the semaphore becomes available. When you have completed your work with the data, you return the semaphore.

For semaphores to work correctly, there are two restrictions that *must* be observed at all times:

- 1) All tasks using shared data that is protected by a semaphore must *always ask for the semaphore first before accessing the data*. If some task accesses the data directly without first going through the semaphore, the data may be corrupted. No task will have safe access to the data.
- 2) A deadlock will occur if a task that owns a semaphore on some data inadvertently calls another task which needs to get a semaphore on that same data. Deadlocks and other such issues are beyond the scope of this manual. For more details on deadlocks and other problems of shared data in a multi-tasking system and the methods used to prevent them, refer to a textbook in computer science such as *Operating Systems* by Tannenbaum (Prentice-Hall).

## The Signal Semaphore

Exec semaphores are signal based. Using signal semaphores is the easiest way to protect shared, single-access resources in the Amiga. Your task will sleep until the semaphore is available for use. The **SignalSemaphore** structure is as follows:

```
struct SignalSemaphore
{
    struct Node ss_Link;
    SHORT      ss_NestCount;
    struct MinList ss_WaitQueue;
    struct SemaphoreRequest ss_MultipleLink;
    struct Task *ss_Owner;
    SHORT      ss_QueueCount;
};
```

### **ss\_Link**

is the node structure used to link semaphores together. The **ln\_Pri** and **ln\_Name** fields are used to set the priority of the semaphore in a list and to name the semaphore for public access. If a semaphore is not public the **ln\_Name** and **ln\_Pri** fields may be left NULL.

### **ss\_NestCount**

is the count of number of locks the current owner has on the Semaphore.

### **ss\_WaitQueue**

is the List Header for the list of other tasks waiting for this semaphore.

### **ss\_MultipleLink**

is the SemaphoreRequest used by **ObtainSemaphoreList()**.

### **ss\_Owner**

is the pointer to the current owning task.

### **ss\_QueueCount**

is the number of other tasks waiting for the semaphore.

A practical application of a **SignalSemaphore** would be to use it as the base of a shared data structure. For example:

```

struct MySharedList
{
    struct SignalSemaphore MY_Semaphore;
    struct MinList        MY_List;
};

```

### Initializing a SignalSemaphore Structure

To initialize a **SignalSemaphore** structure use the **InitSemaphore()** function. This function initializes the list structure and the nesting and queue counters. It does not change the semaphore's name or priority fields.

To create and initialize a semaphore for a data item such as the example **MySharedList** structure above, you would use the following:

```

struct MySharedList *MyList;

if (MyList=AllocMem(sizeof(struct MySharedList),MEMF_PUBLIC|MEMF_CLEAR))
{
    NewList(&MyList->MY_List); /* Get my MinList Header set up... */
    InitSemaphore((struct SignalSemaphore *)MyList);
}
/* If the memory did not get allocated, we must abort... */

```

### Making a SignalSemaphore Available to the Public

A semaphore should be used internally in your program if your program has more than one task operating on shared data structures. There may also be cases when you wish to make a data item public to other applications but still need to restrict its access via semaphores. In that case, you would give your semaphore a unique name and add it to the public **SignalSemaphore** list maintained by Exec. The **AddSemaphore()** function does this for you. This works in a manner similar to **AddPort()** for message ports.

To create and initialize a *public* semaphore for a data item and add it to the public semaphore list maintained by Exec, the following function should be used. (This will prevent the semaphore from being added or removed more than once by separate programs that use the semaphore).

```

struct SignalSemaphore *AddPublicSemaphore(char *Name)
{
    struct SignalSemaphore *sema=NULL;

    Forbid();
    if (!FindSemaphore(Name))
    {
        if (sema=AllocMem(sizeof(struct SignalSemaphore),MEMF_PUBLIC|MEMF_CLEAR))
        {
            sema->ss_Link.ln_Pri=0;
            sema->ss_Link.ln_Name=Name;
            /*
             * Note that we did not make a copy of Name... If a copy
             * is needed, do that here...
             */
            /* AddSemaphore() Fix... */
            InitSemaphore(sema);
            Forbid();
            Enqueue(&SysBase->SemaphoreList,sema);
            Permit();
        }
    }
    Permit();
    return(sema);
}

```

The `AddPublicSemaphore()` function shown above returns the semaphore if it was created, or `NULL` if it was not. A return value of `NULL` means that the semaphore already exists or that there was not enough free memory to create it. Two other functions you need to manage public semaphores, `RemovePublicSemaphore()` and `ObtainPublicSemaphore()` are shown below.

```
struct SignalSemaphore *RemovePublicSemaphore(char *Name)
{
    struct SignalSemaphore *semi;

    Forbid();
    if (semi=FindSemaphore(Name))
    {
        RemSemaphore(semi);      /* So no one else can find it... */
        ObtainSemaphore(semi);   /* Wait for us to be last user... */
        ReleaseSemaphore(semi);  /* Ready for cleanup... */
    }
    Permit();
    return(semi);
}
```

This function returns the semaphore if it was successfully removed or `NULL` if it did not exist. It is up to the program to free the resources that were protected by the semaphore.

### Obtaining a SignalSemaphore

Before using the data item or other resource which is protected by a semaphore, you must first obtain the semaphore. The `ObtainSemaphore()` function does this for you. If another task currently has the semaphore, your task will be put to sleep until the semaphore is released.

### NOTE

**SignalSemaphores** have nesting. That is, if your task already owns the semaphore, it will get a second ownership of that semaphore. This simplifies the writing of routines that must own the semaphore but do not know if the caller has obtained it yet.

To obtain a semaphore use:

```
struct SignalSemaphore *sema;
ObtainSemaphore(sema);
```

To obtain a *public* semaphore, the following code should be used:

```
struct SignalSemaphore *ObtainPublicSemaphore(char *Name)
{
    struct SignalSemaphore *sema;

    Forbid();
    if (sema=FindSemaphore(Name))
    {
        ObtainSemaphore(sema);
    }
    Permit();
    return(sema);
}
```

This returns the semaphore if it was obtained successfully. It returns `NULL` if the semaphore does not exist. This is *only* needed if the semaphore has a chance of going away at any time (i.e. the semaphore is public and might be removed by some other program). If there is a guarantee that the semaphore will not disappear, the semaphore address could be cached, and all that would be needed is a call to the `ObtainSemaphore()` function.

## Releasing a SignalSemaphore

Once you have obtained the semaphore and completed any operations on the semaphore protected object, you should release the semaphore. The **ReleaseSemaphore()** function does this. For each **ObtainSemaphore()** call you make, you *must* have a matching **ReleaseSemaphore()** call.

## Checking a SignalSemaphore

When you attempt to obtain a semaphore with **ObtainSemaphore()**, your task will be put to sleep if the semaphore is not currently available. If you do not want to wait, you can call **AttemptSemaphore()** instead. If the semaphore is available, **AttemptSemaphore()** obtains it for you and returns TRUE. If it is not available, the function returns FALSE immediately instead of waiting for the semaphore to be released.

## Multiple Semaphores

The semaphore system has the ability to ask for ownership of a complete list of semaphores. This can help prevent deadlocks when there are two or more tasks trying to get the same set of semaphores. If task A gets semaphore 1 and tries to obtain semaphore 2 *after* task B has obtained semaphore 2 but *before* it tries to obtain semaphore 1 then both tasks will hang. Exec provides **ObtainSemaphoreList()** and **ReleaseSemaphoreList()** to prevent this problem.

A semaphore list is a list header to a list that contains **SignalSemaphore** structures. The semaphore list must not contain any public semaphores. This is because the semaphore list functions use the standard node structures in the semaphore.

### NOTE

Since the obtain semaphore list feature uses the standard node structures in the semaphore, the semaphores in a semaphore list *must not* be on the public semaphore list.

One way to implement a semaphore list is to make a public semaphore that can be found via **FindSemaphore()** and to have its structure contain the list header for the semaphore list. This also gives you a locking semaphore for protecting the **ObtainSemaphoreList()** call. Once you have gotten access to the list with **ObtainSemaphore()**, you may obtain all the semaphores on the list via **ObtainSemaphoreList()** (or get individual semaphores with **ObtainSemaphore()**). When you are finished with the protected objects, release the semaphores on the list with **ReleaseSemaphoreList()**, and then release the list semaphore via **ReleaseSemaphore()**.

For example:

```
ObtainSemaphore((struct SignalSemaphore *)MySemaphoreList);
ObtainSemaphoreList(MySemaphoreList->MY_List);
/* Your processing of protected objects goes here */
ReleaseSemaphoreList(MySemaphoreList->MY_List);
ReleaseSemaphore((struct SignalSemaphore *)MySemaphoreList);
```

See the **MySharedList** structure above for an example of a semaphore structure with a list header.



## Example

A simple "do nothing" example of Exec signal semaphore use is shown below. When the semaphore is owned by a task, attempted access by other tasks will block. A nesting count is maintained, so the current task can safely call **ObtainSemaphore()** on the same semaphore.

```
#include <exec/types.h>
#include <exec/semaphores.h>

#include <proto/exec.h>

#include <stdio.h>

/* This prevents Lattice ctrl-C processing... */
int CXBRK(VOID) { return(0); }

struct SignalSemaphore LockSemaphore;

VOID main(int argc, char *argv[])
{
    InitSemaphore(&LockSemaphore);

    ObtainSemaphore(&LockSemaphore);
    if (argc) /* Check if CLI */
    {
        printf("This task now owns the semaphore.\n");
    }

    ReleaseSemaphore(&LockSemaphore);
    if (argc) /* Check if CLI */
    {
        printf("This task released the semaphore.\n");
    }
}
```

# **Chapter 21**

## **Exec: Interrupts**

### **Introduction**

Exec manages the decoding, dispatching, and sharing of all system interrupts. This includes control of hardware interrupts, software interrupts, task-relative interrupts (see Exceptions in the “Tasks” chapter), and interrupt disabling/enabling. In addition, Exec supports a more extended prioritization of interrupts than that provided in the 68000.

The proper operation of multitasking depends heavily on the consistent management of the interrupt system. Task activities are often driven by intersystem communication that is originated by various interrupts.

### **SEQUENCE OF EVENTS DURING AN INTERRUPT**

Before useful interrupt handling code can be executed, a considerable amount of hardware and software activity must occur. Each interrupt must propagate through several hardware and software interfaces before application code is finally dispatched:

1. A hardware device decides to cause an interrupt and sends a signal to the interrupt control portions of the 4703 (Paula) custom chip.
2. The 4703 interrupt control logic notices this new signal and performs two primary operations. First, it records that the interrupt has been requested by setting a flag bit in the INTREQ register. Second, it examines the INTENA register to determine whether the corresponding interrupt and the interrupt master are enabled. If both are enabled, the 4703 generates an interrupt request by placing the priority level of the request onto the three 68000 interrupt control input lines (IPL0,IPL1,IPL2).
3. These three signals correspond to seven interrupt priority levels in the 68000. If the priority of the new interrupt is *greater* than the current processor priority, an interrupt sequence is initiated. The priority level of the new interrupt is used to index into the top seven words of the processor address space. The odd byte (a vector number) of the indexed word is fetched and then shifted left by two to create an offset into the processor's auto-vector interrupt table. The vector offsets used are in the range of \$064 to \$07C. These are labeled as *interrupt autovectors* in the 68000 manual. The auto-vector table appears in low memory on a 68000 system, but its location for other 68000 family processors is determined by the processor's CPU Vector Base Register (VBR). VBR can be accessed from supervisor mode with the MOVEC instruction.
4. The processor then switches into *supervisor* mode (if it is not already in that mode), and saves copies of the status register and program counter (PC) onto the top of the *system* stack (additional information may be saved by processors other than the 68000). The processor priority is then raised to the level of the active interrupt.
5. From the low memory vector address (calculated in step three above), a 32-bit *autovector* address is fetched and loaded into the program counter. This is an entry point into Exec's interrupt dispatcher.
6. Exec must now further decode the interrupt by examining the INTREQ and INTENA 4703 chip registers. Once the active interrupt has been determined, Exec indexes into an **ExecBase** array to fetch the interrupt's handler entry point and handler data pointer addresses.
7. Exec now turns control over to the interrupt handler by calling it as if it were a subroutine. This handler may deal with the interrupt directly or may propagate control further by invoking interrupt server chain processing.

You can see from the above discussion that the interrupt autovectors *should never be altered by the user*. If you wish to provide your own system interrupt handler, you must use the Exec **SetIntVector()** function. You should not change the contents of any autovector location.

Task multiplexing usually occurs as the result of an interrupt. When an interrupt has finished and the processor is about to return to user mode, Exec determines whether task-scheduling attention is required. If a task was signaled during interrupt processing, the task scheduler will be invoked. Because Exec uses preemptive task scheduling, it can be said that the interrupt subsystem is the heart of task multiplexing. If, for some reason, interrupts do not occur, a task might execute forever because it cannot be forced to relinquish the CPU.

## INTERRUPT PRIORITIES

Interrupts are prioritized in hardware and software. The 68000 CPU priority at which an interrupt executes is determined strictly by hardware. In addition to this, the software imposes a finer level of *pseudo-priorities* on interrupts with the same CPU priority. These pseudo-priorities determine the order in which simultaneous interrupts of the same CPU priority are processed. Multiple interrupts with the same CPU priority but a different pseudo-priority will not interrupt one another. Interrupts are serviced by either an exclusive handler or by server chains to which many servers may be attached, as shown in the **Type** field of the next table.

The following table summarizes all interrupts by priority.

Table 21-1: Interrupts by Priority

Hardware Priority	Exec Pseudo-Priority		Label	Type
		Description		
1	1	software interrupt	SOFTINT	H
	2	disk block complete	DSKBLK	H
	3	serial transmit buffer empty	TBE	H
2	4	external INT2 & CIAA	PORTS	S
3	5	graphics coprocessor	COPER	S
	6	vertical blank interval	VERTB	S
	7	blitter finished	BLIT	H
4	8	audio channel 2	AUD2	H
	9	audio channel 0	AUD0	H
	10	audio channel 3	AUD3	H
	11	audio channel 1	AUD1	H
5	12	serial receive buffer full	RBF	H
	13	disk sync pattern found	DSKSYNC	H
6	14	external INT6 & CIAB	EXTER	S
	15	special (master enable)	INTEN	-
7	--	non-maskable interrupt	NMI	S

The 8520s (also called CIAs) are Amiga peripheral interface adapter chips that generate the INT2 and INT6 interrupts. For more information about them, see *Amiga Hardware Reference Manual*.

As described in the Motorola 68000 programmer's manual, interrupts may nest only in the direction of higher priority. Because of the time-critical nature of many interrupts on the Amiga, the CPU priority level *must never be changed* by user or system code. When the system is running in user mode (multitasking), the CPU priority level must remain set at zero. When an interrupt occurs, the CPU priority is raised to the level appropriate for that interrupt. Lowering the CPU priority would permit unlimited interrupt recursion on the system stack and would "short-circuit" the interrupt-priority scheme.

Because it is dangerous on the Amiga to hold off interrupts for any period of time, higher-level interrupt code must perform its business and exit promptly. If it is necessary to perform a time-consuming operation as the result of a high-priority interrupt, the operation should be deferred either by posting a *software interrupt* or by signalling a task. In this way, interrupt response time is kept to a minimum. Software interrupts are described in a later section.

## NONMASKABLE INTERRUPT

The 68000 provides a nonmaskable interrupt (NMI) of CPU priority 7. Although this interrupt cannot be generated by the Amiga hardware itself, it can be generated on the expansion bus by external hardware. Because this interrupt does not pass through the 4703 interrupt controller circuitry, it is capable of violating system code critical sections. In particular, it short-circuits the **DISABLE** mutual-exclusion mechanism. Code that uses NMI must not assume that it can access system data structures.

## Servicing Interrupts

Interrupts are serviced on the Amiga through the use of interrupt *handlers* and *servers*. An interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. An interrupt server is one of possibly many system routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide a means of interrupt sharing. This concept is useful for general-purpose interrupts such as vertical blanking.

At system start, Exec designates certain interrupts as handlers and others as server chains. The PORTS, COPER, VERTB, EXTER, and NMI interrupts are initialized as server chains. Therefore, each of these may execute multiple interrupt routines per each interrupt. All other interrupts are designated as handlers and are always used exclusively.

## INTERRUPT DATA STRUCTURE

Interrupt handlers and servers are defined by the Exec **Interrupt** structure. This structure specifies an interrupt routine entry point and data pointer. The C definition of this structure is as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR      is_Data;
    VOID      (*is_Code)();
};
```

Once this structure has been properly initialized, it can be used for either a handler or a server.

## ENVIRONMENT

Interrupts execute in an environment different from that of tasks. All interrupts execute in *supervisor mode* and utilize the single *system stack*. This stack is large enough to handle extreme cases of nested interrupts (of higher priorities). Interrupt processing has no effect on task stack usage.

All interrupt processing code, both handlers and servers, is invoked as assembly code subroutines. Normal assembly code register conventions dictate that the D0, D1, A0, and A1 registers be free for scratch use. In the case of an interrupt handler, some of these registers also contain data that may be useful to the handler code. See the section on handlers below.

**NOTE:** Because interrupt processing executes outside the context of most system activities, certain data structures will not be self-consistent and must be considered off limits for all practical purposes. This happens because certain system operations are not atomic in nature and may be interrupted only after executing part of an important instruction sequence. For example, memory allocation and deallocation routines do not disable interrupts. This results in the finite possibility of interrupting a memory-related routine. In such a case, a memory linked list may be inconsistent when examined from the interrupt code itself. Therefore, interrupt routines must not use any memory allocation or deallocation functions.

In addition, interrupts may not call any system function which might allocate memory, wait, manipulate unprotected lists, or modify `ExecBase->ThisTask` data (example `Forbid` and `Permit`). In practice, this means that very few system calls may be used within interrupt code. The following functions may generally be used safely within interrupts:

**Alert(), Disable(), Enable(), Signal(), Cause(), PutMsg(), ReplyMsg(), FindPort(), FindTask()**

and if you are manipulating your *own* List structures while in an interrupt:

**AddHead(), AddTail(), RemHead(), RemTail(), FindName()**

In addition, certain devices (notably the timer.device) specifically allow limited use of `SendIO()` and `BeginIO()` within interrupts.

## INTERRUPT HANDLERS

As described above, an interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. There can only be one handler per 4703 interrupt. Every interrupt handler consists of an **Interrupt** structure (as defined above) and a single assembly code routine. Optionally, a data structure pointer may also be provided. This is particularly useful for ROM-resident interrupt code.

An interrupt handler is passed control as if it were a subroutine of `Exec`. Once the handler has finished its business, it must return to `Exec` by executing an `RTS` (return from subroutine) instruction rather than an `RTE` (return from exception) instruction. Interrupt handlers should be kept very short to minimize service-time overhead and thus minimize the possibilities of interrupt overruns. As described above, an interrupt handler has the normal scratch registers at its disposal. In addition, `A5` and `A6` are free for use. These registers are saved by `Exec` as part of the interrupt initiation cycle.

For the sake of efficiency, `Exec` passes certain register parameters to the handler (see the list below). These register values may be utilized to trim a few microseconds off the execution time of a handler. All of the following registers (`D0/D1/A0/A1/A5/A6`) may be used as scratch registers by an interrupt handler, and need not be restored prior to returning.

### NOTE

Interrupt servers have different register usage rules (see the `Interrupt Servers` section).

#### Interrupt Handler Register Usage

**D0** contains no valid information.

**D1** contains the 4703 `INTENAR` and `INTREQR` registers values ANDed together. This results in an indication of which interrupts are enabled *and* active.

- A0 points to the base address of the Amiga custom chips. This information is useful for performing indexed instruction access to the chip registers.
- A1 points to the data area specified by the `is_Data` field of the **Interrupt** structure. Because this pointer is always fetched (regardless of whether you use it), it is to your advantage to make some use of it.
- A5 is used as a vector to your interrupt code.
- A6 points to the Exec library base (`SysBas`). You may use this register to call Exec functions or set it up as a base register to access your own library or device.

Interrupt handlers are established by passing the Exec function `SetIntVector()`, your initialized **Interrupt** structure, and the 4703 interrupt bit number of interest. The parameters for this function are as follows:

**SetIntVector(ULONG intNumber, struct Interrupt \*interrupt)**

The first argument is the bit number for which this interrupt server is to respond (example `INTB_VERTB`). The possible bits for interrupts are defined in *hardware/intbits.h*. The second argument is the address of an interrupt server node as described earlier in this chapter.

Keep in mind that certain interrupts are established as server chains and should not be accessed as handlers.

The following example demonstrates initialization and installation of an assembler interrupt handler. See the Resources chapter for more information on allocating resources, and the "Serial Device" chapter for the more common method of serial communications.

```
/* RBFHandler_c.c - C module of interrupt handler example
 *
 * Compiled with Lattice 5.02: LC -b1 -cfist -v -y
 * Linkage: c.o,RBFHandler_c.o,RBFHandler_a.o library LC.lib,amiga.lib
 *
 * See also Serial Device chapter (most applications use the serial.device)
 *
 * To receive characters, this example requires ascii serial input
 * at your Amiga's current serial hardware baud rate (ie. 9600 after
 * reboot, else last baud rate used)
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/execbase.h>
#include <exec/interrupts.h>
#include <resources/misc.h>
#include <hardware/custom.h>
#include <hardware/intbits.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
extern struct Custom far custom; /* defined in amiga.lib */
UBYTE *GetMiscResource(long,UBYTE *);
void FreeMiscResource(long);
#pragma libcall MiscBase GetMiscResource 6 9002
#pragma libcall MiscBase FreeMiscResource c 001
#else
extern struct Custom custom;
/* Without pragmas, you must provide C bindings for misc.resource calls.
 * See the Resources chapter.
 */
#endif
#endif
```

```

/* Our assembler interrupt handler code entry */
extern void RBFHandler();

/* Our C subroutines */
void tryRemSer(void);
void cleanup(void);
void cleanexit(UBYTE *,LONG);

#define BUFLen 256
struct OurData {
    struct Task *maintask;
    ULONG mainsig;
    UWORD bufi;
    UBYTE chbuf[BUFLen+2];
    UBYTE flbuf[BUFLen+2];
    UBYTE ourname[32];
};

struct OurData *ourdata = NULL;
struct Interrupt *RBFInterrupt = NULL;
struct Interrupt *PriorInterrupt = NULL;
struct MiscResource *MiscBase = NULL;
BOOL PriorEnable = FALSE;
BOOL Installed = FALSE;
BOOL GotPort = FALSE, GotBits = FALSE;
BYTE mainsignum = -1;
UBYTE *MyName = "RBF-Example";
extern struct ExecBase *SysBase;

void main(argc,argv)
int argc;
char **argv;
{
    struct Device *dev;
    ULONG signals;
    UBYTE *user;

    /* Try to get the serial hardware resources */
    if (NULL == (MiscBase=(struct MiscResource *)OpenResource("misc.resource")))
        cleanexit("Can't open misc.resource\n",RETURN_FAIL);

    GotPort = ((user = GetMiscResource(MR_SERIALPORT,MyName)) == NULL);
    if (user)
    {
        printf("Serial hardware currently allocated by %s\n",user);
        if (!(strcmp(user,"serial.device")))
        {
            printf("Will try to remove serial device\n");
            Forbid();
            dev=(struct Device *)FindName(&SysBase->DeviceList,"serial.device");
            if(dev) RemDevice(dev);
            Permit();
            GotPort = ((GetMiscResource(MR_SERIALPORT,MyName)) == NULL);
        }
    }

    GotBits = ((GetMiscResource(MR_SERIALBITS,MyName)) == NULL);
    if (GotPort && GotBits) printf("Allocated the serial hardware\n");
    else cleanexit("Can't allocate the serial hardware\n",RETURN_FAIL);

    /* Allocate a signal so interrupts can signal main */
    if (-1 == (mainsignum = AllocSignal(-1)))
        cleanexit("Can't allocate signal\n",RETURN_FAIL);

    /* Allocate an Interrupt node structure: */
    if (NULL == (RBFInterrupt = (struct Interrupt *)
        AllocMem((LONG)sizeof(struct Interrupt), MEMF_PUBLIC|MEMF_CLEAR)))
        cleanexit("Can't allocate interrupt structure\n",RETURN_FAIL);

    /* Allocate our data structure which includes our input buffers */
    if (NULL == (ourdata = (struct OurData *)
        AllocMem((LONG)sizeof(struct OurData), MEMF_PUBLIC|MEMF_CLEAR)))

```



```

        cleanexit("Can't allocate data structure\n",RETURN_FAIL);

/* Initialize ourdata structure */
ourdata->maintask = FindTask(NULL);
ourdata->mainsig = 1L << mainsignum;

/* Initialize the Interrupt node */
RBFInterrupt->is_Node.ln_Type = NT_INTERRUPT;
RBFInterrupt->is_Node.ln_Pri = 0;
strcpy(ourdata->ourname,MyName);
RBFInterrupt->is_Node.ln_Name = ourdata->ourname;

RBFInterrupt->is_Data = (APTR)ourdata;
RBFInterrupt->is_Code = RBFHandler;

/* Save state of RBF interrupt and disable it */
PriorEnable = custom.intena & INTF_RBF ? TRUE : FALSE;
custom.intena = INTF_RBF;

/* Install the new interrupt handler */
PriorInterrupt = SetIntVector(INTB_RBF, RBFInterrupt);
Installed = TRUE;

if (PriorInterrupt) printf("Replaced the %s RBF interrupt handler\n",
        PriorInterrupt->is_Node.ln_Name);

printf("Enabling RBF interrupt...\n");
custom.intena = INTF_SETCLR | INTF_RBF;

printf("Waiting for our handler to fill character buffer. CTRL-C to exit\n");
signals = Wait(ourdata->mainsig | SIGBREAKF_CTRL_C);

if(signals & SIGBREAKF_CTRL_C) printf("Break...\n");
printf("Character buffer contains:\n");
puts(ourdata->chbuf);

printf("\nRestoring previous handler and exiting...\n");
cleanup();
exit(RETURN_OK);
}

void cleanexit(s,e)
UBYTE *s;
LONG e;
{
    if(*s) printf(s);
    cleanup();
    exit(e);
}

void cleanup()
{
    if(Installed)
    {
        /* Disable serial int, restore prior handler and prior state */
        custom.intena = INTF_RBF;
        SetIntVector(INTB_RBF, PriorInterrupt);
        if(PriorEnable) custom.intena = INTF_SETCLR|INTF_RBF;
    }
    if(ourdata) FreeMem(ourdata,(LONG)sizeof(struct OurData));
    if(RBFInterrupt) FreeMem(RBFInterrupt,(LONG)sizeof(struct Interrupt));
    if(mainsignum != -1) FreeSignal(mainsignum);
    if(GotBits) FreeMiscResource(MR_SERIALBITS);
    if(GotPort) FreeMiscResource(MR_SERIALPORT);
}

```

The assembler interrupt handler code, **RBFHandler**, reads the complete word of serial input data from the serial hardware and then separates the character and flag bytes into separate buffers. When the buffers are full, the handler

signals the main process causing main to print the character buffer contents, remove the handler, and exit.

## NOTE

The data structure containing the main signal, main task pointer, and buffers is allocated and initialized by main, and passed to the handler (shown below) via the `is_Data` pointer of the Interrupt structure.

```
*
* RBFHandler_a.asm - Example interrupt handler code
*

INCLUDE "exec/types.i"
INCLUDE "hardware/custom.i"
INCLUDE "hardware/intbits.i"

        XDEF      _RBFHandler

JSRLIB MACRO
    XREF _LVO\1
    JSR  _LVO\1(A6)
ENDM

BUFLEN EQU 256

STRUCTURE OURDATA,0
    APTR od_maintask
    ULONG od_mainsig
    UWORD od_bufi
    STRUCT od_chbuf,BUFLEN+2
    STRUCT od_flbuf,BUFLEN+2
    STRUCT od_ourname,32
    LABEL OURDATA_SIZEOF

CODE

* Entered with:
* D0 == scratch
* D1 == INTENAT & INTREQR (scratch)
* A0 == custom chips (scratch)
* A1 == is_Data which is OURDATA structure (scratch)
* A5 == vector to our code (scratch)
* A6 == pointer to ExecBase (scratch)
*
* Note - This simple handler just receives one buffer full of serial
* input data, signals main, then ignores all subsequent serial data.
*

_RBFHandler:                                ;entry to our interrupt handler

    MOVE.W serdatr(A0),D1                    ;get the input word (flags and char)

    MOVE.W od_bufi(A1),D0                    ;get our buffer index
    CMPI.W #BUFLEN,D0                        ;no more room in our buffer ?
    BEQ.S ExitHandler                        ;yes - just exit (ignore new char)
    LEA.L od_chbuf(A1),A5                    ;else get our character buffer address
    MOVE.B D1,0(A5,D0.W)                     ;store character in our chbuf
    LEA.L od_flbuf(A1),A5                    ;get our flag buffer address
    LSR.W #8,d1                              ;shift flags down
    MOVE.B D1,0(A5,D0.W)                     ;store flags in our flbuf

    ADDQ.W #1,D0                             ;increment our buffer index
    MOVE.W D0,od_bufi(A1)                    ; and replace it
    CMPI.W #BUFLEN,D0                        ;did our buffer just become full ?
    BNE.S ExitHandler                        ;no - we can exit
    MOVE.L A0,-(SP)                          ;yes - save custom
    MOVE.L od_mainsig(A1),D0                 ;get signal allocated by main
    MOVE.L od_maintask(A1),A1                ;and pointer to main task
    JSRLIB Signal                            ;tell main we are full
    MOVE.L (SP)+,A0                          ;restore custom
    ;Note: system call trashed D0-D1/A0-A1
```

```
ExitHandler:
    MOVE.W  #INTF_RBF,intreq(A0)    ;clear the interrupt
    RTS                                ;return to exec

    END
```

## INTERRUPT SERVERS

As mentioned above, an interrupt server is one of possibly many system interrupt routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide an essential mechanism for interrupt sharing.

Interrupt servers must be used for PORTS, COPER, VERTB, EXTER, or NMI interrupts. For these interrupts, all servers are linked together in a chain. Every server in the chain will be called in turn as long as the previous server returned with the processor's Z (zero) flag set. If you determine that an interrupt was specifically for your server, you should return with the processor's Z flag cleared (non-zero condition) so that the remaining servers on the chain will be skipped.

### NOTE

VERTB (vertical blank) servers should *always* return with the Z (zero) flag set. The processor Z flag is used rather than the normal function convention of returning a result in D0 because it may be tested more quickly by Exec upon the server's return.

The easiest way to set the condition code register is to do an immediate move to the D0 register as follows:

```
SetZflag_Calls_Next:
    MOVEQ   #0,D0
    RTS

ClrZflag_Ends_Chain:
    MOVEQ   #1,D0
    RTS
```

The same Exec **Interrupt** structure used for handlers is also used for servers. Also, like interrupt handlers, servers must terminate their code with an RTS instruction.

Interrupt servers are called in priority order. The priority of a server is specified in its `is_Node.In_Pri` field. Higher-priority servers are called earlier than lower-priority servers. Adding and removing interrupt servers from a particular chain is accomplished with the Exec `AddIntServer()` and `RemIntServer()` functions. These functions require you to specify both the 4703 interrupt number and a properly initialized **Interrupt** structure.

Servers have different register values passed than handlers do. A server cannot count on the D0, D1, A0, or A6 registers containing any useful information. However, the highest priority system vertical blank server currently expects to receive a pointer to the custom chips A0. Therefore, if you install a vertical blank server at priority 10 or greater, you must place custom (\$DFF000) in A0 before exiting. Other than that, a server is free to use D0-D1 and A0-A1/A5-A6 as scratch.

### Interrupt Server Register Usage

D0 scratch

D1 scratch

A0 scratch except in certain cases (see note above)

A1 points to the data area specified by the `is_Data` field of the `Interrupt` structure. Because this pointer is always fetched (regardless of whether you use it), it is to your advantage to make some use of it. (scratch)

A5 points to your interrupt code (scratch)

A6 scratch

In a server chain, the interrupt is cleared automatically by the system. Having a server clear its interrupt is not recommended and not necessary (clearing could cause the loss of an interrupt on `PORTS` or `EXTER`).

Here is an example of a program to install and remove a low-priority vertical blank interrupt server:

```
/* Vertb_c.c - C module of interrupt server example
 * Compiled with Lattice 5.02: LC -b1 -cfist -v -y
 * Linkage: c.o,vertb_c.o,vertb_a.o library LC.lib,amiga.lib
 */
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/interrupts.h>
#include <hardware/custom.h>
#include <hardware/intbits.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
#endif

extern void VertBServer(); /* our assembler interrupt server */

struct Interrupt *VBInterrupt;
ULONG counter;

void main(argc,argv)
int argc;
char **argv;
{
    ULONG finalcount;

    /* Allocate an Interrupt node structure: */
    if (NULL == (VBInterrupt = (struct Interrupt *)
        AllocMem((LONG)sizeof(struct Interrupt), MEMF_PUBLIC|MEMF_CLEAR)))
    {
        printf("Can't allocate interrupt structure\n");
        exit(RETURN_FAIL);
    }

    /* Initialize the Interrupt node: */
    VBInterrupt->is_Node.ln_Type = NT_INTERRUPT;
    VBInterrupt->is_Node.ln_Pri = -60;
    VBInterrupt->is_Node.ln_Name = "VertB-example";
    VBInterrupt->is_Data = (APTR)&counter;
    VBInterrupt->is_Code = VertBServer;

    /* put the new interrupt server into action: */
    AddIntServer(INTB_VERTB, VBInterrupt);
    printf("VBlank server will increment a counter every frame.\n");
    printf("Counter now zero - wait a few seconds then press CTRL-C\n");
    counter = 0L;

    Wait(SIGBREAKF_CTRL_C);

    finalcount = counter;
    printf("\n%d vertical blanks occurred.\n",finalcount);
}
```

```

RemIntServer(INTB_VERTB, VBInterrupt);
FreeMem(VBInterrupt, sizeof(struct Interrupt));
}

```

This is the assembler **VertBServer** installed by the C example:

```

*
* Vertb_a.asm - assembler vertical blank server
* Passed is_Data pointer in A1 (pointer to counter in C code above)
*
        XDEF      _VertBServer
        CODE
_VertBServer:
        ADDI.L    #1, (A1) ;increment the counter is_Data points to
        MOVEQ.L   #0,D0    ;set Z flag to continue to process other vb-servers
        RTS
        END

```

## Software Interrupts

Exec provides a means of generating *software interrupts*. This type of interrupt is useful for creating special-purpose asynchronous system contexts. Software interrupts execute at a priority higher than that of tasks but lower than that of hardware interrupts, so they are often used to defer hardware interrupt processing to a lower priority. Software interrupts use the same **Interrupt** data structure as hardware interrupts. As described above, this structure contains pointers to both interrupt code and data, and should be initialized as node type **NT\_INTERRUPT** (not **NT\_SOFTINT** which is an internal Exec flag).

A software interrupt is usually activated with the **Cause()** function. If this function is called from a task, the task will be interrupted and the software interrupt will occur. If it is called from a hardware interrupt, the software interrupt will not be processed until the system exits from its last hardware interrupt. If a software interrupt occurs from within another software interrupt, it is not processed until the current one is completed.

### NOTE

Software interrupts execute in an environment almost identical to that of hardware interrupts, and the same restrictions on allowable system function calls (as described earlier) apply to both.

Software interrupts are prioritized. Unlike interrupt servers, software interrupts have only five allowable priority levels: -32, -16, 0, +16, and +32. The priority should be put into the **ln\_Pri** field prior to calling **Cause()**.

Software interrupts can also be generated by message arrival at a **PA\_SOFTINT** message port. The applications of this technique are limited since it is not permissible, with most devices, to send IO requests from within interrupt code. However, the timer.device does allow such interactions, so a self-perpetuating **PA\_SOFTINT** timer port can provide an application with quite consistent timing under varying multitasking loads. The following example demonstrates initialization of a software interrupt, and use of a **PA\_SOFTINT** port. See the Exec "Messages and Ports" chapter for more information about messages and ports.

```

/*
* TimerSoftInt.c - PA_SOFTINT MsgPort example
* Compiled with Lattice 5.02: LC -b1 -cfist -v -y
* Linkage: c.o,timersoftint.o library LC.lib,amiga.lib
*/

#include <exec/types.h>
#include <exec/interrupts.h>
#include <exec/memory.h>
#include <devices/timer.h>

```

```

#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
#endif

#define MIC_DELAY 1000

/* our functions */
void *tsoftcode(void); /* our timer softint code */
void cleanexit(UBYTE *, LONG);
void cleanup(void);
void begintr(struct timerequest *);

/* Opens and allocations we must clean up.
 * Note - casts of timerequest pointers to (struct IOStdReq *)
 * in this example are to eliminate compiler warnings.
 * A timerequest starts with but is larger than an IOStdReq.
 */
struct Interrupt *tint = NULL;
struct timerequest *treq = NULL;
struct MsgPort *tport = NULL;
BOOL OpenedTimer = FALSE;

/* Global variables shared with the softint code
 * If our tsoftcode was in assembler, we could use is_Data
 * pointer instead to tell softint code where shared data is
 */
#define OFF 0
#define ON 1
#define STOPPED 2
BOOL SFlag=OFF;
ULONG counter;

char tportname[] = "RKM_timersoftint";

void main(argc, argv)
int argc;
char **argv;
{
    ULONG endcount;

    if(!argc) exit(RETURN_FAIL); /* CLI only example...Uses printf */

    /* Allocate our MsgPort and Interrupt structures
     * Since we can't use CreatePort, we'll build the port ourselves.
     * Create Port creates a PA_SIGNAL and allocs a signal.
     * We want a PA_SOFTINT port;
     */

    if(!(tport = (struct MsgPort *)
        AllocMem(sizeof(struct MsgPort),MEMF_PUBLIC|MEMF_CLEAR)))
        cleanexit("Can't allocmem msgport\n",RETURN_FAIL);

    if(!(tint = (struct Interrupt *)
        AllocMem(sizeof(struct Interrupt),MEMF_PUBLIC|MEMF_CLEAR)))
        cleanexit("Can't allocmem interrupt\n",RETURN_FAIL);

    /* Set up the (software) interrupt structure.
     * Note that we are priority 0. Software interrupts may only be
     * priority -32, -16, 0, +16, or +32. Also note that the
     * correct node type for a software interrupt is NT_INTERRUPT.
     * (NT_SOFTINT is an internal flag of Exec's)
     * This is the same setup as that for a software interrupt
     * which you Cause(). If our interrupt code was in assembler,
     * you could initialize is_Data here to contain a pointer
     * to shared data structures. An assembler software interrupt
     * routine would receive the is_Data pointer in A1.

```

```

*/
tint->is_Code = (VOID (*)( ))tsoftcode; /* Our softint routine */
tint->is_Node.ln_Type = NT_INTERRUPT;
tint->is_Node.ln_Pri = 0;

/* Set up the PA_SOFTINT msgport */
tport->mp_Node.ln_Type = NT_MSGPORT;
tport->mp_Node.ln_Name = (char *)tportname;
tport->mp_Flags = PA_SOFTINT;
tport->mp_SigTask = (struct Task *)tint; /* Ptr to interrupt struct */

/* Not using CreatePort, so we must add the port ourselves */
AddPort(tport);

/* Now Create the IO request */
if(! (treq=(struct timerequest *)
    CreateExtIO(tport,sizeof(struct timerequest))))
    cleanexit("Can't create ioreq\n", RETURN_FAIL);

/* Open the timer device - Note 0 return means success */
if(OpenDevice("timer.device",UNIT_MICROHZ,(struct IOStdReq *)treq,0))
    cleanexit("Can't open timer device\n",RETURN_FAIL);

OpenedTimer = TRUE; /* Flag for closing in cleanup */

/* Now, let's do something with it */
counter = 0L;
SFlag = ON;
begintr(treq); /* Prime the pump with first timer request */

printf("Timer softint is counting milliseconds. Press CTRL-C to exit...");
Wait(SIGBREAKF_CTRL_C);

endcount = counter;
printf("\n\nSoftint counted %ld milliseconds\n",endcount);
printf("Stopping timer and exiting\n");

SFlag = OFF;
while(SFlag != STOPPED) Delay(10);

cleanup();
exit(RETURN_OK);
}

/* Routine called as software interrupt */
VOID *tsoftcode()
{
    struct timerequest *tr;

    /* Remove our message from our port */
    tr = (struct timerequest *)GetMsg(tport);

    /* If main hasn't flagged us to stop, keep the ball rolling */
    if((tr)&&(SFlag==ON))
    {
        /* Increment the counter and send the timer request out again.
        *
        * IMPORTANT: This self-perpetuating technique of calling
        * BeginIO during a software interrupt may only be used with
        * the audio and timer device.
        */
        counter++;
        begintr(tr);
    }
    /* Else flag main we have indeed stopped */
    else(SFlag=STOPPED);
    return(0);
}

```

```

/* begintr(tr)
 * Sets up and sends off timer request.
 * IMPORTANT: Do not BeginIO to any device other than timer or audio
 * from within a software or hardware interrupt. The BeginIO code
 * of other devices may allocate memory, wait, or perform other
 * functions which are illegal or dangerous during interrupts.
 */
void begintr(struct timerequest *tr)
{
    /* Set up the timer command */
    tr->tr_node.io_Command = TR_ADDREQUEST;
    tr->tr_time.tv_micro = MIC_DELAY;
    BeginIO((struct IOStdReq *)tr);
}

/* Prints message if any, cleans up, and exits */
void cleanexit(UBYTE *s, LONG n)
{
    if(*s) printf(s);
    cleanup();
    exit(n);
}

/* Close/deallocate everything opened/allocated */
void cleanup()
{
    if(OpenedTimer) CloseDevice((struct IOStdReq *)treq);
    if(treq) DeleteExtIO((struct IOStdReq *)treq);
    if(tport)
    {
        RemPort(tport);
        FreeMem(tport, sizeof(struct timerequest));
    }
    if(tint) FreeMem(tint, sizeof(struct Interrupt));
}

```

## Disabling Interrupts

As mentioned in the “Tasks” chapter, it is sometimes necessary to disable interrupts when examining or modifying certain shared system data structures. However, for proper system operation, interrupts should never be disabled unless absolutely necessary, and never for more than 250 microseconds. Interrupt disabling is controlled with the **Disable()** and **Enable()** functions. Although assembler **DISABLE** and **ENABLE** macros are provided, we strongly suggest that you use the system functions rather than the macros for upwards compatibility and smaller code size.

In some system code, there are nested disabled sections. Such code requires that interrupts be disabled with the first **Disable()** and not re-enabled until the *last* **Enable()**. The system **Enable** and **Disable** functions are designed to permit this sort of nesting.

**Disable()** increments a counter to track how many levels of disable have been issued. Only 126 levels of nesting are permitted. **Enable()** decrements the counter, and reenables interrupts when the last disable level has been exited.



# Chapter 22

## Exec: ROM-Wack

### Introduction

*Wack* is a keystroke-interactive, run-time debugger used with Amiga hardware and software. *ROM-Wack* is a small, Kickstart resident version primarily useful for examining data structures after a system crash. ROM-Wack is a functional subset of Wack.

### Getting to ROM-Wack

ROM-Wack is invoked by Exec automatically upon a fatal system error, or it can be explicitly invoked through the Exec **Debug(0)** function. Once invoked, communication is performed through the built-in serial port. ROM-Wack requires the use of an external terminal or another Amiga set to 9600 baud, 8 bits, no parity, and one stop bit.

When a fatal system error occurs, ROM-Wack can be used to examine memory in an attempt to locate the source of the failure. The machine will be frozen at the point in which the error occurred. ROM-Wack will not disturb the state of the system beyond using a small amount of supervisor stack, memory between \$200 and \$400, and the serial data port.

ROM-Wack may be invoked from the secret Workbench menu. To get to the secret Workbench menu, use "LoadWB -debug" instead of "LoadWB" from the CLI or startup-sequence. The secret menu will be to the right of the normal menus. A program may explicitly invoke ROM-Wack by calling the Exec **Debug(0)** function. This is useful during the debug phase of development for establishing program breakpoints. **Debug()** may be called from any context, including interrupts, supervisor mode or during **Forbid()** or **Disable()**. For future compatibility, **Debug()** should be called with a single, null parameter. RAM loaded debuggers may also hook into the **Debug()** vector.

**NOTE:** When ROM-Wack is called from a program, system interrupts continue to process, but multi-tasking is disabled. Generally this is not harmful to the system. Your graphics will still display, keys may be typed, and so on. However, many interrupts deposit raw data into bounded or circular buffers. These interrupts often signal related device tasks to further process these buffers. If too many interrupts occur, device buffers may begin to overflow or wrap around. You should limit the number of interrupt actions (such as typing keys on the Amiga keyboard) you perform while executing in ROM-Wack.

Finally, certain system failures are so serious that the system is forced to reboot. Before the reboot, the power LED will flash slowly. By typing a DEL (\$7F) character on the remote terminal, the system will drop into ROM-Wack.

## Keystrokes, Numbers, and Symbols

ROM-Wack performs a function for every keyboard keystroke. In ROM-Wack, these functions are permanently bound to certain keys. For example, typing ">" will immediately result in the execution of the **next-word** function. This type of operation gives a keystroke-interactive feel to most of the common ROM-Wack commands. The Wack feature of arbitrary key binding is not available in ROM-Wack.

Most punctuation marks are bound to simple actions, such as displaying a memory frame, moving the frame pointer, or altering a single word. These actions are always performed immediately. In contrast, the keys A-Z, a-z, and 0-9 are bound to a function that collects the keys as a string. When such a string is terminated with <RETURN>, the keys are interpreted as a single *symbol* or *number*. During the "collection" of a symbol or number string, typing a backspace deletes the previous character. Typing <CTRL-X> deletes the entire line.

If a string of keys forms a number, that number is treated as a hexadecimal value. If a string of keys is neither a number nor a known symbol, the message "unknown symbol" is presented.

## Register Frame

When ROM-Wack is invoked for any reason, a *register frame* is displayed:

```
ROM-Wack
PC: 00F00AB4  SR: 0000  USP: 001268  SSP: 07FFE8  XCPT: 0000  TASK: 0008B8
DR: 00000001  00000004  0000000C  00000AB4  00000001  0000001C  00000914  00000914
AR: 00000AB4  00F0D348  00011A80  00000B9C  00F20770  00F20380  00000604
SF: 0000 00F0 0AB4 0014 00F0 0AB4 0014 00F0 0AB4 0004 00F0 0AB4 0000 0004 0000
```

This frame displays the current processor state and system context from which you entered ROM-Wack. If you are familiar with the 68000 processors, most of this frame should be obvious: PC for program counter, SR for status register, USP for user stack pointer, SSP for system stack pointer, etc.

The XCPT field indicates the 68000 exception vector number that forced entry into ROM-Wack.

Standard 68000 exception vector (XCPT) names & numbers	
0	Normal entry.
2	Bus error and/or access to non-existent memory.
3	Address error. Instructions or data accessed at an odd address. The 68020 only takes address errors for instruction access.
4	Illegal instruction (other than \$AXXX or \$FXXX).
5	Zero divide.
6	CHK instruction. Bounds check failed.
7	TRAPV instruction. Trap on overflow.
8	Privilege violation. Supervisor instruction by user code.
9	Trace (single step).
A	Line 1010 emulator (instruction starting with \$A).
B	Line 1111 emulator (instruction starting with \$F).
2X	TRAP instruction number X (\$2F for breakpoint).

#### NOTE

Exec also uses the term *exception* for asynchronous task events driven by the signal system. See the "Tasks" chapter for details.

The TASK field indicates from which task the system entered ROM-Wack. If this field is zero, the system entered from supervisor mode.

The SF line provides a backtrace of the current stack frame. This is often useful for determining the current execution context (last function called, for example). The user stack is displayed for entry from a task; the system stack for entry from supervisor mode.

## Display Frames

ROM-Wack displays memory in fixed size *frames*. A frame may vary in size from 0 to 64K bytes. Frames normally show addresses, word size hex data, and ASCII equivalent characters. By default, ROM-Wack will pack as much memory content as it can onto a single line. Sometimes it is preferable to see more or less than this default frame size. The frame size may be modified with `:n`. Here "n" represents the number of bytes (rounded to the next unit size) that will be displayed.

```
:4
F000C4 6578 6563  e x e c
:20
F000C4 6578 6563 2E6C 6962 7261 7279 0000 4AFC  e x e c . l i b r a r y ...
F000D4 00F0 00D2 00F0 2918 0019 0978 00F0 00C4  ..... ) ^ X . . ^ Y ^ I . . . . .
```

A `":0"` frame size is required for altering the write-only custom chip registers. Custom chip registers are either *read-only* or *write-only*.

## Relative Positioning

.	forward a frame
,	backward a frame
>	forward a word
<	backward a word
+n	forward n bytes
-n	backward n bytes
<RETURN>	redisplay current frame
<SPACE>	forward a word
<BKSP>	backward a word
:XX	Set frame size to XX. Zero is allowed

## Absolute Positioning

There are a few commands that perform absolute positioning. Typing a hex number sets the current address. ROM-Wack also maintains an indirection stack to help you walk down linked lists of absolute pointers:

```
4
000004 0000 11ec 00f0 0a8e 00f0 0a90 00f0 0a92 ....^Q.....^J.....^J.....
[      (use current longword as the next address)
0011ec 0000 18f6 0000 1332 0900 00f0 086a 0000 ....^X.....^S 2^I.....
]      (return to the previous "indirected" address)
000004 0000 11ec 00f0 0a8e 00f0 0a90 00f0 0a92 ....^Q.....^J.....^J.....
```

## Altering Memory & Stored Registers

The = command lets you modify your current memory word. ROM-Wack will prompt for the new value.

```
20134
020134 0000 0000 0000 .....
020134 0000 =767
020134 0767 0000 0000 ^G g.....
```

If the frame size is zero, the contents of the word will not be displayed prior to the modification:

```
:0
dff180
DFF180 xxxx =0ff0
```

If you decide not to modify the contents after typing <= >, press <RETURN> without typing a number. If you have already typed a number, type <CTRL-X>.

The alter command performs a repeated <= > which is handy for setting up tables. While in this mode, the less-than and greater-than keys (angle brackets) will move you forward or backward one word. To exit from this mode, type a <RETURN> with no preceding number.

```

alter
001400 0280 =222
001402 00C8 =<
001400 0222 =333
001402 00C8 =444
001404 0000 =0
001406 3700 =>
001408 0000 =666
00140A 0000 =<RETURN>

```

You can modify registers when single-stepping or breakpointing. Typing < ! > followed by the register name (D0-D7, A0-A6, etc.) lets you make modifications. SR and SSP cannot be modified.

The **fill** command fills memory with a given pattern from the current location to an upper bound. The **limit** command determines the upper bound of the fill. The size of the fill pattern determines the number of bytes the pattern occupies in memory. For example, typing "45" fills bytes with \$45. Typing "045" fills words with \$0045, and "0000045" fills longs with \$00000045.

The **find** command searches from the current position to the upper bound. The pattern may be from one to four bytes in length. The pattern is not affected by the alignment of memory; that is, byte alignment is used for all searches regardless of the pattern size.

### CAUTION

Using the **fill** or **find** commands without properly setting the limit can destroy data in memory. To set the upper bound, go to an address, then type **limit** or press the carat-key.

## Execution Control

<b>go</b>	<b>execute from current memory address</b>
<b>resume</b>	<b>resume at current address from the PC (exit wack)</b>
<b>^D</b>	<b>same as resume</b>
<b>^I</b>	<b>(tab) single instruction step</b>
<b>boot</b>	<b>reboot system (cold-reset)</b>

## Breakpoints

ROM-Wack has the ability to perform limited RAM-based program breakpoints. Up to 16 breakpoints may be set. The breakpoint commands are as follows:

<b>set</b>	<b>set breakpoint at current address</b>
<b>show</b>	<b>show all breakpoint addresses</b>
<b>clear</b>	<b>clear breakpoint at current address</b>
<b>reset</b>	<b>clear all breakpoints</b>

To set a breakpoint, position the address pointer to the break address and type **set**. Resume program execution with **go** or **resume**. When your breakpoint has been reached, ROM-Wack will display a register frame. The breakpoint is automatically cleared once the breakpoint is reached.

## Other Commands

<b>alter</b>	<b>Like =, but allows modification of multiple locations</b>
<b>find</b>	<b>Search for pattern</b>
<b>fill</b>	<b>Fill memory with bytes, words or longs up to the limit</b>
<b>limit</b>	<b>Set limit for use with commands like fill</b>
<b>list</b>	<b>Display the contents of an Exec list structure</b>
<b>regs</b>	<b>List the current register set</b>
<b>user</b>	<b>Try to restart multitasking</b>
<b>?</b>	<b>Print help</b>
<b>[</b>	<b>Indirect from current address</b>
<b>]</b>	<b>Exdirect</b>
<b>^</b>	<b>Bound to limit</b>
<b>!</b>	<b>Set register (type register number)</b>
<b>=</b>	<b>Change memory</b>
<b>^X</b>	<b>Cancel gathering keys</b>

## Returning to Multitasking After a Crash

The **user** command forces the machine back into multitasking mode after a crash that invoked ROM-Wack. This gives your system a chance to flush disk buffers before you reset, thus securing your disk's super-structures.

Once you type **user**, you cannot exit from ROM-Wack, so you should use this command only when you want to reboot after debugging. Give your disk a few seconds to write out its buffers. If your machine is in serious trouble, the **user** command may not work.

# **Chapter 23**

## **Graphics: Primitives**

This chapter describes the basic graphics tools. It covers the graphics support structures, display routines, and drawing routines. Many of the operations described in this section are also performed by the Intuition software. See the “Intuition” chapter for more information.

### **Introduction**

The Amiga has two basic types of graphics support routines: display routines and drawing routines. These routines are very versatile and allow you to define any combination of drawing and display areas you may wish to use.

The first section of this chapter defines the display routines. These routines show you how to form and manipulate a display, including the following aspects of display use:

- How to identify the memory area that you wish to have displayed
- How to position the display area window to show only a certain portion of a larger drawing area
- How to split the screen into as many vertically stacked slices as you wish

- Whether to use the high-resolution or low-resolution display mode for a particular screen segment, and whether to use interlaced or non-interlaced mode
- How to determine the correct number of pixels across and lines down for a particular section of the display
- How to specify how many color choices per pixel are to be available in a specific section of the display

The next section of the chapter explains all of the available modes of drawing supported by the system software, including how to do the following:

- Reserve memory space for use by the drawing routines
- Define the colors that can be drawn into a drawing area
- Define the colors of the drawing pens (foreground pen, background pen for patterns, and outline pen for area-fill outlines)
- Define the pen position in the drawing area
- Drawing primitives; lines, rectangles, circles and ellipses
- Define vertex points for area-filling, and specify the area-fill color and pattern
- Define a pattern for patterned line drawing
- Change drawing modes
- Read or write individual pixels in a drawing area
- Copy rectangular blocks of drawing area data from one drawing area to another
- Use a template (predefined shape) to draw an object into a drawing area

## **COMPONENTS OF A DISPLAY**

In producing a display, you are concerned with two primary components: sprites and the playfield. Sprites are the easily movable parts of the display. The playfield is the static part of the display and forms a backdrop against which the sprites can move and with which the sprites can interact.

This chapter covers the creation of the background. Sprites are described in the “Animation” chapter.

## **INTRODUCTION TO RASTER DISPLAYS**

There are three television standards in common use around the world: NTSC, PAL, and SECAM. NTSC is used primarily in the United States; PAL and SECAM are primarily used in Europe. The Amiga currently supports both NTSC and PAL. The major differences between the two systems are refresh frequency and the number of scan lines produced. Where necessary, the differences will be described and any special considerations will be mentioned.



The Amiga produces its video displays on standard television or video monitors by using raster display techniques. The picture you see on the video display screen is made up of a series of horizontal video lines stacked one on top of another, as illustrated in the following figure. Each line represents one sweep of an electronic video beam, which “paints” the picture as it moves along. The beam sweeps from left to right, producing the full screen one line at a time. After producing the full screen, the beam returns to the top of the display screen.

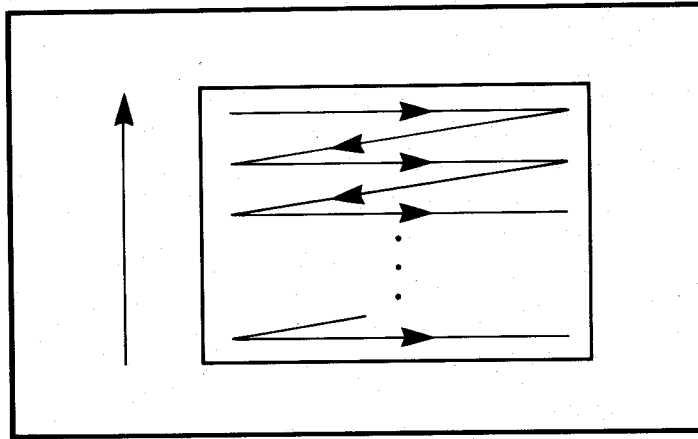


Figure 23-1: How the Video Display Picture Is Produced

The diagonal lines in the figure show how the video beam returns to the start of each horizontal line.

#### Effect of Display Overscan on the Viewing Area

To assure that the picture entirely fills the viewable region of the screen, the manufacturer of the video display usually creates a deliberate *overscan*. That is, the video beam is swept across a larger section than the front face of the screen can actually display. The video beam actually covers 262 vertical lines (312 for PAL). The user, however, sees only the portion of the picture that is within the center region of the display, which is about 200 rows on a NTSC machine, as illustrated in the following figure. The graphics system software allows you to specify, for a NTSC display, up to 241 lines (482 interlace). For a PAL display; up to 283 lines for a non-interlaced screen, 566 for an interlaced screen.

Overscan also restricts the amount of video data that can appear on each display line. The system software allows you to specify a display width of up to 362 pixels (or 724 in high-resolution mode) per horizontal line. Generally, however, you should use the standard values of 320 (or 640 in high-resolution mode) for most applications.

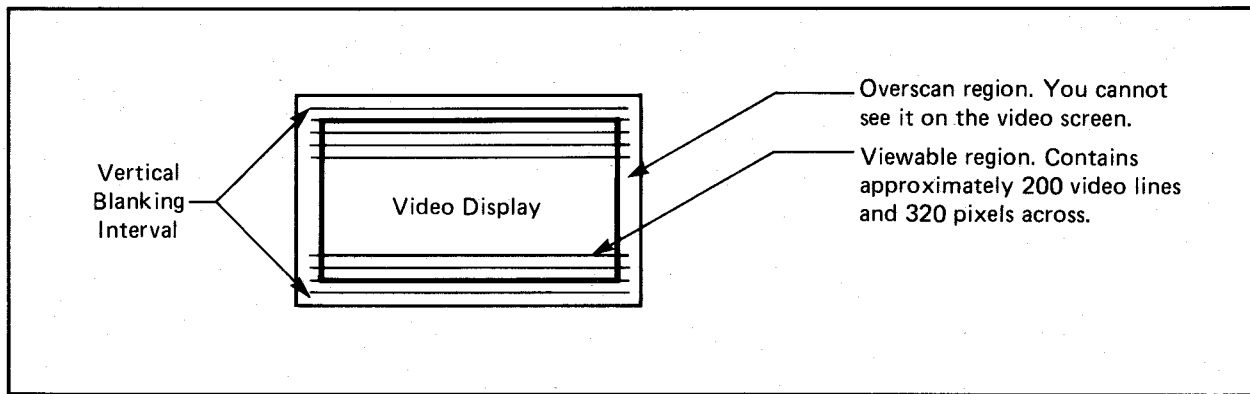


Figure 23-2: Display Overscan Restricts Usable Picture Area

The time during which the video beam is in the region below the bottom line of the viewable area and above the top line of the next display field is called the *vertical blanking interval*. A practical minimum to allow for this interval is 21 lines for NTSC, 29 lines for PAL.

### Color Information for the Video Lines

The hardware reads the system display memory to obtain the color information for each line. As the video display beam sweeps across the screen producing the display line, it changes color, producing the images you have defined.

### INTERLACED AND NON-INTERLACED MODES

In producing the complete display (262 video lines), the video display device produces the top line, then the next lower line, then the next, until it reaches the bottom of the screen. When it reaches the bottom, it returns to the top to start a new scan of the screen. Each complete set of 262 lines is called a *display field*. It takes about 1/60th (for NTSC; PAL takes 1/50th) of a second to produce a complete display field.

The Amiga has two vertical display modes: *interlaced* and *non-interlaced*. In non-interlaced mode, the video display produces the same picture for each successive display field. A non-interlaced NTSC display normally has about 200 lines in the viewable area (for a full-screen size display), while a PAL display will show 256 lines.

Interlaced mode allows you to double the viewable area. On a NTSC display this amounts to 400 lines, while on a PAL display it amounts to 512 lines.

For interlaced mode, the video beam scans the screen at the same rate (1/60th of a second per complete NTSC video display field); however, it takes two display fields to form a complete video display picture. During the first of each pair of display fields, the system hardware shows the odd-numbered lines of an interlaced display (1, 3, 5, and so on). During the second display field, it shows the even-numbered lines (2, 4, 6 and so on). During the display, the hardware moves the second display field's lines downward slightly from the position of the first, so that the lines in the second field are "interlaced" with those of the first field, giving the higher vertical resolution of this mode. For an interlaced display, the data in memory defines twice as many lines as for a non-interlaced display, as shown in the following figure.

DATA AS DISPLAYED	DATA IN MEMORY
Odd field — Line 1	Line 1
Even field — Line 1	Line 2
Odd field — Line 2	Line 3
Even field — Line 2	Line 4
.	.
.	.
Odd field — Last line	Line 399
Even field — Last line	Line 400

Figure 23-3: Interlaced Mode — Display Fields and Data in Memory

The following figure shows a display formed as display lines 1, 2, 3, 4, ... 400. The 400-line interlaced display uses the same physical display area as a 200-line non-interlaced display.

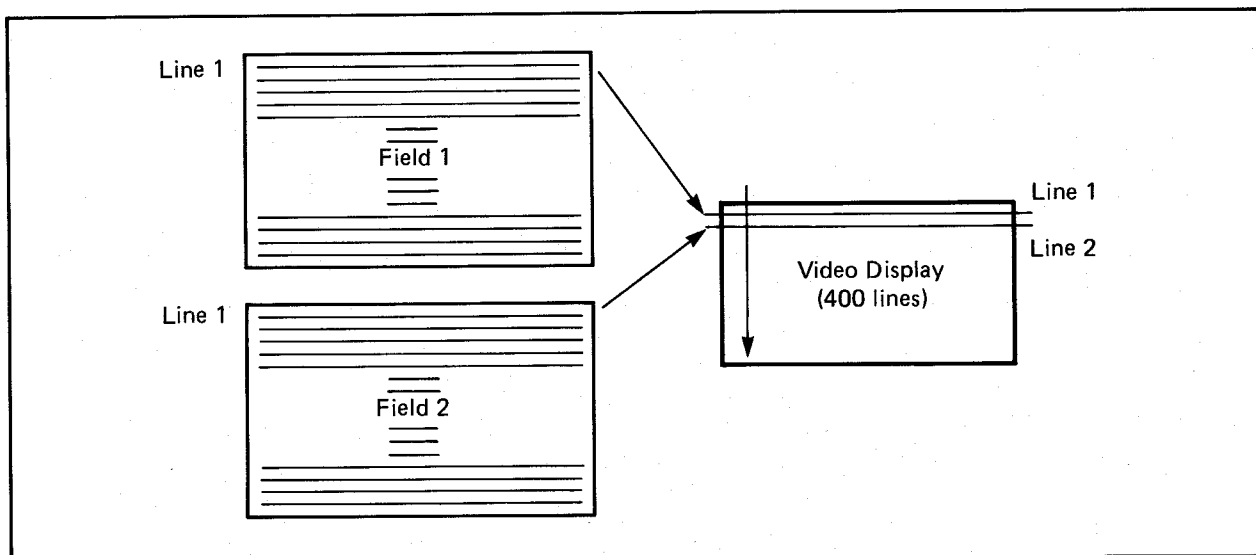


Figure 23-4: Interlaced Mode Doubles Vertical Resolution

During an interlaced display, it appears that both display fields are present on the screen at the same time and form one complete picture. This phenomenon is called *video persistence*. Interlaced displays will appear to flicker if adjacent (odd and even) scan lines have contrasting brightness. This phenomenon can be reduced by using a long-persistence monitor, or alleviated completely with a hardware “screen de-interlacer”.

## HIGH- AND LOW-RESOLUTION MODES

The Amiga also has two horizontal display modes: *high-resolution* and *low-resolution*. High-resolution mode provides (nominally) 640 distinct pixels (picture elements) across a horizontal line. Low-resolution provides (nominally) 320 pixels across each line. Out of a total of 4,096 colors, low-resolution mode allows up to 32 colors and high-resolution mode allows up to 16 colors on-screen at one time.

Two other low-resolution display modes also affect the number of colors you can display at one time: *Extra-Half-Brite* and *Hold-And-Modify*.

Extra-Half-Brite allows for 64 colors on-screen at once; 32 colors plus 32 additional colors that are half the intensity of the first 32. For example, if color 1 is defined as 0xffff (white), then color 33 is 0x7777 (grey).

Hold-And-Modify (HAM) allows you to display the entire palette of 4,096 colors on-screen at once with certain restrictions, explained later.

## FORMING AN IMAGE

To create an image, you write data (that is, you “draw”) into a memory area in the computer. From this memory area, the system can retrieve the image for display. You tell the system exactly how the memory area is organized, so that the display is correctly produced. You use a block of memory words at sequentially increasing addresses to represent a rectangular region of data bits. The following figure shows the contents of three example memory words: 0 bits are shown as blank rectangles, and 1 bits as filled-in rectangles.

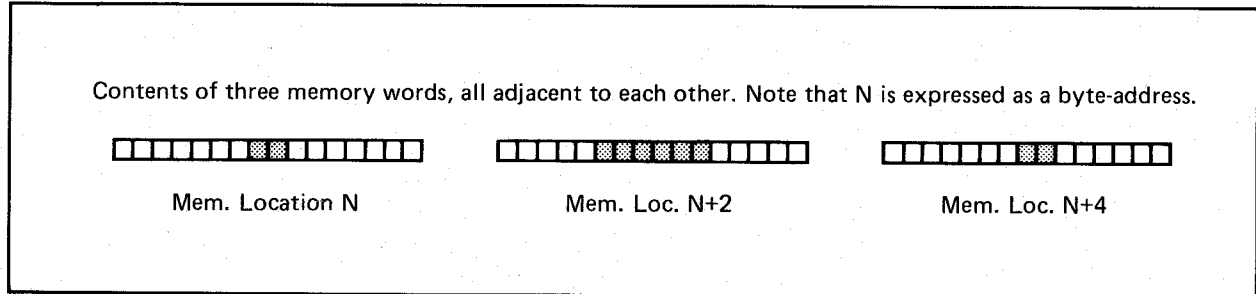


Figure 23-5: Sample Memory Words

The system software lets you define linear memory as rectangular regions, called *bit-planes*. The following figure shows how the system views the same three words as a bit-plane, wherein the data bits form an x-y plane.

Three memory words, organized as a bit-plane.

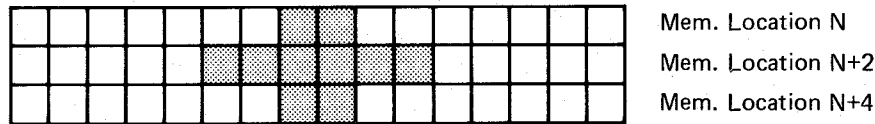


Figure 23-6: A Rectangular “Look” at the Sample Memory Words

The following figure shows how 4,000 words (8,000 bytes) of memory can be organized to provide enough bits to define a single bit-plane of a full-screen, low-resolution video display (320 x 200).

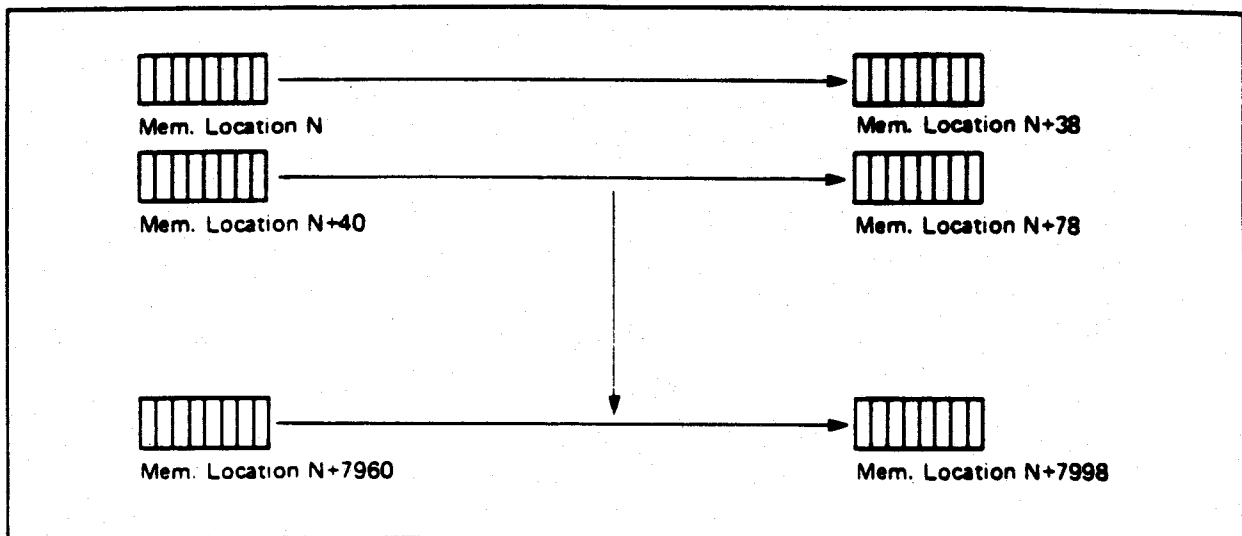


Figure 23-7: Bit-Plane for a Full-screen, Low-resolution Display

Each memory data word contains 16 data bits. The color of each pixel on a video display line is directly related to the value of one or more data bits in memory, as follows:

- If you create a display in which each pixel is related to only one data bit, you can select from only two possible colors, because each bit can have a value of only 0 or 1.
- If you use two bits per pixel, there is a choice of four different colors because there are four possible combinations of the values of 0 and 1 from each of the two bits.
- If you specify three, four, or five bits per pixel, you will have eight, sixteen, or thirty-two possible choices of a color for a pixel.

- If you use six bits per pixel, then depending on the video mode (EXTRA\_HALFBRITE or HAM), you will have sixty-four or 4,096 possible choices for a pixel.

To create multicolored images, you must tell the system how many bits are to be used per pixel. The number of bits per pixel is the same as the number of bit-planes used to define the image.

As the video beam sweeps across the screen, the system retrieves one data bit from each bit-plane. Each of the data bits is taken from a different bit-plane, and one or more bit-planes are used to fully define the video display screen. For each pixel, data-bits in the same x,y position in each bit-plane are combined by the system hardware to create a binary value. This value determines the color that appears on the video display for that pixel.

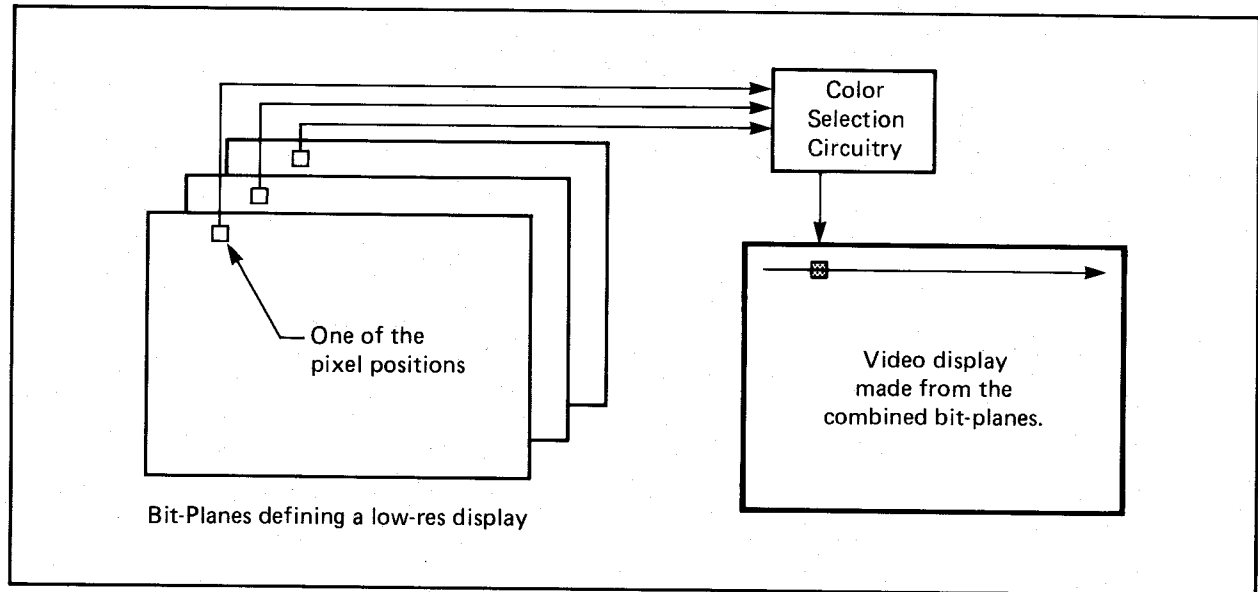


Figure 23-8: Bits from Each Bit-Plane Select Pixel Color

You will find more information showing how the data bits actually select the color of the displayed pixel in the section called “ViewPort Color Selection.”

## ROLE OF THE COPPER (COPROCESSOR)

The Amiga has a special-purpose coprocessor, called the *Copper*, that can control nearly the entire graphics system. The Copper can control register updates, reposition sprites, change the color palette, and update the blitter. The graphics and animation routines use the Copper to set up lists of instructions for handling displays, and advanced users can create their own “user Copper lists.”

# Display Routines and Structures

## CAUTION

This section describes the lowest-level graphics interface to the system hardware. If you use any of the routines and the data structures described in these sections, your program will essentially take over the entire display. It will not, therefore, be compatible with the multiwindow operating environment, known as Intuition, which is used by AmigaDOS.

The descriptions of the display routines, as well as those of the drawing routines, occasionally use the same terminology as that in the "Intuition" chapter. These routines and data structures are the same ones that Intuition software uses to produce its displays.

The computer produces a display from a set of instructions you define. You organize the instructions as a set of parameters known as the **View** structure. The following figure shows how the system interprets the contents of a **View** structure. This drawing shows a complete display composed of two different component parts, which could (for example) be a low-resolution, multicolored part and a high-resolution, two-colored part.

A complete display consists of one or more **ViewPorts**, whose display sections are separated from each other by at least one blank scan line. (If the system must make many changes to the display during the transition from one **ViewPort** to the next, there may be two or more blank scanlines between the **ViewPorts**.) The viewable area defined by each **ViewPort** is a rectangular portion from the same size (or larger) raster. You are essentially defining a display consisting of a number of vertically stacked display areas in which separate sections of graphics rasters can be shown.

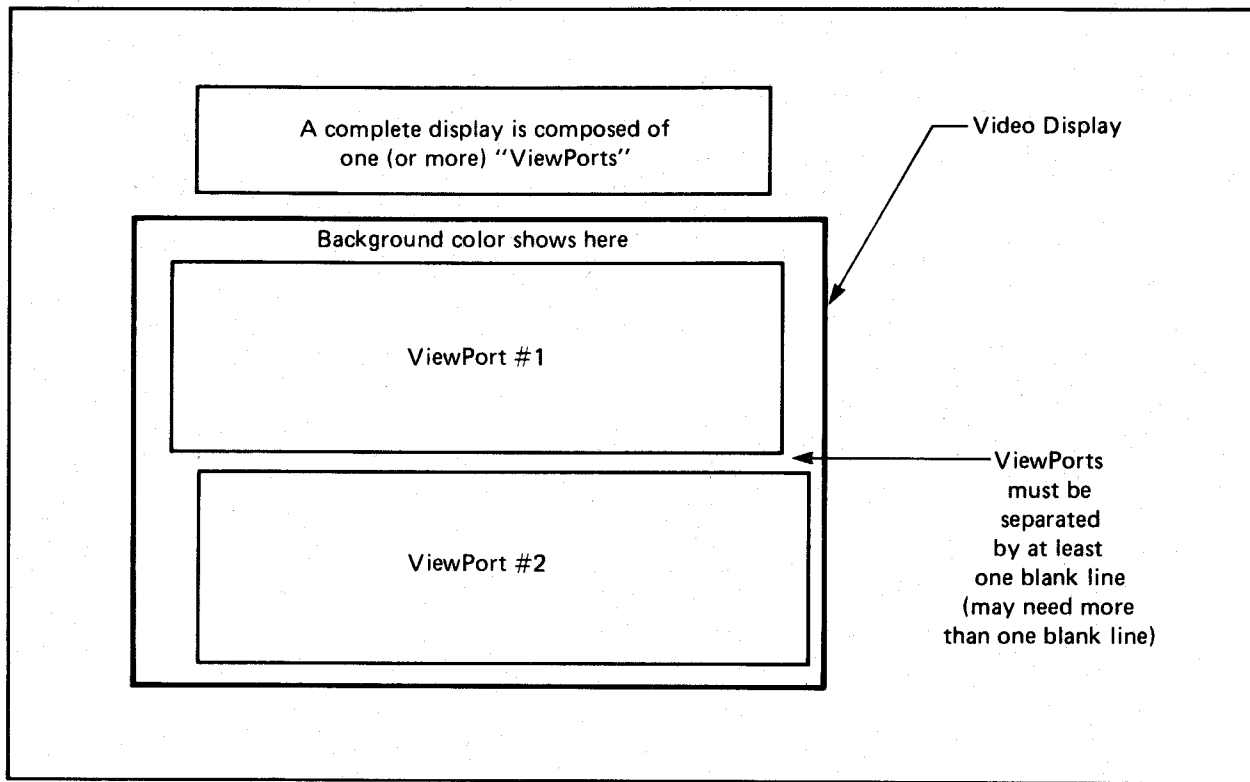


Figure 23-9: The Display Is Composed of ViewPorts

### LIMITATIONS ON THE USE OF VIEWPORTS

The system software for defining **ViewPorts** allows only vertically stacked fields to be defined. The following figure shows acceptable and unacceptable display configurations. If you want to create overlapping windows, define a single **ViewPort** and manage the windows yourself within that **ViewPort**.



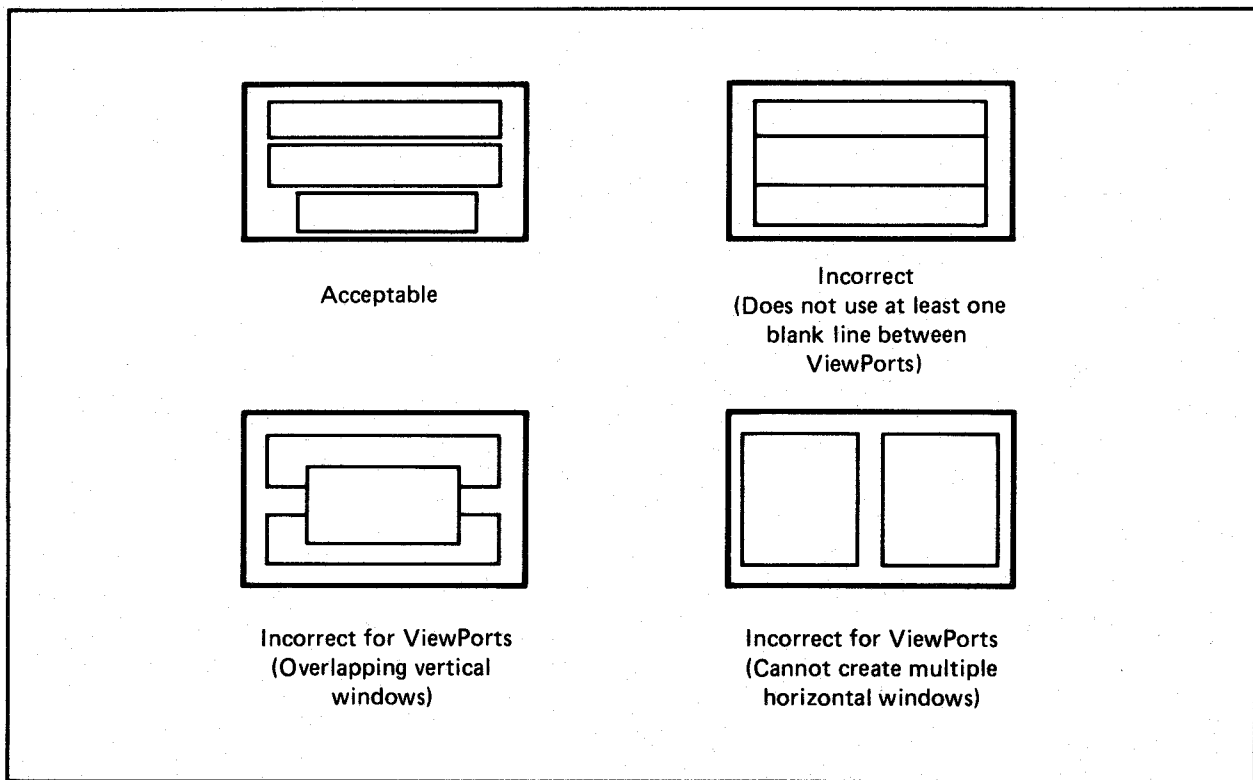


Figure 23-10: Correct and Incorrect Uses of ViewPorts

A **ViewPort** is related to the custom screen option of Intuition. In a custom screen, you can split the screen into slices as shown in the “correct” illustration of the above figure. Each custom screen can have its own set of colors, use its own resolution, and show its own display area.

### CHARACTERISTICS OF A VIEWPORT

To describe a **ViewPort** fully, you need to set the following parameters: height, width, and display mode.

In addition to these parameters, you must tell the system the location in memory from which the data for the **ViewPort** display should be retrieved, and how to position the final **ViewPort** display on the screen.

### VIEWPORT SIZE SPECIFICATIONS

The following figure illustrates that the variables **DHeight**, and **DWidth** specify the size of a **ViewPort**.

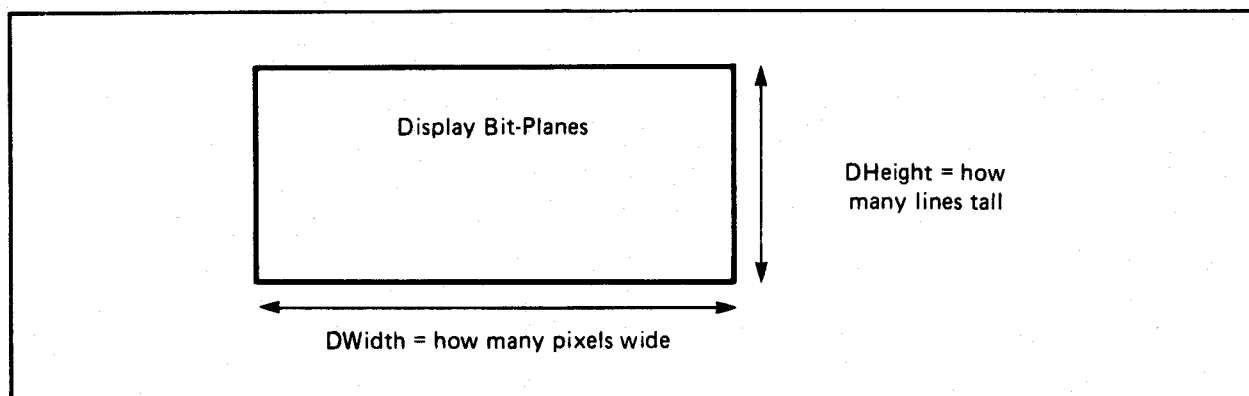


Figure 23-11: Size Definition for a ViewPort

### ViewPort Height

The variable **DHeight** determines how many video lines will be reserved to show the height of this display segment. The size of the actual segment depends on whether you define a non-interlaced or an interlaced display. An interlaced **ViewPort** displays twice as many lines as does a non-interlaced **ViewPort** in the same physical vertical height.

For example, a **View** consisting of two **ViewPorts** might be defined as follows:

- **ViewPort #1** is 150 lines, high-resolution mode (uses the top three-quarters of the display).
- **ViewPort #2** is 49 lines of low-resolution mode (uses the bottom quarter of the display and allows the space for the required blank line between **ViewPorts**).

To set your **ViewPort** to the maximum intuition-supported height, use the following code fragment (double the value for an interlace screen):

```
viewPort.DHeight = GfxBase->NormalDisplayRows;
```

### ViewPort Width

The **DWidth** variable determines how wide, in pixels, the display segment will be. To determine the maximum intuition-supported high-resolution width, use the following fragment:

```
viewPort.DWidth = GfxBase->NormalDisplayColumns;
```

To compute the maximum low-resolution width, divide the **NormalDisplayColumns** by two. You may specify a smaller value of pixels per line to produce a narrower display segment.

Although the system software allows you define low-resolution displays as wide as 362 pixels and high-resolution displays as wide as 724 pixels, you should not exceed the normal values of 320 or 640, respectively. Because of display overscan, many video displays will not be able to show all of a wider display, and sprite display may be affected. If you are using hardware sprites or VSprites with your display, and you specify **ViewPort** widths

exceeding 320 or 640 pixels (for low- or high-resolution, respectively), it is likely that some hardware sprites will not be properly rendered on the screen. These sprites may not be rendered because playfield DMA (direct memory access) takes precedence over sprite DMA when an extra-wide display is produced.

## VIEWPORT COLOR SELECTION

The maximum number of colors that a **ViewPort** can display is determined by the depth of the **BitMap** that the **ViewPort** displays. The depth is specified when the **BitMap** is initialized. See the section below called "Preparing the **BitMap** Structure."

Depth determines the number of bit-planes used to define the colors of the rectangular image you are trying to build (the raster image) and the number of different colors that can be displayed at the same time within a **ViewPort**. For any single pixel, the system can display any one of 4,096 possible colors.

The following table shows depth values and the corresponding number of possible colors for each value.

Table 23-1: Depth Values and Number of Colors in the ViewPort

Colors	Depth Value	
2	1	
4	2	
8	3	
16	4	(Note 1)
32	5	(Notes 1,2)
16	6	(Note 3)
64	6	(Note 4)
4,096	6	(Notes 1,2,5)

### Notes:

1. Single-playfield mode only - DUALPF *not* specified in **ViewPort Modes**.
2. Low-resolution mode only - HIRES *not* specified in **ViewPort Modes**.
3. Dual Playfield mode - DUALPF specified in **ViewPort Modes**. Up to eight colors (in three planes) for each playfield.
4. Extra-Half-Brite mode - EXTRA\_HALFBRITE specified in **ViewPort Modes**.
5. Hold-And-Modify mode only - HAM specified in **ViewPort Modes**.

The color palette used by a **ViewPort** is specified in a **ColorMap**. See the section called "Preparing the **ColorMap**" for more information.

Depending on whether single- or dual-playfield mode is used, the system will use different color register groupings for interpreting the on-screen colors. The table below details how the depth and the **Modes** variable in the **ViewPort** structure affect the registers the system uses.

Table 23-2: Single-playfield Mode (DUALPF *not* specified in Modes variable)

Depth	Color Registers Used	
1	0,1	
2	0-3	
3	0-7	
4	0-15	
5	0-31	
6	0-31	(if EXTRA_HALFBRITE is specified in Modes)
6	0-16	(if HAM is specified in Modes)

The following table shows the five possible combinations when DUALPF *is* specified in the Modes variable.

Table 23-3: Dual-playfield Mode (DUALPF specified in Modes variable)

Depth (PF-1)	Color Registers	Depth (PF-2)	Color Registers
1	0,1	1	8,9
2	0-3	1	8,9
2	0-3	2	8-11
3	0-7	2	8-11
3	0-7	3	8-15

## VIEWPORT DISPLAY MODES

The system has eight different display modes that you can specify for each **ViewPort**. The eight constants that control the modes are DUALPF, PFBA, HIRES, LACE, HAM, SPRITES, VP\_HIDE and EXTRA\_HALFBRITE. A mode becomes active if you specify the corresponding constant in the Modes variable of the **ViewPort** structure. After you initialize the **ViewPort**, you can combine the constants for the modes you want (binary-or them) and place the result in the Modes variable. (See the section called "Preparing the ViewPort Structure" for more information about initializing a **ViewPort**.)

Modes DUALPF and PFBA are related. DUALPF tells the system to treat the raster specified by this **ViewPort** as the first of two independent and separately controllable playfields. It also modifies the manner in which the pixel colors are selected for this raster.

When PFBA is specified, it indicates that the second playfield has video priority over the first one. Playfield relative priorities can be controlled when the playfield is split into two overlapping regions. Single-playfield and dual-playfield modes are discussed in "Advanced Topics" below.

HIRES tells the system that the raster specified by this **ViewPort** is to be displayed with 640 horizontal pixels rather than 320 horizontal pixels.

LACE tells the system that the raster specified by this **ViewPort** is to be displayed in interlaced mode. If the **ViewPort** is non-interlaced and the **View** is interlaced, the **ViewPort** will be displayed at its specified height and will look only slightly different than it would look when displayed in a non-interlaced **View**. See "Interlaced Mode versus Non-interlaced Mode" below for more information.

HAM tells the system to use "hold-and-modify" mode, a special mode that lets you display up to 4,096 colors on screen at the same time. It is described in the "Advanced Topics" section.

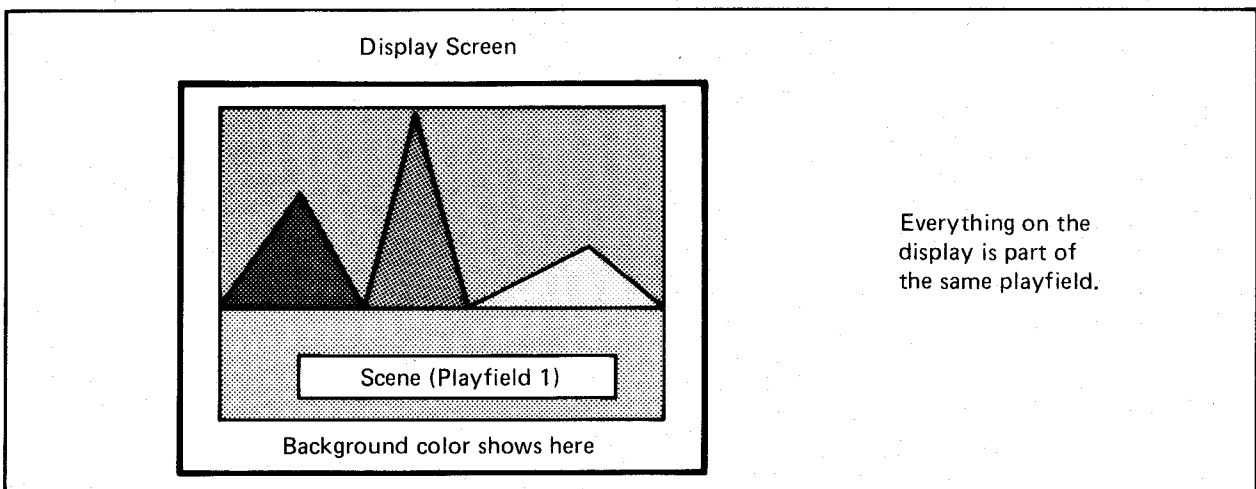
SPRITES tells the system that you are using sprites in this display (either VSprites or Simple Sprites). The system will load color registers for the sprites. See the "Animation" chapter for more information about sprites.

VP\_HIDE tells the system that this **ViewPort** is obscured by other **ViewPorts**. When a **View** is constructed, no display instructions are generated for this **ViewPort**.

EXTRA\_HALFBRITE tells the system to use the Extra-Half-Brite mode, a special mode that allows you display 64 colors on screen at the same time. It is described in the "Advanced Topics" section.

### Single-playfield Mode versus Dual-playfield Mode

When you specify single-playfield mode you are asking that the system treat all bit-planes as part of the definition of a single playfield image. Each of the bit-planes defined as part of this **ViewPort** contributes data bits that determine the color of the pixels in a single playfield.



If you use dual-playfield mode (DUALPF specified in **ViewPort.Modes**), you can define two independent, separately controllable playfield areas.

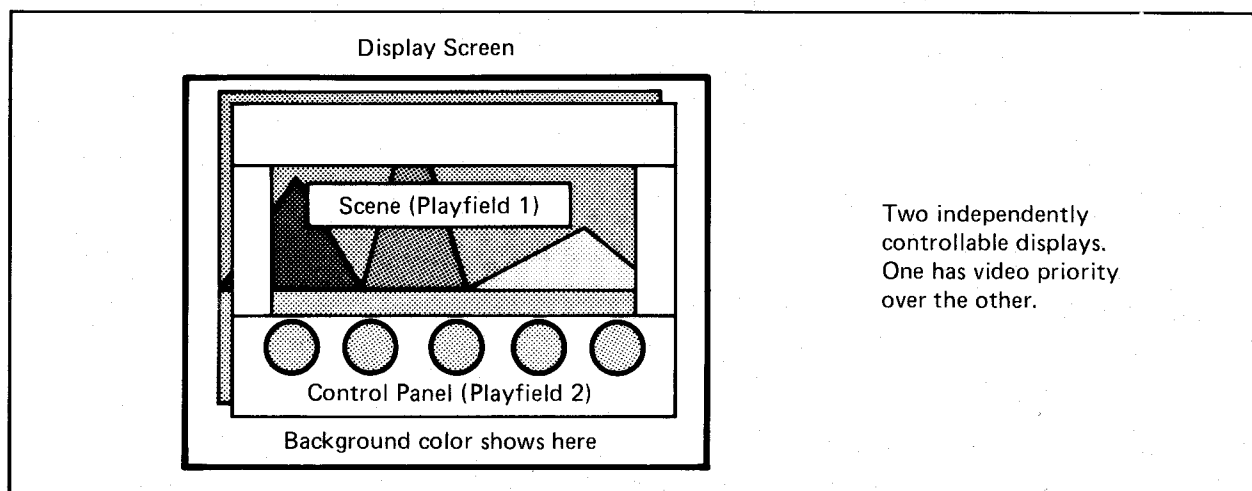


Figure 23-13: A Dual-playfield Display

In the previous figure, PFBA was included in the display mode. If PFBA had not been included, the relative priorities would have been reversed; playfield 2 would have appeared to be behind playfield 1.

### Low-resolution Mode versus High-resolution Mode

In low-resolution mode, horizontal lines of 320 pixels fill most of the ordinary viewing area. The system software lets you define a screen segment width up to 362 pixels in this mode, or you can define a screen segment as narrow as you desire (minimum of 16 pixels). In high-resolution mode (also called "normal" resolution), 640 pixels fill a horizontal line. In this mode you can specify any width from 16 to 724 pixels. Overscan normally limits you to showing only 16 to 320 pixels per line in low-resolution mode or 16 to 640 pixels per line in high-resolution mode. Intuition assumes the nominal 320-pixel or 640-pixel width.

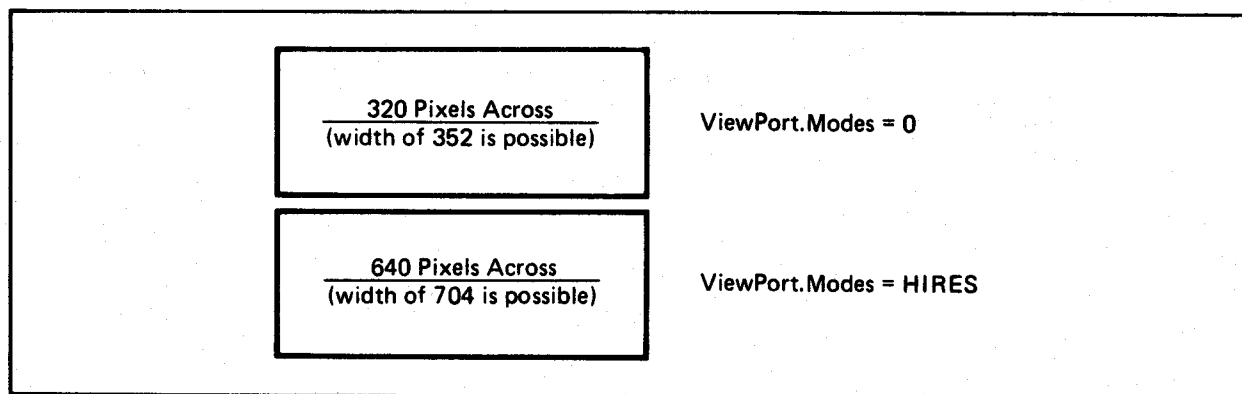


Figure 23-14: How HIRES Affects Width of Pixels

## Interlaced Mode versus Non-interlaced Mode

In interlaced mode, there are twice as many lines available as in non-interlaced mode, providing better vertical resolution in the same display area.

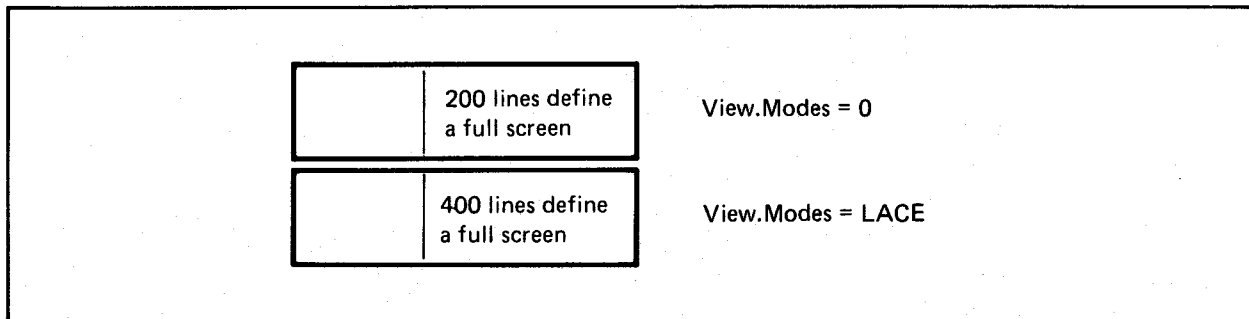


Figure 23-15: How LACE Affects Vertical Resolution

If the **View** structure does not specify **LACE**, and the **ViewPort** specifies **LACE**, you may see only every other line of the **ViewPort** data. If the **View** structure specifies **LACE** and the **ViewPort** is non-interlaced, the same **ViewPort** data will be repeated in both fields. The height of the **ViewPort** display is the height specified in the **ViewPort** structure. If both the **View** and the **ViewPort** are interlaced, the **ViewPort** will be built with double the normal vertical resolution. That means it will need twice as much data space in memory as a non-interlaced picture for this display.

## VIEWPORT DISPLAY MEMORY

The picture you create in memory can be larger than the screen image that can be displayed within your **ViewPort**. This big picture (called a raster and represented by the **BitMap** structure) can have a maximum size of 1,024 by 1,024 pixels. Because a picture this large cannot fit fully on the display, you specify which piece of it to display. Once you have selected the piece to be shown, you can specify where it is to appear on the screen.

The example in the following figure introduces terms that tell the system how to find the display data and how to display it in the **ViewPort**. These terms are **RHeight**, **RWidth**, **RyOffset**, **RxOffset**, **DHeight**, **DWidth**, **DyOffset** and **DxOffset**.

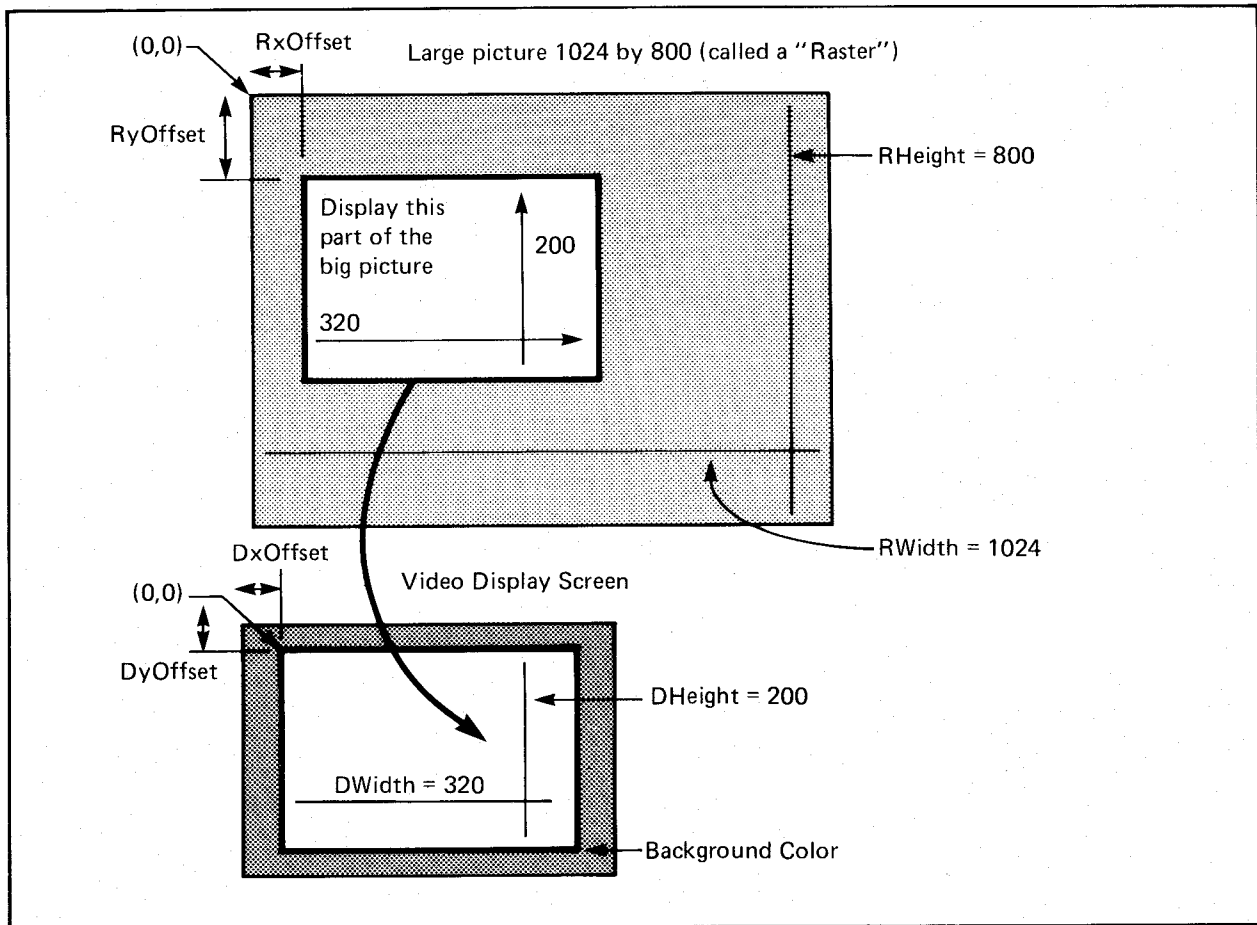


Figure 23-16: ViewPort Data Area Parameters

The terms RHeight and RWidth do not appear in actual system data structures. They refer to the dimensions of the raster and are used here to relate the size of the raster to the size of the display area. RHeight is the number of rows in the raster, and RWidth is the number of bytes per row times 8. The raster shown in the figure is too big to fit entirely in the display area, so you tell the system which pixel of the raster should appear in the upper left corner of the display segment specified by your ViewPort. The variables that control that placement are RyOffset and RxOffset.

To compute RyOffset and RxOffset, you need RHeight, RWidth, DHeight, and DWidth. The DHeight and DWidth variables define the height and width in pixels of the portion of the display that you want to appear in the ViewPort. The example shows a full-screen, low-resolution mode (320-pixel), non-interlaced (200-line) display formed from the larger overall picture.

Normal values for RyOffset and RxOffset are defined by the formulas:

$$0 \leq \text{RyOffset} \leq (\text{RHeight} - \text{DHeight})$$

$$0 \leq \text{RxOffset} \leq (\text{RWidth} - \text{DWidth})$$



Once you have defined the size of the raster and the section of that raster that you wish to display, you need only specify where to put this **ViewPort** on the screen. This is controlled by the variables **DyOffset** and **DxOffset**. A value of 0 for each of these offsets places a normal-sized picture in a centered position at the top, bottom, left and right on the display screen. Possible NTSC values for **DyOffset** range from -23 to +217 (-46 to +434 if **View.Modes** includes LACE), PAL values range from -15 to +267 (-30 to +534 for LACE). Possible values for **DxOffset** range from -18 to +362 (-36 to +724 if **ViewPort.Modes** includes HIRES), when the **View** is in its default, initialized position.

The parameters shown in the figure above are distributed in the following data structures:

- **RasInfo** (information about the raster) contains the variables **RxOffset** and **RyOffset**. It also contains a pointer to the **BitMap** structure.
- **View** (information about the whole display) includes the variables that you use to position the whole display on the screen. The **View** structure contains a **Modes** variable used to determine if the whole display is to be interlaced or non-interlaced. It also contains pointers to its list of **ViewPorts** and pointers to the Copper instructions produced by the system to create the display you have defined.
- **ViewPort** (information about this piece of the display) includes the values **DxOffset** and **DyOffset** that are used to position this portion relative to the overall **View**. The **ViewPort** also contains the variables **DHeight** and **DWidth**, which define the size of this portion; a **Modes** variable; and a pointer to the local **ColorMap**. Each **ViewPort** also contains a pointer to the next **ViewPort**. You create a linked list of **ViewPorts** to define the complete display.
- **BitMap** (information about memory usage) tells the system where to find the display and drawing area memory and shows how this memory space is organized.

You must allocate enough memory for the display you define. The memory you use for the display may be shared with the area control structures used for drawing. This allows you to draw into the same areas that you are currently displaying on the screen.

As an alternative, you can define two **BitMaps**. One of them can be the active structure (that being displayed) and the other can be the inactive structure. If you draw into one **BitMap** while displaying another, the user cannot see the drawing taking place. This is called *double-buffering* of the display. See "Advanced Topics" below for an explanation of the steps required for double-buffering. Double-buffering takes twice as much memory as single-buffering because two full displays are produced.

To determine the amount of required memory for each **ViewPort** for single-buffering, you can use the following formula.

```
bytes_per_ViewPort = Depth * RASSIZE(Width, Height);
```

**RASSIZE** is a system macro attuned to the current design of the system memory allocation for display rasters. See the *graphics/gfx.h* include file for the formula with which **RASSIZE** is calculated.

For example, a 32-color **ViewPort** (depth = 5), 320 pixels wide by 200 lines high uses 40,000 bytes (as of this writing). A 16-color **ViewPort** (depth = 4), 640 pixels wide by 400 lines high uses 128,000 bytes (as of this writing).

## FORMING A BASIC DISPLAY

This section offers an example that shows how to create a single **ViewPort** with a size of 200 lines, in which the area displayed is the same size as the big picture (raster) stored in memory. The example also shows how this **ViewPort** becomes the single display segment of a **View** structure. Following the description of the individual operations, the “Graphics Example Program” section pulls all of the pieces into a complete executable program. Instead of linking these routines to drawing routines, the example allocates memory specifically and only for the display (instead of sharing the memory with the drawing routines) and writes data directly to this memory. This keeps the display and the drawing routines separate for purposes of discussion.

Here are the data structures that you need to define to create a basic display:

```
struct View view;
struct ViewPort viewPort;
struct BitMap bitMap;
struct RasInfo rasInfo;
```

### Opening the Graphics Library

Most of the system routines used here are located in the graphics library. When you compile your programs, you must provide a way to tell the compiler to link your calling sequences into the graphics library. You accomplish this by declaring the variable called **GfxBase**. Then, by opening the graphics library, you provide the value (address of the library) that the system needs for linking with your program. See the “Libraries” chapter for more information.

Here is a typical sequence:

```
struct GfxBase *GfxBase = NULL;

GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
if (GfxBase == NULL)
    cleanExit(RETURN_FAIL);
```

### Preparing the View Structure

The following code section prepares the **View** structure for further use:

```
InitView(&view);    /* Initialize the View. */
```

### Preparing the ViewPort Structure

The following code section prepares the **ViewPort** structure for further use:

```

InitVPort(&viewPort);    /* Initialize the ViewPort. */
viewPort.RasInfo = &rasInfo;
viewPort.DWidth = WIDTH;
viewPort.DHeight = HEIGHT;

/* Init ColorMap. */
/* 2 planes deep, so 4 entries (2 raised to the #_planes power). */
viewPort.ColorMap = GetColorMap(4L);
if (viewPort.ColorMap == NULL)
    cleanExit(RETURN_WARN);

```

The **InitVPort()** routine presets certain default values. The defaults include:

- **Modes** variable set to zero—this means you select a low-resolution display.
- **Next** variable set to zero—no other **ViewPort** is linked to this one. If you want to have multiple **ViewPorts** in a single **View**, you must create the link yourself. The last **ViewPort** in the chain must have a **Next** value of 0.

If you have defined two **ViewPorts**, such as

```

struct ViewPort viewPortA;
struct ViewPort viewPortB;

```

and you want them to both be part of the same display, you must create a link between them, and a **NULL** link at the end of the chain of **ViewPorts**:

```

viewPortA.Next = &viewPortB;    /* Tell first one the address of the second. */
viewPortB.Next = NULL;          /* There are no others after this one. */

```

## Preparing the BitMap Structure

The **BitMap** structure tells the system where to find the display and drawing memory and how this memory space is organized. The following code section prepares a **BitMap** structure, including allocation of memory for the bit-map. This is done with two functions: **InitBitMap()** and **AllocRaster()**. **InitBitMap()** takes four arguments: a pointer to a **BitMap** and the depth, width, and height of the desired bit-map. Once the bit-map is initialized, memory for its bit-planes must be allocated. **AllocRaster()** takes two arguments: width and height. Here is a fragment of code:

```

/* Init BitMap for RasInfo. */
InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);

/*
   Set the plane pointers to NULL so the cleanup
   routine will know if they were used.
*/
for(depth=0; depth<DEPTH; depth++)
    bitMap.Planes[depth] = NULL;

/* Allocate space for BitMap. */
for(depth=0; depth<DEPTH; depth++)
{
    bitMap.Planes[depth] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
    if (bitMap.Planes[depth] == NULL)
        cleanExit(RETURN_WARN);
}

```

This code allocates enough memory to handle the display area for as many bit-planes as the depth you have defined.

## Preparing the RasInfo Structure

The **RasInfo** structure provides information to the system about the location of the **BitMap** as well as the positioning of the display area as a window against a larger drawing area. Use the following steps to prepare the **RasInfo** structure:

```
/* Initialize the RasInfos. */
rasInfo.BitMap = &bitMap; /* Attach the corresponding BitMap. */
rasInfo.RxOffset = 0; /* Align upper left corners of display */
rasInfo.RyOffset = 0; /* with upper left corner of drawing area. */
rasInfo.Next = NULL; /* for a single playfield display, there
                      * is only one RasInfo structure present */
```

The system may be made to reinterpret the **RxOffset** and **RyOffset** values in a **ViewPort**'s **RasInfo** structure by calling **ScrollVPort()** with the address of the **ViewPort**. Changing one or both offsets and calling **ScrollVPort()** has the effect of scrolling the **ViewPort**.

## Preparing the ColorMap Structure

When the **View** is created, Copper instructions are generated to change the current contents of each color register just before the topmost line of a **ViewPort** so that this **ViewPort**'s color registers will be used for interpreting its display.

Here are the steps normally used for initializing a **ColorMap**:

```
/* RGB values for the four colors used. */
#define BLACK 0x000
#define RED 0xf00
#define GREEN 0x0f0
#define BLUE 0x00f

/* Define some colors in an array of UWORDS. */
static UWORD colortable[] = { BLACK, RED, GREEN, BLUE };

/* Init ColorMap. */
/* 2 planes deep, so 4 entries (2 raised to the #_planes power). */
viewPort.ColorMap = GetColorMap(4L);
if (viewPort.ColorMap == NULL)
    cleanExit(RETURN_WARN);

/* Change colors to those in colortable. */
LoadRGB4(&viewPort, colortable, 4);
```

### NOTE

The "4" in the name **LoadRGB4()** refers to the fact that each of the red, green, and blue values in a color table entry consists of four bits. It has nothing to do with the fact that this particular color table contains four entries, which is a result of the choice of **DEPTH=2** for this example. The call **GetRGB4()** returns the RGB value of a single entry of a **ColorMap**. **SetRGB4CM()** allows individual control of the entries in the **ColorMap** before or after linking it into the **ViewPort**.

The **LoadRGB4()** call above could be replaced with the following:

```

register USHORT entry;

/* Operate on the same four ColorMap entries as above. */
for (entry = 0; entry < 4; entry++)
{
    /* Call SetRGB4CM() with the address of the ColorMap, the entry to be
       changed, and the Red, Green, and Blue values to be stored there.
    */
    SetRGB4CM(viewPort.ColorMap, entry,
    /* Extract the three color values from the one colortable entry. */
        ((colortable[entry] & 0x0f00) >> 8),
        ((colortable[entry] & 0x00f0) >> 4),
        (colortable[entry] & 0x000f));
}

```

#### NOTE

Notice above how the four bits for each color are masked out and shifted right to become values from 0 to 15.

It is important that the standard system **ColorMap**-related calls are used to access the **ColorMap** entries. These calls will remain compatible with possible future enhancements of the **ColorMap** structure.

From the section called "Viewport Color Selection," notice that you might need to specify more colors in the color map than you think. If you use a dual-playfield display (covered later in this chapter) with a depth of 1 for each of the two playfields, this means a total of four colors (two for each playfield). However, because playfield 2 uses color registers starting from number 8 on up when in dual-playfield mode, the color map must be initialized to contain at least 10 entries. That is, it must contain entries for colors 0 and 1 (for playfield 1) and color numbers 8 and 9 (for playfield 2). Space for sprite colors must be allocated as well. For Amiga system software version 1.3 and earlier, *when in doubt*, allocate a **ColorMap** with 32 entries, just in case.

### Creating the Display Instructions

Now that you have initialized the system data structures, you can request that the system prepare a set of display instructions for the Copper using these structures as input data. During the one or more blank vertical lines that precede each **Viewport**, the Copper is busy changing the characteristics of the display hardware to match the characteristics you expect for this **Viewport**. This may include a change in display resolution, a change in the colors to be used, or other user-defined modifications to system registers.

Here is the code that creates the display instructions:

```

/* Construct preliminary Copper instruction list. */
MakeVPort( &view, &viewPort );

```

In this line of code, **&view** is the address of the **View** structure and **&viewPort** is the address of the first **Viewport** structure. Using these structures, the system has enough information to build the instruction stream that defines your display.

**MakeVPort()** creates a special set of instructions that controls the appearance of the display. If you are using animation, the graphics animation routines create a special set of instructions to control the hardware sprites and the system color registers. In addition, the advanced user can create special instructions (called user Copper instructions) to change system operations based on the position of the video beam on the screen.

All of these special instructions must be merged together before the system can use them to produce the display you have designed. This is done by the system routine **MrgCop()** (which stands for "Merge Coprocessor Instructions"). Here is a typical call:

```
/* Merge preliminary lists into a real Copper list in the view structure. */
MrgCop( &view );
```

## LOADING AND DISPLAYING THE VIEW

To display the View, you need to load it using `LoadView()` and turn on the direct memory access (DMA). A typical call is shown below.

```
LoadView(&view);
```

where `&view` is the address of the View structure defined in the example above.

Two macros control display DMA: `ON_DISPLAY` and `OFF_DISPLAY` (defined in *graphics/gfxmacros.h*). They simply turn the display DMA control bit in the DMA control register on or off. After you have loaded a new View, you use `ON_DISPLAY` to allow the system DMA to display it on the screen.

If you are drawing to the display area and do not want the user to see intermediate steps in the drawing, you can turn off the display. Because `OFF_DISPLAY` shuts down the display DMA and possibly speeds up other system operations, it can be used to provide additional memory cycles to the blitter or the 68000. The distribution of system DMA, however, allows four-channel sound, disk read/write, and a sixteen-color, low-resolution display (or four-color, high-resolution display) to operate at the same time with no slowdown (7.1 megahertz effective rate) in the operation of the 68000. Using `OFF_DISPLAY` in a multitasking environment may, however, be an unfriendly thing to do to the other running processes. Use `OFF_DISPLAY` with discretion.

## GRAPHICS EXAMPLE PROGRAM

The program below creates and displays a single-playfield display that is 320 pixels wide, 200 lines high, and two bit-planes deep.

```
/* RGBBoxes
   A self-contained example of a single playfield display.
   For Lattice, compile and link with: LC -bl -cfist -L -v -y RGBBoxes.c
*/

#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <graphics/view.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdlib.h>

#define DEPTH 2    /* The number of bitplanes. */
#define WIDTH 320
#define HEIGHT 200

struct GfxBase *GfxBase = NULL;

/* Construct a simple display. */
struct View view;
struct ViewPort viewPort;
struct BitMap bitMap;

/* Pointer for writing to BitMap memory. */
UBYTE *displaymem = NULL;
```

```

/* Pointer to old View so it can be restored. */
struct View *oldview = NULL;

/* RGB values for the four colors used. */
#define BLACK 0x000
#define RED   0xf00
#define GREEN 0x0f0
#define BLUE  0x00f

/*
   Draw a WIDTH/2 by HEIGHT/2 box of color "fillcolor" into the given plane.
*/
VOID drawFilledBox(WORD fillcolor, WORD plane)
{
    UBYTE value;
    WORD boxHeight, boxWidth, width;

    /*
       Divide (WIDTH/2) by eight because each UBYTE that
       is written stuffs eight bits into the BitMap.
    */
    boxWidth = (WIDTH/2)/8;
    boxHeight = HEIGHT/2;

    value = ((fillcolor & (1 << plane)) != 0) ? 0xff : 0x00;

    for( ; boxHeight; boxHeight--)
    {
        for(width=0 ; width < boxWidth; width++)
            *displaymem++ = value;

        displaymem += (bitMap.BytesPerRow - boxWidth);
    }

    /*
       Return user- and system-allocated memory to system memory manager.
    */
    VOID freeMemory(VOID)
    {
        WORD depth;

        /* Free the drawing area. */
        for(depth=0; depth<DEPTH; depth++)
        {
            if (bitMap.Planes[depth])
                FreeRaster(bitMap.Planes[depth], WIDTH, HEIGHT);
        }

        /* Free the color map created by GetColorMap(). */
        if (viewPort.ColorMap)
            FreeColorMap(viewPort.ColorMap);

        /* Free all intermediate Copper lists from ViewPort. */
        FreeVPortCopLists(&viewPort);

        /* Deallocate the hardware Copper list. */
        if (view.LOFCprList)
            FreeCprList(view.LOFCprList);
    }

    /*
       Clean up and exit.
    */
    VOID cleanExit(int exitStatus)
    {
        if (oldview)
        {
            LoadView(oldview);    /* Put back the old View. */
        }
    }

```

```

    /* Wait until the the view is being rendered to free our memory. */
    WaitTOF();
}

freeMemory(); /* Give back what was borrowed. */

if (GfxBase)
    CloseLibrary((struct Library *)GfxBase);

exit(exitStatus);
}

VOID main(VOID)
{
    WORD depth, box;
    /* Offsets in BitMap where boxes will be drawn. */
    static SHORT boxoffsets[] = { 802, 2010, 3218 };
    static UWORD colortable[] = { BLACK, RED, GREEN, BLUE };
    struct RasInfo rasInfo;

    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
    if (GfxBase == NULL)
        cleanExit(RETURN_FAIL);

    /* Example steals screen from Intuition if Intuition is around. */
    oldview = GfxBase->ActiView; /* Save current View to restore later. */

    InitView(&view); /* Initialize the View. */
    InitVPort(&viewPort); /* Initialize the ViewPort. */
    view.ViewPort = &viewPort; /* Link the ViewPort into the View. */

    /* Init BitMap for RasInfo. */
    InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);

    /*
    Set the plane pointers to NULL so the cleanup
    routine will know if they were used.
    */
    for(depth=0; depth<DEPTH; depth++)
        bitMap.Planes[depth] = NULL;

    /* Init RasInfo. */
    rasInfo.BitMap = &bitMap;
    rasInfo.RxOffset = 0;
    rasInfo.RyOffset = 0;
    rasInfo.Next = NULL;

    viewPort.RasInfo = &rasInfo;
    viewPort.DWidth = WIDTH;
    viewPort.DHeight = HEIGHT;

    /* Init ColorMap. */
    /* 2 planes deep, so 4 entries (2 raised to the #_planes power). */
    viewPort.ColorMap = GetColorMap(4L);
    if (viewPort.ColorMap == NULL)
        cleanExit(RETURN_WARN);

    /* Change colors to those in colortable. */
    LoadRGB4(&viewPort, colortable, 4);

    /* Allocate space for BitMap. */
    for (depth=0; depth<DEPTH; depth++)
    {
        bitMap.Planes[depth] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
        if (bitMap.Planes[depth] == NULL)
            cleanExit(RETURN_WARN);
    }

    /* Construct preliminary Copper instruction list. */
    MakeVPort(&view, &viewPort);

```



```

/* Merge preliminary lists into a real Copper list in the view structure. */
MrgCop( &view );

for(depth=0; depth<DEPTH; depth++)
{
    displaymem = (UBYTE *)bitMap.Planes[depth];
    BltClear(displaymem, RASSIZE(WIDTH, HEIGHT), 0);
}

LoadView(&view);

/*
    Now fill some boxes so that user can see something.
    Always draw into both planes to assure true colors.
*/
for (box=1; box<=3; box++)    /* Three boxes; red, green, and blue. */
{
    for (depth=0; depth<DEPTH; depth++)    /* Two planes. */
    {
        displaymem = bitMap.Planes[depth] + boxoffsets[box-1];
        drawFilledBox(box, depth);
    }
}

Delay(10L * TICKS_PER_SECOND);    /* Pause for 10 seconds, */

cleanExit(RETURN_OK);    /* then exit. */

}    /* End of main(). */

```

## Exiting Gracefully

The sample program above provides a way of exiting gracefully, returning to the memory manager all dynamically-allocated memory chunks. Notice the calls to **FreeRaster()** and **FreeColorMap()**. These calls correspond directly to the allocation calls **AllocRaster()** and **GetColorMap()** located in the body of the program. Now look at the calls within **freeMemory()** to **FreeVPortCopLists()** and **FreeCprList()**. When you call **MakeVPort()**, the graphics system dynamically allocates some space to hold intermediate instructions from which a final Copper instruction list is created. When you call **MrgCop()**, these intermediate Copper lists are merged together into the final Copper list, which is then given to the hardware for interpretation. It is this list that provides the stable display on the screen, split into separate **ViewPorts** with their own colors and resolutions and so on.

When your program completes, you must see that it returns all of the memory resources that it used so that those memory areas are again available to the system for reassignment to other projects. Therefore, if you use the routines **MakeVPort()** or **MrgCop()**, you must also arrange to use **FreeCprList()** (pointing to each of those lists in the **View** structure) and **FreeVPortCopLists()** (pointing to the **ViewPort** that is about to be deallocated). If your view is interlaced, you will also have to call **FreeCprList(&view.SHFCprList)** because an interlaced view has a separate Copper list for each of the two fields displayed. Do not confuse **FreeVPortCopLists()** with **FreeCopList()**. The former works on intermediate Copper lists for a specific **ViewPort**, the latter directly on an intermediate Copper list.

As a final caveat, notice that when you do free everything, the memory manager or other programs may immediately change the contents of the freed memory. Therefore, if the Copper is still executing an instruction stream (as a result of a previous **LoadView()**) when you free that memory, the display will malfunction. Once another **View** has been installed via **LoadView()**, do a **WaitTOF()** for the new **View** to begin displaying, and then you can begin freeing up your resources. **WaitTOF()** waits for the vertical blanking period to begin and all vertical blank interrupts to complete before returning to the caller. The routine **WaitBOVP()** "WaitBottomOfViewPort" waits until the vertical beam reaches the bottom of the specified **ViewPort** before returning to the caller.

## Advanced Topics

### CREATING A DUAL-PLAYFIELD DISPLAY

In dual-playfield mode, you have two separately controllable playfields. In this mode, you always define two **RasInfo** data structures. Each of these structures defines one of the playfields. There are seven different ways you can configure a dual-playfield display, because there are five different distributions of the bit-planes which the system hardware allows. The following table shows these distributions.

Table 23-4: Bit-Plane Assignment in Dual-playfield Mode

Number of Bit-planes	Playfield 1 Depth	Playfield 2 Depth
0	0	0
1	1	0
2	1	1
3	2	1
4	2	2
5	3	2
6	3	3

Recall that if PFBA is included in the **ViewPort Modes** variable, you can swap playfield priority and display playfield 2 in front of playfield 1. In this way, you can get more bit-planes in the background playfield than you have in the foreground playfield. If you create a display with multiple **ViewPorts**, only for this **ViewPort** will the playfield priority be changed.

Playfield 1 is defined by the first of the two **RasInfo** structures. Playfield 2 is defined by the second of the two **RasInfo** structures.

When you call **MakeVPort()**, you use parameters as follows:

```
MakeVPort( &view, &viewPort );
```

The **ViewPort Modes** variable must include the DUALPF bit. This tells the graphics system that there are two **RasInfo** structures to be used.

In summary, to create a dual-playfield display you must do the following things:

- Allocate one **View** structure
- Allocate two **BitMap** structures
- Allocate two **RasInfo** structures (linked together), each pointing to different **BitMaps**

- Allocate one **ViewPort** structure
- Set up a pointer in the **ViewPort** structure to the playfield 1 **RasInfo**
- Initialize each **BitMap** structure to describe one playfield, using one of the permissible bit-plane distributions shown in the above table and allocate memory for the bit-planes themselves.

#### NOTE

**BitMap 1** and **BitMap 2** need *not* be the same width and height.

- Initialize the **ViewPort** structure
- Set the DUALPF (and possibly the PFBA) bit in the **ViewPort Modes** variable
- Call **MakeVPort()**
- Call **MrgCop()**
- Call **LoadView()** to display the newly created **ViewPort**.

For display purposes, each of the two **BitMaps** is assigned to a separate playfield display.

To draw separately into the **BitMaps**, you must also assign these **BitMaps** to two separate **RastPorts**. The section called “Initializing the **RastPort**” shows you how to use a **RastPort** data structure to control your drawing routines.

### CREATING A DOUBLE-BUFFERED DISPLAY

To produce smooth animation or other such effects, it is occasionally necessary to double-buffer your display. To prevent the user from seeing your graphics rendering while it is in progress, you will want to draw into one memory area while actually displaying a different area.

Here are two methods of creating and displaying a double-buffered display. The first is more complicated, but uses less memory than the second.

#### Method 1:

This method consists of creating two separate display areas and two sets of pointers to those areas for a single **View**.

- Allocate two **BitMap** structures
- Allocate one **RasInfo** structure
- Allocate one **ViewPort** structure
- Allocate one **View** structure
- Initialize each **BitMap** structure to describe one drawing area and allocate memory for the bit-planes themselves

- Create a pointer for each **BitMap**
- Create a pointer for the **View** long-frame Copper list (**LOFCprList**) and short-frame Copper list (**SHFCprList**) for each of two alternate display fields. The **SHFCprList** is for interlaced displays only.
- Initialize the **RasInfo** structure, setting the **BitMap** pointer to point to one of the two **BitMaps** you have created
- Call **MakeVPort()**
- Call **MrgCop()**
- Call **LoadView()**

When you call **MrgCop()**, the system uses all of the information you have provided in the various data structures to create a list of instructions for the Copper to execute. This list tells the Copper how to split the display and how to specify colors for the various portions of the display. When the steps shown above have been completed, the system will have allocated memory for a long-frame (LOF) Copper list (and for interlaced displays a short-frame (SHF) Copper list) and will have set the pointer(s) called **LOFCprList** (and **SHFCprList**) in the **View** structure. The long-frame Copper list is normally used for all non-interlaced displays, and the short-frame Copper list is used only when interlaced mode is turned on. The pointers point to the two sets of Copper instructions.

The **LOFCprList** and **SHFCprList** pointers are initialized when **MrgCop()** is called. The instruction stream referenced by these pointers includes references to the first **BitMap**.

You must now do the following:

- Save the current values in back-up pointers and set the values of **LOFCprList** and **SHFCprList** in the **View** structure to zero. When you next perform **MrgCop()**, the system automatically allocates another memory area to hold a new list of instructions for the Copper.
- Install the pointer to the other **BitMap** structure in the **RasInfo** structure before your call to **MakeVPort()**, and then call **MakeVPort()** and **MrgCop()**.

Now you have created two sets of instruction streams for the Copper, one of which you have saved in a pair of pointer variables. The other has been newly created and is in the **View** structure. You can save this new set of pointers as well, swapping in the set that you want to use for display, while drawing into the **BitMap** that is not on the display. Remember that you will have to call **FreeCprList()** on both sets of Copper lists when you have finished.

Method 2:

A simpler, but more memory-hungry, method is to create two complete **Views** and switch back and forth between them with **LoadView()** and **WaitTOF()**.

## EXTRA-HALF-BRITE MODE

In the Extra-Half-Brite mode you can create a single playfield display in which 64 colors can be displayed at a time. This requires that your **ViewPort** be defined using six bit-planes and that you specify **EXTRA\_HALFBRITE** in the **ViewPort Modes** variable.

If you draw using color numbers 0 through 31, the pixel you draw will be the color specified in that particular system color register. If you draw from a color number from 32 to 63, the color displayed is half the intensity of corresponding color numbers from 0 to 31. For example, if color register 0 is set to 0xFFFF (white), then color number 32 would be 0x777 (grey).

When setting up the color palette for the Extra-Half-Brite display, you can only specify values for registers 0 to 31.

The Extra-Half-Brite mode uses all six bit-planes. The color register (0 through 31) is obtained from the bit combinations from planes 5 to 1, in that order of significance. Plane 6 is used to determine whether the full intensity (bit value 0) color or half-intensity (bit value 1) color is to be displayed.

## **HOLD-AND-MODIFY MODE**

In hold-and-modify mode you can create a single-playfield display in which 4,096 different colors can be displayed simultaneously. This requires that your **ViewPort** be defined using six bit-planes and that you specify HAM in the **ViewPort Modes** variable.

When you draw into the **BitMap** associated with this **ViewPort**, you can choose one of four different ways of drawing into the **BitMap**. (Drawing into a **BitMap** is shown in the next section, "Drawing Routines.") If you draw using color numbers 0 to 15, the pixel you draw will appear in the color specified in that particular system color register. If you draw with any other color value from 16 to 63, the color displayed depends on the color of the pixel that is to the immediate left of this pixel on the screen. For example, hold constant the contents of the red and the green parts of the previously produced color, and take the rest of the bits of this new pixel's color register number as the new contents for the blue part of the color. Hold-and-modify means hold part and modify part of the preceding defined pixel's color.

### **NOTE**

A particular hold-and-modify pixel can only change one of the three color values at a time. Thus, the effect has a limited control.

In hold-and-modify mode, you use all six bit-planes. Planes 5 and 6 are used to modify the way bits from planes 1 through 4 are treated, as follows:

- If the 6-5 bit combination from planes 6 and 5 for any given pixel is 00, normal color selection procedure is followed. Thus, the bit combinations from planes 4 to 1, in that order of significance, are used to choose one of 16 color registers (registers 0 through 15).

**If only five bit-planes are used, the data from the sixth plane is automatically supplied with the value as 0.**

- If the 6-5 bit combination is 01, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit combinations from planes 4 through 1 are used to replace the four "blue" bits in the pixel color without changing the value in any color register.
- If the 6-5 bit combination is 10, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit combinations from planes 4 through 1 are used to replace the four "red" bits.
- If the 6-5 bit combination is 11, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit combinations from planes 4 through 1 are used to replace the four "green" bits.

- At the leftmost edge of each line, hold-and-modify begins with the background color. The color choice does *not* carry over from the preceding line.

## Drawing Routines

Most of the graphics drawing routines require information about how the drawing is to take place. For this reason, the graphics support routines provide a data structure called a **RastPort**, which contains information essential to the graphics drawing functions. In using most of the drawing functions, you must pass them a pointer to your **RastPort** structure. Associated with the **RastPort** is another data structure called a **BitMap**, which contains a description of the organization of the data in the drawing area.

### INITIALIZING A BITMAP STRUCTURE

The **RastPort** contains information for controlling the drawing. In order to use the graphics, you also need to tell the system the memory area location where the drawing will occur. You do this by initializing a **BitMap** structure, defining the characteristics of the drawing area, as shown in the following example. This was already shown in the section called “Forming a Basic Display,” but it is repeated here because it relates to drawing as well as to display routines. You need not necessarily use the same **BitMap** for both the drawing and the display.

```
#define DEPTH 2      /* Two planes deep. */
#define WIDTH 320    /* Width in pixels. */
#define HEIGHT 200   /* Height in lines. */

struct BitMap bitMap;

/* Initialize the BitMap. */
InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);
```

### INITIALIZING A RASTPORT STRUCTURE

Before you can use a **RastPort** for drawing, you must initialize it. Here is a sample initialization sequence:

```
struct BitMap bitMap;
struct RastPort rastPort;

/* Initialize the RastPorts and link the BitMaps to them. */
InitRastPort(&rastPort);
rastPort.BitMap = &bitMap;
```

### NOTE

You cannot link the bit-map in until after the **RastPort** has been initialized.

The **RastPort** data structure can be found in the include files *graphics/rastport.h* and *graphics/rastport.i*. It contains the following information:

- Area-filling information
- Graphics elements information for animation

- A write mask
- Drawing pens
- Drawing modes
- Patterns
- Text attributes and font information
- Current pen position
- Some graphics private data
- A pointer for user extensions

The following sections explain each of the items in the **RastPort** structure.

### Area-fill Information

Two structures in the **RastPort**—**AreaInfo** and **TmpRas**—define certain information for area filling operations. The **AreaInfo** pointer is initialized by a call to the routine **InitArea()**.

```
#define AREA_SIZE 200

register USHORT i;
WORD areaBuffer[AREA_SIZE];
struct AreaInfo areaInfo;

/* Clear areaBuffer before calling InitArea(). */
for (i=0; i<AREA_SIZE; i++)
    areaBuffer[i] = 0;

InitArea(&areaInfo, areaBuffer, (AREA_SIZE*2)/5);
```

The area buffer *must* start on a *word* boundary. That is why the sample declaration shows **areaBuffer** as composed of unsigned words (200), rather than unsigned bytes (400). It still reserves the same amount of space, but aligns the data space correctly.

To use area fill, you must first provide a work space in memory for the system to store the list of points that define your area. You must allow a storage space of 5 bytes per vertex. To create the areas in the work space, you use the functions **AreaMove()**, **AreaDraw()**, and **AreaEnd()**.

Typically, you prepare the **RastPort** for area-filling by following the steps in the code fragment above and then linking your **AreaInfo** into the **rastPort**, as so:

```
rastPort->AreaInfo = &areaInfo;
```

In addition to the **AreaInfo** structure in the **RastPort**, you must also provide the system with some work space to build the object whose vertices you are going to define. This requires that you initialize a **TmpRas** structure, then point to that structure for your **RastPort** to use.

Here is a code fragment that builds and initializes a **TmpRas**. First the **TmpRas** structure is initialized (via

**InitTmpRas()** then it is linked into the **RastPort** structure.

#### NOTE

The area to which **TmpRas.RasPtr** points must be at least as large as the area (width times height) of the largest rectangular region you plan to fill. Typically, you allocate a space as large as a single bit-plane (usually 320 by 200 bits for low-resolution mode, 640 by 200 bits for high-resolution mode).

```
UWORD height, width;
PLANEPTR planePtr = NULL;
struct RastPort *rastPort = window->RPort;
struct TmpRas tmpRas;

height = GfxBase->NormalDisplayRows;
width = GfxBase->NormalDisplayColumns;

planePtr = AllocRaster(width, height);
if (planePtr)
{
    InitTmpRas(&tmpRas, planePtr, RASSIZE(width, height));
    rastPort->TmpRas = &tmpRas;
    /* The TmpRas can be used now. */
    FreeRaster(planePtr, width, height);    /* Free raster when done. */
}
```

When you use functions that dynamically allocate memory from the system, you must remember to return these memory blocks to the system before your program exits. See the description of **FreeRaster()** in *The Amiga ROM Kernel Manual: Includes and Autodocs*.

#### Graphics Element Pointer

The graphics element pointer in the **RastPort** structure is called **GelsInfo**. If you are doing graphics animation using the GELS system, this pointer must refer to a properly initialized **GelsInfo** structure. See the chapter on "Animation" for more information.

#### Write Mask

The write mask is a **RastPort** variable that determines which of the bit-planes are currently writable. For most applications, this variable contains all 1's (hex FF). This means that all bit-planes defined in the **BitMap** are affected by a graphics writing operation. You can selectively disable one or more bit-planes by simply specifying a 0 bit in that specific position in the control byte. For example:

```
#include <graphics/gfxmacros.h>

SetWrMsk(&rastPort, 0xFB);    /* disable bit-plane 2 */
```

A useful application for the Mask is to set or clear plane 6 while in the Extra-Half-Brite display mode to create shadow effects. For example:

```
SetWrMsk(&rastPort, 0xE0);    /* Disable planes 1 through 5. */
SetAPen(&rastPort, 0);        /* Clear the Extra-Half-Brite bit */
RectFill(&rastPort, 20, 20, 40, 30); /* in the old rectangle. */
SetAPen(&rastPort, 32);        /* Set the Extra-Half-Brite bit */
RectFill(&rastPort, 30, 25, 50, 35); /* in the new rectangle. */
SetWrMsk(&rastPort, 0xFF);    /* Re-enable all planes. */
```



## Drawing Pens

The Amiga has three different drawing “pens” associated with the graphics drawing routines. These are:

- **FgPen**—the foreground or primary drawing pen. For historical reasons, it is also called the A-Pen.
- **BgPen**—the background or secondary drawing pen. For historical reasons, it is also called the B-Pen.
- **AOIPen**—the area outline pen. For historical reasons, it is also called the O-Pen.

A drawing pen variable in the **RastPort** contains the current value (range 0-255) for a particular color choice. This value represents a color register number whose contents are to be used in rendering a particular type of image. The effect of the pen value is dependent upon the drawing mode and can be influenced by the pattern variables and the write mask as described below. Always use the system calls (e.g. **SetAPen()**) to set the different pens, never store values directly into the pen fields of the **RastPort**.

### NOTE

The Amiga 500/1000/2000 contains only 32 color registers. Any range beyond that repeats the colors in 0-31. For example, pen numbers 32-63 refer to the colors in registers 0-31 (except when you are using Extra-Half-Brite mode).

The drawing routines support **BitMaps** up to eight planes deep, allowing for future expansion of the hardware.

The color in **FgPen** is used as the primary drawing color for rendering lines and areas. This pen is used when the drawing mode is JAM1 (see the next section for drawing modes). JAM1 specifies that only one color is to be “jammed” into the drawing area.

You establish the color for **FgPen** using the statement:

```
SetAPen(&rastPort, newcolor);
```

The color in **BgPen** is used as the secondary drawing color for rendering lines and areas. If you specify that the drawing mode is JAM2 (jamming two colors) and a pattern is being drawn, the primary drawing color (**FgPen**) is used where there are 1s in the pattern. The secondary drawing color (**BgPen**) is used where there are 0s in the pattern.

You establish the drawing color for **BgPen** using the statement:

```
SetBPen(&rastPort, newcolor);
```

The area outline pen **AOIPen** is used in two applications: area fill and flood fill. (See “Area Fill Operations” below.) In area fill, you can specify that an area, once filled, can be outlined in this **AOIPen** color. In flood fill (in one of its operating modes) you can fill until the flood-filler hits a pixel of the color specified in this pen variable.

You establish the drawing color for **AOIPen** using the statement:

```
SetOPen(&rastPort, newcolor);
```

## Drawing Modes

Four drawing modes may be specified:

- JAM1** Whenever you execute a graphics drawing command, one color is jammed into the target drawing area. You use only the primary drawing pen color, and for each pixel drawn, you *replace* the color at that location with the **FgPen** color.
- JAM2** Whenever you execute a graphics drawing command, two colors are jammed into the target drawing area. This mode tells the system that the pattern variables (both line pattern and area pattern—see the next section) are to be used for the drawing. Wherever there is a 1 bit in the pattern variable, the **FgPen** color replaces the color of the pixel at the drawing position. Wherever there is a 0 bit in the pattern variable, the **BgPen** color is used.

### COMPLEMENT

For each 1 bit in the the pattern, the corresponding bit in the target area is complemented—that is, its state is reversed. As with all other drawing modes, the write mask can be used to protect specific bit-planes from being modified. Complement mode is often used for drawing and then erasing lines.

### INVERSVID

This is the drawing mode used primarily for text. If the drawing mode is (JAM1|INVERSVID), the text appears as a transparent letter surrounded by the **FgPen** color. If the drawing mode is (JAM2|INVERSVID), the text appears as in (JAM1|INVERSVID) except that the **BgPen** color is used to draw the text character itself. In this mode, the roles of **FgPen** and **BgPen** are effectively reversed.

You set the drawing modes using the statement:

```
SetDrMd(&rastPort, newmode);
```

## Patterns

The **RastPort** data structure provides two different pattern variables that it uses during the various drawing functions: a line pattern and an area pattern. The line pattern is 16 bits wide and is applied to all lines. When you initialize a **RastPort**, this line pattern value is set to all 1s (hex FFFF), so that solid lines are drawn. You can also set this pattern to other values to draw dotted lines if you wish. For example, you can establish a dotted line pattern with the statement:

```
SetDrPt(&rastPort, 0xCCCC);
```

where 0xCCCC is a bit-pattern, 1100110011001100, to be applied to all lines drawn. If you draw multiple, connected lines, the pattern cleanly connects all the points.

The area pattern is 16 bits wide and its height is some power of two. This means that you can define patterns in heights of 1, 2, 4, 8, 16, and so on. To tell the system how large a pattern you are providing, include this statement:

```
SetAfPt(&rastPort, &areaPattern, power_of_two);
```

where **&areaPattern** is the address of the first word of the area pattern and **power\_of\_two** specifies how many words are in the pattern. For example:

```
USHORT ditherData[] =
{
    0x5555, 0xAAAA
};

SetAfPt(&rastPort, ditherData, 1);
```

This example produces a small cross-hatch pattern, useful for shading. Because **power\_of\_two** is set to 1, the pattern is 2 to the 1st, or 2, rows high.

To clear the fill pattern, use:

```
SetAfPt(&rastPort, NULL, 0);
```

### Pattern Positioning

The pattern is always positioned with respect to the upper left corner of the **RastPort** drawing area (the 0,0 coordinate). If you draw two rectangles whose edges are adjacent, the pattern will be continuous across the rectangle boundaries.

### Multicolored Patterns

The last example above produces a two-color pattern with one color where there are 1s and the other color where there are 0s in the pattern. A special mode allows you to develop a pattern having up to 256 colors. To create this effect, specify **power\_of\_two** as a negative value instead of a positive value.

The following initialization establishes an 8-color checkerboard pattern where each square in the checkerboard has a different color. The checkerboard is 2 squares wide by 4 squares high.

```
USHORT areaPattern[3][8] =
{
    /* plane 0 pattern */
    {
        0x0000, 0x0000, 0xffff, 0xffff,
        0x0000, 0x0000, 0xffff, 0xffff
    },
    /* plane 1 pattern */
    {
        0x0000, 0x0000, 0x0000, 0x0000,
        0xffff, 0xffff, 0xffff, 0xffff
    },
    /* plane 2 pattern */
    {
        0xff00, 0xff00, 0xff00, 0xff00,
        0xff00, 0xff00, 0xff00, 0xff00
    }
};

SetAfPt(&rastPort, &areaPattern, -3);

/* when doing this, it is best to set three other parameters as follows: */
SetAPen(&rastPort, 255);
SetBPen(&rastPort, 0);
SetDrMd(&rastPort, JAM2);
```

If you use this multicolored pattern mode, you must provide as many planes of pattern data as there are planes in your **BitMap**.

### Current Pen Position

The graphics drawing routines keep the current position of the drawing pen in the variables **cp\_x** and **cp\_y**, for the horizontal and vertical positions, respectively. The coordinate location 0,0 is in the upper left corner of the drawing area. The x value increases proceeding to the right; the y value increases proceeding toward the bottom of the drawing area.

### Pen Size

The variables **PenWidth** and **PenHeight** are not currently implemented.

### Text Attributes

Text attributes and font information are set by calls to the font routines. These are covered separately in the chapter on "Text."

## USING THE GRAPHICS DRAWING ROUTINES

This section shows you how to use the Amiga drawing routines. All of these routines work either on their own or with the windowing system and layer library. For details about using the layer library and windows, see the chapters on "layers" (clipping) and "Intuition" (windows).

### NOTE

The graphics.library rendering and data movement routines generally wait to get access to the blitter, start their blit, and then exit. Therefore, you must **WaitBlit()** after a graphics rendering or data movement call if you intend to immediately deallocate, examine, or perform order-dependent processor operations on the memory used in the call.

As you read this section, keep in mind that to use the drawing routines, you need to pass them a pointer to a **RastPort**. You can define the **RastPort** directly, as shown in the sample program segments in preceding sections, or you can get a **RastPort** from your **Window** structure using code like the following:

```
struct Window *window;
struct RastPort *rastPort;

window = OpenWindow(&newWindow);
if (window)
    rastPort = window->RPort;
```

You can also get the **RastPort** from the layer structure, if you are not using Intuition.

## Drawing Individual Pixels

You can set a specific pixel to a desired color by using a statement like this:

```
LONG result;  
result = WritePixel(&rastPort, x, y);
```

**WritePixel()** uses the primary drawing pen and changes the pixel at that x,y position to the desired color if the x,y coordinate falls within the boundaries of the **RastPort**. A value of 0 is returned if the write was successful; a value of -1 is returned if x,y was outside the range of the **RastPort**. **WritePixel()** waits for the blitter to complete its operation between each of the blits.

## Reading Individual Pixels

You can determine the color of a specific pixel with a statement like this:

```
LONG result;  
result = ReadPixel(&rastPort, x, y);
```

**ReadPixel()** returns the value of the pixel color selector (from 0 to 255) at the specified x,y location. If the coordinates you specify are outside the range of your **RastPort**, this function returns a value of -1.

## Drawing Ellipses

Two functions are associated with drawing ellipses: **DrawCircle()** and **DrawEllipse()**. **DrawCircle()** (a macro that calls **DrawEllipse()**) will draw a circle from the specified center point using the specified radius. This function is executed by the statement:

```
DrawCircle(&rastPort, center_x, center_y, radius);
```

**DrawEllipse()** draws an ellipse with the specified radii from the specified center point. The function call is:

```
DrawEllipse(&rastPort, center_x, center_y, horiz_r, vert_r);
```

Neither function performs clipping on a non-layered **RastPort**.

## Drawing Lines

Two functions are associated with line drawing: **Move()** and **Draw()**. **Move()** simply moves the cursor to a new position. It is like picking up a drawing pen and placing it at a new location. This function is executed by the statement:

```
Move(&rastPort, x, y);
```

**Draw()** draws a line from the current x,y position to a new x,y position specified in the statement itself. The drawing pen is left at the new position. This is done by the statement:

```
Draw(&rastPort, x, y);
```

**Draw()** uses the pen color specified for **FgPen**. Here is a sample sequence that draws a line from location (0,0) to (100,50).

```
SetAPen(&rastPort, COLOR1);    /* Set A pen color. */
Move(&rastPort, 0, 0);          /* Move to this location. */
Draw(&rastPort, 100,50);        /* Draw to a this location. */
```

### CAUTION

If you attempt to draw a line outside the bounds of the **BitMap**, using the basic initialized **RastPort**, you may crash the system. You must either do your own software clipping to assure that the line is in range, or use the layer library. Software clipping means that you need to determine if the line will fall outside your **BitMap** *before* you draw it, and render only the part which falls inside the **BitMap**.

### Drawing Patterned Lines

To turn the example above into a patterned line draw, simply set a drawing pattern, such as:

```
SetDrPt (&rastPort, 0xAAAA);
```

Now all lines drawn appear as dotted lines (0xAAAA = 1010101010101010 in binary). To resume drawing solid lines, execute the statement:

```
SetDrPt (&rastPort, ~0);
```

Because ~0 is defined as 1111111111111111 in binary.

### Drawing Multiple Lines with a Single Command

You can use multiple **Draw()** statements to draw connected line figures. If the shapes are all definable as interconnected, continuous lines, you can use a simpler function, called **PolyDraw()**. **PolyDraw()** takes a set of line endpoints and draws a shape using these points. You call **PolyDraw()** with the statement:

```
PolyDraw(&rastPort, count, arraypointer);
```

**PolyDraw()** reads the array of points and draws a line from the first pair of coordinates to the second, then a connecting line to each succeeding pair in the array until **count** points have been connected. This function uses the current drawing mode, pens, line pattern, and write mask specified in the target **RastPort**; for example:

```
SHORT linearray[] =
{
    3, 3,
    15, 3,
    15, 15,
    3, 15,
    3, 3
};
```

```
PolyDraw(&rastPort, 5, &linearray[0]);
```

draws a rectangle, using the five defined pairs of x,y coordinates.

## Area-fill Operations

Assuming that you have properly initialized your **RastPort** structure to include a properly initialized **AreaInfo**, you can perform area fill by using the functions described in this section.

**AreaMove()** tells the system to begin a new polygon, closing off any other polygon that may already be in process by connecting the end-point of the previous polygon to its starting point. **AreaMove()** is executed with the statement:

```
LONG result;
result = AreaMove(&rastPort, x, y);
```

**AreaMove()** returns 0 if successful, -1 if there was no more space left in the vector list. **AreaDraw()** tells the system to add a new vertex to a list that it is building. No drawing takes place until **AreaEnd()** is executed. **AreaDraw** is executed with the statement:

```
LONG result;
result = AreaDraw(&rastPort, x, y);
```

**AreaDraw()** returns 0 if successful, -1 if there was no more space left in the vector list. **AreaEnd()** tells the system to draw all of the defined shapes and fill them. When this function is executed, it obeys the drawing mode and uses the line pattern and area pattern specified in your **RastPort** to render the objects you have defined.

## NOTE

To fill an area, you do not have to **AreaDraw()** back to the first point before calling **AreaEnd()**. **AreaEnd()** automatically closes the polygon. **AreaEnd()** is executed with the following statement:

```
LONG result;
result = AreaEnd(&rastPort);
```

**AreaEnd()** returns 0 if successful, -1 if there was an error.

Here is an example routine that demonstrates **AreaInfo** initialization. It draws a pair of disconnected triangles, using the currently defined **FgPen**, **BgPen**, **AOIPen**, **DrawMode**, **LinePtrn**, and **AreaPtrn**:

```
/* AreaInfoExample
   Insert this routine into the "wrapper" code at the end of the TEXT chapter.
*/

#define AREA_SIZE 200

BOOL example(struct Window *window)
{
    register USHORT i;
    WORD areabuffer[AREA_SIZE];
    UWORD height, width;
    PLANEPTR planePtr = NULL;
    struct RastPort *rastPort = window->RPort;
    struct TmpRas tmpRas;
    struct AreaInfo areaInfo;

    height = GfxBase->NormalDisplayRows;
    width = GfxBase->NormalDisplayColumns;
```

```

planePtr = AllocRaster(width, height);
if (planePtr)
{
    for (i=0; i<AREA_SIZE; i++)
        areabuffer[i] = 0;

    InitArea(&areaInfo, areabuffer, (AREA_SIZE*2)/5);
    rastPort->AreaInfo = &areaInfo;

    InitTmpRas(&tmpRas, planePtr, RASSIZE(width, height));
    rastPort->TmpRas = &tmpRas;

    /* Area routines need a temporary raster buffer at least as large as the
     * largest object to be drawn. If a single task uses multiple RastPorts,
     * it is sometimes possible to share the same TmpRas structure among
     * multiple RastPorts. Multiple tasks, however, cannot share a TmpRas,
     * as each task won't know when another task has a drawing partially
     * completed.
     */

    AreaMove(rastPort, 0, 0);
    AreaDraw(rastPort, 0, 100);
    AreaDraw(rastPort, 100, 100);

    AreaMove(rastPort, 50, 10);
    AreaDraw(rastPort, 50, 50);
    AreaDraw(rastPort, 100, 50);

    AreaEnd (rastPort);

    FreeRaster(planePtr, width, height);
}
return(WAIT_FOR_CLOSE);
}

```

If you had executed the statement “SetOPen( &rastPort, 3)” in the area-fill example, then the areas that you had defined would have been outlined in pen color 3. To turn off the outline function, you have to set the **RastPort Flags** variable back to 0 by:

```

#include "graphics/gfxmacros.h"

struct RastPort *rastPort;

BNDRYOFF(rastPort);

```

Otherwise, every subsequent area-fill or rectangle-fill operation will outline their rendering with the outline pen.

Two functions are associated with drawing filled ellipses: **AreaCircle()** and **AreaEllipse()**. **AreaCircle()** (a macro that calls **AreaEllipse()**) will draw a circle from the specified center point using the specified radius. This function is executed by the statement:

```
AreaCircle(&rastPort, center_x, center_y, radius);
```

**AreaEllipse()** draws a filled ellipse with the specified radii from the specified center point. The function call is:

```
AreaEllipse(&rastPort, center_x, center_y, horiz_r, vert_r);
```

Outlining with **SetOPen()** is not currently supported by the **AreaCircle()** and **AreaEllipse()** routines.

### CAUTION

If you attempt to fill an area outside the bounds of the **BitMap**, using the basic initialized **RastPort**, it may crash the system. You must either do your own software clipping to assure that the area is in range, or use the layer library.



## Flood-fill Operations

Flood fill is a technique for filling an arbitrary shape with a color. The Amiga flood-fill routines can use a plain color or do the fill using a combination of the drawing mode, **FgPen**, **BgPen**, and the area pattern.

Flood-fill requires a **TmpRas** at least as large as the **RastPort** in which the flood-fill is being done. This is to ensure that even if the flood-filling "leaks", it will not flow outside the **TmpRas** and corrupt another task's memory.

There are two different modes for flood fill:

- In *outline mode* you specify an x,y coordinate, and from that point the system searches outward in all directions for a pixel whose color is the same as that specified in the area outline pen. All horizontally or vertically adjacent pixels *not* of that color are filled with a colored pattern or plain color. The fill stops at the outline color. Outline mode is selected when the **mode** variable is a 0.
- In *color mode* you specify an x,y coordinate, and whatever pixel color is found at that position defines the area to be filled. The system searches for all horizontally or vertically adjacent pixels whose color is the same as this one and replaces them with the colored pattern or plain color. Color mode is selected when the **mode** variable is a 1.

You use the **Flood()** routine for flood fill. The syntax for this routine follows.

```
Flood(&rastPort, mode, x, y);
```

where

**&rastPort**

is a pointer to the **RastPort**

**mode**

tells how to do the fill

**x,y**

is the starting coordinate in the **BitMap**

The following sample program fragment creates and then flood-fills a triangular region. The overall effect is exactly the same as shown in the preceding area-fill example above, except that flood-fill is slightly slower than area-fill. Mode 0 (fill to a pixel that has the color of the outline pen) is used in the example.

```
SHORT oldAPen;
struct RastPort *rastPort = Window->RPort;

oldAPen = rastPort->FgPen;
SetAPen(rastPort, rastPort->AOIPen);
/* using mode 0 */
/* triangular shape */
Move(rastPort, 0, 0);
Draw(rastPort, 0, 100);
Draw(rastPort, 100, 100);
Draw(rastPort, 0, 0); /* close it */

SetAPen(rastPort, oldAPen);
Flood(rastPort, 0, 10, 50); /* Start Flood() inside triangle. */
```

This example saves the current **FgPen** value and draws the shape in the same color as **AOIPen**. Then **FgPen** is restored to its original color so that **FgPen**, **BgPen**, **DrawMode**, and **AreaPtrn** can be used to define the fill within the outline.

## Rectangle-fill Operations

The final fill function, **RectFill()**, is for filling rectangular areas. The form of this function follows:

```
RectFill(&rastPort, xmin, ymin, xmax, ymax);
```

where

**&rastPort**

points to the **RastPort** that receives the filled rectangle

**xmin and ymin**

represent the upper left corner of the rectangle

**xmax and ymax**

represent the lower right corner of the rectangle.

### NOTE

The variable **xmax** must be equal to or greater than **xmin**, and **ymax** must be equal to or greater than **ymin**.

Rectangle-fill uses **FgPen**, **BgPen**, **AOIPen**, **DrawMode** and **AreaPtrn** to fill the area you specify. Remember that the fill can be multicolored as well as single- or two-colored. When the **DrawMode** is **COMPLEMENT**, it complements all bit planes, rather than only those planes in which the foreground is non-zero.

The three sets of statements in the following example routine create exactly the same drawing.

```
/* BoxFill
   Three methods of rendering a filled rectangle.
   Insert this routine into the "wrapper" code at the end of the TEXT chapter.
*/

#include <graphics/gfxmacros.h>

#define AREA_SIZE 200

#define COLOR0 0
#define COLOR1 1
#define COLOR3 3

BOOL example(struct Window *window)
{
    register USHORT i;
    SHORT xLow, xHigh, yLow, yHigh;
    WORD areabuffer[AREA_SIZE];
    UWORD height, width;
    PLANEPTR planePtr = NULL;
    struct RastPort *rastPort = window->RPort;
    struct TmpRas tmpRas;
    struct AreaInfo areaInfo;

    xLow = window->BorderLeft;
    xHigh = window->GZZWidth - window->BorderRight;
    yLow = window->BorderTop;
    yHigh = window->GZZHeight - window->BorderBottom;

    height = GfxBase->NormalDisplayRows;
    width = GfxBase->NormalDisplayColumns;
```

```

/* The AreaInfo and TmpRas are only needed by the Area...() calls. */
planePtr = AllocRaster(width, height);
if (planePtr)
{
    for (i=0; i<AREA_SIZE; i++)
        areabuffer[i] = 0;

    InitArea(&areaInfo, areabuffer, (AREA_SIZE*2)/5);
    rastPort->AreaInfo = &areaInfo;

    InitTmpRas(&tmpRas, planePtr, RASSIZE(width, height));
    rastPort->TmpRas = &tmpRas;

    SetRast(rastPort, COLOR0);

    /* Area-fill a rectangular area. */
    TITLE(window, "AreaMove(), AreaDraw(), AreaEnd()");
    Delay(2L * TICKS_PER_SECOND);
    SetAPen(rastPort, COLOR1);
    SetOPen(rastPort, COLOR3);
    AreaMove(rastPort, xLow, yLow);
    AreaDraw(rastPort, xLow, yHigh);
    AreaDraw(rastPort, xHigh, yHigh);
    AreaDraw(rastPort, xHigh, yLow);
    /* AreaEnd() will complete the rectangle automatically. */
    AreaEnd(rastPort);

    Delay(5L * TICKS_PER_SECOND);
    SetRast(rastPort, COLOR0);

    /* Flood-fill a rectangular area. */
    TITLE(window, "Move(), Draw(), Flood()");
    Delay(2L * TICKS_PER_SECOND);
    SetAPen(rastPort, COLOR3);
    BNDRYOFF(rastPort);
    Move(rastPort, xLow, yLow);
    Draw(rastPort, xLow, yHigh);
    Draw(rastPort, xHigh, yHigh);
    Draw(rastPort, xHigh, yLow);
    /* Must complete the rectangle or flood will leak. */
    Draw(rastPort, xLow, yLow);
    SetAPen(rastPort, COLOR1);
    /* Start Flood() in the middle of the rectangle,
       and replace all pixels of the same color as x,y).
    */
    Flood(rastPort, 1L, (xHigh-xLow)/2, (yHigh-yLow)/2);

    Delay(5L * TICKS_PER_SECOND);
    SetRast(rastPort, COLOR0);

    /* Rectangle-fill a rectangular area. */
    TITLE(window, "RectFill()");
    Delay(2L * TICKS_PER_SECOND);
    SetAPen(rastPort, COLOR1);
    SetOPen(rastPort, COLOR3);
    RectFill(rastPort, xLow, yLow, xHigh, yHigh);

    FreeRaster(planePtr, width, height);
}

return(WAIT_FOR_CLOSE);
}

```

Not only is the RectFill() routine the shortest, it is also the fastest to execute.

## Data Move Operations

### NOTE

The graphics library rendering and data movement routines generally wait to get access to the blitter, start their blit, and then exit. Therefore, you must **WaitBlit()** after a graphics rendering or data movement call if you intend to immediately deallocate, examine, or perform order-dependent processor operations on the memory used in the call.

The graphics support functions include several routines for simplifying the handling of the rectangularly organized data that you would encounter when doing raster-based graphics. These routines do the following:

- Clear an entire segment of memory
- Set a raster to a specific color
- Scroll a subrectangle of a raster
- Draw a pattern "through a stencil"
- Extract a pattern from a bit-packed array and draw it into a raster
- Copy rectangular regions from one bit-map to another
- Control and utilize the hardware-based data mover, the blitter

The following sections cover these routines in detail.

### Clearing a Memory Area

For memory that is accessible to the blitter (that is, internal CHIP memory), the most efficient way to clear a range of memory is to use the blitter. You use the blitter to clear a block of memory with the statement:

```
BltClear(memblock, bytecount, flags);
```

where **memblock** is a pointer to the location of the first byte to be cleared, and **bytecount** is the number of bytes to set to zero. For the usage of the **flags** variable, refer to *The Amiga ROM Kernel Manual: Includes and Autodocs*.

This command accepts the starting location and count and clears that block to zeros.

### Setting a Whole Raster to a Color

You can preset a whole raster to a single color by using the function **SetRast()**. A call to this function takes the following form:

```
SetRast(&rastPort, pen);
```

where

**&rastPort**

is a pointer to the **RastPort** you wish to use

**pen** is the pen value to fill the **RastPort** with.

### Scrolling a Sub-rectangle of a Raster

You can scroll a sub-rectangle of a raster in any direction—up, down, left, right, or diagonally. To perform a scroll, you use the **ScrollRaster()** routine and specify a **dx** and **dy** (delta-x, delta-y) by which the rectangle image should be moved towards the (0,0) location.

As a result of this operation, the data within the rectangle will become physically smaller by the size of **delta-x** and **delta-y**, and the area vacated by the data when it has been cropped and moved is filled with the background color (color in **BgPen**). **ScrollRaster()** is affected by the **Mask** setting.

Here is the syntax of the **ScrollRaster()** function:

```
ScrollRaster(&rastPort, dx, dy, xmin, ymin, xmax, ymax);
```

where

**&rastPort**

is a pointer to a **RastPort**

**dx, dy**

are the distances (positive, 0, or negative) to move the rectangle

**xmin, xmax, ymin, ymax**

specify the outer bounds of the sub-rectangle

Here are some examples that scroll a sub-rectangle:

```
/* scroll up 2 */
ScrollRaster(&rastPort, 0, 2, 10, 10, 50, 50);

/* scroll right 1 */
ScrollRaster(&rastPort, -1, 0, 10, 10, 50, 50);
```

When scrolling left or right while using a Super-BitMap Window **ScrollRaster()** requires a properly initialized **TmpRas**. Refer to the Windows section of the “Intuition” chapter for information on Super-BitMap windows. The **TmpRas** must be initialized to the size of one bit-plane with a width and height the same as the Super-BitMap, using the technique described in the “Area-Fill Information” section. When scrolling a Simple-Refresh Window (or other layered **RastPort**), **ScrollRaster()** scrolls the appropriate existing damage region. Refer to the “Layers” chapter for an explanation of damage regions. If you are using a Super-BitMap Layer, it is possible that the information in the BitMap is not fully reflected in the layer and vice-versa. Two graphics calls: **CopySBitMap()** and **SyncSBitMap()** remedy these situations. Refer to *The Amiga ROM Kernel Manual: Includes and Autodocs* for more information.

## Drawing through a Stencil

The routine **BitPattern()** allows you to change only a very selective portion of a drawing area. Basically, this routine lets you define the rectangular region to be affected by this drawing operation and a mask of the same size that defines how that area will be affected.

The following figure shows an example of what you can do with **BitPattern()**. The 0 bits are represented by blank rectangles; the 1 bits by filled-in rectangles.

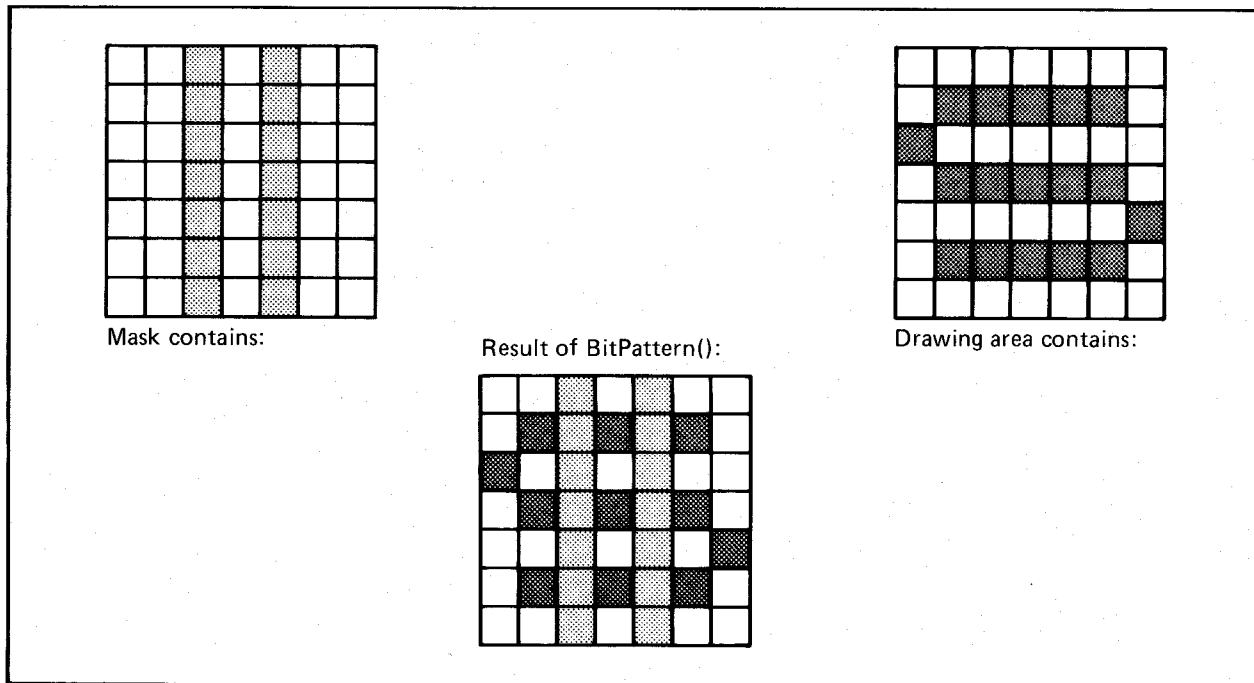


Figure 23-17: Example of Drawing Through a Stencil

In the “Result” drawing, the lighter squares show where the target drawing area has been affected. Exactly *what* goes into the drawing area where the mask has 1’s is determined by your **FgPen**, **BgPen**, **DrawMode**, and **AreaPtrn**.

The variables that control this function are:

- &rastport** a pointer to the drawing area
- mask** a pointer to the mask (mask layout explained below). May be NULL, in which case a rectangular region is modified.
- xl, maxx** upper left corner x, and lower right corner x
- yl, maxy** upper left corner y, and lower right corner y

**bytecnt**      number of bytes per row for the mask (*must* be an even number of bytes)

You call **BltPattern()** with:

```
BltPattern(&rastport, mask, xl, yl, maxx, maxy, bytecnt)
```

The **mask** parameter is a rectangularly organized, contiguously stored pattern. This means that the pattern is stored in linearly increasing memory locations stored as  $(maxy - yl + 1)$  rows of **bytecnt** bytes per row.

#### NOTE

These patterns must obey the same rules as **BitMaps**. This means that they must consist of an even number of bytes per row. For example, a mask such as:

```
01000010000000000
00100100000000000
00011000000000000
00100100000000000
```

is stored in memory beginning at a legal *word* address.

#### Extracting from a Bit-packed Array

You use the routine **BltTemplate()** to extract a rectangular area from a source area and place it into a destination area. The following figure shows an example.

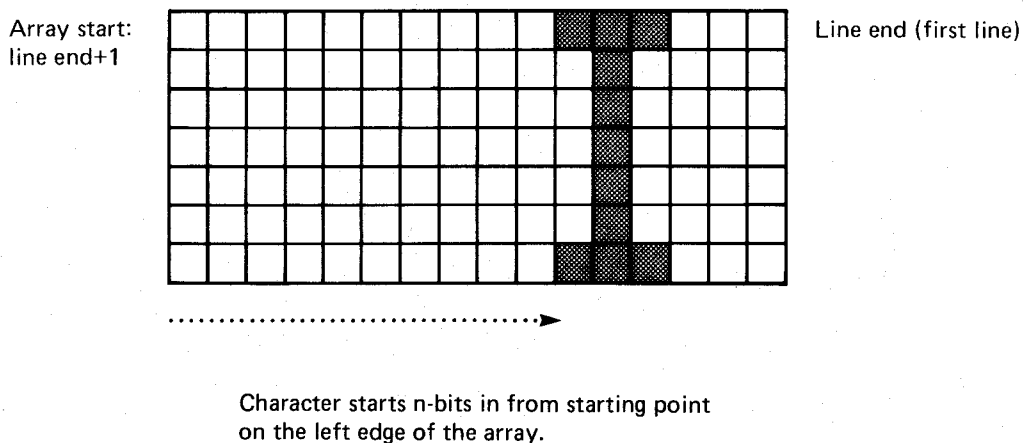


Figure 23-18: Example of Extracting from a Bit-Packed Array

If the rectangular bit array is to be represented as a rectangle within a larger, rectangularly organized bit array, the system must know how the larger array is organized. This allows the system to extract each line of the object properly. For this extraction to occur properly, you need to tell the system the modulo for the array. The modulo is the value that must be added to the address pointer so that it points to the correct word in the next line in this rectangularly organized array.

The following figure represents a single bit-plane and the smaller rectangle to be extracted. The modulo in this instance is 4, because at the end of each line, you must add 4 to the address pointer to make it point to the first word in the smaller rectangle.

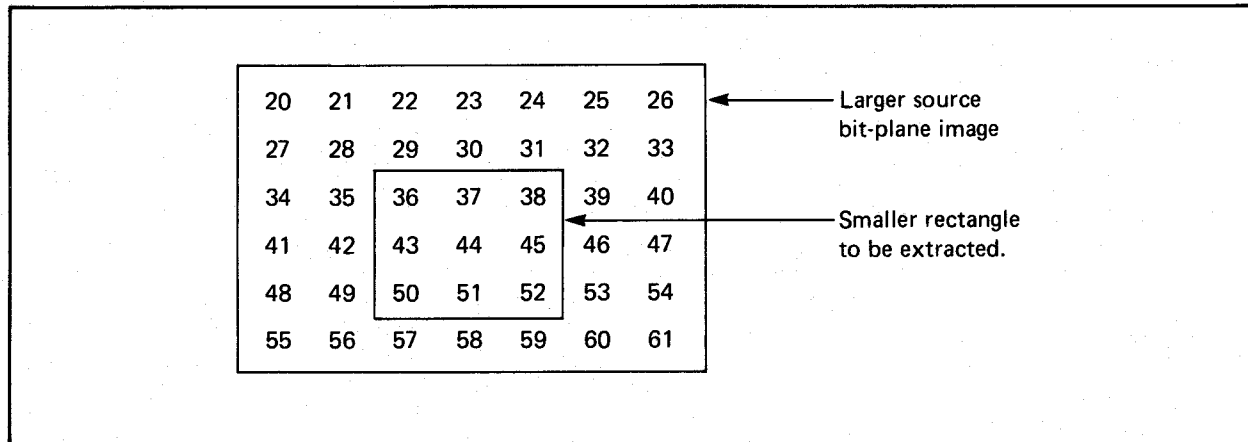


Figure 23-19: Modulo

#### NOTE

The modulo value must be an even number of bytes.

**BltTemplate()** takes the following arguments:

```
BltTemplate(source, srcX, srcMod, &destRastPort, destX, destY, sizeX, sizeY);
```

<b>source</b>	the source pointer for the array
<b>srcX</b>	source X (bit position) in the array at which the rectangle begins
<b>srcMod</b>	source modulo so it can find the next part of the source rectangle
<b>destRastPort</b>	a pointer to the destination <b>RastPort</b>
<b>destX, destY</b>	destination x and y, showing where to put the rectangle
<b>sizeX, sizeY</b>	size x and y, indicating how much data to move

**BltTemplate()** uses **FgPen**, **BgPen**, **DrawMode** and **Mask** to place the template into the destination area. This routine differs from **BltPattern()** in that only a solid color is deposited in the destination drawing area, with or without a second solid color as the background (as in the case of text). Also, the template can be arbitrarily bit-aligned and sized in x.

#### Copying Rectangular Areas

Four routines copy rectangular areas from one section of a **BitMap** to another: **BltBitMap()**, **BltBitMapRastPort()**, **BltMaskBitMapRastPort()**, and **ClipBlit()**. **BltBitMap()** is the basic routine, taking **BitMaps** as part of its arguments. It allows you to define a rectangle in a source region and copy it to a destination area of the same size in another (or even the same) **BitMap**. This routine is used in graphics rendering.



**BltBitMap()** returns the number of planes actually involved in the blit. The syntax for the function is:

```
ULONG planes;  
  
planes = BltBitMap(&srcBM, srcX, srcY, &dstBM, dstX, dstY,  
                 sizeX, sizeY, minterm, mask, tempA);
```

**BltBitMapRastPort** takes most of the same arguments, but its destination is a **RastPort**, instead of a **BitMap**. The syntax for the function is:

```
VOID BltBitMapRastPort(&srcBM, srcX, srcY, &dstRP, dstX, dstY,  
                    sizeX, sizeY, minterm);
```

**BltMaskBitMapRastPort()** works like **BltBitMapRastPort()**, except that a single bit-plane mask also controls the blit. The syntax for the function is:

```
VOID BltMaskBitMapRastPort(&srcBM, srcX, srcY, &dstRP, dstX, dstY,  
                        sizeX, sizeY, minterm, bltmask);
```

#### NOTE

The data area **bltmask** must be in CHIP memory.

**ClipBlit()** takes most of the same arguments as **BltBitMap()**, but it works with the **RastPorts** and layers. Before **ClipBlit()** moves data, it looks at the area from which and to which the data is being copied (**RastPorts**, not **BitMaps**) and determines if there are overlapping areas involved. It then splits up the overall operation into a number of bit maps to move the data in the way you request. To execute **ClipBlit()**:

```
VOID ClipBlit(&srcRP, srcX, srcY, &dstRP, dstX, dstY, XSize, YSize, minterm);
```

The following code fragments show how to save and restore an undo buffer.

```
/* Save work rastport to an undo rastport */  
ClipBlit(&drawRP, 0, 0, &undoRP, 0, 0, areaWidth, areaHeight, 0xC0);  
  
/* restore undo rastport to work rastport */  
ClipBlit(&undoRP, 0, 0, &drawRP, 0, 0, areaWidth, areaHeight, 0xC0);
```

The **minterm** variable is an unsigned byte value which represents the action to be performed during the move. This routine uses the blitter to move the data and can therefore logically combine or change the data as the move is made. The most common operation is a direct copy from source area to destination, which is the hex value C0.

You can determine how to set the **minterm** variable by using the logic equations shown in the following table.

Table 23-5: Minterm Logic Equations

Logic Term in Leftmost 4 Bits	Logic Term Included in Final Output
8	BC
4	$\overline{B}C$
2	$B\overline{C}$
1	$\overline{B}\overline{C}$

Source B contains the data from the source rectangle, and source C contains the data from the destination area. If you choose bits 8 and 4 from the logic terms (C0), in the final destination area you will have data that occurs in source B only. Thus, C0 means a direct copy. The logic equation for this is:

$$BC + \overline{B}C = B(C + \overline{C}) = B$$

Logic equations may be used to decide on a number of different ways of moving the data. For your convenience, a few of the most common ones are listed in the following table.

Table 23-6: Some Common Logic Equations for Copying

Hex Value	Mode
30	Replace destination area with inverted source B.
50	Replace destination area with inverted version of original of destination.
60	Put B where C is not, put C where B is not (cookie cut).
80	Only put bits into destination where there is a bit in the same position for both source and destination (sieve operation).
C0	Plain vanilla copy from source to destination.

Refer to the listing for **BltBitMap()** in *The Amiga ROM Kernel Manual: Includes and Autodocs*.

### Accessing the Blitter in a Multitasking Environment

To use the blitter, you must first be familiar with how its registers control its operation. This topic is covered thoroughly in the *The Amiga Hardware Reference Manual* and is not repeated here.

Five routines may be used to access the blitter:

- **WaitBlit()** returns as soon as the blitter is idle. It is used to ensure that the blitter has completed its current operation before being given another operation. It should normally only be used when dealing with the blitter in a synchronous manner, such as when using **OwnBlitter()** and **DisownBlitter()**. **WaitBlit()** does not wait for all blits queued up using **QBlit()** or **QBSBlit()**. You should call **WaitBlit()** if you are just about to free some memory that you have used with the blitter.

#### NOTE

Many graphic calls start up the blitter and return to the caller. The CPU does not need to wait for the blitter to finish before returning. When examining bits with the CPU immediately after a blit, or when freeing temporary memory used by the blitter, a **WaitBlit()** may be required.

- **OwnBlitter()** allows your task to obtain exclusive use of the blitter.

#### NOTE

The system uses the blitter extensively for disk and display operation. While your task is using the blitter, many other system processes will be locked out. Therefore, use it only for brief periods and relinquish it as quickly as possible, using **DisownBlitter()**.

- **DisownBlitter()** returns the device to shared operation.
- **QBlit()** and **QBSBlit()** let your task queue up requests for the use of the blitter on a non-exclusive basis. You share the blitter with system tasks.

To use **QBlit()** and **QBSBlit()**, you provide a data structure called a **bltnode** (blitter node). The system can use this structure to link blitter usage requests into a first-in, first-out (FIFO) queue. When your turn comes, your own blitter routine can be repeatedly called until your routine says it is finished using the blitter.

Two separate queues are formed. One queue is for the **QBlit()** routine. You use **QBlit()** when you simply want something done and you do not necessarily care when it happens. This may be the case when you are moving data in a memory area that is not currently being displayed.

The second queue is maintained for **QBSBlit()**. QBS stands for “queue-beam-synchronized”. **QBSBlit()** forms a beam-synchronized FIFO. When the video beam gets to a predetermined position, your routine is called. Beam synchronization takes precedence over the simple FIFO. This means that if the beam sync matches, the beam-synchronous blit will be done before the non-synchronous blit in the first position in the queue. You might use **QBSBlit()** to draw into an area of memory that is currently being displayed to modify memory that has already been “passed-over” by the video beam. This avoids display flicker as an area is being updated.

The input to each routine is a pointer to a **bltnode** data structure. The required items of the data structure are:

- A pointer to a **bltnode**
- A pointer to a function to perform
- A **beamsync** value (used if this is a **beamsync** blit)
- A status flag indicating whether the blitter control should perform a “clean-up” routine when the last blit is finished
- The address of the clean-up routine

The **bltnode** data structure is contained in the include file *hardware/blit.h*. Here is a copy of that data structure, followed by details about the items you must initialize:

```
struct bltnode
{
    struct bltnode *n;
    int      (*function) ( );
    char      stat;
    short     blitsize;
    short     beamsync;
    int      (*cleanup) ( );
};
```

The contents of **bltnode** are as follows:

**struct bltnode \*n;**

This is a pointer to the next **bltnode**, which, for most applications will be zero. You should not link **bltnodes** together. This is to be performed by the system by way of a separate call to **QBlit()** or **QBSBlit()**.

**int (\*function)();**

This position is occupied by the address of a function that the blitter queuer will call when your turn comes up. Your routine must be formed as a subroutine, with an RTS at the end. Using the C-language convention, the returned value will be in D0 (C returns its value by the **return(value)** statement).

If you return a nonzero value, the system will call your routine the next time the blitter is done until you finally return 0. This is to allow you to maintain control over the blitter; for example, it allows you to handle all five bit-planes if you are blitting an object that spans that number of planes. For display purposes, if you are blitting multiple objects and then saving and restoring the background, you must be sure that all planes of the object are positioned before another object is overlaid. This is the reason for the lockup in the blitter queue; it allows all work per object to be completed before going on to the next one.

Assembly language programmers take note:

Actually, the system tests the *status codes* for a condition of **EQUAL** or **NOTEQUAL**. When the C language returns the value of 0, it sets the status codes to **EQUAL**. When it returns a value of -1, it sets the status codes to **NOTEQUAL**, so they would be compatible. Functions **(\*function)()** that are written for **QBlit()** and **QBSBlit()** are not normally written in C. They are usually written in assembly language, as they then can take advantage of the ability of the queue routines to pass them parameters in the system registers. The register passing conventions for these routines are as follows:

- Register A0 receives a pointer to the system hardware registers so that all hardware registers can be referenced as an offset from that address.
- Register A1 contains a pointer to the current **bltnode**. You may have queued up multiple blits, each of which perhaps uses the same blitter routine. You can access the data for this particular operation as an offset from the value in A1. A typical user of these routines will precalculate the hardware register values that are stuffed into the registers and, during the routine, simply stuff them. For example, you can create a new structure such as the following:

```
INCLUDE "exec/types.i"
INCLUDE "hardware/blit.i"

STRUCTURE mybltnode,0
    ; Make this new structure compatible with a bltnode
    ; by making the first element a bltnode structure.
STRUCTURE bltnode,bn_SIZEOF
    UWORD bltcon1    ; Blitter control register 1.
    UWORD fwmask     ; First and last word masks.
```

```

        UWORD    lwmask
        UWORD    bltmda        ; Modulos for sources a, b, and c.
        UWORD    bltmdb
        UWORD    bltmdc
        UWORD    any_more_data ; add anything else you want
LABEL mbn_SIZEOF

```

Other forms of data structures are certainly possible, but this should give you the general idea.

**char stat;**

Tells the system whether or not to execute the clean-up routine at the end. This byte should be set to CLEANUP (0x40) if cleanup is to be performed. If not, then the **bltnode cleanup** variable can be zero.

**short beamsync;**

The value that should be in the VBEAM counter for use during a beam-synchronous blit before the **function()** is called.

The system cooperates with you in planning when to start a blit in the routine **QBSBlit()** by not calling your routine until, for example, the video beam has already passed by the area on the screen into which you are writing. This is especially useful during single buffering of your displays. There may be time enough to write the object between scans of the video display. You will not be visibly writing while the beam is trying to scan the object. This avoids flicker (part of an old view of an object along with part of a new view of the object).

**int (\*cleanup)();**

The address of a routine that is to be called after your last return from the **QBlit()** routine. When you finally return a zero, the queuer will call this subroutine (ends in **RTS** or **return()**) as the clean-up. Your first entry to the function may have dynamically allocated some memory or may have done something that must be undone to make for a clean exit. This routine must be specified.

## User Copper Lists

The Copper coprocessor allows you to produce mid-screen changes in certain hardware registers in addition to changes that the system software already provides. For example, it is the Copper that allows the Amiga to split the viewing area into multiple draggable screens, each with its own independent set of colors.

To create your own mid-screen (or mid-Intuition-Screen) effects on the system hardware registers, you provide "user Copper lists" that can be merged into the system Copper lists.

In the **ViewPort** data structure there is a pointer named **UCopIns**. If this pointer value is non-NULL, it points to a user Copper list that you have dynamically allocated and initialized to contain your own special hardware-stuffing instructions.

You allocate a user Copper list by an instruction sequence such as the following:

```

struct UCopList *uCopList = NULL;

/* Allocate memory for the Copper list */
/* Make certain that the initial memory is cleared. */
uCopList = (struct UCopList *)
    AllocMem(sizeof(struct UCopList), MEMF_PUBLIC|MEMF_CLEAR);
if (uCopList == NULL)
    return (FALSE);

```

## NOTE

User Copper lists do *not* have to be in CHIP RAM.

Once this pointer to a user Copper list is available, you can use it with system macros (*graphics/gfxmacros.h*) to instruct the system what to add to its own list of things for the Copper to do within a specific **ViewPort**.

The file *graphics/gfxmacros.h* provides the following four macro functions that implement user Copper instructions.

**CINIT** initializes the Copper list buffer. It is used to specify how many instructions are going to be placed in the Copper list.

```
CINIT(uCopList, num_entries);
```

**CWAIT** waits for the video beam to reach a particular horizontal and vertical position. Its format is:

```
CWAIT(uCopList, v, h)
```

where

**uCopList**

is the pointer to the Copper list

**v** is the vertical position for which to wait, specified relative to the top of the **ViewPort**. The legal range of values (for both NTSC and PAL) is from 0 to 255.

**h** is the horizontal position for which to wait. The legal range of values (for both NTSC and PAL) is from 0 to 226.

**CMOVE** installs a particular value into a specified system register. Its format is:

```
CMOVE(uCopList, reg, value)
```

where

**uCopList** is the pointer to the Copper list

**reg** is the register to be affected, specified in this form: "custom.register" (see *hardware/custom.h*)

**value** is the value to place in the register

**CBump** increments the user Copper list pointer to the next position in the list. It is usually invoked for the programmer as part of the macro definitions **CWAIT** or **CMOVE**. Its format is:

```
CBump(uCopList)
```

where **uCopList** is the pointer to the user Copper list.

**CEND** terminates the user Copper list. Its format is:

```
CEND(uCopList)
```

where **uCopList** is the pointer to the user Copper list.

Executing any of the user Copper list macros causes the system to dynamically allocate special data structures called intermediate Copper lists that are linked into your user Copper list (the list to which **uCopList** points) describing the operation. When you call the function **MakeVPort(&view, &viewport)** as shown in the section called "Forming A Basic Display," the system uses all of its intermediate Copper lists to sort and merge together the real Copper lists for the system (**LOFCprList** and **SHFCprList**).

When your program exits, you must return to the system all of the memory that you allocated or caused to be allocated. This means that you must return the intermediate Copper lists, as well as the user Copper list data structure. Here are two different methods for returning this memory to the system.

```
/* Returning memory to the system if you have NOT
 * obtained the viewport from Intuition. */
```

```
FreeVPortCopLists(viewPort);
FreeVPortCopLists(&viewport);
```

```
/* Returning memory to the system if you HAVE
 * obtained the viewport from Intuition. */
```

```
CloseScreen(&screen); /* Intuition only */
```

The example program below shows the use of user Copper lists under Intuition.

```
/* UserCopperExample
   User Copper List Example
   For Lattice, compile and link with: LC -bl -cfist -L -v -y uCopperExample.c
*/
```

```
#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <intuition/intuition.h>
#include <hardware/custom.h>
#include <libraries/dos.h>
```

```
#include <proto/all.h>
#include <stdlib.h>
```

```
/* Use this structure to gain access to the custom registers. */
extern struct Custom far custom;
```

```
VOID openAll(VOID), cleanExit(int);
BOOL loadCopper(VOID);
```

```
struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Window *window = NULL;
struct IntuiMessage *intuiMessage;
struct ViewPort *viewPort = NULL;
struct UCopList *uCopList = NULL;
```

```
struct NewWindow newWindow =
{
    40,0, /* Upper Left Corner */
    175,10, /* Width and Height */
    -1,-1, /* Use screen default pens */
    CLOSEWINDOW, /* IDCMP flags */
    WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|NOCAREREFRESH, /* Flags */
    NULL, /* no gadgets */
    NULL, /* use default checkmark image */
}
```

```

    "Copper!",      /* Title */
    NULL,           /* using WorkBench screen */
    NULL,           /* no super-bitmap */
    0,0,0,0,        /* no sizing */
    WBENCHSCREEN    /* it goes on the WorkBench screen */
};

#define NUMCOLORS    32
#define NUMLINES_EACH 8

UWORD colors[] =
{
    0x0604, 0x0605, 0x0606, 0x0607, 0x0617, 0x0618, 0x0619,
    0x0629, 0x072a, 0x073b, 0x074b, 0x074c, 0x075d, 0x076e,
    0x077e, 0x088f, 0x07af, 0x06cf, 0x05ff, 0x04fb, 0x04f7,
    0x03f3, 0x07f2, 0x0bf1, 0x0ff0, 0x0fc0, 0x0ea0, 0x0e80,
    0x0e60, 0x0d40, 0x0d20, 0x0d00
};

VOID main(VOID)
{
    openAll(); /* Open the libraries and a window. */
    if (loadCopper()) /* Do the Copper specific stuff. */
    {
        /* Wait until the user clicks in the close gadget */
        (VOID) Wait(1<<window->UserPort->mp_SigBit);
        while (intuiMessage = (struct IntuiMessage *)GetMsg(window->UserPort))
            ReplyMsg((struct Message *)intuiMessage);
    }
    cleanExit(RETURN_OK);
}

VOID openAll(VOID)
{
    IntuitionBase=(struct IntuitionBase *)OpenLibrary("intuition.library", 33L);
    if (IntuitionBase == NULL)
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);
    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
    if (GfxBase == NULL)
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);
    if (! (window = OpenWindow(&newWindow))) /* Open a Window. */
        cleanExit(ERROR_NO_FREE_STORE);
    viewPort = ViewPortAddress(window); /* Get a pointer to the ViewPort. */
}

VOID cleanExit(int retval)
{
    if (uCopList) /* Free the memory allocated for the Copper. */
    {
        FreeVPortCopLists(viewPort);
        RemakeDisplay();
    }
    if (window)
        CloseWindow(window);
    if (IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);
    if (GfxBase)
        CloseLibrary((struct Library *)GfxBase);
    exit(retval);
}

BOOL loadCopper(VOID)
{
    register USHORT i;

    /* Allocate memory for the Copper list */
    /* Make certain that the initial memory is cleared. */

```



```

uCopList = (struct UCopList *)
    AllocMem(sizeof(struct UCopList), MEMF_PUBLIC|MEMF_CLEAR);
if (uCopList == NULL)
    return(FALSE);
CINIT(uCopList, NUMCOLORS); /* Initialize the Copper list buffer. */
for (i=0; i<NUMCOLORS; i++) /* Load in each color. */
{
    CWAIT(uCopList, (i*NUMLINES_EACH), 0);
    CMOVE(uCopList, custom.color[0], colors[i]);
}
CEND(uCopList); /* End the Copper list */
/* Forbid task switching while changing the Copper list. */
Forbid();
viewPort->UCopIns=uCopList;
Permit(); /* Permit task switching again. */
RethinkDisplay(); /* Display the new Copper list. */
return(TRUE);
}

```

## Advanced Graphics Examples

### DUAL-PLAYFIELDS EXAMPLE

This example is almost identical to the single-playfield demonstration program earlier in this chapter. It has been adapted to show a dual-playfield display with objects drawn in both playfields. The single playfield example wrote directly into the screen's memory. This example adds a `RastPort` so that rectangle-fill routines can be used.

```

/* DualPF
   Dual-PlayFields Example
   For Lattice, compile and link with: LC -bl -cfist -L -v -y DualPF.c
*/

#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/rastport.h>
#include <graphics/view.h>
#include <hardware/dmabits.h>
#include <hardware/custom.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdlib.h>

#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200

struct GfxBase *GfxBase = NULL;

/* Add a second BitMap for dual playfield. */
struct BitMap bitMap, bitMap2;
struct View view, *oldview;

#define BLACK 0x000
#define BLUE 0x00f
#define GREEN 0x0f0
#define RED 0xf00
#define VIOLET 0xf0f
#define ORANGE 0xf80
#define WHITE 0xfff

```

```

#define COLOR0 0
#define COLOR1 1
#define COLOR2 2
#define COLOR3 3

VOID cleanExit(int exitStatus);

VOID main(VOID)
{
    SHORT i, stripe;
    LONG color;

    /* In dual playfield mode, colors 0-7 are dedicated to playfield one
     * and colors 8-15 to playfield two. Since (in this example)
     * there are only 2 planes in each playfield, colors 4-7 and 12-15
     * won't even get used. Colors 4-7 are included below to keep the
     * values of colors 8-11 in their proper locations for LoadRGB4().
     */
    static UWORD colortable[] =
    {
        BLACK, WHITE, ORANGE, RED,
        0, 0, 0, 0, /* The values for colors 4-7 are placeholders. */
        /* The second playfield's BLACK will be transparent. */
        BLACK, GREEN, VIOLET, BLUE
    };

    /* Add a second RasInfo for dual playfield. */
    struct RasInfo rasInfo, rasInfo2;
    struct RastPort rastPort, rastPort2; /* RastPorts for both BitMaps. */
    struct ViewPort viewPort;

    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
    if (GfxBase == NULL)
        cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);

    /* Initialize the BitMaps. */
    InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);
    InitBitMap(&bitMap2, DEPTH, WIDTH, HEIGHT);

    /* Clear the plane pointers. */
    for (i=0; i<DEPTH; i++)
        bitMap.Planes[i] = bitMap2.Planes[i] = NULL;

    /* Allocate space for their Planes. */
    for (i=0; i<DEPTH; i++)
    {
        bitMap.Planes[i] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
        if (bitMap.Planes[i] == NULL)
            cleanExit(ERROR_NO_FREE_STORE);

        bitMap2.Planes[i] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
        if (bitMap2.Planes[i] == NULL)
            cleanExit(ERROR_NO_FREE_STORE);
    }

    /* Initialize the RastPorts and link the BitMaps to them. */
    InitRastPort(&rastPort);
    InitRastPort(&rastPort2);
    rastPort.BitMap = &bitMap;
    rastPort2.BitMap = &bitMap2;

    /* Simple form of setting drawing area to color COLOR0. */
    SetRast(&rastPort, COLOR0);
    SetRast(&rastPort2, COLOR0);

    /* Initialize the RasInfos. */
    rasInfo.BitMap = &bitMap; /* Attach the corresponding BitMap. */
    rasInfo.RxOffset = 0; /* Align upper left corners of display */
    rasInfo.RyOffset = 0; /* with upper left corner of drawing area. */
    rasInfo.Next = &rasInfo2; /* Link second RasInfo structure to the first. */

    /* Initialize second RasInfo for Dual Playfield. */

```

```

rasInfo2.BitMap = &bitMap2;
rasInfo2.RxOffset = 0;
rasInfo2.RyOffset = 0;
rasInfo2.Next = NULL;

InitVPort(&viewPort);    /* Initialize the ViewPort. */

/* Specify ViewPort characteristics. */
viewPort.DWidth = WIDTH;
viewPort.DHeight = HEIGHT;
viewPort.RasInfo = &rasInfo;
viewPort.Modes = DUALPF; /* dual-playfield mode */

/* Initialize the ViewPort's ColorMap. */
viewPort.ColorMap = GetColorMap(12L);
if (viewPort.ColorMap == NULL)
    cleanExit(ERROR_NO_FREE_STORE);
LoadRGB4(&viewPort, colortable, 12);

InitView(&view);    /* Initialize the View. */
view.ViewPort = &viewPort; /* Attach the ViewPort to the View. */

/* Construct (preliminary) Copper instruction list. */
MakeVPort( &view, &viewPort );

/* Merge preliminary lists together into a real
 * Copper list in the View structure.
 */
MrgCop( &view );

/* Save current view to restore later. Example steals
 * the screen from the active view, but restores it when done.
 */
oldview = GfxBase->ActiView;

/* Load the newly created View. */
LoadView( &view );

/* Stripes of color will be drawn in the playfields.
 * Wherever color 0 is used in playfield number 1,
 * the colors of playfield number 2 will show through.
 */

/* Playfield number 1. Vertical stripes. */
stripe = WIDTH/32;
for (color=0L, i=WIDTH-stripe; i>=0; i-=stripe)
{
    SetAPen(&rastPort, color++ % 4L); /* Cycle through the four colors. */

    /* Create a rectangle. The coordinates are inset by one on all
     * four edges to allow for the outline created by RectFill.
     */
    RectFill(&rastPort, i+1, 1, i+stripe-2, HEIGHT-1);

    /* Delay for 1/4 second between stripes.
     * The +1L prevents the possibility of a Delay(0) which
     * would cause problems due to a bug in the timer.device.
     */
    Delay(TICKS_PER_SECOND / 4L + 1L);
}

Delay(1L * TICKS_PER_SECOND); /* Pause for one second. */

/* Playfield number 2. Horizontal stripes. */
stripe = HEIGHT/20;
for (color=0L, i=HEIGHT-stripe; i>=0; i-=stripe)
{
    SetAPen(&rastPort2, color++ % 4L);
    RectFill(&rastPort2, 1, i+1, WIDTH-1, i+stripe-2);
    Delay(TICKS_PER_SECOND / 4L + 1L);
}

```

```

Delay(10L * TICKS_PER_SECOND);
cleanExit(RETURN_OK);

}    /* end of main() */

VOID cleanExit(int exitStatus)
{
    SHORT i;

    if (oldview)
    {
        LoadView(oldview);    /* Put the original View back again. */
        WaitTOF();    /* Wait for that View to return. */
    }

    /* Free the drawing area. */
    for(i=0; i<DEPTH; i++)
    {
        if (bitMap.Planes[i])
            FreeRaster(bitMap.Planes[i], WIDTH, HEIGHT);

        if (bitMap2.Planes[i])
            FreeRaster(bitMap2.Planes[i], WIDTH, HEIGHT);
    }

    /* Free the color map created by GetColorMap(). */
    if (view.ViewPort->ColorMap)
        FreeColorMap(view.ViewPort->ColorMap);

    /* Free dynamically created structures. */
    if (view.ViewPort)
        FreeVPortCopLists(view.ViewPort);
    if (view.LOFCprList)
        FreeCprList(view.LOFCprList);

    if (GfxBase)
        CloseLibrary((struct Library *)GfxBase);

    exit(exitStatus);
}

```

## HOLD-AND-MODIFY MODE EXAMPLE

This example demonstrates the Amiga's hold-and-modify mode, showing at all times a different subset of 256 of the 4,096 colors available on the Amiga. At any moment, no two squares are the same color.

```

/*
    Hold-And-Modify Example
    For Lattice, compile and link with: LC -b1 -cfist -L -v -y HAMExample.c
*/

#include <exec/types.h>
#include <intuition/intuitionbase.h>
#include <graphics/gfxbase.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define XSIZE 11    /* Color box sizes. */
#define YSIZE 6

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

```

```

struct TextAttr textAttr =
{
    "topaz.font",    /* Standard system font. */
    8,
    0,
    0
};

struct Window *window = NULL;
struct Screen *screen = NULL;

struct NewScreen newScreen =
{
    0, 0,
    320, 200, 6,
    0, 1,
    HAM,
    CUSTOMSCREEN,
    &textAttr,
    (UBYTE *) "256 different colors out of",
    NULL,
    NULL
};

struct NewWindow newWindow =
{
    0, 11,                /* LeftEdge, TopEdge, */
    320, 186,             /* Width, Height, */
    -1, -1,               /* DetailPen, BlockPen, (-1 == "use Screen's") */
    MOUSEBUTTONS|CLOSEWINDOW, /* IDCMPFlags, */
    ACTIVATE|WINDOWCLOSE,  /* Flags, */
    NULL,                 /* FirstGadget, */
    NULL,                 /* CheckMark, */
    (UBYTE *) "4,096 at any given moment.", /* Title, */
    NULL,                 /* Screen, */
    NULL,                 /* BitMap, */
    0, 0,                 /* MinWidth, MinHeight, */
    320, 186,             /* MaxWidth, MaxHeight, */
    CUSTOMSCREEN           /* Type */
};

#define BLACK 0x000
#define RED 0xf00
#define GREEN 0x0f0
#define BLUE 0x00f
#define WHITE 0xfff

#define COLOR0 0
#define COLOR1 1
#define COLOR2 2
#define COLOR3 3

VOID cleanExit(LONG exitStatus);
VOID hamBox(struct RastPort *rastPort, LONG color, LONG x, LONG y);
VOID prompt(struct RastPort *rastPort);
VOID
    colorWheel(struct RastPort *rastPort, SHORT xpos[], SHORT ypos[], BOOL textneeded);
VOID
    colorFull(struct RastPort *rastPort, SHORT xpos[], SHORT ypos[], BOOL textneeded);
extern ULONG RangeRand(ULONG);

VOID main(VOID)
{
    BOOL wheelmode = TRUE, textneeded = TRUE;
    SHORT xpos[16], ypos[16];
    USHORT code, i;
    static UWORD colors[] = {BLACK, RED, GREEN, BLUE, WHITE};
    ULONG class;
    struct RastPort *rastPort;    /* Graphics structures. */
    struct ViewPort *viewPort;
    struct IntuiMessage *intuiMessage;

```

```

for(i=0; i<16; i++)      /* Establish color square positions. */
{
    xpos[i] = (XSIZE + 4) * i + 20;
    ypos[i] = (YSIZE + 3) * i + 21;
}

IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 33L);
if (IntuitionBase == NULL)
    cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);

GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
if (GfxBase == NULL)
    cleanExit(ERROR_INVALID_RESIDENT_LIBRARY);

screen = OpenScreen(&newScreen);
if (screen == NULL)
    cleanExit(ERROR_NO_FREE_STORE);

newWindow.Screen = screen;      /* Open window in our new screen. */
window = OpenWindow(&newWindow);
if (window == NULL)
    cleanExit(ERROR_NO_FREE_STORE);

viewPort = &(screen->ViewPort); /* Set colors in screen's ViewPort. */
rastPort = window->RastPort;    /* Render into the window's RastPort. */

/* Set the color registers: Black, Red, Green, Blue, White. */
LoadRGB4(viewPort, colors, 5L);

SetBPen(rastPort, COLOR0);      /* Insure clean text. */

/* Process any and all messages in the queue, then update the display
 * colors once, then come back here to look at the queue again. If you
 * see a left-mouse-button-down event, then switch display modes. If you
 * see a Close-Window-gadget event, then clean up and exit the program.
 * NOTE: This is a BUSY LOOP so the colors will cycle as quickly as possible.
 */
while (1)
{
    while(intuiMessage=(struct IntuiMessage *)GetMsg(window->UserPort))
    {
        /* Can't reply until done using it! */
        class = intuiMessage->Class;
        code = intuiMessage->Code;
        ReplyMsg((struct Message *)intuiMessage);

        if (class == CLOSEWINDOW) /* Exit the program. */
            cleanExit(RETURN_OK);

        if (class == MOUSEBUTTONS && code == SELECTDOWN) /* Swap modes. */
        {
            wheelmode = NOT wheelmode;

            SetAPen(rastPort, COLOR0); /* Clear the drawing area. */
            SetDrMd(rastPort, JAM1);
            RectFill(rastPort, 3, 12, 318, 183);
            textneeded = TRUE;
        }
    }
    if (wheelmode)
        colorWheel(rastPort, xpos, ypos, textneeded);
    else
        colorFull(rastPort, xpos, ypos, textneeded);

    textneeded = FALSE;
}

/*
 * Display a randomized set of colors.
 */

```

```

VOID
colorFull(struct RastPort *rastPort, SHORT xpos[], SHORT ypos[], BOOL textneeded)
{
    SHORT usesquare;
    ULONG sChoice, cChoice;
    LONG usecolor;
    static SHORT sStop, cStop;
    static SHORT squares[16 * 16];
    static LONG squarecolor[16 * 16], freecolors[4096-(16*16)];

    if(textneeded) /* First call since mode change? */
    {
        prompt(rastPort);
        sStop = 255; /* Top of list of squares yet to change. */
        cStop = 4095 - 256; /* Top of list of colors still needing use. */

        for(usecolor=0; usecolor<256; usecolor++) /* Initialize colors. */
        {
            usesquare = usecolor;
            squares[usesquare] = usesquare;
            squarecolor[usesquare] = usecolor;
            hamBox(rastPort, usecolor, xpos[usesquare % 16], ypos[usesquare / 16]);
        }

        for(usecolor=256; usecolor<4095; usecolor++) /* Ones not yet used. */
            freecolors[usecolor - 256] = usecolor;
    }

    /*****
    * Randomly choose next square to change such that all squares change color
    * at least once before any square changes twice. squares[0] through squares
    * [sStop] are the square numbers that have not yet changed in this pass.
    * RangeRand(r) is an integer function provided in "amiga.lib" that produces
    * a random result in the range 0 to (r-1) given an integer r in the range 1 to 65535.
    *****/

    sChoice = RangeRand(sStop + 1); /* Pick a remaining square. */

    usesquare = squares[sChoice]; /* Extract square number. */
    squares[sChoice] = squares[sStop]; /* Swap it with sStop slot. */
    squares[sStop] = usesquare;

    if (NOT sStop--)
        sStop = 255; /* Only one change per pass. */

    /*****
    * Randomly choose new color for selected square such that all colors are
    * used once before any color is used again, and such that no two squares
    * simultaneously have the same color. freecolors[0] through freecolors[cStop]
    * are the colors that have not yet been chosen in this pass. Note that
    * the 256 colors in use at the end of the previous pass are not available
    * for choice in this pass.
    *****/

    cChoice = RangeRand(cStop + 1);

    usecolor = freecolors[cChoice];
    freecolors[cChoice] = freecolors[cStop];
    freecolors[cStop] = squarecolor[usesquare];
    squarecolor[usesquare] = usecolor;

    if (NOT cStop--)
        cStop = 4095 - 256;

    hamBox(rastPort, usecolor, xpos[usesquare % 16], ypos[usesquare / 16]);
}

/*
    Display an ordered set of colors.
*/

```

```

VOID
    colorWheel(struct RastPort *rastPort, SHORT xpos[], SHORT ypos[], BOOL textneeded)
{
    SHORT i, j;
    static SHORT sequence;
    static UBYTE *number[] =
    {
        "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"
    };
    static UBYTE red[] = "Red", blue[] = "Blue", green[] = "Green";
    if(textneeded)
    {
        prompt(rastPort);

        SetAPen(rastPort, COLOR2); /* Green pen for green color numbers. */
        Move(rastPort, 260, ypos[15]+17);
        Text(rastPort, green, strlen(green));
        for(i=0; i<16; i++)
        {
            Move(rastPort, xpos[i]+3, ypos[15]+17);
            Text(rastPort, number[i], strlen(number[i]));
        }

        SetAPen(rastPort, COLOR3); /* Blue pen for blue color numbers. */
        Move(rastPort, 4, 18);
        Text(rastPort, blue, strlen(blue));
        for(i=0; i<16; i++)
        {
            Move(rastPort, 7, ypos[i]+6);
            Text(rastPort, number[i], strlen(number[i]));
        }

        SetAPen(rastPort, COLOR1); /* Red pen for red color numbers. */
        Move(rastPort, 271, 100);
        Text(rastPort, red, strlen(red));

        sequence = 0;
    }

    SetAPen(rastPort, COLOR1); /* Identify the red color in use. */
    SetDrMd(rastPort, JAM2);
    Move(rastPort, 280, 115);
    Text(rastPort, number[sequence], strlen(number[sequence]));

    for(j=0; j<16; j++) /* Update all of the squares. */
    {
        for(i=0; i<16; i++)
            hamBox(rastPort, (sequence<<8 | i<<4 | j), xpos[i], ypos[j]);
    }

    if (++sequence == 16)
        sequence = 0;
}

/* Display mode changing prompt. */
VOID prompt(struct RastPort *rastPort)
{
    static UBYTE text[] = "[left mouse button = new mode]";

    SetDrMd(rastPort, JAM2);
    SetAPen(rastPort, 4);
    Move(rastPort, 23, 183);
    Text(rastPort, text, strlen(text));
}

/*****
 * hamBox() -- routine to draw a colored box in Hold and Modify mode. Draws a
 * box of size XSIZE by YSIZE with an upper left corner at (x,y). The
 * desired color is achieved in 3 steps on each horizontal line of the box.
 */

```



```

*   First we set the red component, then the green, then the blue. We
*   achieve this by drawing a vertical line of Modify-Red, followed by a
*   vertical line of Modify-Green, followed by a rectangle of Modify-Blue.
*   Note that the resulting color for the first two vertical lines depends
*   upon the color(s) of the pixels immediately to the left of that
*   line. By the time we reach the rectangle we are assured of getting
*   (and maintaining) the desired color because we have set all 3
*   components (R, G, and B) straight from the bit map.
*****/

VOID hamBox(struct RastPort *rastPort, LONG color, LONG x, LONG y)
{
    SHORT c;

    SetDrMd(rastPort, JAM1); /* Establish Drawing Mode in RastPort. */

    c = (color & 0xf00) >> 8; /* Extract desired Red color component. */
    SetAPen(rastPort, c + 0x20); /* Hold G, B from previous pixel. Set R=n. */
    Move(rastPort, x, y);
    Draw(rastPort, x, y+YSIZE);

    x++;
    c = (color & 0xf0) >> 4; /* Extract desired Green color component. */
    SetAPen(rastPort, c + 0x30); /* Hold R, B from previous pixel. Set G=n. */
    Move(rastPort, x, y);
    Draw(rastPort, x, y+YSIZE);

    x++;
    c = (color & 0xf); /* Extract desired Blue color component.*/
    SetAPen(rastPort, c + 0x10); /* Hold R, G from previous pixel. Set B=n. */
    RectFill(rastPort, x, y, x+XSIZ-2, y+YSIZE);
}

/*
Clean up and exit.
*/
VOID cleanExit(LONG exitStatus)
{
    struct IntuiMessage *intuiMessage;

    if (window)
    {
        /* Reply to any pending IntuiMessages. */
        while(intuiMessage=(struct IntuiMessage *)GetMsg(window->UserPort))
            ReplyMsg((struct Message *)intuiMessage);
        CloseWindow(window);
    }
    if (screen)
        CloseScreen(screen);
    if (GfxBase)
        CloseLibrary((struct Library *)GfxBase);
    if (IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);

    exit(exitStatus);
}

```

# Chapter 24

## Graphics: Text

### Introduction

Text on the Amiga is simply another graphics primitive. Because of this, you can easily intermix text and graphics on the same screen. Typically, a 320-by-200 graphics screen can contain 40-column, 25-line text using a text font defined in an 8-by-8 matrix. The same type of font can be used to display 80-column text if the screen resolution is extended to 640 by 200. Window borders and other graphics embellishments may reduce the actual available area.

The text support routines use the **RastPort** structure to hold the variables that control the text rendering process. Therefore, any changes you make to **RastPort** variables affect both the drawing routines and the text routines.

In addition to the basic fonts provided in the ROMs, you can link your own font into the system, and ask that it be used along with the other system fonts.

This chapter shows you how to:

- Print text into a drawing area
- Specify the character color

- Specify which font to use
- Access disk-based fonts
- Define a new font
- Define a disk-based font

## Printing Text into a Drawing Area

The placement of text in the drawing area depends on several variables. Among these are the current position for drawing operations, the font width and height, and the placement of the font baseline within that height.

### CURSOR POSITION

Text position and drawing position use the same variables in the **RastPort** structure—**cp\_x** and **cp\_y**, the current horizontal and vertical pen position. The text character begins at this point. You use the graphics call **Move(rastPort, x, y)** to establish the **cp\_x** and **cp\_y** position.

### BASELINE OF THE TEXT

The **cp\_y** position of the drawing pen specifies the position of the baseline of the text. In other words, all text printed into a **RastPort** using a single “write string” command is positioned relative to this **cp\_y** as the text baseline. The following figure shows some sample text that includes a character that has 1 dot below the baseline and a maximum of 7 dots above and including the baseline.

For clarity, blank squares and shaded squares, rather than 0s and 1s, are used for the figure.

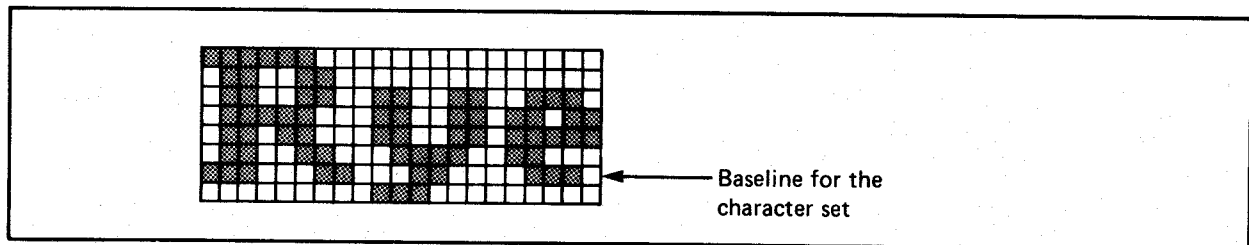


Figure 24-1: Text Baseline

The figure shows that for this font, the baseline value is 6. The baseline value is the number of lines from the top of the character to the baseline.

When the text routines output a character to a **RastPort**, the leftmost edge of the character position is specified by the **cp\_x** (current horizontal position) variable.

After all characters have been written to the **RastPort**, the variable **cp\_y** is unchanged. The value of **cp\_x** will be changed by the number of horizontal positions that were needed to write all characters of the specified text. Both fixed-width and proportionally spaced character sets are accommodated.

The default fonts in the system are all designed to be above and below the baseline, where the baseline position in this example is at line 6 of the character font. This means that you must specify a **cp\_y** value of at least 6 when you request that text be printed to a **RastPort** in order to assure that you stay within the memory bounds of the **RastPort** itself.

You should exercise caution when requesting that the text routines write beyond the outer bounds of the **RastPort** memory, either horizontally or vertically. Text written outside the **RastPort** bounds may be clipped if the **RastPort** supports clipping (most do). Clipping means that the system will actually render only that portion of the text that is written within the boundaries of the **RastPort**. If it is *not* clipped, the contents of memory may be corrupted.

## SIZE OF THE FONT

Font design is covered later in this chapter. For now, simply note that the width and height of the font affect how many characters you may print on a line. The position of the baseline affects where the text is rendered.

## PRINTING THE TEXT

Text is rendered into a **RastPort** with the **Text()** routine. A typical call to this routine is:

```
Text(rastPort, string, count);
```

where

<b>rastPort</b>	is a pointer to the <b>RastPort</b> structure where the text is to be rendered
<b>string</b>	is the address of the string to be rendered
<b>count</b>	is the string length

```
/* ShowDefaultFont
   An example that prints the name and size of the default font.
   Insert this routine into the "wrapper" code at the end of the chapter.
*/

BOOL example(struct Window *window)
{
    UBYTE textNum[6];    /* A temporary buffer. */
    struct TextAttr textAttr;
    struct RastPort *rastPort = window->RPort;
    SHORT indentLeft = window->BorderLeft, indentTop = window->BorderTop;

    TITLE(window, "This Window's default font");
    SetAPen(rastPort, COLOR1);    /* Set the pen color. */

    /* AskFont() will fill textAttr with the default font's attributes. */
    AskFont(rastPort, &textAttr);

    /* Move the pen's current position such that the characters
       will be rendered below the Window's top border.
    */
    Move(rastPort, indentLeft, indentTop + rastPort->TxBaseline);
```

```

/* Render this string (which is 20 characters long). */
Text(rastPort, "The default font is ", 20);

/* The pen's cp_y has been updated to where the last rendering finished.
   If we don't Move() it, the next string will continue from there.
*/
Text(rastPort, textAttr.ta_Name, strlen(textAttr.ta_Name));

/* Use sprintf() to copy the numeric value to a string buffer. */
sprintf(textNum, "%d", textAttr.ta_YSize);
Text(rastPort, textNum, strlen(textNum));

Text(rastPort, " pixels high.", 13);

/* Wait for the user to click in the close gadget before returning. */
return(WAIT_FOR_CLOSE);
}

```

## Selecting the Font

Two default character fonts are provided in the ROMs. One font produces either 40- or 80-column text (depending on the use of a 320 or 640 horizontal resolution, respectively). The other font produces either 32- or 64-column text. Each font has a name associated with it. The names and specifications of these default fonts are shown in the following table.

Table 24-1: Default Character Fonts

Name	Height	No. of Columns
topaz.font	8	40/80
topaz.font	9	32/64

Although both fonts have the same name, they are of different size, allowing the system to differentiate between them.

To specify which font the system should use, you call the system routine **OpenFont()** or **OpenDiskFont()**, followed by **SetFont()**. Typical calls to **OpenFont()** and **SetFont()** can be found in the example routine "ShowOpenFont", a fragment of which follows.

```

struct TextFont *textFont;

/* Request this font from the system. */
textFont = OpenFont(&textAttr);
if (textFont)
{
    /* Make this font the RastPort's default font. */
    SetFont(rastPort, textFont);
}

```

where

### **textFont**

is a pointer to a **TextFont** data structure, returned by either **OpenFont()** or **OpenDiskFont()**. **TextFont** is defined in the include file *graphics/text.h*.

### **textAttr**

is a **TextAttr** structure (it is defined in the include file *graphics/text.h*). It contains a pointer to a null-terminated string that specifies the name of the font, the font height, font style bits, and font flag bits.

### **rastPort**

is a pointer to the **RastPort** that is being told to use the font described by **textFont** as its default font.

The call to **OpenFont()** or **OpenDiskFont()** says "give me a font with these characteristics". The system attempts to fulfill your request by providing the font whose characteristics best match your request. If **OpenFont()** cannot be satisfied, it returns a 0.

### **NOTE**

In the "Graphics Primitives" chapter, you saw that the routine **InitRastPort()** initializes certain variables to default values. The **InitRastPort()** routine automatically sets the default font to **topaz.font** in the size set by the user from the "Preferences" program.

```
/* ShowOpenFont
   An example illustrating how to request a font from OpenFont().
   Insert this routine into the "wrapper" code at the end of the chapter.
*/

BOOL example(struct Window *window)
{
    SHORT indentLeft = window->BorderLeft, indentTop = window->BorderTop;
    struct RastPort *rastPort = window->RPort;
    struct TextAttr textAttr;
    struct TextFont *textFont, *oldTextFont;

    TITLE(window, "Using OpenFont() to request fonts");

    SetAPen(rastPort, COLOR1);

    /* Save the address of the RastPort's current font for replacement later.
       It cannot "disappear" during this time as it still has at least
       one "accessor".
    */
    oldTextFont = rastPort->Font;

    /* Fill the TextAttr structure with the desired characteristics. */
    /* These just happen to be the attributes of the standard topaz 80 font. */
    textAttr.ta_Name = "topaz.font";
    textAttr.ta_YSize = 8;
    textAttr.ta_Style = FS_NORMAL;
    textAttr.ta_Flags = FPF_DESIGNED | FPF_ROMFONT;

    /* Request this font from the system. */
    textFont = OpenFont(&textAttr);
    if (textFont)
    {
        /* Make this font the RastPort's default font. */
        SetFont(rastPort, textFont);

        /* Move the pen's current position such that the characters
           will be rendered below the Window's top border.
        */
        indentTop += rastPort->TxBaseline;
        Move(rastPort, indentLeft, indentTop);
    }
}
```

```

Text(rastPort, "topaz.font, 8 dots high", 23 );

/* Set this RastPort's font back to its original font. */
SetFont(rastPort, oldTextFont);

/* Close the font when completely done with it. */
CloseFont(textFont);
}

textAttr.ta_YSize = 9; /* Change the textAttr structure, */
textFont = OpenFont(&textAttr); /* and request this new font. */
if (textFont)
{
    SetFont(rastPort, textFont);
    /* Move down to avoid overlapping with the previous line.
       Use the new font's YSize plus one pixel.
    */
    indentTop += textFont->tf_YSize + 1;
    Move(rastPort, indentLeft, indentTop);
    Text(rastPort, "topaz.font, 9 dots high", 23);
    SetFont(rastPort, oldTextFont);
    CloseFont(textFont);
}
return(WAIT_FOR_CLOSE);
}

```

## Selecting the Text Color

You can select which color to use for the text you print by using the graphics calls **SetAPen()** and **SetBPen()** and by selecting the drawing mode in your **RastPort** structure. The combination of those values determines exactly how the text will be printed.

## Selecting a Drawing Mode

The **DrawMode** variable of a **RastPort** determines how the text will be combined with the graphics in the destination area.

### NOTE

With the exception of **INVERSVID**, the **DrawMode** selections are mutually exclusive. You can select from any *one* of the following drawing modes.

If **DrawMode** is **JAM1**, it means that the text will be drawn in the color of **FgPen** (the foreground, or primary, drawing pen). Wherever there is a 1-bit in the character's image definition, the **FgPen** color will overwrite the data present at the text position in the **RastPort**. This is called overstrike mode.

If **DrawMode** is **JAM2**, it means that the **FgPen** color will be used for the text, and the **BgPen** color (the background or secondary drawing color pen) will be used as the background color for the text. The rectangle of data bits that defines the text-character completely overlays the destination area in your **RastPort**. Where there is a 1 bit in the character's image definition, the **FgPen** color is used. Where there is a 0 bit in the image, the **BgPen** color is used. This mode draws text with a colored background.

If **DrawMode** is **COMPLEMENT**, it means that wherever the text character is drawn, a position occupied by a 1 bit causes bits in the destination **RastPort** to be changed as follows (see also the figure which follows):

- If a text-character 1 bit is to be written over a destination area 0 bit, it changes the destination area to a 1 bit.
- If a text-character 1 bit is to be written over a destination area 1 bit, the result of combining the source and destination is a 0 bit. In other words, whatever the current state of a destination area bit, a 1 bit in the source changes it to the opposite state.
- Zero bits in the text character definition have no effect on the destination area.

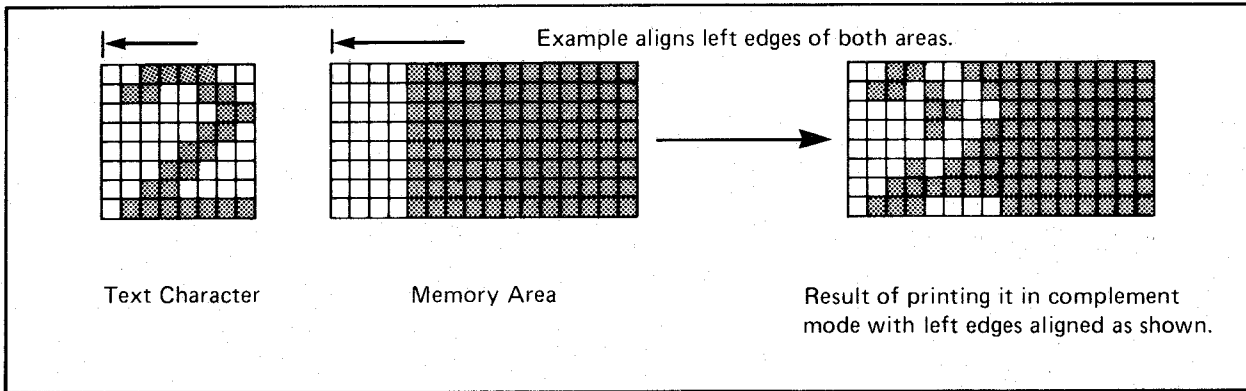


Figure 24-2: Complement Mode

If you combine any of the draw modes with the **INVERSVID** flag, it will change all 1 bits to 0 bits and vice versa in a text or other **RastPort** writing operation before writing them into the destination area.

```

/* ShowDrawModes
   Example to illustrate the different DrawModes.
   Insert this routine into the "wrapper" code at the end of the chapter.
*/

BOOL example(struct Window *window)
{
    struct RastPort *rastPort = window->RPort;
    /* x and y will be used to position the text. d (the amount to move
       down to keep subsequent lines of text from overlapping) is 1 more
       than the height of the default font for this RastPort.
    */
    SHORT x = window->BorderLeft;
    SHORT y = window->BorderTop + rastPort->TxBaseline;
    SHORT d = rastPort->TxHeight + 1;
    static UBYTE mode, modes[] =
    {
        JAM1, JAM2, COMPLEMENT
    };
    static UBYTE *modeText[] =
    {
        " This is JAM1 DrawMode ",
        " This is JAM2 DrawMode ",
        " This is COMPLEMENT DrawMode "
    };

    TITLE(window, "The various DrawModes");

    /* Set the A pen color to one which will stand out from both the
       standard foreground and background Workbench colors.
    */
    SetAPen(rastPort, COLOR3);
    /* The B pen color will be the standard background color. */
    SetBPen(rastPort, COLOR0);

```



```

/* Fill the RastPort with a color other than the standard background color. */
SetRast(rastPort, COLOR1);

/* Run through the three modes. y is incremented by d pixels for each line. */
for (mode=0; mode < 3; mode++, y+=d)
{
    Move(rastPort, x, y); /* Set the starting position. */
    SetDrMd(rastPort, modes[mode] ); /* Set the draw mode. */
    Text(rastPort, modeText[mode], strlen(modeText[mode]));
    /* cp_x (current position for x) has increased by the length of the
       string (in pixels), cp_y is unchanged. The text that is about to
       be rendered will begin at this position.
    */
    SetDrMd(rastPort, modes[mode] | INVERSVID ); /* "or" in INVERSVID. */
    Text(rastPort, " and this is it in inverse. ", 28);
}
return(WAIT_FOR_CLOSE);
}

```

## Effects of Specifying Font Style

When you call `OpenFont()`, specifying certain style characteristics, the system searches the loaded fonts to find the closest match to the font you requested. If the remainder of the characteristics match what you have requested, but the style does not match, the text routines `AskSoftStyle()` and `SetSoftStyle()` create a font styled as you have requested by modifying the existing font (that is, modifying a normal font to italic or bold by modifying its characters). Because many fonts do not lend themselves to such modifications, it is always preferred that the font of the specific style be loaded for use. The system always tries to find the exact specified font before attempting to modify another to fit your request.

If there is a font present in the system that matches your `OpenFont()` request both in name and size, but not in style, (as determined by looking at the font style field), you may use `SetSoftStyle()` to generate the selected style algorithmically as follows:

### NORMAL

The font is used exactly as defined.

### NOTE

A font that is *designed* to be italic, bold, etc. cannot have those attributes removed by `SetSoftStyle()`.

### UNDERLINED .

An underline is generated one pixel below the baseline position.

### BOLD

The character is binary or-ed with a copy of itself, shifted right by the bold shift attribute (`tf_BoldSmear`) of the font (usually 1).

### ITALIC

The character is given a slant to the right, starting from the bottom line, and shifting subsequent upward line positions to the right one bit position for every second count up from the bottom of the character.

## EXTENDED

This attribute cannot be set with `SetSoftStyle()`. See “Font Style” below.

If you use a font that has the various style characteristics built in, rather than generated, the internal spacing and kerning tables tell the system how to leave the proper amount of space between characters if you are simply printing them one at a time.

If you ask `Text()` to output the characters individually, `Text()` calculates character width and positioning based on the width and inter-character spacing that it finds in the font descriptor. After printing one or more characters, it automatically positions the drawing pen (`cp_x`) at the position it believes to be correct for the next output character. In some fonts, this may cause adjacent characters to overlap when printed individually. Moreover, overlap may occur when italic or bold is generated via `SetSoftStyle()`.

There is a solution to this problem. You can build your output string of characters before calling `Text()` to output it. `Text()` can handle character strings, correctly generating the desired style with correct inter-character spacing.

To increase inter-character spacing, you can set a field called `rp_TxSpacing` in the `RastPort`. The spacing is specified in pixels.

## Using a Disk Font

To use an existing disk font, you must open the diskfont library and then open a disk font. Refer to the “wrapper” program at the end of this chapter for an example of opening the diskfont library.

The code to open and use a font that is on disk is remarkably similar to that of a font already in memory. The reader is encouraged to try modifying the “ShowOpenFont” routine to make it use `OpenDiskFont()` to load the “sapphire” font (found on the Workbench disk) in sizes 14 and 19. This is as simple as replacing the information saved in the `TextAttr` structure and changing the `OpenFont()` calls to `OpenDiskFont()` calls. The “diskfont” library must be opened before calls to `OpenDiskFont()` can be made. The “wrapper” code used for the examples in this chapter does this for you. `CloseFont()` is used to close fonts opened with either `OpenFont()` or `OpenDiskFont()`.

### NOTE

There is an important difference between disk-based and memory-resident font usage. A “process” can call either `OpenFont()` or `OpenDiskFont()`, while a “task” is limited to calling `OpenFont()`. Refer to the “task” section of the Exec chapter for more information on tasks and processes.

## USING FONTS IN OTHER DIRECTORIES

`OpenDiskFont()` has a feature which makes it easy to access fonts in directories other than `FONTS:`. If the name in the `TextAttr` structure (which gets passed to `OpenDiskFont()`) is a full pathname, (e.g. `Work:special/accounting font` or `df0:Phunny!Phonts/quack.font`) then `OpenDiskFont()` will use that full pathname to access the font contents file.

## Finding Out Which Fonts Are Available

The function AvailFonts() fills in a memory area designated by you with a list of all of the fonts available to the system. AvailFonts() searches the AmigaDOS directory path currently assigned to *FONTSD:* and locates all available fonts. If you haven't issued a DOS Assign command to change the *FONTSD:* directory path, it defaults to the *sys:fonts* directory.

The example code "FontParade" provides a list of the fonts you can use and shows how to find the appropriate items to put into the text attribute data structure for the call to OpenDiskFont().

```
/* FontParade
   Example to display all fonts available to the system.
   Insert these routines into the "wrapper" code at the end of the chapter.
*/

/* The buffer for concatenating text strings will be this big. */
#define FONTSTRINGSIZE (MAXFONTNAME + 40L)
/* The buffer for the first attempt at AvailFonts() will be this big. */
#define AFH_SIZE_DEFAULT 400L

/* Pointer to buffer for AvailFonts() information. */
struct AvailFontsHeader *afHeader = NULL;
LONG afhSize = AFH_SIZE_DEFAULT;
UBYTE *fontName = NULL, *fontString = NULL; /* String pointers. */

/*
   Allocate a buffer to hold information returned by AvailFonts().
   Call AvailFonts(). If the buffer wasn't large enough, deallocate it,
   allocate a larger buffer and try again. Allocate memory for text strings.
   Return TRUE if completely successful, FALSE otherwise.
*/
BOOL getMem(VOID)
{
    LONG needMore;

    /* AvailFonts() returns a value indicating the amount of memory it
       would need (above what you have given it) to successfully fulfill
       the request. If this value is greater than zero, it is necessary
       to reallocate a larger chunk of memory and try again.
    */
    do
    {
        needMore = 0L;
        afHeader = (struct AvailFontsHeader *)AllocMem(afhSize, MEMF_PUBLIC);
        if (afHeader)
        {
            needMore =
                AvailFonts((UBYTE *)afHeader, afhSize, AFF_MEMORY | AFF_DISK);
            if (needMore)
            {
                FreeMem(afHeader, afhSize);
                afhSize += needMore;
            }
        }
    } while (needMore);

    if (afHeader)
    {
        fontString = (UBYTE *)AllocMem(FONTSTRINGSIZE, MEMF_PUBLIC);
        if (fontString)
        {
            fontName = (UBYTE *)AllocMem(MAXFONTNAME, MEMF_PUBLIC);
        }
    }
}
```

```

        if (fontName)
            return(TRUE);
    }
return(FALSE);
}

/*
Free any memory allocated in getMem().
*/
VOID returnMem(VOID)
{
    if (fontName)
        FreeMem(fontName, MAXFONTNAME);
    if (fontString)
        FreeMem(fontString, FONTSTRINGSIZE);
    if (afHeader)
        FreeMem(afHeader, afhSize);
}

/*
Walk through the linked-list of AvailFont structures filled in by
AvailFonts(). For each font, use SetSoftStyle() to set it to
each of eight styles (normal, underlined, italic, bold and all
logical permutations) in turn. Create a text string which is
composed of the font name, size, and style and render it into
the window in that particular size and style.
*/
BOOL example(struct Window *window)
{
    BOOL success, done = FALSE;
    SHORT right, left, bottom, xPos;
    SHORT style;
    UBYTE sizeString[5];
    UWORD entry;
    struct RastPort *rPort;
    struct AvailFonts *aFonts;
    struct TextFont *tFont, *oldTextFont;
    struct TextAttr tAttr;
    struct IntuiMessage *intuiMessage;

    /* The eight different styles, in #define form (from graphics/text.h). */
    static ULONG styles[ ] =
    {
        FS_NORMAL, FSF_UNDERLINED, FSF_ITALIC, FSF_ITALIC | FSF_UNDERLINED,
        FSF_BOLD, FSF_BOLD | FSF_UNDERLINED, FSF_BOLD | FSF_ITALIC,
        FSF_BOLD | FSF_ITALIC | FSF_UNDERLINED
    };

    /* The eight different styles, in descriptive text form. */
    static UBYTE *descriptions[ ] =
    {
        "Normal", "Normal Underlined", "Italic", "Italic Underlined",
        "Bold", "Bold Underlined", "Bold Italic", "Bold Italic Underlined"
    };

    /* Set window title to indicate activity. */
    TITLE(window, "Scanning Fonts:");
    if (success = getMem())
    {
        TITLE(window, "FontParade");

        rPort = window->RPort; /* Get the window's rastport. */
        /* Save the original TextFont pointer. */
        oldTextFont = rPort->Font;
        SetAPen(rPort, COLOR1); /* Set the A pen color. */
        SetDrMd(rPort, JAM1); /* Set the drawing mode. */

        /*
        Calculate the amount of space to indent from the

```

```

sides of the window, to keep the text from being hidden
behind the borders. If this were not a GimmeZeroZero
window, window->Width and window->Height would be used
instead of window->GZZWidth and window->GZZHeight.
Reverse-path fonts are rendered from right to left,
and so must begin on the right side of the window.
*/
left = window->BorderLeft;
right = window->GZZWidth - (SHORT>window->BorderRight;
bottom = window->GZZHeight - (SHORT>window->BorderBottom;

/* Skip over the header to the first of the AvailFont structures. */
aFonts = (struct AvailFonts *)&afHeader[1];

for (entry = afHeader->afh_NumEntries; entry && !done; entry--, aFonts++)
{
/*
It is possible that AvailFonts() will find two entries for a
single font, one of "af_Type 1" saying that the font is memory-
resident, and the other of "af_Type 2" saying the font is disk-based.
This happens because another process has previously called
OpenDiskFont() requesting that font.

The second part of the if-statement lets you tell the two apart if
you are scanning the list for unique elements; it says "if it's in
memory and it's also from disk, then don't list it because you'll find
another entry in the table that says it is not in memory, but is on
disk".
*/
if ( ! ((aFonts->af_Attr.ta_Flags & FPF_REMOVED) ||
(aFonts->af_Type & AFF_MEMORY) &&
(aFonts->af_Attr.ta_Flags & FPF_DISKFONT)))
{
/*
Copy the contents of the AvailFonts' TextAttr structure
to tAttr. Modify this copy to allow for both ROMFONT and
DISKFONT fonts (because both types are wanted). Then use
tAttr to request a font with these attributes from
OpenDiskFont(). An exact match will be found since all
the attributes came from an existing font.
*/
strcpy(fontName, aFonts->af_Attr.ta_Name);
tAttr.ta_Name = fontName;
tAttr.ta_YSize = aFonts->af_Attr.ta_YSize;
tAttr.ta_Style = aFonts->af_Attr.ta_Style;
tAttr.ta_Flags = aFonts->af_Attr.ta_Flags |
(FPF_ROMFONT | FPF_DISKFONT);

tFont = (struct TextFont *)OpenDiskFont(&tAttr);

if (tFont)
{
/* Make this font the RastPort's font. */
SetFont(rPort, tFont);
/* Step through the eight styles, one at a time. */
for (style=0; style<8; style++)
{
/*
The user may want to exit before the program has finished.
The following code is a simple method of checking for a
single class of message without putting the program to
sleep with the Wait() call. It is not as multitasking-
friendly as Wait(). Note that this routine does not check
what class of message has arrived since only one class of
message was asked for in the window's IDCMP field
(CLOSEWINDOW). See the Intuition chapter for information
on fully utilizing Intuition's IDCMP.
*/

/* If a message has arrived... */
intuiMessage =
(struct IntuiMessage *)GetMsg(window->UserPort);

```

```

        if (intuiMessage)
        {
            /* reply to it, */
            ReplyMsg((struct Message *)intuiMessage);
            done = TRUE; /* set the "done" flag and */
            break; /* break out of the "for" loop. */
        }

        /* If a font already has certain attributes (bold,
        italic, etc.) don't bother re-displaying it with
        those styles. Those attributes were evident
        when the font was displayed in its "normal" style.
        */
        if (styles[style] & tFont->tf_Style)
            continue;

        /* Create a string from the font's name, size and style. */
        strcpy(fontString, " ");
        strcat(fontString, tFont->tf_Message.mn_Node.ln_Name);
        sprintf(sizeString, " %ld", tFont->tf_YSize);
        strcat(fontString, sizeString);
        strcat(fontString, " point, ");
        strcat(fontString, descriptions[style]);
        strcat(fontString, " ");

        /* Set the style. */
        (VOID)SetSoftStyle(rPort, styles[style], ~0L);

        /* Set starting point for rendering. If font is
        reverse-path, start on the right side.
        */
        xPos = (tFont->tf_Flags & FPF_REVPATH) ? right : left;
        Move(rPort, xPos,
            bottom-(tFont->tf_YSize-tFont->tf_Baseline));

        /*
        Erase any previous text by clearing the window's
        rastport to the background color.
        */
        SetRast(rPort, COLOR0);

        /* Render the text string. */
        Text(rPort, fontString, (ULONG)strlen(fontString));

        /* Pause. */
        Delay(1L * TICKS_PER_SECOND);
    }

    /* Reset the RastPort's original font. */
    SetFont(rPort, oldTextFont);
    CloseFont(tFont);
}

}

else
/* Alert the user to the low-memory condition. */
TITLE(window, "Not enough free memory.");

/* Return any memory that was successfully allocated. */
returnMem();

if (success)
    /* Close immediately if there was no error. */
    return(DONT_WAIT);
else
    /* Wait for the user to read the error message. */
    return(WAIT_FOR_CLOSE);

```

The text in “FontParade” was rendered without regard to its size. If the text being rendered happened to go beyond the edge of the window, it disappeared from view as it was clipped by the layers system. It is possible to determine the character advance width (in pixels) of a string rendered by `Text()` without actually rendering the text. This is useful in determining the largest font that can be used to render text into a space of known size. The call is `TextLength()`; it takes the same arguments as the `Text()` call: a pointer to a `RastPort`, a pointer to a string, and the length of the string. It returns a `SHORT` which represents the change in `cp_x` that would take place if the text were to be rendered in the given `RastPort` in its default font.

Some fonts have characters that intrinsically render outside of the “character box” for their font. With certain fonts, even algorithmically styled italic and bold characters may render outside the font’s character box. In these cases, `TextLength()` is insufficient for determining whether a text string can be rendered wholly within a given area.

## Contents of a Font Directory

In a font directory, you will usually find two names for each font type. A typical pair of entries in the fonts directory might be:

```
sapphire.font  
sapphire (dir)
```

The file named *sapphire.font* does not contain the actual font. It contains the description of the contents of that font family. The contents are described by a `FontContentsHeader` and one or more `FontContents` data structure entries. The `FontContentsHeader` structure is defined in *libraries/diskfont.h* as:

```
struct FontContentsHeader  
{  
    UWORD fch_FileID; /* FCH_ID */  
    UWORD fch_NumEntries; /* the number of FontContents elements */  
    /* struct FontContents fch_FC[]; */  
};
```

where

### **fch\_FileID**

is a numeric identifier for this file type. It is defined as `0x0f00`.

### **fch\_NumEntries**

indicates how many entries of type `FontContents` follow this header.

The **FontContents** structure is defined as follows:

```
struct FontContents
{
    char  fc_FileName[MAXFONTPATH];
    UWORD fc_YSize;
    UBYTE fc_Style;
    UBYTE fc_Flags;
};
```

where

**fc\_FileName**

is the pathname that AmigaDOS must follow to find the actual diskfont descriptive header, along with the **TextFont** data structure of which this font is composed. In Amiga system software version 1.2 and earlier, this path is relative to *FONTS:*. In version 1.3 and later, it is relative to the directory containing the font contents (e.g. "*ruby.font*") file.

**fc\_YSize, fc\_Style, and fc\_Flags**

correspond to their equivalents in the **TextAttr** data structure (**ta\_YSize**, **ta\_Style**, and **ta\_Flags**).

As an example, a typical entry in *sapphire.font* is:

"sapphire/14",	a null-terminated string, padded out with zeros for a length of MAXFONTPATH bytes,
14,	the value for fc_YSize,
00,	the value for fc_Style,
60 (hex)	the value for fc_Flags.

This entry indicates that the actual **DiskFontHeader** for the font to be loaded is in *sapphire/14*. This means that there must be a file named *14* in the directory *sapphire* which is a subdirectory of the directory containing *sapphire.fonts*.

Beginning with Amiga system software version 1.3, the information in a **FontContentsHeader** structure can be easily generated programmatically with the **NewFontContents()** function call.

**NewFontContents()** is called with two parameters: an AmigaDOS Lock on the directory where a font contents file and associated font directory are located, and a pointer to the name of a font contents file in that "Locked" directory. If successful, **NewFontContents()** returns a pointer to a **FontContentsHeader** structure. If the font contents file could not be opened, or memory could not be allocated for the **FontContentsHeader**, **NewFontContents()** returns a 0.

At some time after a successful call to **NewFontContents()**, a call to **DisposeFontContents()** must be made to return the resources which **NewFontContents()** allocated from the system.

**DisposeFontContents()** is called with one parameter: a pointer to a **FontContentsHeader** structure.

Both **NewFontContents()** and **DisposeFontContents()** are part of the diskfont library, which must be successfully opened with version number 34 before they can be called.



## The Disk Font

A disk font is constructed as a loadable, executable module. In this manner, AmigaDOS can be used to perform **LoadSegment()** and **UnloadSegment()** on it. AmigaDOS can therefore allocate memory for the font, and return the memory when the font is unloaded. The contents of the **DiskFontHeader** structure are described in the include-file *libraries/diskfont.h*. The most significant item in this structure, the embedded **TextFont** structure, is described below in the topic “Defining a Font”.

## Defining a Font

The characteristics of a font are contained in a **TextFont** structure. The **TextFont** structure is specified in the include file named *graphics/text.h*. The following topics show the meaning of the items in a **TextFont** structure. Following the structure description is an example showing a four-character font, which is defined using this structure and can be linked into the system using **AddFont()**. Once a font has been added to the system font list with **AddFont()** it is available to any task that calls **OpenFont()** with a **TextAttr** request which is satisfied by the font’s attributes. **AddFont()** is passed one argument, a pointer to a **TextFont** structure. The **TextFont** structure passed must be in public memory (specify **MEMF\_PUBLIC** to **AllocMem()**). **AddFont()** has no return value. **RemFont()** will remove a font previously added with **AddFont()**, ensuring that access to it is restricted to those tasks that already have an active pointer to it; i.e. no new **SetFont()** requests for this font will be satisfied. **RemFont** takes one argument, a pointer to a **TextFont** structure. It has no return value. The memory used for the **TextFont** structure cannot be reclaimed until the font’s **tf\_Accessors** count has dropped to zero, indicating that there are no more users of this font.

### THE TEXT NODE

The first item in the **TextFont** structure is a **Node** structure with which the system can link this font into the system **TextFonts** list. The name of the font is referenced using the **ln\_Name** pointer in the **Node** structure.

### FONT HEIGHT

You specify the height in the **tf\_YSize** variable. All characters of the font must be defined using this number of lines of data even if they do not require that many lines to contain all font data. Variable-height fonts are not supported.

## FONT STYLE

You specify the style of the font by specifying certain constants in the `TextFont tf_Style` variable. The value of `tf_Style` is determined by the binary or-ing of the style constants, defined as:

<code>UNDERLINED</code>	The font is underlined.
<code>BOLD</code>	The font is bold.
<code>ITALIC</code>	The font is italic.
<code>EXTENDED</code>	The font is stretched out (width).

In the font structure, these bits indicate style attributes as an intrinsic part of the font; that is, the font already has them and one can never take them away. A font that has none of these characteristics is considered "NORMAL".

## FONT FLAGS

This variable provides additional information that tells the font routines how to create or access the characters. The `tf_Flags` variable is composed of the binary or-ing of the flag constants, defined as follows:

### `FPF_ROMFONT`

The font is located in ROM. Do not use this flag unless you are burning new system ROMs yourself.

### `FPF_DISKFONT`

The font must be loaded from disk.

### `FPF_REVPATH`

The font is designed to be rendered primarily from right to left (for example, Hebrew).

### NOTE

The actual rendering direction is determined by the kern and space tables.

### `FPF_PROPORTIONAL`

The characters in the font are not guaranteed to be `tf_XSize` wide (see "Font Width" below).

### `FPF_DESIGNED`

This font was specifically designed to have the size and attributes it has. Choosing this font for its characteristics will generally yield better results than allowing the system to algorithmically modify another font to have those characteristics.

### `FPF_REMOVED`

This font has been removed from the system, making it unavailable to other tasks.

## FONT WIDTH

The `tf_XSize` variable specifies the nominal width of the font.

## FONT ACCESSORS

Due to the Amiga's multitasking abilities, it is possible that more than one task will be accessing a character font. A variable in the font structure keeps track of how many accessors this font currently has. Whenever a call to `OpenFont()` or `OpenDiskFont()` is made, this variable is incremented for the font and later is decremented by `CloseFont()`. The font accessor value should never be reduced below zero. This accessor count should be initialized to zero *before* you first link a new font into the system, but it is managed by the system after the link is performed.

See the description of the `RemFont()` call for information on purging a font from the system.

## CHARACTERS REPRESENTED BY THIS FONT

It is possible to create a font consisting of from 0 to 256 characters. Some fonts can be exceedingly large because of their design and the size of the characters. For this reason, the text system allows the design and loading of fonts that may consist of only a few of the characters. The variables `tf_LoChar` and `tf_HiChar` specify the ASCII numerical values for the characters represented in this font.

In the example that is being built for this chapter, a font consisting of four playing card suits is being constructed. This font consists of only four items, one for each of the suits. Here is a fragment from the "suits8" assembler code which appears in full later in this chapter:

```
DC.B      97          ; tf_LoChar
DC.B     100         ; tf_HiChar
```

The range of 97 to 100 indicates that there are 4 characters represented in this font (`tf_HiChar - tf_LoChar + 1`). As part of the character data, in addition to defining the included character numbers, you must also define a character representation to be used as the image of a character number requested but not defined in this font. This character is placed at the end of the font definition.

For this example, any character number outside the range of 97 to 100 (inclusive) would print this "not in this font" character.

## THE CHARACTER DATA

The font structure includes a pointer to the character set data along with descriptions of how the data is packed into an array. The variables used are defined in *graphics/text.h*; their usage is as follows:

### `tf_CharData`

This is a pointer to the memory location at which the font data begins. This is the bit-packed array of character information.

### tf\_Modulo

This is the row modulo for the font. It must be an even number of bytes. The font is organized with the bits for the top line of the first character directly adjacent to the bits for the top line of the second character and so on.

For example, if the bit-packed character set needs 10 words of 16 bits each to hold the top line of all of the characters in the set, then the value of the modulo will be 20 (bytes). Twenty is the number which must be added to the pointer into the character matrix to go from the first line to the second line of a specific character.

### tf\_CharLoc

This is a pointer to an array of paired values. The values are the bit offset into the bit-packed character array for this character, and the size of the character in bits. Expressed in C language, this array of values can be expressed with a structure as:

```
struct charDef
{
    WORD charOffset;
    WORD charBitWidth;
};
```

In the program definition, the array to which **tf\_CharLoc** points can be expressed as:

```
/* Define an array of bit-packed placement and width information
   for four characters and one 'not a character' */
struct charDef suitDef[5];
```

For all fonts, there must be one set of descriptors for each character defined in the character set.

### tf\_CharSpace

This is a pointer to an array of words of proportional spacing information. It represents the advance for the character position from the starting position (after kerning, if applicable).

For example, a narrow character may still be stored within a wide space (see the following figure).

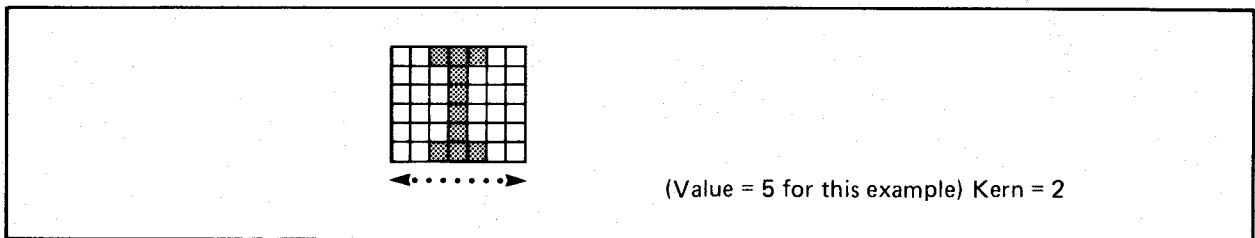


Figure 24-3: CharSpace

If this pointer is null, the nominal width for each character (**tf\_XSize**) is used.

### tf\_CharKern

This is a pointer to an array of words of character kerning data. Kerning is the offset from the starting character position to the start of the bit data (see the following figure). If this pointer is null, kerning is zero.

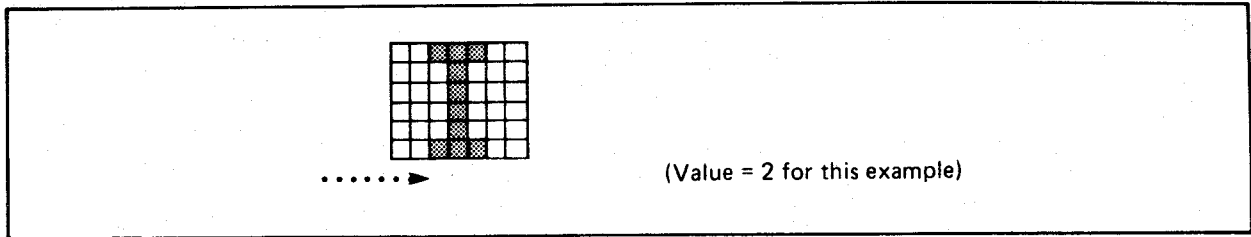


Figure 24-4: CharKern

## A COMPLETE SAMPLE FONT

The sample font below pulls together all of the pieces from the above sections. It defines a font whose contents are the four suits from a set of playing cards: hearts, spades, diamonds, and clubs.

The suits are defined as proportionally spaced to provide a complete example, even though each suit could as easily have been defined in a 14-wide-by-8-high matrix. There is an open-centered-square character, which is used if you ask for a character not defined in this font.

```
*
* A sparse (but complete) sample font. To use this font, do the following:
*
* 1. Assemble this file (assumed to have been saved as "suits8.asm").
*    For example, if you have the CAPE 680x0 assembler, and you have
*    assigned "include:" to the directory containing your include files,
*    use:
*    CAsm -a "suits8.asm" -o "suits8.o" -i "include:"
*
*    Link "suits8.o". For example, if you have Lattice, use:
*    BLink from "suits8.o" to "suits8"
*
* 2. Create the subdirectory "Fonts:suits".
*    Copy the file "suits8" (created in step 1.) to "Fonts:suits/8".
*
* 3. Create a font contents file for the font. You can do this
*    by three methods:
*    a. Run the program "Workbench1.3:System/FixFonts" which
*       will create the file "Fonts:suits.font" automatically.
*    b. Use the NewFontContents() call in the diskfont library to
*       create a FontContentsHeader structure, which can be saved
*       in the Fonts: directory as "suits.font". This is essentially
*       what FixFonts does.
*    c. Create the file manually by following these directions:
*
*    Create a file in the Fonts: directory named "suits.font" the
*    contents of which are as follows:
*
*    The first word (two bytes) must contain the font header identifier, FCH_ID
*    FCH_ID is defined in "libraries/diskfont.i" and is currently 0x0f00.
*
*    The next word contains the number of FontContents entries. There will
*    be only one font descriptor file in this sample font's directory, so the
*    value 0x0001 should be used.
*
*    Follow this information with the hex values for the string "suits/8";
*    the path AmigaDOS should follow to reach this font size. The path is
*    relative to "Fonts:"; "suits" is the directory in which AmigaDOS will find
*    this font's font descriptor files and "8" is a font descriptor file.
*    This should be followed by enough null characters (0x00) to pad the
```

```

* pathname to MAXFONTPATH bytes in length. MAXFONTPATH is defined in
* "libraries/diskfont.i" and is currently 256 bytes. The string in this
* example is 7 bytes long, therefore 249 bytes of padding are needed.
* Note that the font pathname must have at least one null character after
* it, making the maximum pathname length MAXFONTPATH-1.
*
* The next word contains the font YSize; in this case, 0x0008.
*
* The next byte contains the font Flags, in this case 0x00.
*
* The last byte contains the font characteristics, in this case 0x60.
* This says it is a disk-based font (bit 1 set) and the font has been
* removed (bit 7 set), saying that the font is not currently resident.
*
* Summary of suits.font file:
*
* Name: fch_FileID fch_NumEntries fc_FileName      fc_YSize fc_Flags fc_Style
* Size: word      word          MAXFONTPATH bytes word      byte      byte
* Hex: 0f00      0001          73756974732F3800 0008      00        60
* ASCII:                s u i t s / 8 \0
*
* The correct length of a font file may be calculated with this formula:
* length := ((number of font contents entries) * (MAXFONTPATH+4)) + 4.
* In this case (one entry), this becomes (MAXFONTPATH + 8) or 264.
*
* To try out this example font, do the following.
* Start up the Notepad program or any other program which allows the
* user to select fonts. Choose the "suits" font in size 8 and type "abcd".
* This example font defines ASCII characters 'a' 'b' 'c' and 'd' only.
* All other characters map to a rectangle, meaning "character unknown".

```

```

INCLUDE      "exec/types.i"
INCLUDE      "exec/nodes.i"
INCLUDE      "libraries/diskfont.i"

```

```

* Provide an easy exit in case this file is "Run" instead of merely loaded.

```

```

MOVEQ      #-1,D0
RTS

```

```

* The following five entries comprise a Node structure, used by the system
* to link disk fonts into a list. See the definition of the "DiskFontHeader"
* structure in the "libraries/diskfont.i" include file for more information.

```

```

DC.L      0          ; ln_Succ
DC.L      0          ; ln_Pred
DC.B      NT_FONT    ; ln_Type
DC.B      0          ; ln_Pri
DC.L      fontName   ; ln_Name

DC.W      DFH_ID     ; FileID
DC.W      1          ; Revision
DC.L      0          ; Segment

```

```

* The next MAXFONTNAME bytes are a placeholder. The name of the
* font contents file (e.g. "suits.font") will be copied here after this
* font descriptor is LoadSeg-ed into memory. The Name field could have
* been left blank, but inserting the font name and size (or style) allows
* one to tell something about the font by using "Type OPT H" on the file.

```

```

fontName:
DC.B      "suits8"   ; Name

```

```

* If your assembler needs an absolute value in place of the "length"
* variable, simply count the number of characters in Name and use that.

```

```

length EQU      *-fontName      ; Assembler calculates Name's length.
DCB.B      MAXFONTNAME-length,0 ; Padding of null characters.

```

```

* The rest of the information is a TextFont structure.
* See the "graphics/text.i" include file for more information.

```

```

font:
    DC.L      0          ; ln_Succ
    DC.L      0          ; ln_Pred
    DC.B      NT_FONT    ; ln_Type
    DC.B      0          ; ln_Pri
    DC.L      fontName   ; ln_Name
    DC.L      0          ; mn_ReplyPort
    DC.W      0          ; (Reserved for 1.4 system use.)
    DC.W      8          ; tf_YSize
    DC.B      0          ; tf_Style
    DC.B      PPF_DESIGNED!PPF_PROPORTIONAL!PPF_DISKFONT ; tf_Flags
    DC.W      14         ; tf_XSize
    DC.W      6          ; tf_Baseline

* tf_Baseline must be no greater than tf_YSize-1, otherwise algorithmically-
* generated styles (italic in particular) can corrupt system memory.

    DC.W      1          ; tf_BoldSmear
    DC.W      0          ; tf_Accessors
    DC.B      97         ; tf_LoChar
    DC.B      100        ; tf_HiChar
    DC.L      fontData   ; tf_CharData
    DC.W      8          ; tf_Modulo, no. of bytes to add to the
                        ; data pointer to go from one row of
                        ; a character to the next row of it.
    DC.L      fontLoc    ; tf_CharLoc, bit position in the font
                        ; data at which the character begins.
    DC.L      fontSpace  ; tf_CharSpace
    DC.L      fontKern   ; tf_CharKern

* The four characters of this font define the four playing-card suit symbols.
* The heart, club, diamond, and spade map to the lower-case ASCII characters
* 'a', 'b', 'c', and 'd' respectively. The fifth entry in the table is the
* character to be output when there is no entry defined in the character set
* for the requested ASCII value.
*
*
*   97 (a)      98 (b)      99 (c)      100 (d)      255
*   <          X          X          X          X          >
*   .eee...eee. ....e..... ..e... ..e...eee... eeeeeeeeeeee
*   eeeee.eeeee. ....e..... ..e... ..e...eee... ee.....ee
*   .eeeeeeeeee. .eeeeeeeeee. .eeeeee. .ee..e..ee. ee.....ee
*   ..eeeeeeee.. eeeeeeeeeeee eeeeeeee eeeeeeeeeeee ee.....ee
*   ...eeeeee... .eee.e. .... .eeeeee. .ee..e..ee. ee.....ee
*   ..eeeeee... ..e..... ..e... ..e...eee... ee.....ee
*   .....e..... ..eeeeee... ..e... ..e...eee... eeeeeeeeeeee
*   .....
*
* Font data is bit-packed edge to edge to save space.

fontData:
    DC.W      $071C0,$08040,$070FF,$0F000
    DC.W      $0FB3,$0E0E0,$0F8C0,$03000
    DC.W      $07FCF,$0F9F3,$026C0,$03000
    DC.W      $03F9F,$0FFFF,$0FFC0,$03000
    DC.W      $01F0E,$0B9F3,$026C0,$03000
    DC.W      $00E00,$080E0,$020C0,$03000
    DC.W      $00403,$0E040,$0F8FF,$0F000
    DC.W      $00000,$00000,$00000,$00000
    DC.W      $00000,$00000,$00000,$00000

* The fontLoc information is used to "unpack" the fontData.
* Each pair of words specifies how the characters are bit-packed. For
* example, the first character starts at bit position 0x0000, and is 0x000B
* (11) bits wide. The second character starts at bit position 0x000B and
* is 0x000B bits wide, and so on. This tells the font handler how to unpack
* the bits from the array.

fontLoc:
    DC.L      $00000000B,$0000B000B,$000160007,$0001D000B,$00028000C

* fontSpace array: Use a space this wide to contain this character when it

```

\* is printed. For reverse-path fonts these values would be small or negative.

```
fontSpace:
    DC.W          000012,000012,000008,000012,000013
```

\* fontKern array: Place a space this wide after the corresponding character  
\* to separate it from the following character. For reverse-path fonts these  
\* values would be large negative numbers, approximately the width of the  
\* characters.

```
fontKern:
    DC.W          000001,000001,000001,000001,000001
```

```
fontEnd:
    END
```

## Wrapper code

The following code is meant to be used with the 'C' programming examples in this chapter. It contains routines needed by many of the examples, but is printed only once to save space. Insert a section of example code into the space indicated in the wrapper program, compile, link and run it.

```
/* Wrapper
   Support routines needed by Text chapter examples.
   Insert routine(s) where indicated below, then compile, link, and run.
   For Lattice, compile and link with: LC -bl -cfist -L -v -y wrapper.c
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <exec/nodes.h>
#include <graphics/text.h>
#include <graphics/rastport.h>
#include <graphics/gfxbase.h>
#include <intuition/intuitionbase.h>
#include <libraries/dos.h>
#include <libraries/diskfont.h>
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Library pointers.
   Pointers are initialized to NULL so that the Close...()
   and Free...() routines can tell whether the pointers were used.
*/

struct GfxBase *GfxBase = NULL;
struct IntuitionBase *IntuitionBase = NULL;
struct Library *DiskfontBase = NULL;

/* The Workbench pen color registers. */
#define COLOR0 0
#define COLOR1 1
#define COLOR2 2
#define COLOR3 3

/* do/don't wait for the user to click the close gadget before exiting. */
#define WAIT_FOR_CLOSE FALSE
#define DONT_WAIT TRUE

/*
```



```

    Set the titlebar of the specified Window (w) to the specified string (s).
*/
#define TITLE(w, s) (SetWindowTitles((w), (UBYTE *) (s), (UBYTE *) ~0))

/*
=====
                        INSERT EXAMPLE CODE AFTER HERE
=====
*/

/*
=====
                        INSERT EXAMPLE CODE BEFORE HERE
=====
*/

/*
    Open the graphics, intuition, and diskfont libraries.
    Return TRUE if completely successful, FALSE otherwise.
*/
BOOL openLibs(VOID)
{
    GfxBase = (struct GfxBase *) OpenLibrary("graphics.library", 33L);
    if (GfxBase)
    {
        IntuitionBase =
            (struct IntuitionBase *) OpenLibrary("intuition.library", 33L);
        if (IntuitionBase)
        {
            DiskfontBase = OpenLibrary("diskfont.library", 33L);
            if (DiskfontBase)
                return(TRUE);
        }
    }
    return(FALSE);
}

/*
    Open a half-height, full-width window on the Workbench screen.
    Return a Window pointer if successful, NULL otherwise.
*/
struct Window *doWindow(VOID)
{
    static struct NewWindow nw =
    {
        0, 0, 0, 0, -1, -1, CLOSEWINDOW,
        WINDOWCLOSE|WINDOWDEPTH|WINDOWDRAG|SMART_REFRESH|ACTIVATE|GIMMEZEROZERO,
        NULL, NULL, NULL, NULL, NULL, ~0, ~0, ~0, ~0, WBENCHSCREEN
    };
    struct Window *window;

    nw.Height = GfxBase->NormalDisplayRows / 2;
    nw.Width = GfxBase->NormalDisplayColumns;
    window = OpenWindow(&nw);
    return(window);
}

/*
    If requested, wait for the user to click on the Window's close gadget.
    Reply to any messages which may have accumulated, and close the Window.
*/
VOID undoWindow(struct Window *window, BOOL immediate)
{
    struct IntuiMessage *intuiMessage;
    if (window)
    {

```

```

    if (! immediate)
        (VOID) Wait(1<<window->UserPort->mp_SigBit);
    while (intuiMessage = (struct IntuiMessage *)GetMsg(window->UserPort))
        ReplyMsg((struct Message *)intuiMessage);
    CloseWindow(window);
}

/*
    Close any libraries opened in openLibs().
*/
VOID closeLibs(VOID)
{
    if (DiskfontBase)
        CloseLibrary(DiskfontBase);
    if (IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);
    if (GfxBase)
        CloseLibrary((struct Library *)GfxBase);
}

/*
    Open the libraries, open a window, and call the example routine.
    Returns one of several values to the calling environment:
    RETURN_OK if successful, RETURN_ERROR if unable to open a window,
    and RETURN_FAIL if unable to open the needed libraries.
*/
VOID main(int argc, char *argv[])
{
    int exitVal = RETURN_OK;
    BOOL immediate = TRUE;
    struct Window *window = NULL;

    if (openLibs())
    {
        if (window = doWindow())
            immediate = example(window);
        else
            exitVal = RETURN_ERROR;
        undoWindow(window, immediate);
    }
    else
        exitVal = RETURN_FAIL;
    closeLibs();
    exit(exitVal);
}

```

## Chapter 25

# Graphics: Sprites, Bobs and Animation

This chapter describes how to use the functions provided by the graphics library to manipulate the various Graphic Elements. It is divided into six sections:

- An overview of the animation system, including fundamental terms and structures
- Explanation of **Simple** (hardware) **Sprites** and an example showing their usage
- Explanation of **VSprites** and an example showing their usage
- Explanation of **Bobs** and an example showing their usage
- Discussion of topics that apply to all **Graphic Elements (GELs)** such as collision detection and user structure extension.
- Discussion of animation, using **AnimComps** and **AnimObs** and an example showing their usage

## Introduction — Some Terms

Before going into details, a quick glossary: **PlayField** is a term that encompasses the **View**, **ViewPort**, and **RastPort** structures. It refers to the background area that Sprites appear *over*, and Bobs appear *in*. PlayFields can be created and controlled at several levels. Please see the “Graphics Primitives” and “Layers” chapters for details on lower level PlayField control. The “Intuition” section should be examined for details on screens and windows, which give higher level access to PlayFields. The graphics animation functions provide several levels of control over the manipulation of Graphic Elements, or GELs. GELs are objects that can be moved easily around the display. The GELs system is compatible with all PlayField modes, including Dual-PlayField. GELs are defined by structures found in *graphics/gels.h*.

The GEL types are (in order of increasing complexity):

<b>VSprites</b>	for Virtual Sprites
<b>Bobs</b>	for Blitter Objects
<b>AnimComps</b>	for Animation Components

not really GELs, but described here:

<b>AnimObs</b>	for Animation Objects, are used to group AnimComps.
----------------	---

### Simple Sprites

**Simple Sprites** (also known as Hardware Sprites) are *not* part of the animation system (they are *not* GELs). If Simple Sprites and GELs are used in the same display, the GELs system must be told specifically which Simple Sprites to avoid. Simple Sprites are described in this chapter because they are the basis of VSprites.

The Amiga hardware has the ability to handle up to eight Sprite objects. Each Sprite is produced by one of the eight hardware Sprite DMA channels. They are 16-bits wide and arbitrarily tall. The Amiga software offers a choice of how to use these hardware elements. After a Sprite DMA channel has displayed the last line of a Sprite, the system can reuse the channel for a different Sprite lower on the screen. This is how VSprites are implemented.

### VSprites

The **Virtual Sprite** is the most elemental structure in the GEL system. It contains just a bit more information than is needed to define a Simple Sprite. If the **VSprite** structure turns out to be a true VSprite, the system temporarily assigns each VSprite to a Simple Sprite, as needed. This makes it appear to application code that it has a *virtually* unlimited supply of VSprites. Another function of the **VSprite** structure is to serve as the root of the more complex GEL types.

When the system is deciding what a given root VSprite structure *really* is, it does some simple tests:

- If the base **VSprite** structure has the **VSPRITE** flag set in its **Flags** field, the system knows this GEL is a true VSprite, and will convert it to a Simple Sprite to display it. If, however, the **VSPRITE** flag is not set, the system knows that this is not a Sprite based GEL, it is a Blitter based GEL.
- It then looks at the VSprite's **VSBob** pointer to find the Bob part of this GEL. If the Bob does not have the **BOBISCOMP** flag set, the system knows that this GEL is just a basic Bob, and uses the Blitter to display it.
- If the **BOBISCOMP** flag is set, the system knows to look at the Bob's **BobComp** pointer to find the **AnimComp** and **AnimOb** structures, which contain the variables needed for sequenced animation.

## Types Of Animation

Using the Amiga system tools, two fundamental kinds of image animation can be performed: **Sprite animation** and **PlayField animation**.

### Sprite Animation — VSprites

True **VSprites** are implemented by the system as **Simple Sprites**. So, rules that apply to Simple Sprites apply to true VSprites too. Sprites are not rendered into the underlying BitMap, and so do not affect any bits in the BitMap they appear to be over. Because they are hardware based, *they are positioned in the display absolutely, and are not affected by the movement of screens*. The starting position of a Sprite must not occur before line -20, because of certain hardware DMA time constraints. Sprites are also quick.

### PlayField Animation — Bobs and AnimComps

The *Blitter* is special Amiga hardware that is used to move data quickly and efficiently, optionally performing logical operations as it does. **Bobs** (and, therefore, **AnimComps**) are rendered using this device. In PlayField animation, the underlying PlayField (the RastPort's BitMap, actually) is modified. The background BitMap where the GEL was may need to be restored, the BitMap where it's going may need to be saved, and the GEL must be rendered into the BitMap. Doing this repeatedly, perhaps while changing the image or the entire Bob, creates an animation effect. The system uses pointers to link the **VSprite** and **Bob** structures, "extending" the **VSprite** structure to include Bobs.

### VSprites vs. Bobs

If you are going to manage the movement and sequencing of GELS yourself, you need to decide if VSprites or Bobs best suit your needs. If you've got simple requirements or lots of coding time, you may even opt to use only Simple Sprites, and control them yourself. If you elect to have the system manage your animations, *AnimComps must* be used, and they are Bobs at heart.

Some fundamental differences between Sprites and Bobs:

- Sprite images and coordinates are low resolution sized pixels, even on a high resolution display. Bob images and coordinates are the same resolution as the PlayField that they're rendered into.

- Sprites have a maximum width of 16 (low resolution) pixels. Attached Sprites have a maximum width of 32 pixels. Sprites may be displayed any width up to these maximums. Bobs can be any width. Display time creates a practical limit for Bob sizes. Large Bobs can have a strong impact on performance, due to the number of Blits required to move a Bob.
- The height of either Sprites or Bobs can be as tall as the display.
- Sprites have a maximum of three colors, or fifteen if they're attached. Because the system uses the Copper to control Sprite colors on the fly, the colors are not necessarily the same as those in the background PlayField. Bobs can use any or all of the colors in the background playfield. Limiting factors include playfield resolution and display time. Bobs with more colors take longer to display.
- Sprites are positioned using absolute display coordinates, and don't move with screens. Bobs follow screen movement.

In general: Sprites offer speed, Bobs offer flexibility.

## AnimComps

The **AnimComp** (for Animation Component) is a structure that has all the variables needed to implement simple sequenced animation. The system uses pointers to link the **Bob** and **AnimComp** structures, "extending" the Bob structure to include AnimComps. Part of the **AnimComp** structure includes pointers to other AnimComps. When many AnimComps are built, these pointers must be arranged so that the AnimComps form a list. Once so set up, the system will cycle through this list to achieve unattended animation, displaying each AnimComp in turn.

## AnimObs

The **AnimOb** (for Animation Object) is a structure that is used to provide an offset for one AnimComp or many AnimComps. For example; our AnimOb consists of two AnimComps, one that looks like a planet and another containing a sequence of AnimComps that describe orbiting moons. As the AnimOb moves the planet across our display, the moons travel along with it, orbiting the planet the entire time. If the x, y pair in the AnimOb is modified, the system automatically manages the movement of all the associated AnimComps. If they are to be moved separately, the planet AnimComp and the moon sequence must each have their own AnimOb.

## The GELs System

All GELs have a **VSprite** structure at their core. The system keeps track of all the GELs that it will display (the active GELs) by using a standard list structure to link their core VSprites. This list is accessed via **GelsInfo**, which is associated with the RastPort. The **GelsInfo** structure is defined in the file *graphics/rastport.h*.

When **AddGel()** is called to introduce a new GEL to the system, the GEL is linked into the GelsInfo list. The new GEL is added immediately ahead of the first existing GEL whose y, x value is greater than or equal to that of the new GEL, always trying to keep the list sorted.

As GELs are moved about the screen, their y, x values are constantly changing. **SortGList()** re-sorts this list by the y, x values.

Although this is a list of **VSprite** structures, bear in mind that some or all may really be Bobs or AnimComps.

The basic set up of the **GelsInfo** structure requires three important fields: **sprRsrvd**, **gelHead** and **gelTail**.

**sprRsrvd** tells the system which Hardware Sprites not to use when managing true VSprites.

**gelHead** and **gelTail** are **VSprite** structures that are used to manage the list of gels. They are never displayed. To activate or deactivate a GEL, a system call is made to add it to or delete it from this list.

Other fields must be set up to provide for collision detection, color optimization, and other features. A complete example for setting up the **GelsInfo** structure is given below.

The following figure is a visualization of the various GEL configurations.

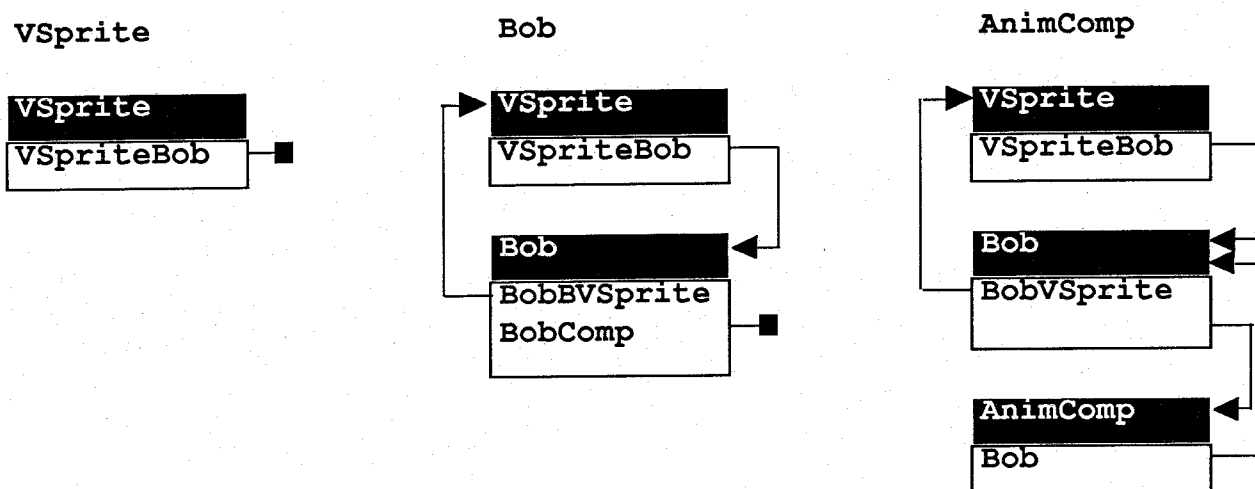


Figure 25-1: GEL Structure Layout

### Preparing To Use Graphics Animation

Because the animation functions have been designed to interact with a **PlayField**, one must be provided. This means that the system requires that access be provided to **View**, **ViewPort**, and **RastPort** structures. These structures may be allocated and initialized in one of several ways. For the bulk of the examples provided, the **Intuition** library is used for this purpose.

### Initializing the GEL System

To initialize the animation system, call the system function **InitGels()**. It takes the form:

```

struct VSprite *vsHead;
struct VSprite *vsTail;
struct GelsInfo *gInfo;

InitGels(vsHead, vsTail, gInfo);

```

where

**vsHead** is a pointer to the **VSprite** structure to be used as the GEL list head.

**vsTail** is a pointer to the **VSprite** structure to be used as the GEL list tail.

**gInfo** is a pointer to the **GelsInfo** structure to be initialized.

**InitGels()** forms these structures into a linked list of GELs that is empty except for these two dummy elements. It gives the head **VSprite** the maximum negative y and x positions and the tail **VSprite** the maximum positive y and x positions. This is to aid the system in keeping the list sorted by y, x values, so GELs that are closer to the top and left of the display are nearer the head of the list. The memory space that the **VSprites** and **Gelsinfo** structures take up must already have been allocated. This can be done either by declaring them statically or explicitly allocating memory for them.

Here is a sample function that sets up the **GelsInfo** structure through explicit allocation. This is a fragment of a larger file called "animtools.c". See the complete animation example at the end of this chapter for a complete listing of this file:

```

/*-----
** set up the gels system. After this call is made you can use
** vsprites, bobs, anim comps, and anim obs.
**
** note that this links the GelsInfo structure into the rast port,
** and calls InitGels( ).
**
** all resources are properly freed on failure.
**
** It uses information in your RastPort structure to establish
** boundary collision defaults at the outer edges of the raster.
**
** This function sets up for everything - collision detection and all.
**
** You must already have run LoadView before setupGelsSys is called.
*/
struct GelsInfo *setupGelsSys(struct RastPort *rPort, BYTE reserved)
{
    struct GelsInfo *gInfo;
    struct VSprite *vsHead;
    struct VSprite *vsTail;

    if (NULL != (gInfo =
        (struct GelsInfo *)AllocMem((LONG)sizeof(struct GelsInfo), MEMF_CLEAR)))
    {
        if (NULL != (gInfo->nextLine =
            (WORD *)AllocMem((LONG)sizeof(WORD) * 8, MEMF_CLEAR)))
        {
            if (NULL != (gInfo->lastColor =
                (WORD **)AllocMem((LONG)sizeof(LONG) * 8, MEMF_CLEAR)))
            {
                if (NULL != (gInfo->collHandler =
                    (struct collTable *)AllocMem((LONG)sizeof(struct collTable),
                        MEMF_CLEAR)))
                {
                    if (NULL != (vsHead = (struct VSprite *)AllocMem(
                        (LONG)sizeof(struct VSprite), MEMF_CLEAR)))
                    {
                        if (NULL != (vsTail = (struct VSprite *)AllocMem(
                            (LONG)sizeof(struct VSprite), MEMF_CLEAR)))

```



```

    {
        gInfo->sprRsrvd   = reserved;
        gInfo->leftmost   = 0;
        gInfo->rightmost  =
            (rPort->BitMap->BytesPerRow << 3) - 1;
        gInfo->topmost    = 0;
        gInfo->bottommost = rPort->BitMap->Rows - 1;

        rPort->GelsInfo = gInfo;

        InitGels(vsHead, vsTail, gInfo);

        return(gInfo);
    }
    FreeMem(vsHead, (LONG)sizeof(*vsHead));
}
FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
}
FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
}
FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
}
FreeMem(gInfo, (LONG)sizeof(*gInfo));
}
return(NULL);
}

```

Once the **GelsInfo** structure has been allocated and initialized, Gels can be added to the system. To remove and free the **GelsInfo** structure, use code like the following. This is a fragment of a larger file called "animtools.c". See the complete animation example at the end of this chapter for a complete listing of this file:

```

/*-----
** free all of the stuff allocated by setupGelsSys( ).
** only call this function if setupGelsSys( ) returned successfully.
** the GelsInfo structure is the one returned by setupGelsSys( ).
**
** It also unlinks the GelsInfo from the RastPort.
*/
VOID cleanupGelsSys(struct GelsInfo *gInfo, struct RastPort *rPort)
{
    rPort->GelsInfo = NULL;

    FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
    FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
    FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
    FreeMem(gInfo->gelHead, (LONG)sizeof(struct VSprite));
    FreeMem(gInfo->gelTail, (LONG)sizeof(struct VSprite));
    FreeMem(gInfo, (LONG)sizeof(*gInfo));
}

```

## Using Simple (Hardware) Sprites

First the structure is described, then the system functions that manipulate and display Simple Sprites will be discussed.

### NOTE

See the *Amiga Hardware Reference Manual* for a more complete description of Simple Sprites, including information on attached Sprites.

This **SimpleSprite** structure is found in *graphics/sprite.h*. It has fields that indicate the height and position of the Sprite and a number that indicates its associated hardware Sprite.

Simple Sprites are always 16 bits wide, which is why there is no width member in the **Sprite** structure. As mentioned in the introduction, Sprites *always* appear as low resolution pixels, and their position is specified in the same way. If the Sprite is being moved across a high-resolution display in single pixel increments, it will appear to move two pixels for each increment. In low-resolution mode, single pixel movement will be seen. Similarly, in an interlaced display, the y direction motions are in two line increments. The same image of the Sprite is placed into both even and odd fields of the interlaced display, so the Sprite will appear to be the same size in any display mode.

The upper left corner of the **ViewPort** area has coordinates (0,0). The motion of the Sprite is relative to this position.

The Sprite pairs 0/1, 2/3, 4/5, and 6/7 share color registers. Please see *VSprite Advanced Topics*, later in this chapter, for precautions to take if Simple Sprites and VSprites are used at the same time.

The following figure shows which color registers are used by Sprites.

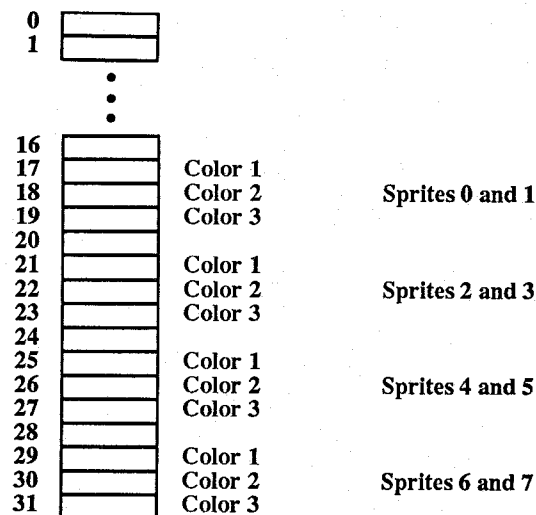


Figure 25-2: Sprite Color Registers

Sprites do not have *exclusive* use of the color registers. If the ViewPort is 5 bit-planes deep, all 32 of the system color registers will still be used by the PlayField display hardware.

#### NOTE

Color zero for all Sprites is always a “transparent” color, and the colors in registers 16, 20, 24, and 28 are not used by Sprites. These colors will be seen only if they are rendered into a PlayField. For further information, see the *Amiga Hardware Reference Manual*.

If there are two ViewPorts with different color sets on the same display, a Sprite will switch colors when it is moved across their boundary. For example, Sprite 0 and 1 will appear in colors 17-19 of whatever ViewPort they happen to be over. This is because the system jams *all* the ViewPort’s colors into the display hardware at the top of each ViewPort.

To use Simple Sprites, fill in their data structures and use the following functions:

- GetSprite()** attempts to allocate a Sprite for exclusive use
- ChangeSprite()** modifies the Sprite's appearance
- MoveSprite()** changes the Sprite's position
- FreeSprite()** returns the Sprite to the virtual Sprite machine

These functions are described in detail in the following paragraphs.

To use these Simple Sprite functions or the VSprite functions, the **SPRITE** flag must have been set in the **NewScreen** structure for **OpenScreen()**. If Intuition is not being used, this flag must be specified in the **View** and **ViewPort** data structures before **MakeViewPort()** is called.

### Controlling Sprite DMA

The graphics macros **ON\_SPRITE** and **OFF\_SPRITE** can be used to control Sprite DMA. **OFF\_SPRITE** prevents the system from displaying any hardware Sprites, whether Simple Sprites or VSprites. **ON\_SPRITE** restores the Sprite display.

#### NOTE

The Intuition pointer is a Sprite. Thus, if **OFF\_SPRITE** is used, Intuition's pointer will disappear too. Use care when calling **OFF\_SPRITE**. The macro turns off sprite DMA, so that no new sprite data is fetched. Whatever sprite data was being displayed at this point is displayed for EVERY line on the screen. This may lead to a vertical color bar if a Sprite is being displayed when **OFF\_SPRITE** is called.

### Accessing A Hardware Sprite

**GetSprite()** is used to gain access to a new hardware Sprite. **GetSprite()** allocates a hardware Sprite for exclusive use. The VSprite allocator can no longer assign this Sprite. Please see the *Includes & Autodocs Manual* for details on parameters and error returns. The call is made like this:

```
struct SimpleSprite *sprite;
SHORT                number, sprite_num;

if (-1 == (sprite_num = GetSprite(sprite, number)))
    return_code = RETURN_WARN; /* did not get the sprite */
```

The inputs to the **GetSprite()** function are:

- Sprite** A pointer containing the address of a **SimpleSprite** structure.
- number** The number (0-7) of the hardware Sprite to be accessed. If **number** is -1, the system gets the first available Sprite. If **number** is -1, a returned value of -1 means that all Sprites are allocated.

A value of 0-7 is returned if the request was granted, specifying which Sprite was allocated. A returned value of -1 means that this Sprite is already allocated.

## Changing The Appearance Of A Simple Sprite

The **ChangeSprite( )** function can completely alter the appearance of a Simple Sprite. **ChangeSprite( )** substitutes a new data content for that currently used to display a Simple Sprite. It is called by the following sequence:

```
struct ViewPort      *vp;
struct SimpleSprite *sprite;
APTR                 newdata;

ChangeSprite(vp, sprite, newdata);
```

The inputs to this function are:

- vp**            A pointer to the ViewPort for this Sprite or 0 if this Sprite is relative only to the current View.
- sprite**       A pointer to a SimpleSprite structure.
- newdata**      A pointer to a data structure containing the new data to be used. The data must reside in chip (MEMF\_CHIP) memory.

The structure for the new data is shown below, and can be found in the *Includes & Autodocs Manual*.

```
struct spriteimage
{
    /* position and control information for this Sprite */
    UWORD posctl[2];
    /* two words per line of Sprite height, first of the two words contains
     * MSB for color selection, second word contains LSB (colors 0,1,2,3
     * from allowable color register selection set). Color '0' for any
     * Sprite pixel makes it transparent.
     */
    UWORD data[height][2];                      /* actual Sprite image */

    /* reserved, initialize to 0, 0 */
    UWORD reserved[2];
};
```

## Moving A Simple Sprite

**MoveSprite( )** repositions a Simple Sprite. After this function is called, the Simple Sprite is moved to a new position relative to the upper left corner of the ViewPort. It is called as follows:

```
struct ViewPort      *vp;
struct SimpleSprite *sprite;
SHORT                x, y;

MoveSprite(vp, sprite, x, y);
```

The inputs to **MoveSprite( )** are as follows:

- vp**            A pointer to the ViewPort with which this Sprite interacts or 0 if this Sprite's position is relative only to the current View.

<b>sprite</b>	A pointer to a SimpleSprite structure
<b>x, y</b>	Pixel position to which a Sprite is to be moved.

## Relinquishing A Simple Sprite

The `FreeSprite()` function returns an allocated Sprite to the Virtual Sprite machine. The virtual Sprite machine can now reuse this Sprite to allocate Virtual Sprites. The syntax of this function is

```
SHORT sprite_number;
FreeSprite(sprite_number);
```

where `sprite_number` is the number (0-7) of the Sprite to be returned to the system.

### NOTE

Sprites *must* be freed if allocated using `GetSprite()`. If they are not freed when the task ends, the system will have no way of reallocating those Sprites until the system is rebooted.

## Complete Sprite Example

The following example demonstrates how to move a Simple Sprite.

```
/* ssprite.c 19oct89
** lattice c 5.04
** lc -bl -cfist -v -y ssprite.c
** blink FROM LIB:c.o ssprite.o LIB LIB:lc.lib LIB:amiga.lib TO ssprite
*/
#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/sprite.h>
#include <hardware/custom.h>
#include <hardware/dmabits.h>
#include <libraries/dos.h>

#include <stdlib.h>
#include <proto/all.h>

struct GfxBase *GfxBase = NULL;
extern struct Custom far custom;

/* real boring sprite data */
UWORD chip_sprite_data[ ] =
{
    0, 0, /* position control */
    0xffff, 0x0000, /* image data line 1, color 1 */
    0xffff, 0x0000, /* image data line 2, color 1 */
    0x0000, 0xffff, /* image data line 3, color 2 */
    0x0000, 0xffff, /* image data line 4, color 2 */
    0x0000, 0x0000, /* image data line 5, transparent */
    0x0000, 0xffff, /* image data line 6, color 2 */
    0x0000, 0xffff, /* image data line 7, color 2 */
    0xffff, 0xffff, /* image data line 8, color 3 */
    0xffff, 0xffff, /* image data line 9, color 3 */
    0, 0 /* reserved, must init to 0 0 */
};
```

```

VOID main(int argc, char **argv)
{
    struct SimpleSprite    sprite;
    struct ViewPort        *viewport;

    SHORT sprite_num;
    SHORT delta_move;
    SHORT ktr1;
    SHORT ktr2;
    SHORT color_reg;

    int return_code;

    return_code = RETURN_OK;

    if (NULL == (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L)))
        return_code = RETURN_FAIL;
    else
    {
        /* opened library, need a viewport to do a sprite
        ** in real life you use the viewport of the screen that you open
        ** for your application.
        */
        viewport = GfxBase->ActiView->ViewPort ;

        if (-1 == (sprite_num = GetSprite(&sprite, 2)))
            return_code = RETURN_WARN;
        else
        {
            /* got a sprite.
            ** calculate the correct base color register number and
            ** set up the color registers.
            */
            color_reg = 16 + ((sprite_num & 0x06) << 1);
            SetRGB4(viewport, (long)color_reg + 1, 12, 3, 8);
            SetRGB4(viewport, (long)color_reg + 2, 13, 13, 13);
            SetRGB4(viewport, (long)color_reg + 3, 4, 4, 15);

            sprite.x = 0;          /* initialize position and size info */
            sprite.y = 0;          /* to match that shown in sprite_data */
            sprite.height = 9;     /* so system knows layout of data later */

            /* install sprite data and move sprite to start position. */
            ChangeSprite(NULL, &sprite, sprite_data);
            MoveSprite(NULL, &sprite, 30, 0);

            /* move the sprite to and fro */
            for ( ktr1 = 0, delta_move = 1;
                  ktr1 < 6;
                  ktr1++, delta_move = -delta_move)
            {
                for ( ktr2 = 0; ktr2 < 185; ktr2++)
                {
                    MoveSprite(NULL, &sprite, (LONG)(sprite.x + delta_move),
                               (LONG)(sprite.y + delta_move));
                    WaitTOF(); /* one move per video frame */
                    if (ktr2 == 80)
                        OFF_SPRITE ;
                    if (ktr2 == 100)
                        ON_SPRITE ;
                }
            }

            /* NOTE: if you turn off the sprite at the wrong time
            ** (when it is being displayed), the sprite will appear as
            ** a vertical bar on the screen. To really get rid of the
            ** sprite, you must OFF_SPRITE while it is not displayed.
            ** this is hard in a multi-tasking system (the solution is
            ** not addressed in this program).
            */
            ON_SPRITE ; /* just to be sure */
            FreeSprite((LONG)sprite_num);
        }
    }
}

```

```

    CloseLibrary((struct Library *)GfxBase);
}
exit(return_code);
}

```

## True VSprites

The following paragraphs describe how to set up the **VSprite** structure so that it represents a true **VSprite**. True **VSprites** are managed by the system, which converts them to Simple Sprites and displays them. The restrictions imposed by hardware Sprites on **VSprites** are overcome by Bobs. Later sections describe how a **VSprite** structure must be set up for Bobs and AnimComps.

## VSprite Setup

Before the system is told of the **VSprite**'s existence, space for the structure must be allocated and set up to *correctly* represent something the system recognizes as a **VSprite**. The importance of giving correct and complete structures to the system cannot be overstressed; because speed is of the essence, no structure cohesiveness or validity checking is done. The result of handing the system a bogus structure is usually a fireworks display, followed by a system failure.

The system software provides a way to detect collisions between **VSprites** and other on-screen objects. There is also a method of extending the **VSprite** structure so that it incorporates user defined variables. These subjects are applicable to all GELs, and are explained in the "Common GEL Topics" sub-section below.

### SPECIFICATION OF VSPRITE STRUCTURE

Assuming that the "C" programming language is being used, there are two primary ways to allocate and fill in space for **VSprites**. They can be statically declared, or a memory allocation function can be called, and they can be filled in programmatically. The method used is entirely up to you, as long as they're complete when the system sees them.

The declaration to statically set up a **VSprite** structure is listed below. The **VSprite** structure definition was taken from the *Includes & Autodocs Manual*; please see that book for a fully commented definition:

```

/* VSprite structure definition */
struct VSprite
{
    struct VSprite *NextVSprite;
    struct VSprite *PrevVSprite;
    struct VSprite *DrawPath;
    struct VSprite *ClearPath;
    WORD            OldY, OldX;
    WORD            Flags;
    WORD            Y, X;
    WORD            Height;
    WORD            Width;
    WORD            Depth;
    WORD            MeMask;
    WORD            HitMask;
    WORD            *ImageData;
    WORD            *BorderLine;
    WORD            *CollMask;
}

```

```

        WORD            *SprColors;
        struct Bob      *VSBob;
        BYTE            PlanePick;
        BYTE            PlaneOnOff;
        VUserStuff      VUserExt;
    };

/* VSprite static data definition.
** must set the following for TRUE VSprites:
**     VSPRITE flag.
**     Width to 1.
**     Depth to 2.
**     VSBob to NULL.
**/
struct VSprite myVSprite =
{
    NULL, NULL, NULL, NULL, 0, 0, VSPRITE, 0, 0, 5, 1, 2, 0, 0,
    &myImage, 0, 0, &mySpriteColors, NULL, 0x3, 0, 0
};

```

This static allocation gives the required VSprite structure, but does not allocate or set up collision masks for the VSprite.

Here is a function to dynamically allocate and initialize a VSprite structure. This is a fragment of a larger file called "animtools.c". See the complete animation example at the end of this chapter for a complete listing of this file:

```

/* define a new structure for makeVSprite( ).
** This can be initialized and passed to makeVSprite( ) to create
** a new VSprite.
**
** data structure to hold information for a new vsprite.
** note that:
**     NEWVSPRITE myNVS;
** is equivalent to:
**     struct newVSprite myNVS;
**/
typedef struct newVSprite
{
    WORD            *nvs_Image;          /* image data for the vsprite */
    WORD            *nvs_ColorSet;       /* color array for the vsprite */
    SHORT           nvs_WordWidth;       /* width in words */
    SHORT           nvs_LineHeight;      /* height in lines */
    SHORT           nvs_ImageDepth;      /* depth of the image */
    SHORT           nvs_X;               /* initial x position */
    SHORT           nvs_Y;               /* initial y position */
    SHORT           nvs_Flags;           /* vsprite flags */
} NEWVSPRITE;

/*-----
** create a VSprite from the information given in nVSprite.
** use freeVSprite( ) to free this gel.
**/
struct VSprite *makeVSprite(NEWVSPRITE *nVSprite)
{
    struct VSprite *vsprite;
    LONG           line_size;
    LONG           plane_size;

    line_size = (LONG)sizeof(WORD) * nVSprite->nvs_WordWidth;
    plane_size = line_size * nVSprite->nvs_LineHeight;

    if (NULL != (vsprite =
        (struct VSprite *)AllocMem((LONG)sizeof(struct VSprite), MEMF_CLEAR)))
    {
        if (NULL != (vsprite->BorderLine =
            (WORD *)AllocMem(line_size, MEMF_CHIP)))
        {
            if (NULL != (vsprite->CollMask =
                (WORD *)AllocMem(plane_size, MEMF_CHIP)))

```



```

    {
        vsprite->Y          = nVSprite->nvs_Y;
        vsprite->X          = nVSprite->nvs_X;
        vsprite->Flags      = nVSprite->nvs_Flags;
        vsprite->Width      = nVSprite->nvs_WordWidth;
        vsprite->Depth      = nVSprite->nvs_ImageDepth;
        vsprite->Height     = nVSprite->nvs_LineHeight;
        vsprite->MeMask     = 1;
        vsprite->HitMask    = 1;
        vsprite->ImageData  = nVSprite->nvs_Image;
        vsprite->SprColors  = nVSprite->nvs_ColorSet;
        vsprite->PlanePick  = 0x00;
        vsprite->PlaneOnOff = 0x00;

        InitMasks(vsprite);
        return(vsprite);
    }
    FreeMem(vsprite->BorderLine, line_size);
}
FreeMem(vsprite, (LONG)sizeof(*vsprite));
}
return(NULL);
}

```

To free a VSprite that has been allocated with the above function, use the function below. This is a fragment of a larger file called "animtools.c". See the complete animation example at the end of this chapter for a complete listing of this file:

```

/*-----
** free the data created by makeVSprite( )
**
** assumes images deallocated elsewhere.
**/
VOID freeVSprite(struct VSprite *vsprite)
{
    LONG    line_size;
    LONG    plane_size;

    line_size = (LONG)sizeof(WORD) * vsprite->Width;
    plane_size = line_size * vsprite->Height;

    FreeMem(vsprite->BorderLine, line_size);
    FreeMem(vsprite->CollMask, plane_size);

    FreeMem(vsprite, (LONG)sizeof(*vsprite));
}

```

The following paragraphs fill in and discuss each member in turn.

#### NOTE

The VSprite structure does not need to reside in CHIP memory. Also the use of MEMF\_CLEAR lets one rely on the structure members being pre-set to NULL or 0, saving set-up code, though the examples given here will be explicit.

#### RESERVED VSPRITE MEMBERS

These VSprite structure members are reserved for system use:

**NextVSprite and PrevVSprite**      These are used as links in the GelsInfo list.

## **DrawPath and ClearPath**

These are used for Bobs, not true VSprites.

## **OldY and OldX**

Previous position holder, the system uses these for double buffered Bobs, but application programs can read them too.

The values can be set like this:

```
myVSprite.NextVSprite = NULL;  
myVSprite.PrevVSprite = NULL;  
myVSprite.DrawPath = NULL;  
myVSprite.ClearPath = NULL;  
myVSprite.OldY = 0;  
myVSprite.OldX = 0;
```

## **USING VSPRITE FLAGS**

The Flags member of the VSprite structure is both read and written by the system. Some bits are used by the application to inform the system; others are used by the system to indicate things to the application. The only bits that are used by true VSprites are:

### **VSPRITE**

This may be set to indicate to the system that it should treat the structure as a true VSprite, not part of a Bob. This affects the interpretation of the data layout and the use of various system variables.

### **VSOVERFLOW**

The system sets this bit in the true VSprites that it is unable to display. This happens when there are too many in the same scan line, and the system has run out of Simple Sprites to assign. It indicates that this VSprite has not been displayed. If no Sprites are reserved, this means that more than eight Sprites touch one scan line. This bit will not be set for Bobs, and should not be changed by the application.

### **GELGONE**

If the system has set GELGONE bit in the Flags member, then the GEL associated with this VSprite is not on the display at all, it is entirely outside the GEL boundaries. This area is defined by the GelsInfo members — **topmost**, **bottommost**, **leftmost**, and **rightmost**, defined in *graphics/rastport.h*.

On the basis of that information, the application may decide that the object need no longer be part of the GEL list and may decide to remove it to speed up the consideration of other objects. Use **RemVSprite( )** (or **RemBob( )**, if it's a Bob) to do this. This bit should not be changed by the application.

The value may be set like this:

```
myVSprite.Flags = VSPRITE;
```

## **VSPRITE POSITION**

To control the position of a VSprite, the y and x variables in the VSprite structure are used. These specify where the upper left corner of the VSprite will be, relative to the upper left corner of the drawing area it appears over. So if VSprites are used under Intuition and within a screen, they will be positioned relative to the upper left-hand corner of the screen. In a 320 by 200 screen, a y value of 0 puts the Sprite at the top of that display, a y value of (200 - Sprite height) puts the Sprite at the bottom. And an x value of 0 puts the Sprite at the left of that display, an x value of (320 - Sprite width) puts the Sprite at the right. Values of less than (0,0) or greater than (200, 320) may be used

to move the VSprite partially or entirely off the screen, if desired. See the “Graphics Primitives” chapter for more information on display coordinates and “Normal” display size. See the *Hardware Reference Manual* for more information on hardware sprites.

#### NOTE

It is important that the starting position of true VSprites is not less than -20 in the y direction, which is the start of the active display area for Sprites. Also, if they are moved too far to the left, true VSprites may not have enough DMA time to be displayed.

The values may be set like this to put it in the upper-left:

```
myVSprite.Y = 0;  
myVSprite.X = 0;
```

#### VSPRITE IMAGE SIZE

A true VSprite is always one word (16 pixels) wide and may be any number of lines high. It can be made to appear thinner by making some pixels transparent. Like Simple Sprites, VSprite pixels are always the size of a pixel in low-resolution mode (320x200); regardless of the resolution the display is set to. To specify how many lines make up the VSprite image, the VSprite structure member, **Height**, is used. VSprites always have a **Depth** of two, allowing for three colors. The values may be set like this:

```
myVSprite.Width  = 1;      /* ALWAYS 1 for true VSprites. */  
myVSprite.Height = 5;      /* The example height. */  
myVSprite.Depth  = 2;      /* ALWAYS 2 for true VSprites. */
```

#### VSPRITE MEMBERS FOR COLLISION DETECTION

The following members are all used for collision detection. To set up for no collision detection, zero them:

```
myVSprite.HitMask    = 0;  
myVSprite.MeMask     = 0;  
myVSprite.BorderLine = 0;  
myVSprite.CollMask   = 0;
```

The **VUserExt** member is for user extension of the VSprite structure. To ignore it for now, set it to zero too:

```
myVSprite.VUserExt = 0;
```

These are all explained in the “Common GEL Topics” sub-section below.

The **VSBob** member is not used for true VSprites. It is used for the more complex GELs, and is explained in the next sub-section. For use with VSprites:

```
myVSprite.VSBob = NULL;
```

#### VSPRITE IMAGEDATA

The **ImageData** pointer must be initialized with the address of the first word of the image data array. The image data array must be in chip memory. It takes two sequential 16-bit words to define each line of a true VSprite. This means that the data area containing the VSprite image is always *Height x 2* (= 10 in the example case) words long.

A VSprite image is defined just like a real hardware Sprite.

The combination of bits in corresponding locations in the two data words that define each line select the color for that pixel. The first of the pair of words supplies the low-order bit of the color selector for that pixel; the second word supplies the high-order bit.

These binary values select colors as follows:

- 00 - selects "transparent"
- 01 - selects the first of three VSprite colors
- 10 - selects the second VSprite color
- 11 - selects the third VSprite color

In those areas where the combination of bits yields a value of 0, the VSprite is transparent. The background, all Bobs and AnimComps, and any VSprite whose priority is lower than this VSprite will show through in transparent sections. For example:

```
(&VSprite->ImageData)    1010 0000 0000 0000
(&VSprite->ImageData + 1) 0110 0000 0000 0000
```

Reading from left to right, the combinations of these two sequential data words form the binary values of 01, 10, 11, and then all 00s. This VSprite's first pixel will be color 1, the next color 2, the third color 3. The rest will be transparent, making this VSprite appear to be three pixels wide. Thus, a three-color image, with some transparent areas, can be formed from a data set like the following sample:

#### VSprite Data

mem	1111 1111 1111 1111	Defines top line — colors selected
mem + 1	1111 1111 1111 1111	3333 3333 3333 3333
mem + 2	0011 1100 0011 1100	
mem + 3	0011 0000 0000 1100	0033 1100 0011 3300
mem + 4	0000 1100 0011 0000	
mem + 5	0000 1111 1111 0000	0000 3322 2233 0000
mem + 6	0000 0010 0100 0000	
mem + 7	0000 0011 1100 0000	0000 0032 2300 0000
mem + 8	0000 0001 1000 0000	Defines last line —
mem + 9	0000 0001 1000 0000	0000 0003 3000 0000

The VSprite Height for this sample image is 5.

#### SPECIFYING THE COLORS OF A VSPRITE

The system software provides a great deal of versatility in the choice of colors for Virtual Sprites. Each VSprite has *its own set* of three colors, pointed to by **SprColors**, which the system jams into the display's copper list as needed.

**SprColors** points to the first of three 16-bit values. The first value represents the color used for the VSprite bits that select color 1, the second value is color 2, and the third value is color 3. When the system assigns a hardware Sprite to carry the VSprite's image, it jams these color values into the Copper list (the intermediate Copper list, NOT the color table), so that the View's colors will be correct for this Sprite at the time the Sprite is displayed. It doesn't jam the original palette's colors back after the VSprite is done. If there is another VSprite later, that VSprite's colors will get jammed; if there is not another VSprite, the colors will remain the same until the next ViewPort's colors get loaded.

If the **SprColors** pointer is set to **NULL**, that VSprite does not generate a color-change instruction stream for the Copper. Instead, the VSprite appears drawn in whatever color set that hardware Sprite happens to have in it already.

Since the registers are initially loaded with the colors from the ViewPort's ColorMap, if all VSprites have **NULL SprColors**, they will appear in the ViewPort's colors.

To continue our example, a set of colors can be declared with the statement:

```
WORD mySpriteColors[] = { 0x0000, 0x00f0, 0x0f00 };
```

and set the VSprite colors with the statement:

```
myVSprite.SprColors = mySpriteColors;
```

Please see "Advanced Topics" below, for a much more detailed description.

## ADDING A VSPRITE

Once a true VSprite has been initialized, the obvious next step is to give it to the system by adding it to the GEL list. A typical system call for this purpose follows:

```
struct VSprite myVSprite;  
struct RastPort myRastPort;  
  
AddVSprite(&myVSprite, &myRastPort);
```

where **myVSprite** is the fully initialized **VSprite** structure and **myRastPort** is the **RastPort** with which it is to be associated. Please see the *Includes & Autodocs Manual* for much more detailed description of this, and all the following functions.

## REMOVING A VSPRITE

To remove a VSprite from the list of controlled objects, use the system function **RemVSprite()**. This function takes the following form:

```
struct VSprite myVSprite;  
  
RemVSprite(&myVSprite);
```

where **&myVSprite** is a pointer to the VSprite to be removed from the GEL list. Don't try to **RemVSprite()** a VSprite that has not been added to the system with **AddVSprite()**.

## CHANGING VSPRITES

Once the VSprite has been added to the GEL list and is in the display, some of its characteristics can be changed dynamically by:

- Changing y, x to a new VSprite position
- Changing ImageData to point to a new VSprite image
- Changing SprColors to point to a new VSprite color set

Study the next two sections to find out how to reserve hardware Sprites for use outside the VSprite system and how to assign the VSprites.

## VSPRITE ADVANCED TOPICS

### VSPRITE MACHINE

This section describes advanced topics pertaining to VSprites. It contains details about reserving hardware Sprites for use outside of the VSprite system, information about how VSprites are assigned, and more information about VSprite colors.

#### Reserving Hardware Sprites

To prevent the VSprite system from using specific hardware Sprites, set the **sprRsrvd** member of the **GelsInfo** structure. The pointer to the **GelsInfo** structure is contained in the **RastPort** structure. If the contents of this 8-bit value are zero, then all of the hardware Sprites may be used by the VSprite system. If any of the bits is a 1, the Sprite corresponding to that bit will not be utilized by VSprites.

#### NOTE

Reserving sprites increases the likelihood of the system not being able to display a VSprite (VSOVERFLOW).

See the next section, "How VSprites are Assigned," for further details on this topic.

To reserve Sprite zero only, set **sprRsrvd** to 0x01;  
to reserve Sprite one only, set **sprRsrvd** to 0x02;  
Etc.

An application program would typically include the following kinds of statements:

```
struct RastPort myRastPort;    /* the View structure is defined */  
myRastPort.GelsInfo->sprRsrvd = 0x03;    /* reserve 0 and 1 */
```

If a hardware Sprite is reserved, the system will not consider that hardware Sprite when it makes VSprite assignments. Remember, hardware Sprite pairs share color register sets. If a hardware Sprite is reserved, its mate should probably be reserved too, otherwise the reserved Sprite's colors will change as the unreserved mate is assigned different VSprites.

For example, it is common practice to reserve Sprites 0 and 1, so that the Intuition pointer (Sprite 0) is left alone.

The **GfxBase** structure may be examined to find which Sprites are already in use. This may, at your option, impact what Sprites you reserve. If Intuition is running, Sprite 0 will already be in use as its pointer.

The reserved Sprite status is accessible as

```
currentreserved = GfxBase->SpriteReserved;
```

The next section presents a few trouble-shooting techniques for VSprite assignment.

### How VSprites Are Assigned

Although VSprites are Sprites managed for you, some underlying limitations are still felt, and the system *may* run out of Sprites.

Here are some reasons that the VSprite system can appear to have problems, and some suggestions on how to avoid them. There are 8 *real* Sprite DMA channels.

As the system goes through the GEL list during **DrawGList()**, whenever it finds a true VSprite, it goes through the following procedure. If there is a Simple Sprite available (after the reserved Sprites and preceeding VSprites are accounted for), copper instructions are added that will load the Sprite hardware with this VSprite's data at the right point on the screen. It may need to add a Copper instruction sequence to load the display's colors associated with the Sprite as well.

The system will run out of hardware Sprites if it is asked to display more than eight VSprites on one scan line. This limit goes down to four when the VSprites have different SprColor pointers. During the time that there is a conflict, the VSprites that could not be put into Simple Sprites will disappear. They will reappear when (as the VSprites are moved about the screen) circumstances permit.

One can alleviate these problems by taking some precautions:

- Minimize the number of VSprites to appear on a single horizontal line.
- If colors for some Virtual Sprites are the same, make sure that the pointer for each of the VSprite structures for these Virtual Sprites points to the same memory location, rather than to a duplicate set of colors elsewhere in memory. The system will know to map these into Sprite pairs.

If a VSprite's SprColors are set to NULL, the VSprite will appear in the ViewPort's ColorMap colors. If SprColors points to a color set, the system will jam SprColors into the display hardware (via the Copper list), effectively overriding those ColorMap registers. The values in the ColorMap are *not* overwritten, but anything in the background display that used to appear in the ColorMap colors will appear in SprColors colors.

The system will display a VSprite in any one of a set of four different possible color groupings as indicated in the Simple Sprite sub-section above, if SprColors is NULL.

### How VSprite and PlayField Colors Interact

At the start of each display, the system loads the colors from the ViewPort's ColorTable into the display's hardware registers, so whatever is rendered into the BitMap is displayed correctly. But if the VSprite system is used, and the colors are specified (via SprColors) for each VSprite, the SprColors will be loaded by the system into the display hardware, as needed. The system does this by generating Copper instructions that will jam the colors into the hardware at specific moments in the display cycle. Any BitMap rendering, including Bobs which share colors with VSprites, may change colors constantly as the video display beam progresses down the screen.

This color changing can be avoided by taking one of the following precautions:

- Use a four bit plane playfield, which only allows the lower 16 colors to be rendered into the BitMap (and allows hi-res).
- If a 32-color playfield display is being used, avoid rendering in colors 17-19, 21-23, 25-27, and 29-32, which are colors affected by the VSprite system.
- Specify the VSprite SprColors pointer as a value of NULL to avoid changing the contents of any of the hardware Sprite color registers. This may cause the VSprites to change colors depending on their positions relative to each other, as described in the previous section.

### GETTING THE VSPRITE LIST IN ORDER

When the system has displayed the last line of a VSprite, it is able to reassign the hardware Sprite to another VSprite located at a lower position on the screen. The system allocates hardware Sprites in the order in which it encounters the VSprites in the list. Therefore, the list of VSprites must be sorted before the system can assign the use of the hardware Sprites correctly.

The function `SortGList( )` must be used to get the GELs in the correct order before the system is asked to display them. *This sorting step is essential!* It should be done before calling `DrawGList( )`, whenever a GEL has changed position. This function is called as follows:

```
struct RastPort myRastPort;  
  
SortGList (&myRastPort);
```

where `&myRastPort` is a pointer to the `RastPort` structure containing the `GelsInfo`.

#### NOTE

There may be a GEL list in more than one `RastPort`. All of them must be initialized, sorted and displayed. Since there may be many `RastPorts` in any display, care must be taken to use only a single `RastPort` for any given gel.



## DISPLAYING THE VSPRITES

The next few sections explain how to display the VSprites. The following system functions are used:

- DrawGList()** to draw the elements into the current RastPort
- MrgCop()** to install the VSprites into the display
- LoadView()** to ask the system to display the new View
- WaitTOF()** to synchronize the functions with the display

### Turning on the Display

Before a display can be viewed on the screen, the system direct memory access must be enabled for both the hardware Sprites and the PlayField display. To enable the display of both PlayField and VSprites, use the system macro calls:

```
ON_DISPLAY;  
ON_SPRITE;
```

These do not need to be used if you are using the Intuition library to manage your display.

### Drawing the Graphics Elements

The system function called **DrawGList()** looks through the list of controlled GELS. It prepares necessary instructions and memory areas to display the data according to requirements. This function is called as follows:

```
struct RastPort myRastPort;  
struct ViewPort myViewPort;  
  
DrawGList(&myRastPort, &myViewPort);
```

where

**&myRastPort** is a pointer to the RastPort

**&myViewPort** is a pointer to the ViewPort

Because the system links VSprites to a View, the use of a RastPort is not significant for them. However, **DrawGList()** is used for Bobs as well as VSprites, so it is required that the pointer to the RastPort be passed to the function. **DrawGList()** actually draws Bobs into that RastPort.

Once **DrawGList()** has prepared the necessary instructions and memory areas to display the data, the VSprites are installed into the display with **MrgCop()**.

## Merging VSprite Instructions

Remember that the call to **DrawGList( )** did not actually draw the VSprites. It simply provided a new set of instructions that the system uses to assign the VSprite images to real hardware Sprites, based on their positions. The **View** structure already has a set of instructions that specifies how to construct the display area. It includes pointers to the set of VSprite instructions that was made by the call to **DrawGList( )**. To install the current VSprites into the display area, the function **MrgCop( )** is called to merge together all of the display-type instructions in the **View** structure. This function is called as follows:

```
struct View *view;  
MrgCop(view);
```

where **view** is a pointer to the **View** structure whose Copper instructions are to be merged

## Loading the New View

Now that the display instructions include the definition of the VSprites, the system may be used to display this newly configured **View**. This is done with the following system function:

```
struct View *view;  
LoadView(view);
```

where **view** is a pointer to the **View** that contains the pointer to the Copper instruction list

The Copper instruction lists are double-buffered, so this instruction does not actually take effect until the next display field occurs. This avoids the possibility of some function trying to update the Copper instruction list while the Copper is trying to use it to create the display.

## Synchronizing with the Display

To synchronize application functions with the display, call the system function **WaitTOF( )**. Although the application functions may be capable of generating more than 60 complete display fields per second, the system itself is limited to 60 displays per second. Therefore, after generating a complete display, a wait may be used until the next display is ready to be shown on the screen. **WaitTOF( )** holds your task until the vertical-blanking interval (blank area at the top of the screen) has begun. At that time, the system has retrieved the current Copper instruction list and is ready to allow generation of a new list.

The call to the vertical-blanking synchronization function takes the following form:

```
WaitTOF( );
```

## Complete VSprite Example

This function requires animtools.c, animtools.h, and animtools\_proto.h. These files are defined at the end of this chapter in the complete animation example.

```
/* vsprite.c 19oct89
**
** lattice c 5.04
** lc -bl -cfist -v -y vsprite.c
** blink FROM LIB:c.o vsprite.o /animtools/animtools.o LIB LIB:lc.lib TO vsprite
*/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gels.h>
#include <graphics/collide.h>
#include <exec/memory.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdlib.h>

#include "/animtools/animtools.h"
#include "/animtools/animtools_proto.h"

VOID borderCheck(struct VSprite *hitVSprite, LONG borderflags);
VOID process_window(struct Window *win, struct VSprite *MyVSprite);
VOID do_VSprite(struct Window *win);
VOID vspriteDrawGList(struct Window *win);

struct GfxBase      *GfxBase;      /* pointer to Graphics library */
struct IntuitionBase *IntuitionBase; /* pointer to Intuition library */

int return_code;

/* number of lines in the vsprite */
#define GEL_SIZE      4

/* vsprite data - there are two sets that are alternated between.
** note that this data is always displayed as low resolution.
*/
WORD chip vsprite_data1[2 * GEL_SIZE] =
    { 0x7ffe, 0x80ff, 0x7c3e, 0x803f, 0x7c3e, 0x803f, 0x7ffe, 0x80ff, };

WORD chip vsprite_data2[2 * GEL_SIZE] =
    { 0x7ffe, 0xff01, 0x7c3e, 0xfc01, 0x7c3e, 0xfc01, 0x7ffe, 0xff01, };

WORD mySpriteAltColors[] =
    { 0x000f, 0x0f00, 0x0ff0 };

WORD mySpriteColors[] =
    { 0x0000, 0x00f0, 0x0f00 };

/* information for the new vsprite */
NEWVSPRITE myNewVSprite =
{
    vsprite_data2, /* image data */
    mySpriteColors, /* sprite color array */
    1, /* word width (must be 1 for true vsprite) */
    GEL_SIZE, /* line height */
    2, /* image depth (must be 2 for true vsprite) */
    160, 100, /* x, y position */
    VSPRITE, /* flags (VSPRITE == true vsprite) */
};

/* information for the new window */
```

```

struct NewWindow myNewWindow =
{
    80, 20, 400, 150, -1, -1, CLOSEWINDOW | INTUITICKS,
    ACTIVATE | WINDOWCLOSE | WINDOWDEPTH | RMBTRAP,
    NULL, NULL, "VSprite", NULL, NULL, 0, 0, 0, 0, WBENCHSCREEN
};

/*-----
**
*/
VOID vspriteDrawGList(struct Window *win)
{
    SortGList(win->RPort);
    DrawGList(win->RPort, ViewPortAddress(win));
    /* These calls are not Intuition compatible...
    **   MrgCop(view);
    **   LoadView(view);
    ** use RethinkDisplay( ) in Intuition environment.
    */
    RethinkDisplay( );
    /* WaitTOF( ); done by RethinkDisplay */
}

/*-----
** collision function for vsprite hitting border.
** note that when the collision is vsprite to vsprite (or bob
** to bob, bob to AnimOb, etc), then the parameters are both
** pointers to a vsprite:
**
** VOID collCheck(struct VSprite *vsp1, struct VSprite *vsp2)
**
*/
VOID borderCheck(struct VSprite *hitVSprite, LONG borderflags)
{
    if (borderflags & RIGHTHIT)
    {
        hitVSprite->SprColors = mySpriteAltColors;
        hitVSprite->VUserExt = -40;
    }
    if (borderflags & LEFTHIT)
    {
        hitVSprite->SprColors = mySpriteColors;
        hitVSprite->VUserExt = 20;
    }
}

/*-----
** process window and dynamically change vsprite:
** - get messages.
** - go away on CLOSEWINDOW.
** - update and redisplay vsprite on INTUITICKS.
** - wait for more messages.
**
*/
VOID process_window(struct Window *win, struct VSprite *myVSprite)
{
    struct IntuiMessage *msg;

    FOREVER
    {
        Wait(1L << win->UserPort->mp_SigBit);

        while (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
            /* only CLOSEWINDOW and INTUITICKS are active */
            if (msg->Class == CLOSEWINDOW)
            {
                ReplyMsg((struct Message *)msg);
                return;
            }

            /* must be an INTUITICKS: change x and y values on the fly.
            ** note offset by window left and top edge--sprite relative
            ** to the screen, not window.

```

```

    */
    myVSprite->X = win->LeftEdge + msg->MouseX + myVSprite->VUserExt;
    myVSprite->Y = win->TopEdge + msg->MouseY + 1;
    ReplyMsg((struct Message *)msg);
}

/* got a message, change image data on the fly */
myVSprite->ImageData = (myVSprite->ImageData == vsprite_data1) ?
    vsprite_data2 : vsprite_data1;

SortGList(win->RPort);
DoCollision(win->RPort);
vspriteDrawGList(win);
}

}

/*-----
** working with the vsprite:
** - set up the gel system, and get a new vsprite (makeVSprite( )).
** - add the vsprite to the system and display.
** - use the vsprite.
** - when done, remove vsprite and update the display without the vsprite.
** - cleanup everything.
*/
VOID do_VSprite(struct Window *win)
{
    struct VSprite      *myVSprite;
    struct GelsInfo     *my_ginfo;

    if (NULL == (my_ginfo = setupGelSys(win->RPort, 0xfc)))
        return_code = RETURN_WARN;
    else
    {
        if (NULL == (myVSprite = makeVSprite(&myNewVSprite)))
            return_code = RETURN_WARN;
        else
        {
            AddVSprite(myVSprite, win->RPort);
            vspriteDrawGList(win);

            myVSprite->VUserExt = 20;
            myVSprite->HitMask = 1L << BORDERHIT;
            SetCollision(BORDERHIT, borderCheck, win->RPort->GelsInfo);

            process_window(win, myVSprite);

            RemVSprite(myVSprite);
            freeVSprite(myVSprite);
        }
        vspriteDrawGList(win);
        cleanupGelSys(my_ginfo, win->RPort);
    }
}

/*-----
** example vsprite program:
** - first open up the libraries and a window.
*/
VOID main(int argc, char **argv)
{
    struct Window      *win;

    return_code = RETURN_OK;

    if (NULL == (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L)))
        return_code = RETURN_FAIL;
    else
    {
        if (NULL == (IntuitionBase =
            (struct IntuitionBase *)OpenLibrary("intuition.library", 33L)))
            return_code = RETURN_FAIL;
        else

```

```

    {
    if (NULL == (win = OpenWindow(&myNewWindow)))
        return_code = RETURN_WARN;
    else
    {
        do_VSprite(win);

        CloseWindow(win);
    }
    CloseLibrary(IntuitionBase);
    }
    CloseLibrary(GfxBase);
    }
exit(return_code);
}

```

## Using Bobs

The following section describes how to define a Bob.

The components common to all GELs — height, collision-handling information, position in the drawing area, and pointers to the image definition — are part of the **VSprite** structure. The added features — such as drawing sequence, data about saving and restoring the background, and other features not applicable to VSprites — are part of the **Bob** structure instead.

### THE VSPRITE STRUCTURE AND BOBS

The root **VSprite** structure is set up as described for true VSprites, with the following exceptions, the more complex of which will be described in detail soon:

<b>Y, X</b>	Bob position is always in pixels that are the same resolution as the display.
<b>Flags</b>	SAVEBACK and/or OVERLAY may be used. Don't set VSPRITE for Bobs.
<b>Height, Width</b>	Bob pixels are the size of the background pixels. The <b>Width</b> of Bobs may be greater than one word.
<b>Depth</b>	The <b>Depth</b> of a Bob may be up to as deep as the playfield, provided that enough image data is provided.
<b>ImageData</b>	This pointer still is set to point to the image, but the data there is organized differently.
<b>SprColors</b>	This pointer should be set to NULL for Bobs.
<b>VSBob</b>	This pointer is set up as described below.

### VSPRITE FLAGS AND BOBS

The bits in the **VSprite Flags** member that are used for Bobs follow.

## VSPRITE Flag

When using the **VSprite** structure to describe a Bob, set **VSPRITE** to zero.

## SAVEBACK Flag

To have the GEL routines save the background before the Bob is drawn and restore the background after the Bob is removed, specify the **SAVEBACK** (for "save the background") flag in the **VSprite** structure **Flags** field.

If this flag is set, the **SaveBuffer** must have been allocated, which is where the system puts this saved background area. The buffer must be large enough to save all the background bitplanes, regardless of how many planes the Bob has.

```
size = Bob.Width * Bob.Height * RastPort.BitMap.Depth;
```

To allocate this space, the graphics function **AllocRaster()** may be used. **AllocRaster()** takes the width in bits, so it is a convenient way to allocate the space needed. The **makeBob()** routine below shows another way to correctly allocate this buffer.

### NOTE

The **SaveBuffer** must be allocated from CHIP memory. The **AllocRaster()** function does this for you.

For example:

```
/* space for 16 bits times 5 lines times 5 bit-planes */  
myBob.SaveBuffer = AllocRaster(16, 5 * 5);
```

### NOTE

The **AllocRaster()** function rounds the width value up to the next integer multiple of 16 bits which is greater than or equal to the current value.

## OVERLAY Flag

If the system should use the Bob shadow mask when it draws the Bob into the background, specify the **OVERLAY** flag in the **VSprite** structure **Flags** field. If this flag is set, it means that the background's original pixels show through in any area where there are 0 bits in the Sprite shadow mask.

If this flag is set, the space for the **ImageShadow** shadow mask must have been allocated and initialized. The **ImageShadow** mask must be allocated from CHIP memory.

If the **OVERLAY** bit is cleared, the system uses the *entire rectangle of words* that define the Bob image to *replace* the playfield area at the specified y,x coordinates. See the paragraphs below called "ImageShadow."

## THE BOB STRUCTURE

The following structure is taken from the *Includes & Autodocs* manual. Please see that manual for a fully commented structure definition. The Bob structure is:

```
struct Bob
{
    WORD          Flags;          /* general purpose flags          */
    WORD          *SaveBuffer;    /* buffer for background save    */
    WORD          *ImageShadow;   /* shadow mask of image         */
    struct Bob     *Before;       /* draw this Bob before Bobs on this list */
    struct Bob     *After;        /* draw this Bob after Bobs on this list */
    struct VSprite *BobVSprite;   /* this Bob's VSprite definition */
    struct AnimComp *BobComp;     /* pointer to this Bob's AnimComp def */
    struct DBufPacket *DBuffer;   /* pointer to this Bob's dBuf packet */
    BUserStuff     BUserExt;     /* Bob user extension           */
};
```

The (global) static declaration of a Bob structure:

```
struct Bob myBob =
{
    0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0
}
```

Since most of the Bob structure members are pointers, it is much more common to allocate and set the Bob up dynamically. Here is a sample routine to allocate Bobs (this is a fragment of a larger file called *animtools.c*. See the complete animation example at the end of this chapter for the complete file):

```
/* from:
** animtools.c 19oct89
** original code by Dave Lucas.
** rework by CATS
** lattice c 5.04
** lc -bl -cfist -v -y animtools.c
**/

#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/gfx.h>
#include <graphics/gels.h>
#include <graphics/clip.h>
#include <graphics/rastport.h>
#include <graphics/view.h>
#include <graphics/gfxbase.h>

/* data structure to hold information for a new vsprite.
*/
typedef struct newVSprite
{
    WORD          *nvs_Image;     /* image data for the vsprite */
    WORD          *nvs_ColorSet;  /* color array for the vsprite */
    SHORT         nvs_WordWidth;  /* width in words              */
    SHORT         nvs_LineHeight; /* height in lines             */
    SHORT         nvs_ImageDepth; /* depth of the image          */
    SHORT         nvs_X;          /* initial x position          */
    SHORT         nvs_Y;          /* initial y position          */
    SHORT         nvs_Flags;      /* vsprite flags               */
} NEWVSPRITE;

/* data structure to hold information for a new Bob.
** note that:
```



```

**      NEWBOB myNBob;
** is equivalent to:
**      struct newBob myNBob;
**
typedef struct newBob
{
    WORD      *nb_Image;      /* image data for the Bob      */
    SHORT     nb_WordWidth;   /* width in words              */
    SHORT     nb_LineHeight;  /* height in lines             */
    SHORT     nb_ImageDepth;  /* depth of the image          */
    SHORT     nb_PlanePick;   /* planes that get image data  */
    SHORT     nb_PlaneOnOff;  /* unused planes to turn on    */
    SHORT     nb_BFlags;      /* Bob flags                   */
    SHORT     nb_DBuf;        /* 1=double buf, 0=not        */
    SHORT     nb_RasDepth;    /* depth of the raster         */
    SHORT     nb_X;           /* initial x position          */
    SHORT     nb_Y;           /* initial y position          */
} NEWBOB ;

#include <proto/all.h>

/* defined above */
struct VSprite *makeVSprite(NEWVSPRITE *nVSprite)
VOID freeVSprite(struct VSprite *vsprite)

/*-----
** create a Bob from the information given in nBob.
** use freeBob( ) to free this gel.
**
** A VSprite is created for this Bob.
** This routine properly allocates all double buffered information
** if it is required.
*/
struct Bob *makeBob(NEWBOB *nBob)
{
    struct Bob      *bob;
    struct VSprite  *vsprite;
    NEWVSPRITE      nVSprite ;
    LONG            rassize;

    rassize = (LONG)sizeof(WORD) *
              nBob->nb_WordWidth * nBob->nb_LineHeight * nBob->nb_RasDepth;

    if (NULL != (bob =
        (struct Bob *)AllocMem((LONG)sizeof(struct Bob), MEMF_CLEAR)))
    {
        if (NULL != (bob->SaveBuffer = (WORD *)AllocMem(rassize, MEMF_CHIP)))
        {
            nVSprite.nvs_WordWidth = nBob->nb_WordWidth;
            nVSprite.nvs_LineHeight = nBob->nb_LineHeight;
            nVSprite.nvs_ImageDepth = nBob->nb_ImageDepth;
            nVSprite.nvs_Image      = nBob->nb_Image;
            nVSprite.nvs_X          = nBob->nb_X;
            nVSprite.nvs_Y          = nBob->nb_Y;
            nVSprite.nvs_ColorSet   = NULL;
            nVSprite.nvs_Flags      = nBob->nb_BFlags;

            if ((vsprite = makeVSprite(&nVSprite)) != NULL)
            {
                vsprite->PlanePick = nBob->nb_PlanePick;
                vsprite->PlaneOnOff = nBob->nb_PlaneOnOff;

                vsprite->VSBob      = bob;
                bob->BobVSprite     = vsprite;
                bob->ImageShadow     = vsprite->CollMask;
                bob->Flags           = 0;
                bob->Before          = NULL;
                bob->After           = NULL;
                bob->BobComp         = NULL;

                if (nBob->nb_DBuf)
                {

```

```

        if (NULL != (bob->DBuffer = (struct DBufPacket *)AllocMem(
            (LONG)sizeof(struct DBufPacket), MEMF_CLEAR)))
        {
            if (NULL != (bob->DBuffer->BufBuffer =
                (WORD *)AllocMem(rassize, MEMF_CHIP)))
            {
                return(bob);
            }
            FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
        }
    }
    else
    {
        bob->DBuffer = NULL;
        return(bob);
    }

    freeVSprite(vsprite);
}
FreeMem(bob->SaveBuffer, rassize);
}
FreeMem(bob, (LONG)sizeof(*bob));
}
return(NULL);
}

```

Here is a sample routine to free Bobs that were allocated using makeBob( ). This is a fragment of a larger file called *animtools.c*. See the complete animation example at the end of this chapter for a complete listing of this file.

```

/*-----
** free the data created by makeBob( )
**
** it's important that rasdepth match the depth you
** passed to makeBob( ) when this gel was made.
** assumes images deallocated elsewhere.
*/
VOID freeBob(struct Bob *bob, LONG rasdepth)
{
    LONG    rassize;

    rassize = (LONG)sizeof(WORD) *
        bob->BobVSprite->Width * bob->BobVSprite->Height * rasdepth;

    if (bob->DBuffer != NULL)
    {
        FreeMem(bob->DBuffer->BufBuffer, rassize);
        FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
    }
    FreeMem(bob->SaveBuffer, rassize);
    freeVSprite(bob->BobVSprite);
    FreeMem(bob, (LONG)sizeof(*bob));
}

```

The **Bob** structure does not need to be in CHIP memory.

## LINKING A BOB TO A VSPRITE STRUCTURE

The **VSprite** and **Bob** structures must point to one another, so that the system can find the entire GEL. The structures are linked with statements like:

```
myBob.BobVSprite = &myVSprite;  
myVSprite.VSBob = &myBob;
```

Now the system (and the application program) can go back and forth between the two structures to obtain the various **Bob** variables.

## USING BOB FLAGS

The following paragraphs describe the **Bob Flags**.

### SAVEBOB Flag

To tell the system not to erase the old image of the **Bob** when the **Bob** is moved, specify the **SAVEBOB** flag in the **Bob** structure **Flags** field. This makes the **Bob** behave like a paintbrush. It has the opposite effect of **SAVEBACK**.

#### NOTE

It takes longer to preserve and restore the raster image than simply to draw a new **Bob** image wherever required.

### BOBISCOMP Flag

If this **Bob** is part of an **AnimComp**, set the **BOBISCOMP** flag in the **Bob** structure to 1. If the flag is a 1, the pointer named **BobComp** must have been initialized. Otherwise, the system ignores the pointer, and it may be left alone (though it's good practice to initialize it to **NULL**). See "Animation Structures and Controls" for a discussion of **AnimComps**.

### BWAITING Flag

This flag is used solely by the system, and should be left alone. When a **Bob** is waiting to be drawn, the system sets the **BWAITING** flag in the **Bob** structure to 1. This occurs only if the system has found a **Before** pointer in this **Bob's** structure that points to another **Bob**. Thus, the system flag **BWAITING** provides current draw-status to the system. Currently, the system clears this flag on return from each call to **DrawGList()**.

### BDRAWN Flag

The **BDRAWN** system status flag in the **Bob** structure tells the system that this **Bob** has already been drawn. Therefore, in the process of examining the various **Before** and **After** flags, the drawing routines may determine the drawing sequence. The system clears this flag on return from each call to **DrawGList()**.

### BOBSAWAY Flag

To initiate the removal of a Bob during the next call to **DrawGList()**, set **BOBSAWAY** to 1. Either the application or the system may set this Bob structure system flag. The system restores the background where it has last drawn the Bob. The system will unlink the Bob from the system GEL list the next time **DrawGList()** is called, unless the application is using double-buffering. In that case, the Bob will not be unlinked and completely removed until two calls to **DrawGList()** have occurred and the Bob has been removed from both buffers. The **RemBob()** macro sets the **BOBSAWAY** flag.

### **BOBNIX Flag**

When a Bob has been completely removed, the system sets the **BOBNIX** flag to 1 on return from **DrawGList()**. In other words, when the background area has been fully restored and the Bob has been removed from the GEL list, this flag is set to a 1. **BOBNIX** is especially significant when double-buffering, because once the application asks that a Bob be removed, the system must remove it from the active drawing buffer *and* from the display buffer. Once **BOBNIX** has been set for a double-buffered Bob, it has been removed from both buffers and the application is free to reuse or deallocate it.

### **SAVEPRESERVE Flag**

The **SAVEPRESERVE** flag is a double-buffer version of the **SAVEBACK** flag. If using double-buffering and wishing to save and restore the background, set **SAVEBACK** to 1. **SAVEPRESERVE** is used by the system to indicate whether the Bob in the “other” buffer has been restored; it is for system use only.

### **SPECIFYING THE SIZE OF A BOB**

Bobs do not have the 16 pixel width limit that applies to VSprites. To specify the overall size of a Bob, use the **Height** and **Width** members of the root **VSprite** structure. Specify the **Width** as the number of 16-bit words it takes to fully contain the object.

The number of lines is still specified with the **Height** member in the **VSprite** data structure.

As an example, suppose the Bob is 24 pixels wide and 20 lines tall. Use statements like the following to specify the size:

```
myVSprite.Height = 20; /* 20 lines tall. */
myVSprite.Width  = 2;  /* 24 bits fit into two words. */
```

Because Bobs are drawn into the background playfield, the pixels of the Bob are the same size as the background pixels, and share the color palette of the ViewPort.

### **SPECIFYING THE SHAPE OF A BOB**

The layout of the data of a Bob's image is different from that of a VSprite because of the way the system retrieves data to draw Bobs. VSprite images are organized in a way convenient to the Sprite hardware; Bob images are set up for easy blitter manipulation. The **ImageData** pointer is still initialized to point to the first word of the image

definition.

#### NOTE

As with all image data, a Bob's **ImageData** must be in CHIP memory for access by the blitter.

The sample image below shows the same image defined as a **VSprite** in the "Using VSprites" sub-section above. The data here, however, is laid out for a Bob. The shape is 2 planes deep and is triangular:

```

                                <first bit-plane data>
mem          1111 1111 1111 1111
mem + 1      0011 1100 0011 1100
mem + 2      0000 1100 0011 0000
mem + 3      0000 0010 0100 0000
mem + 4      0000 0001 1000 0000

                                <second bit-plane data>
mem + 5      1111 1111 1111 1111
mem + 6      0011 0000 0000 1100
mem + 7      0000 1111 1111 0000
mem + 8      0000 0011 1100 0000
mem + 9      0000 0001 1000 0000

                                <<more bit-planes of data if Bob is deeper>
                                ...
```

#### SPECIFYING THE COLORS OF A BOB

Typically a five-bit-plane, low-resolution mode display allows playfield pixels (and therefore, Bob pixels) to be selected from any of 32 active colors out of a system palette of 4,096 different color choices. Bob colors are limited to the colors used in the background playfield.

The system ignores the **sprColors** member of the **VSprite** structure when the **VSprite** structure is the root of a Bob. Instead, the Bob's colors are determined by the combination of the **Depth** of the Bob image and its **PlanePick**, **PlaneOnOff** and **ImageShadow** members.

Use the **Depth** member in the **VSprite** structure to indicate how many planes of image data is provided to define the Bob. This also defines how many colors the Bob will have. The combination of bits in corresponding Y,X positions in each bit-plane determines the color of the pixel at that position.

For example, if a **Depth** of one plane is specified, then the bits of that image allow only two colors to be selected: one color for each bit that is a 0, a second color for each bit that is a 1. Likewise, if there are 5 planes of image data, all 32 colors can be used in the Bob. The Bob **Depth** must not exceed the background depth.

Specify **Depth** using a statement such as the following:

```
myVSprite.Depth = 5; /* Allow a 32 color, 5-bit-plane image. */
```

## OTHER ITEMS INFLUENCING BOB COLORS

The three other members in the **VSprite** structure that affect the color of Bob pixels are **ImageShadow**, **PlanePick**, and **PlaneOnOff**.

### ImageShadow

The **ImageShadow** member is a pointer to the shadow mask of a Bob. A shadow mask is the logical *or* of all bitplanes of a Bob image. The system uses the shadow mask in conjunction with **PlaneOnOff**, discussed below, for color selection. It also uses the shadow mask to “cookie cut” the bits that will be overwritten by this Bob, to save and later restore the background.

The following figure shows the shadow mask of the image described above.

mem + 0	1111 1111 1111 1111
mem + 1	0011 1100 0011 1100
mem + 2	0000 1111 1111 0000
mem + 3	0000 0011 1100 0000
mem + 4	0000 0001 1000 0000

Space for the **ImageShadow** must be provided and this pointer initialized to point to it. The amount of memory needed is equivalent to one plane of the image:

```
shadow_size = myBob->BobVSprite->Height * myBob->BobVSprite->Width;
```

The example image is 5 high and 1 word wide, so, 5 words must be made available.

### NOTE

The image shadow memory must be allocated from CHIP memory (MEMF\_CHIP).

### PlanePick

Because the **Depth** of the Bob can be less than the background, the **PlanePick** member is provided so that the application can indicate which background bit planes are to have image data put into them.

The system starts with the least significant plane of the Bob, and scans **PlanePick** starting at the least significant bit, looking for a plane of the RastPort to put it in.

For example, if **PlanePick** has a binary value of:

0000 0011 (0x03)

then the system draws the first plane of the Bob's image into background plane 0 and the second plane into background plane 1.

Alternatively, a **PlanePick** value of:

0001 0010 (0x12)

directs the system to put the first Bob plane into plane 1, and the second Bob plane into plane 4.

### PlaneOnOff

What happens to the background planes that aren't picked? The shadow mask is used to either set or clear the bits in those planes in the exact shape of the Bob if **OVERLAY** is set, otherwise the entire rectangle containing the Bob is used. The **PlaneOnOff** member tells the system whether to put down the shadow mask as zeros or ones for each plane. The relationship between bit positions in **PlaneOnOff** and background plane numbers is identical to **PlanePick**: the least significant bit position indicates the lowest-numbered bit-plane. A zero bit clears the shadow mask shape in the corresponding plane, while a one bit sets the shadow mask shape. The planes Picked by **PlanePick** have image data — not shadow mask — blitted in.

This provides a great deal of color versatility. One image definition can be used for many Bobs. By having different **PlanePick** / **PlaneOnOff** combinations, each Bob can use a different subset of the background color set.

There is a member in the **VSprite** structure called **CollMask** (the collision mask, covered under “Detecting GEL collisions”) for which the application may also reserve some memory space. The **ImageShadow** and **CollMask** pointers usually, but not necessarily, point to the same data, which must be located in CHIP memory. If they point to the same location, obviously, the memory only need be allocated once.

An example of the kinds of statements that accomplish these actions (see the **makeVSprite()** and **makeBob()** examples for more details):

```
#define BOBW 1
#define BOBH 5
#define BOBD 2

/* Data definition from example layout */
WORD chip BobData[]=
{
    0xFFFF, 0x300C, 0x0FF0, 0x03C0, 0x0180,
    0xFFFF, 0x3E7C, 0x0C30, 0x03C0, 0x0180
};

/* Reserve space for the collision mask for this Bob */
WORD chip BobCollision[BOBW * BOBH * BOBD];

myVSprite.Width  = BOBW;      /* Image is 16 pixels wide (1 word) */
myVSprite.Height = BOBH;      /* 5 lines for each plane of the Bob */
myVSprite.Depth  = BOBD;      /* 2 Planes are in ImageData */

/* Show the system where it can find the data image of the Bob */
myVSprite.ImageData = BobData;

/* binary 0101, render image data into bit-planes 0 and 2 */
myVSprite.PlanePick = 0x05;

/* binary 0000, means colors 1, 4, and 5 will be used.
 * binary 0010 would mean colors 3, 6, and 7.
 * "    1000 "    "    "    9, C, and D.
 * "    1010 "    "    "    B, E, and F.
 */
myVSprite.PlaneOnOff = 0x00;

/* Where to put collision mask */
myVSprite.CollMask = BobCollision;
```

```

/* Tell the system where it can assemble a GEL shadow */
/* Point to same area as CollMask */
myBob.ImageShadow = BobCollision;

/* Create the Sprite collision mask in the VSprite structure */
InitMasks(&myVSprite);

```

## BOB PRIORITIES

This sub-section describes the choices for inter-Bob priorities. The inter-Bob priorities tell the system what order to render the Bobs. Bobs rendered earlier will appear to be behind later Bobs. A Bob drawn earlier is said to have the lower priority, and a Bob drawn later is said to have the higher priority. Thus, the highest priority Bob will be drawn last, and will never be obstructed by another Bob.

### Letting the System Decide Priorities

The priority issue can be ignored and the system will render the Bobs as it finds them in the **GelsInfo** list. To do this, set the Bob's **Before** and **After** pointers to **NULL**. Since the **GelsInfo** list is sorted by GEL y,x values, Bobs that are higher on the display will appear behind the lower ones, and Bobs that are more to the left on the display will appear behind Bobs on the right.

As Bobs are moved about the display, their priorities will change.

### Specifying the Drawing Order

To specify the priorities of the Bobs, use the **Before** and **After** pointers. **Before** points to the Bob that this Bob should be drawn before, and **After** points to the Bob that this Bob should be drawn after. By following these pointers, from Bob to Bob, the system can determine the order in which the Bobs should be drawn.

#### NOTE

This terminology is often confusing, but, due to historical reasons, cannot be changed. The system does NOT draw the Bobs on the **Before** list first, it draws the Bobs on the **After** list first. Next, it draws the current Bob, and, finally, the Bobs on the **Before** list.

For example, to assure that myBob1 always appears in front of myBob2, The **Before** and **After** pointers must be initialized so that the system will always draw myBob1 after myBob2.

```

myBob2.Before = &myBob1;    /* draw Bob2 before drawing Bob1 */
myBob2.After  = NULL;       /* draw Bob2 after  no other Bob */
myBob1.After  = &myBob2;    /* draw Bob1 after  drawing Bob2 */
myBob1.Before = NULL;       /* draw Bob1 before no other Bob */

```

As the system goes through the **GelsInfo** list, it checks the Bob's **After** pointer. If this is not **NULL**, it follows the **After** pointer until it hits a **NULL**. Then it starts rendering the Bobs, going back up the **Before** pointers until it hits a **NULL**. then it continues through the **GelsInfo** list. So, it is important that *all* **Before** and **After** pointers of a



group properly point to each other.

### NOTE

In a screen with a number of complex GELs, you may want to specify the **Before** and **After** order for Bobs that are not in the same AnimOb. This will keep large objects together. If you do not do this, you may have an object drawn with half of its Bobs in front of another object! Also, in sequences you only set the **Before** and **After** pointers for the active AnimComp in the sequence.

### ADDING A BOB

To add a Bob to the system GEL list, use the **AddBob( )** routine. The **Bob** and **VSprite** structures must be correct and cohesive when this call is made. See the **makeBob( )** and **makeVSprite( )** examples for a detailed example of setting up Bobs and VSprites. See the **setupGelSys( )** example for a more complete initialization of the GELs system.

For example:

```
struct GelsInfo myGelsInfo;
struct VSprite dummySpriteA, dummySpriteB;
struct Bob myBob;
struct RastPort rastport;

/* done ONCE, for this GelsInfo
** see setupGelSys( ) above for a more complete initialization of
** the Gel system
*/
InitGels(&dummySpriteA, &dummySpriteB, &myGelsInfo);

/* initialize the Bob members here, then AddBob( ) */
AddBob(&myBob, &rastport);
```

### REMOVING A BOB

Two methods may be used to remove a Bob. These paragraphs describe the two system calls.

The first method uses the **RemBob( )** macro. It is called as follows:

```
struct Bob myBob;

RemBob(&myBob);
```

**RemBob( )** is a *macro* which causes the system to remove the Bob during the next call to **DrawGList( )** (or two calls to **DrawGList( )** if the system is double-buffered). **RemBob( )** asks the system to remove the Bob "at its next convenience." See the description of the BOBSAWAY and BOBNIX flags above.

The second method uses the **RemIBob( )** routine.

For example:

```

struct Bob      myBob;
struct RastPort rastport;
struct ViewPort viewport;

RemIBob(&myBob, &rastport, &viewport);

```

**RemIBob( )** tells the system “remove this Bob immediately.” It causes the system to erase the Bob from the drawing area and causes the immediate erasure of any other Bob that had been drawn subsequent to (and on top of) this one. The system then unlinks the Bob from the system GEL list. To redraw the Bobs that were drawn on top of the one just removed, another call to **DrawGList( )** must be made.

## GETTING THE LIST OF BOBS IN ORDER

As described for VSprites, **SortGList( )** re-orders the **GelsInfo** list. For Bobs, the system uses the position information to decide inter-Bob priorities, if not explicitly set by using the **Before** and **After** pointers.

This function is called as follows:

```

struct RastPort myRastPort;

SortGList(&myRastPort);

```

## DISPLAYING BOBS

This sub-section provides the typical sequence of operations for drawing the Bobs on the screen.

When **DrawGList( )** is called the system *actually draws* the Bobs in the **GelsInfo** list.

For example:

```

struct RastPort myRastPort;
struct ViewPort myViewPort;

DrawGList(&myRastPort, &myViewPort);    /* draw the elements */

```

### NOTE

If you are only using Bobs, and not using ANY true VSprites, you do not need to call **MrgCop( )** and **LoadView( )**. **DrawGList( )** will properly draw the Bobs/AnimComps into the RastPort. You still need to call **WaitTOF( )**.

## CHANGING BOBS

The following characteristics of Bobs can be changed dynamically between calls to **DrawGList( )**:

- To change the location in the drawing area, change the y, x values in the associated **VSprite** structure.
- To change their appearance, the pointer to the **ImageData** in the associated **VSprite** structure may be

changed.

#### NOTE

A change in the `ImageData` also requires a change or recalculation of the `ImageShadow`, using `InitMasks()`.

- To change their colors, change a Bob's `PlanePick` and/or `PlaneOnOff` values; the `Depth` parameters may also be changed.
- To change the object priorities, alter the `Before` and `After` pointers in the `Bob` structures.
- To change the Bob into a paintbrush, specify the `SAVEBOB` flag in the `Bob` structure.

#### NOTE

Neither these nor other changes are evident until `SortGList()` and then `DrawGList()` are called.

### COMPLETE BOB EXAMPLE

This routine requires *animtools.c*, *animtools.h*, and *animtools\_proto.h*. These files are defined at the end of this chapter in the complete animation example.

```
/* bob.c 19oct89
**
** lattice c 5.04
** lc -bl -cfist -v -y bob.c
** blink FROM LIB:c.o bob.o /animtools/animtools.o LIB LIB:lc.lib TO bob
**/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gels.h>
#include <exec/memory.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdlib.h>

/* these define the structures like NEWBOB, and the
** prototypes for the animtools routines.
**/
#include "/animtools/animtools.h"
#include "/animtools/animtools_proto.h"

/* routines used for the bob example */
VOID bobDrawGList(struct RastPort *rport, struct ViewPort *vport);
VOID process_window(struct Window *win, struct Bob *myBob);
VOID do_Bob(struct Window *win);

struct GfxBase      *GfxBase;      /* pointer to Graphics library */
struct IntuitionBase *IntuitionBase; /* pointer to Intuition library */

int return_code;

/* number of lines in the bob */
#define GEL_SIZE      4

/* bob data - there are two sets that are alternated between.
** note that this data is at the resolution of the screen.
**/
WORD chip bob_data1[2 * 2 * GEL_SIZE] =
{ /* data is 2 planes by 2 words by GEL_SIZE lines */
  /* plane 1 */
```

```

0xffff, 0x0003, 0xffff, 0x0003, 0xffff, 0x0003, 0xffff, 0x0003,
/* plane 2 */
0x3fff, 0xfffc, 0x3ff0, 0x0ffc, 0x3ff0, 0x0ffc, 0x3fff, 0xfffc,
};

WORD chip bob_data2[2 * 2 * GEL_SIZE] =
{ /* data is 2 planes by 2 words by GEL_SIZE lines */
/* plane 1 */
0xc000, 0xffff, 0xc000, 0x0fff, 0xc000, 0x0fff, 0xc000, 0xffff,
/* plane 2 */
0x3fff, 0xfffc, 0x3ff0, 0x0ffc, 0x3ff0, 0x0ffc, 0x3fff, 0xfffc,
};

/* information for the new bob
** structure defined in animtools.h
*/
NEWBOB myNewBob =
{
    bob_data2,          /* initial image          */
    2,                  /* WORD width             */
    GEL_SIZE,           /* line height            */
    2,                  /* image depth            */
    3, 0,               /* plane pick, plane on off */
    SAVEBACK | OVERLAY, /* vsprite flags          */
    0,                  /* dbuf (0 == false)      */
    2,                  /* raster depth           */
    160, 100,           /* x,y position           */
};

/* information for the new window */
struct NewWindow myNewWindow =
{
    80, 20, 400, 150, -1, -1, CLOSEWINDOW | INTUITICKS,
    ACTIVATE | WINDOWCLOSE | WINDOWDEPTH | RMBTRAP,
    NULL, NULL, "Bob", NULL, NULL, 0, 0, 0, 0, WBENCHSCREEN
};

/*-----
** draw the bobs into the rast port.
*/
VOID bobDrawGList(struct RastPort *rport, struct ViewPort *vport)
{
    /* only need to MrgCop( ) LoadView( ) if using TRUE VSprites */
    SortGList(rport);
    DrawGList(rport, vport);
    WaitTOF( );
}

/*-----
** process window and dynamically change bob:
** - get messages.
** - go away on CLOSEWINDOW.
** - update and redisplay bob on INTUITICKS.
** - wait for more messages.
*/
VOID process_window(struct Window *win, struct Bob *myBob)
{
    struct IntuiMessage *msg;

    FOREVER
    {
        Wait(1L << win->UserPort->mp_SigBit);

        while (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
            /* only CLOSEWINDOW and INTUITICKS are active */
            if (msg->Class == CLOSEWINDOW)
            {
                ReplyMsg((struct Message *)msg);
                return;
            }
        }
    }
}

```

```

/* must be INTUITICKS: change x and y values on the fly
** note: do not have to add window offset, bob is relative
** to the window (sprite relative to screen).
*/
myBob->BobVSprite->X = msg->MouseX + 20;
myBob->BobVSprite->Y = msg->MouseY + 1;
ReplyMsg((struct Message *)msg);
}

/* after getting a message, change image data on the fly */
myBob->BobVSprite->ImageData =
    (myBob->BobVSprite->ImageData == bob_data1) ? bob_data2 : bob_data1;

InitMasks(myBob->BobVSprite); /* set up masks for new image */
bobDrawGLList(win->RPort, ViewPortAddress(win));
}

}

/*-----
** working with the bob:
** - setup the gel system, and get a new bob (makeBob( )).
** - add the bob to the system and display.
** - use the bob.
** - when done, remove the bob and update the display without the bob.
** - cleanup everything.
*/
VOID do_Bob(struct Window *win)
{
    struct Bob      *myBob;
    struct GelsInfo *my_ginfo;

    if (NULL == (my_ginfo = setupGelSys(win->RPort, 0xfc)))
        return_code = RETURN_WARN;
    else
    {
        if (NULL == (myBob = makeBob(&myNewBob)))
            return_code = RETURN_WARN;
        else
        {
            AddBob(myBob, win->RPort);
            bobDrawGLList(win->RPort, ViewPortAddress(win));

            process_window(win, myBob);

            RemBob(myBob);
            bobDrawGLList(win->RPort, ViewPortAddress(win));

            freeBob(myBob, myNewBob.nb_RasDepth);
        }
        cleanupGelSys(my_ginfo, win->RPort);
    }
}

/*-----
** example bob program:
** - first open up the libraries and a window.
*/
VOID main(int argc, char **argv)
{
    struct Window *win;

    return_code = RETURN_OK;

    if (NULL == (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L)))
        return_code = RETURN_FAIL;
    else
    {
        if (NULL == (IntuitionBase =
            (struct IntuitionBase *)OpenLibrary("intuition.library", 33L)))
            return_code = RETURN_FAIL;
        else
        {

```

```

    if (NULL == (win = OpenWindow(&myNewWindow)))
        return_code = RETURN_FAIL;
    else
    {
        do_Bob(win);

        CloseWindow(win);
    }
    CloseLibrary(IntuitionBase);
}
CloseLibrary(GfxBase);
}
exit(return_code);
}

```

## DOUBLE-BUFFERING

Double-buffering is the technique of supplying two different memory areas in which the drawing routines may create images. The system displays one memory space while drawing into the other area. This eliminates the “tearing” that is visible when the a single display is being rendered into at the same time that it is being displayed.

### NOTE

If any of the Bobs are double-buffered, then *all* of them must be double-buffered. The complete animation example at the end of this chapter allows double-buffering to be turned on or off with a command line argument.

To find whether a Bob is to be double-buffered, the system examines the pointer named **DBuffer** in the **Bob** structure. If this pointer has a value of **NULL**, the system does not use double-buffering for this Bob.

For example:

```
myBob.DBuffer = NULL; /* do this if this Bob is NOT double-buffered */
```

## DBufPacket and Double-Buffering

For double-buffering, a place must be provided for the system to store the extra information it needs. The system maintains these data, and does not expect the application to change them. The **DBufPacket** structure consists of the following members:

- BufY, BufX** lets the system keep track of where the object was located “last screen” (as compared to the **Bob** structure members called **oldY** and **oldX** that tell where the object was two screens ago). **BufY** and **BufX** provide for correct restoration of the background within the currently active drawing buffer.
- BufPath** assures that the system restores the backgrounds in the correct sequence; it relates to the **VSprite** members **DrawPath** and **ClearPath**.
- BufBuffer** This field must be set to point to a buffer the same size as the Bob’s **SaveBuffer**. This buffer is used to store the background for later restoration when the system moves the object. This buffer must be allocated from CHIP memory.

The next sub-section shows how to pull all these members together to make a double-buffered Bob.

## Creating a Double-Buffered Bob

To create a double-buffered Bob, execute a code sequence similar to the following:

```
struct Bob      myBob;
struct DBufPacket myDBufPacket;

/* allocate a DBufPacket for myBob
** same size as previous example
*/
if (NULL != (myDBufPacket.BufBuffer = AllocRaster(48, 20 * 5)))
{
    /* tell Bob about its double buff status */
    myBob.DBuffer = myDBufPacket;
}
```

### NOTE

The above example routines makeBob() and freeBob() correctly handle double buffering.

## Collisions And User Structure Extensions

### DETECTING GEL COLLISIONS

All GELs, including VSprites, can participate in the software collision detection. Simple Sprites must use *hardware* collision detection. See the *Amiga Hardware Reference Manual* for information about hardware collision detection.

Two kinds of collisions are handled by the system routines: GEL-to-boundary hits and GEL-to-GEL hits. You can set up as many as 16 different routines to handle different collision combinations; one routine to handle the boundary hits, and up to fifteen more to handle different inter-GEL hits. You supply the actual collision handling routines, and provide their addresses to the system so that it can call them as needed (when the hits are detected). These addresses are kept in the **collision handler table**, a GelsInfo member. Which routine is called depends on the 16 bit **MeMask** and **HitMask** members of the VSprite structures involved in the collision. When you call **DoCollision()**, the system goes through the GelsInfo list, which, is constantly kept sorted by y, x position. If a GEL intersects the boundaries, and the GELs HitMask indicates it appropriate, the boundary collision routine is called. When **DoCollision()** finds that two GELs overlap, it compares the **MeMask** of one with the **HitMask** of the other. If corresponding bits are set in both, it calls the appropriate inter-GEL collision routine at the table position corresponding to the bits in the **HitMask** and **MeMask**, as outlined below.

### Preparing for Collision Detection

Before you can use the system to detect collisions between GELS, you must allocate and initialize a table of collision-detection routines, the Gelsinfo's **CollHandler**. This table is an array of pointers to the actual routines that you have provided for your collision types. You must also prepare some members of the VSprite structure: **CollMask**, **BorderLine**, **HitMask**, and **MeMask**.

## Building a Table of Collision Routines

The Collision Table is a structure, `CollTable`, defined in `graphics/gels.h`. It is accessed as the `CollHandler` member of the `GelsInfo` structure. The table only needs to be as large as the number of bits for which you wish to provide collision processing. It is safest, though, to allocate space for all 16 entries, considering the small amount of space required. When the `View` structure is first initialized, the system sets all of the values of the collision routine pointers to `NULL`. The application must then initialize those table entries that correspond to the `HitMask` and `MeMask` bits that you plan to use.

The application does not set the vectors by directly changing the array; it gives the address to `SetCollision( )` routine, and it manipulates the array for the application. The parameters for this routine are as follows:

```
ULONG          num;
VOID           (*routine)( );
struct GelsInfo *GInfo;

SetCollision(num, routine, GInfo)
```

where

**num**        is the collision vector number

**routine**    is a pointer to the user collision routine

**GInfo**      is a pointer to a `GelsInfo` structure

For example:

```
struct GelsInfo myGelsInfo;

VOID myCollisionRoutine(GELA, GELB)    /* sample collision routine */
struct VSprite *GELA;
struct VSprite *GELB;
{
    /* process gels here - GELA and GELB point to the base VSprites of
    ** the gels, you can use the user extensions to identify what hit
    ** (if you need the info).
    */
}

/* myGelsInfo must be allocated and initialized */
SetCollision(15, myCollisionRoutine, &myGelsInfo);
```

## VSprite Collision Mask

The `CollMask` member of the `VSprite` is a pointer to a memory area that you have allocated for holding the collision mask of that GEL. This area must be in CHIP memory. This area's size is the equivalent of one bit-plane of the GELs image. The collision mask is *usually* the same as the shadow mask of the GEL, formed from a *logical-or* combination of all planes of the image. The following figure shows an example collision mask.



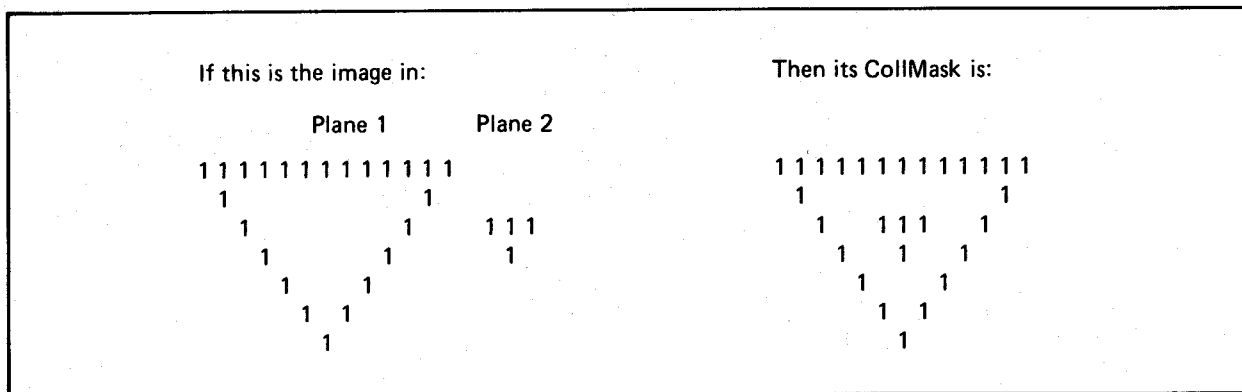


Figure 25-3: A Collision Mask

For example, here are typical program statements to reserve an area for the sprite shadow, initialize the pointer correctly, and then specify that the system uses the same mask for collisions (this example assumes a one-word-wide, five-line-high image):

```
/* reserve 5 16-bit locations for sprite shadow. */
WORD chip myShadowData[5];

myVSprite.ImageShadow = myShadowData; /* and point to it */
myVSprite.CollMask     = myShadowData; /* collision mask is same as shadow */
```

Alternatively, you may have a collision mask that is not derived from the image. In this case, the actual image isn't relevant. The system will not register collisions unless the other objects touch the collision mask. If the collision mask is smaller than the image, other objects will pass through the edges without a collision.

### VSprite BorderLine

For faster collision detection, the system uses the **BorderLine** member of the **VSprite** structure. **BorderLine** specifies the location of the horizontal *logical-or* combination of all of the bits of the object. It may be compared to taking the whole objects' shadow/collision mask and squishing it down into a single horizontal line. You provide the system with a place to store this line. The size of the data area you allocate must be at least as large as the image width.

In other words, if it takes three 16-bit words to hold one line of a GEL, then you must reserve three words for the **BorderLine**. In the **VSprite** examples, the routine `makeVSprite( )` correctly allocates and initializes the collision mask and **borderline**.

For example:

```
/* reserve space for BorderLine for this Bob */
WORD myBorderLineData[3];

/* tell the system where it is */
myVSprite.BorderLine = myBorderLineData;
```

Here is a sample of an object and its **BorderLine** image:

## OBJECT

```
011000001100
001100011000
001100011000
000110110000
000010100000
```

## BORDERLINE IMAGE

```
011110111100
```

Using this squished image, the system can quickly determine if the image is touching the left or rightmost boundary of the drawing area.

To establish a default **BorderLine** and **CollMask** data, make a system call to **InitMasks( )**.

### VSprite HitMask and MeMask

Software collision detection is independently enabled and disabled for each GEL. Further, you can specify which of 16 possible collision routines you wish to have automatically executed.

**DoCollision( )**, in addition to sensing an overlap between objects, uses these masks to determine which routine(s) (if any) the system will call when a collision occurs.

When the system determines a collision, it *ands* the **HitMask** of the upper-leftmost object in the colliding pair with the **MeMask** of the lower-rightmost object of the pair. The bits that are 1s after the *and* operation choose which of the 16 possible collision routines to perform.

- If the collision is with the boundary, bit 0 is a 1 and the system calls the collision handling routine number 0. You assign bit 0 to the condition called "boundary hit." The system uses the flag called **BORDERHIT** to indicate that an object has landed on or moved beyond the outermost bounds of the drawing area (the edge of the clipping region). The complete VSprite example in this chapter uses collision detection to check for border hits.
- If you set any one of the other bits (1 to 15), then the system calls the collision handling routine corresponding to the bit set.

If more than one bit is set in both masks, the system calls the vector corresponding to the rightmost (the least significant) bit *only*.

### Using HitMask and MeMask

This section illustrates the use of the **HitMask** and **MeMask** to define one type of collision.

Suppose there are two classes of objects that you wish to control on the screen: **ENEMYTANK** and **MYMISSILE**. Objects of class **ENEMYTANK** should be able to pass across one another without registering any collisions. Objects of class **MYMISSILE** should also be able to pass across one another without collisions. However, when

MYMISSILE and ENEMYTANK collide, the system should generate a call to a collision routine.

Choose a pair of collision detect bits not yet assigned within **MeMask**, one to represent ENEMYTANK, the other to represent MYMISSILE. You will use the same two bits in the corresponding **HitMask**

	<u>MeMask</u>	<u>HitMask</u>	
Bit #	2 1	2 1	
GEL #1	0 1	1 0	ENEMYTANK
GEL #2	0 1	1 0	ENEMYTANK
GEL #3	1 0	0 1	MYMISSILE

In the example, bit 1 represents ENEMYTANK objects. In the **MeMask**, bit 1 is a 1 for GEL #1 and says "I am an ENEMYTANK." Bit 2 is a zero says this object is *not* a MYMISSILE object.

In bit 1 of the **HitMask** of GEL #1, the 0 bit there says, "I will not register collisions with other ENEMYTANK objects." However, the 1 bit in bit 2 says, "I *will* register collisions with MYMISSILE objects."

Thus when a call to **DoCollision( )** occurs, for any objects that appear to be colliding, the system *ands* the **MeMask** of one object with the **HitMask** of the other object. If there are non-zero bits present, the system will call one of your collision routines.

In this example, suppose that the system senses a collision between ENEMYTANK #1 and ENEMYTANK #2. Suppose also that ENEMYTANK #1 is the top/leftmost object of the pair. Here is the way that the collision testing routine performs the test to see if the system will call any collision-handling routines:

<u>Bit #</u>	<u>2</u>	<u>1</u>
ENEMYTANK #1 MeMask	0	1
ENEMYTANK #2 HitMask	1	0
Result of <i>and</i>	0	0

Therefore, the system does not call a collision routine.

Suppose that **DoCollision( )** finds an overlap between ENEMYTANK #1 and MYMISSILE, and MYMISSILE is the top/leftmost of the pair:

<u>Bit #</u>	<u>2</u>	<u>1</u>
MYMISSILE #1 MeMask	1	0
ENEMYTANK #2 HitMask	1	0
Result of <i>and</i>	1	0

Therefore, the system calls the collision routine at position 2 in the table of collision-handling routines.

## SETTING UP FOR BOUNDARY COLLISIONS

To specify the region in the PlayField that the system will use to define the outermost limits of the GEL boundaries, you use these **GelsInfo** members: **topmost**, **bottommost**, **leftmost**, and **rightmost**. The **DoCollision( )** routine tests these boundaries when determining boundary collisions within this **RastPort**. They have nothing whatsoever to do with graphical clipping. Graphical clipping makes use of the **RastPorts** clipping rectangle.

Here is a typical program segment that assigns the members correctly (for boundaries 50, 100, 80, 240). It assumes that you already have a **RastPort** structure pointer named **myRastPort**.

```
myRastPort->GelsInfo->topmost      = 50;
myRastPort->GelsInfo->bottommost    = 100;
myRastPort->GelsInfo->leftmost      = 80;
myRastPort->GelsInfo->rightmost     = 240;
```

## Parameters Passed To Your Routines

When **DoCollision( )** calls one of your collision routines, it passes parameters which are described in the next paragraphs.

## Parameters To Your Boundary Routines

During the operation of the **DoCollision( )** routine, if you have enabled boundary collisions for a GEL (by setting the least significant bit of its **HitMask**) and it has crossed a boundary, the system calls the boundary routine you have defined.

### NOTE

The system will call the routine once for every GEL that has hit, or gone outside of the boundary.

The system will call your routine with the following two arguments:

- A pointer to the **VSprite** structure of the GEL that hit the boundary
- A flag word containing one to four bits set, representing top, bottom, left and right boundaries, telling you which of the boundaries it has hit or exceeded. To test these bits, compare to the constants **TOPHIT**, **BOTTOMHIT**, **LEFTHIT**, and **RIGHTHIT**.

See the "Complete VSprite Example" above for sample code using boundary collision.

## Parameters To Your Inter-GEL Routines

If, instead of a GEL-to-boundary collision, **DoCollision( )** senses a GEL-to-GEL collision, the system calls your collision routine with the following two arguments:

- Address of the VSprite that is the uppermost (or leftmost if y coordinates are identical) GEL of a colliding pair.
- Address of the VSprite that is the lowermost (or rightmost if y coordinates are identical) GEL of the pair.

### Handling Multiple Collisions

When multiple elements collide within the same display field, the following set of sequential calls to the collision routines occurs:

- The system issues each call in a sorted order for GELs starting at the upper left-hand corner of the screen and proceeding to the right and down the screen.
- For any colliding GEL pair, the system issues only one call, to the collision routine for the object that is the topmost and leftmost of the pair.

### ADDING USER EXTENSIONS TO GEL DATA STRUCTURES

This section describes how to expand the size and scope of the VSprite, Bob and AnimOb data structures. In the definition for these structures, there is an item called UserExt at the end of each. If you want to expand these structures (to hold your own special data), you define the UserExt member before the *graphics/gels.h* file is included. If this member has already been defined when the *graphics/gels.h* file is parsed, the compiler preprocessor will extend the structure definition automatically. If these members have not been defined, the system will make them SHORTs, and you may still consider these as being reserved for your private use.

To show the kind of use you can make of this feature, the example below adds speed and acceleration figures to each GEL by extending the VSprite structure. When your collision routine is called, it could use these values to transfer energy between the two colliding objects (say, billiard balls). You would have to set up additional routines, executed between calls to DoCollision( ), that would add the values to the GELs position appropriately.

You could define a structure that you want to have associated with each of the GELS, similar to the following:

```
struct myInfo
{
    short xvelocity;
    short yvelocity;
    short xaccel;
    short yaccel;
};
```

These members are for example only. You may use any definition for your user extensions.

You would also provide the following line, to extend the VSprites structure:

```
#define VUserStuff struct myInfo
```

To extend the Bobs structure:

```
#define BUserStuff struct myInfo
```

To extend the **AnimObs** structure:

```
#define AUserStuff struct myInfo
```

When the system is compiling the *graphics/gels.h* file with your program, the compiler preprocessor substitutes “struct myInfo” everywhere that **UserExt** is used in the header. The structure is thereby customized to include the items you wish to associate with it.

#### NOTE

You cannot use the “C” construct “typedef” for the above statements. If you want to substitute your own data type for one of the **UserStuff** variables, you must use a “#define”.

## Animation Concepts

To perform animation, an artist must first produce a series of drawings. Each drawing differs from the preceding one so that when each frame in a stack is viewed sequentially, the images appear to flow naturally.

The creation of an animation using just one stack is analogous to a child’s flip-book, which were the earliest form of animation. Later, with the advent of film, classic animation became a refined art; watch any older Warner Brothers Cartoon to appreciate this.

The background for each scene was painted only once. Then, all characters, moving things, and objects that things go behind were painted on transparent sheets of celluloid. Each sheet had just one view of one part on it, and was called a **cell**. These celluloid stills were placed over the background, so that the objects that were to appear closest to the viewer were on the last cells to be put down. The final step was to capture the effect on a **frame** of film. Then, for the next photo, anything that moved had the next cell from its stack put down, taking care that it was at the right depth, so it still appeared in front of and behind the correct things. Then another photo, ad infinitum.

The animation crews realized that they could further break down each object, or character, into its component parts. If the character’s arm was the only thing to move, then that was all they had to redraw. By having all the character’s components share a common registration mark, it could easily be moved about the display, and the arm would always appear in the right place.

If you were animating a walking human, for example, you might have five stacks of celluloids: one stack for each limb, along with one for the trunk and head. You would first put down the cells of the far arm and leg, then the trunk/head, then the near arm and leg. Then, if you got the registration marks lined up correctly, it would appear to be a complete body, and you would photograph the result. Then you would exchange new cells (only those that changed), and repeat the procedure, maybe moving the registration mark to make the whole object move.

## Animation Structures and Controls

There are two structures involved in Amiga animation: **AnimComp** and **AnimOb**.

The **AnimComp** (for Animation Component), is an extension of the **Bob** structure; all the graphic fields of the underlying Bobs are set up as described in the preceding chapters. Its main use is to provide linkages so a stack of **AnimComps** may be sequenced automatically, and so multiple sequences can be grouped to reference a common

registration point. Each AnimComp represents just one version of an object such as a rotating ball, or an arm.

The second structure is the AnimOb, which provides the variables needed for overall control of a set of AnimComps. The AnimOb itself contains no imagery; it provides the common registration point, and specifies how the system should automatically move that point.

AnimComp and AnimOb work by emulating the real-world animation methodology. Every AnimComp is analogous to one sheet of celluloid, representing one (perhaps the only) image of an object, or part of an object. AnimComps have pointers, PrevSeq and NextSeq, that let you group these cells into stacks. AnimComps also have PrevComp and NextComp pointers, that let you group stacks into more complex objects.

A single stack representing a rotating ball, viewed sequentially, could be a complete Boing Ball animation. A sequence of arms would appear to swing back and forth at the shoulder. The system displays the initial AnimComp (the "top of the stack"); then switches to the AnimComp's NextSeq (which is also an AnimComp), then *its* NextSeq, etc. When it reaches the end of the sequence, you can have it start over automatically by having the last AnimComp's NextSeq point to the first AnimComp in the stack (a circular list). So to do a Boing Ball, or a simple inchworm, you could have only one stack that was repeated endlessly.

But to do the multi-component walking human as described previously, you would need to have five stacks, which on the Amiga means five circular lists of AnimComps. To group these stacks into one cohesive unit, you link the stacks via the PrevComp and NextComp pointers, (say, the Arms, Legs, and Trunk sequences) into one larger object (the human figure). All the stacks would share a common AnimOb, so that it can be controlled conveniently.

## ANIMATION TYPES

The GELs system provides several ways of setting up automatic animation, loosely based on some categories of movement in real life. Some things (like balls, arrows, 16 ton weights) can move independently of the background, and look even more realistic if they tumble or rotate as they move; other things (like worms, wheels, and people) must be anchored to the background, or they will appear to *slide* unnaturally.

The system software allows these types of animation through simple motion control, motion control with sequenced drawing, and sequenced drawing using Ring Motion Control.

### Simple Motion Control

To produce motion of a simple object, like a smooth ball, the artist simply moves that cell relative to the overall display, a little at a time. This is simple motion control, and can be accomplished with one AnimComp and one AnimOb, by simply changing the AnimOb's position every N frames. The apparent speed of the object is a combination of how often it is moved (every frame, every other frame, etc.) and how far it is moved (how much the AnimOb's AnX and AnY are changed).

### Sequenced Drawing

To make the ball appear to rotate is a little more complex. To produce apparent movement within the image, sequencing is used. This is done by having a stack of cells that are laid down one after the other, a frame at a time. The stack can be made to repeat automatically, for endless movement. So, when you combine sequenced drawing with simple motion control, you can do things like having a rotating ball bounce around. This can be accomplished with several AnimComps (one for each view of the ball) and one AnimOb, with the system automatically

sequencing the AnimComps, and changing the AnimOb's position every N frames.

## Ring Motion Control

Making a worm appear to crawl is similar to the rotating ball. There is still a stack of frames that are sequenced automatically, and one controlling AnimOb. But each cell is drawn to appear to move relative to the common reference point, which remains stationary through the stack. And, rather than the AnimOb's reference point moving each frame, you tell the system how far to move it at the end of each sequence.

In other words, as the object moves relative to the registration point as the stack is viewed, when the stack is restarted, the registration point must be relocated to maintain fluid apparent motion. Ring motion control looks like:

Draw1, Draw2, Draw3, Move, Draw1, Draw2, Draw3, Move, Draw1...

The following figures show some sequences that illustrate these techniques.

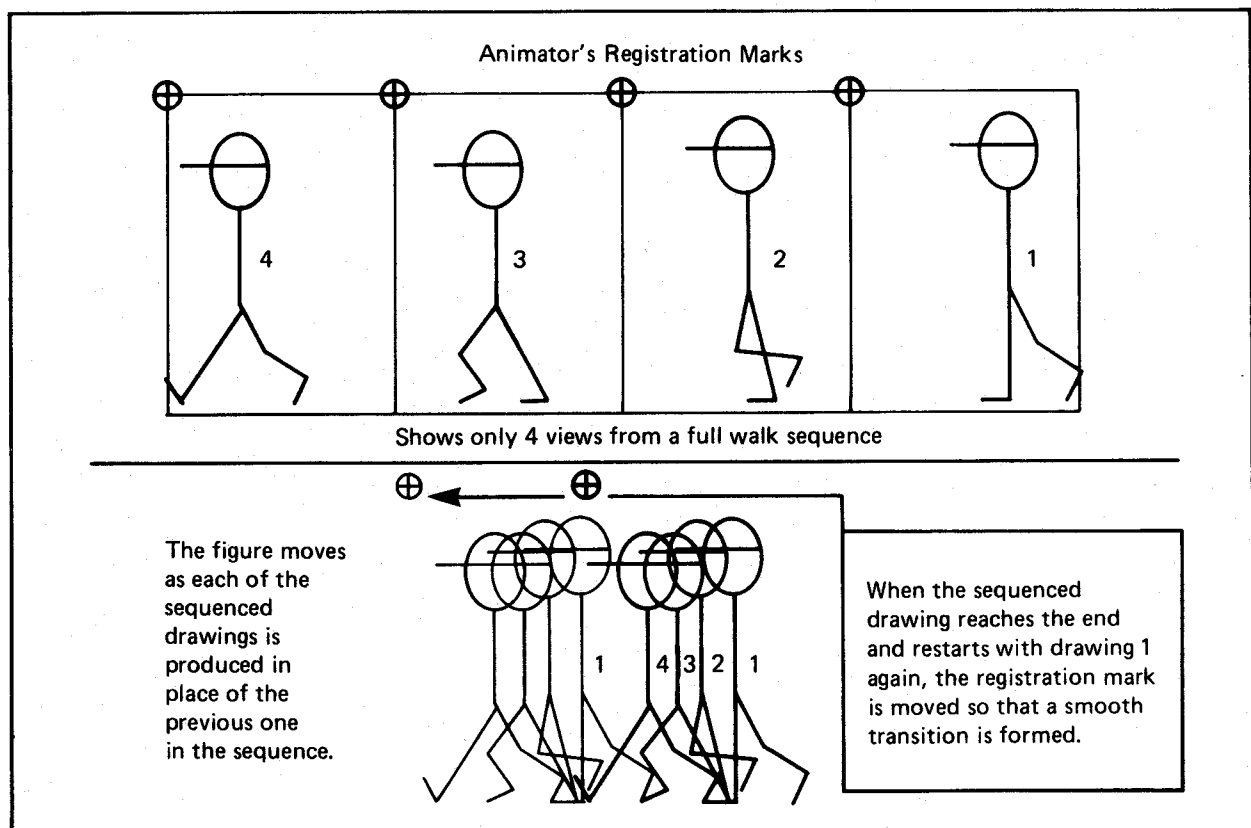


Figure 25-4: Ring Motion Control



## CHARACTERISTICS OF THE ANIMATION SYSTEM

For each Object, you initially specify:

- The starting position of this object
- Its velocity and acceleration (optional).
- A pointer to the first of its animation components.
- A pointer to a special animation routine related to this object (optional).
- Your own extensions to this structure (optional).

For each Object's Components, you initially specify:

- A pointer to the Component's controlling Object
- Initial and alternate views, their timing and order.
- The initial inter-component drawing priorities.
- A pointer to a special animation routine related to this component (optional).
- Your own extensions to this structure (optional).

The animation system automatically:

- Sequences through the alternate views.
- Moves linked objects as a group.
- Maintains inter-object prioritization.
- Calls your routines, if specified.

## SPECIFYING ANIMATION COMPONENTS

When the Components of a complex animation object are built, the PrevComp and NextComp pointers must be initialized *for the initial view of each animation sequence only*. In the previous figure, the pointers shown connecting the five sequences of the 'human' AnimOb illustrate this point. The components that are not active should have their PrevComp and NextComp pointers set to NULL.

### NOTE

You cannot store data in the empty PrevComp and NextComp fields. As the system cycles through the AnimComps, the next AnimComp becomes the head of the sequence. When this happens, the NextComp and PrevComp field for the old head are set to NULL, and the new head AnimComp is linked in, *in place of the old one*.

Whenever the animation system senses that one of the animation components has “timed out”, the next AnimComp in that sequence becomes active. The lists are automatically adjusted so that only the new HeadComp has its PrevComp and NextComp pointers set. The complete animation object is retained.

To find the active animation component at run time, you must look at the AnimOb’s HeadComp field. To find the active AnimComp from any AnimComp, use the HeadOb field to find the AnimOb first.

## SEQUENCING COMPONENTS

To specify the sequencing of any of the individual parts, the pointers called PrevSeq and NextSeq are used to build a doubly-linked list. The sequence can be made circular (and usually is) by linking the first and last components.

The system is designed so that only one of the Components in any given sequence is “active” (being displayed) at a given point in time. It is the only one in the sequence that is (or is about to be) linked into the GelsInfo list. The Timer determines how long each Component in the sequence remains active, as described above.

### Sequence List Traversal

Within each AnimOb is a number of sequences. HeadComp of the AnimOb points to the first AnimComp in the list of sequences.

Each sequence is identified by its “active” AnimComp. There can only be one active AnimComp in each sequence. The sequences are linked together by their active AnimComps; for each of these the NextComp and PrevComp fields link the sequences together to create a list. The first sequence in the list (HeadComp of the AnimOb), has its PrevComp set to NULL. The last sequence in the list has its NextComp set to NULL. None of the inactive AnimComps should have NextComp or PrevComp set.

Within each sequence, the AnimComps are linked together with the NextSeq and PrevSeq fields. The list formed by these links may or may not be a loop. For the list to be a loop, the NextSeq of the last AnimComp must point back to the first AnimComp, and the PrevSeq of the first AnimComp must point to the last AnimComp. If the list is a loop, then the system will continue to cycle through the list until it is stopped. If the list is not a loop, then the program must act to restart the sequence after the last item is displayed. The AnimCRoutine of the last AnimComp may be used to do this.

### Component Ordering

The PrevSeq, NextSeq, PrevComp and NextComp linkages have no bearing on the order in which Components in any given frame are drawn. To specify the inter-Component and inter-Object priorities (so that further things appear behind nearer things), the Before and After pointers in the initially active AnimComp’s underlying Bob structure are linked in to the rest of the system, as described previously in the Bob.

This set-up also needs to be done once, *for the initially active components of the Animation Object only.*

The animation system adjusts the Before and After pointers of the Bob structures to constantly maintain the inter-Component drawing sequence, even though different Components are being made active as sequencing occurs.

These pointers also assure that one complete *object* always has priority over another object. The Bob Before and After pointers are used to link together the last AnimComp’s Bob of one AnimOb to the first AnimComp’s Bob of

the next AnimOb.

### Position of an AnimComp

To specify the placement of each component relative to the registration point (the **AnimOb**), you set the **AnimComp** members **XTrans** and **YTrans**. These values can be positive or negative.

You can have the system call custom routine(s) (that you provide) by specifying an **AnimORoutine** or **AnimCRoutine**. See the section called "Your Own Animation Routine Calls" for details.

The human figure Animation is built of several components (the arms, legs, trunk), all of which are sequenced. The **AnimOb** points to the head component. The "head" component may be any of the components of the object. Notice that, once the system has a pointer to one of the active components, it can find the other active components by following **PrevComp** and **NextComp**. Astute readers will notice that, in this particular example, (because all the sequences have the same number of components, and all the sequences are switched simultaneously) the walking human could be implemented as one sequence, with the entire figure in each pose.

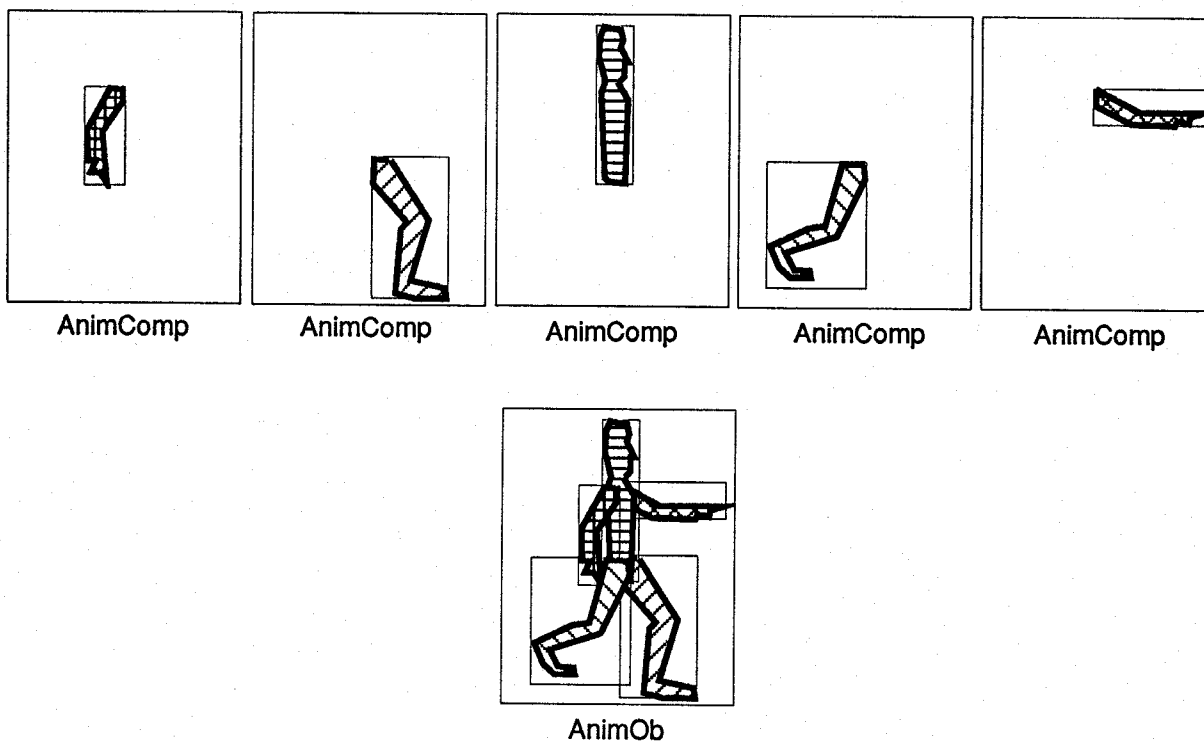


Figure 25-5: Linking AnimComps For a Multiple Component AnimOb

## ANIMATION SEQUENCING

### SPECIFYING TIME FOR EACH IMAGE

The **AnimComp** members **Timer** and **TimeSet** are used to specify how long the system should keep each sequential image on the screen.

When the system makes an animation component active, it copies the value you have put in the **TimeSet** member into the **Timer** member. On each subsequent call to **Animate()**, the system decrements **Timer**; as long as it is greater than zero, that **AnimComp** remains active. When the **Timer** value reaches zero, **Animate()** makes the next **AnimComp** in the sequence active, and the process repeats. If you *initialize the value in TimeSet to zero*, the system will *not sequence this component* (and **Timer** will remain zero). The above figures illustrate how to initialize the “next sequential image” pointers to have the system step from one image to the next.

### Setting Up Simple Motion Control

In this form of animation, you can specify objects that have independently controllable velocities and accelerations in the X and Y directions. Components can still sequence.

The variables that control this motion are located in the **AnimOb** structure and are called:

- **YVel, XVel**—the velocities in the y and x directions. These values are added to the position values on each call to **Animate()**.
- **YAccel, XAccel**—the accelerations in the y and x directions. These values are added to the velocity values on each call to **Animate()**. The velocity values are updated *before* the position values.

### Setting Up Ring Motion Control

To make a given component *trigger* a move of the **AnimOb** (move the registration mark to a new location), you set the **RINGTRIGGER** bit of that component's **Flags**. When the system software encounters this flag, it adds the values of **RingXTrans** and **RingYTrans** found in the Head **AnimOb** structure to its **AnX** and **AnY**. The *next* time you execute **DrawGList()**, the drawing sequence will use the new registration point.

You usually set **RINGTRIGGER** in only one of the animation components in a sequence (the last one); however, you can use this flag and the translation variables any way you wish.

### Using Sequenced Drawing and Motion Control

If you are using Ring Motion Control, you will probably set the velocity and acceleration variables to zero.

Again, consider the example of a person walking. With Ring Motion Control, as each foot falls it is positioned on the ground exactly where originally drawn. If you included a velocity value, the person's foot would not be stationary with respect to the ground, and the person would appear to “skate” rather than walk. If you set the

velocity and acceleration variables at zero, you avoid this problem.

When the system activates a new sequence component, it checks that component's **Flags** to see if the **RINGTRIGGER** bit is set. If so, the system adds **RingYTrans** and **RingXTrans** to **AnY** and **AnX** respectively. See the section called "Animation Types" for details.

## ANOTHER LOOK AT THE ANIMOB

The **AnimOb** is the structure that gives overall control. All **AnimComps** associated with an **AnimOb** use the **AnimOb's** Y,X position as a reference point into the display. This way, by changing one variable, you can control the (possibly many) Component pieces, and the entire object will always appear on the display as a cohesive group.

### Position of an AnimOb

To specify where the common registration point is located, use the **AnimOb** structure members **AnX** and **AnY**. The following figure illustrates that each component has its own offset from the object's registration point.

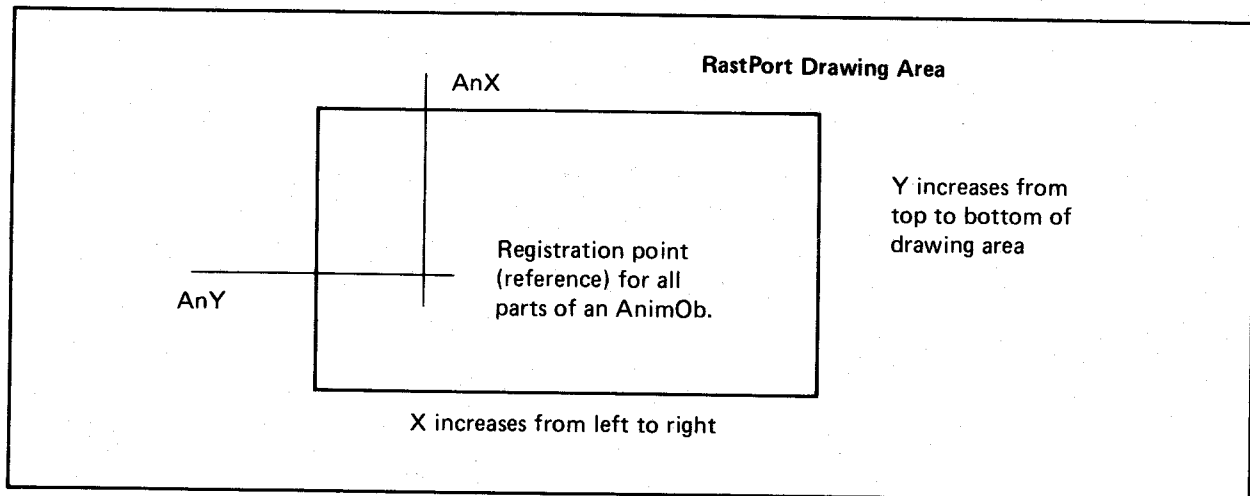


Figure 25-6: Specifying an AnimOb Position

When you change the animation object's **AnX** and **AnY**, all of the component parts will be redrawn relative to it the next time **DrawGList()** is called.

## ADDING ANIMATION OBJECTS

Use the routine **AddAnimOb( )** To add animation objects to the controlled object list. This routine will link the **PrevOb** and **NextOb** pointers to chain all the **AnimObs** that the system is controlling.

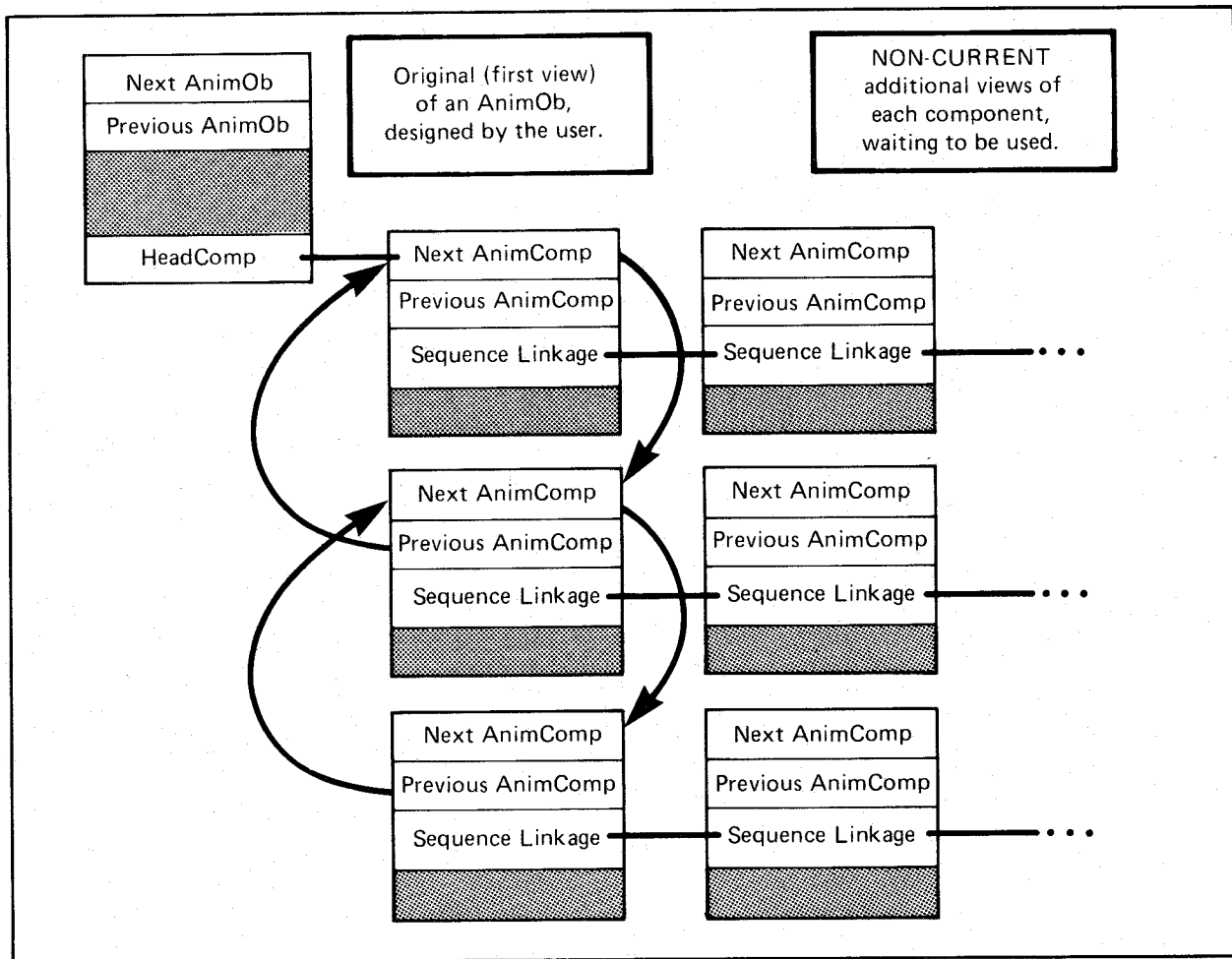


Figure 25-7: Linking of an AnimOb

## YOUR OWN ANIMATION ROUTINE CALLS

The **AnimOb** and **AnimComp** structures include pointers to your own routines that you want the system to call. If you want a routine to be called, you put the address of the routine in this member. If no routine is to be called, you must set this variable to **NULL**. These routines are passed one parameter, a pointer to the **AnimOb** or **AnimComp** it was related to. You can use user structure extensions to hold the variables you need for your own routines.

For example, if you provide a routine:

```
VOID MyOCode (anOb)
struct AnimOb *anOb;
{
/* whatever needs to be done */
}
```

Then, if you put the address of the routine in an **AnimObs** structure:

```
myAnimOb.AnimORoutine = MyOCode;
```

It will be called when **Animate()** processes this **AnimOb**, and it will be passed the address of this **AnimOb**.

The **AnimORoutines** and **AnimCRoutines** are called as follows:

For every **AnimOb** in the list:

- Update its location and velocities.
- Call the **AnimOb**'s special routine if one is supplied.
- For each component of the **AnimOb**:
  - If this sequence times out, switch to the new one.
  - Call this component's special routine if one is supplied.
  - Set the sequence's **VSprite**'s y,x coordinates based on whatever these routines have caused.

## MOVING THE OBJECTS

When you have defined all of the structures and have established all of the links, you can call the **Animate()** routine to move the objects.

## THE ANIMATE() SYSTEM CALL

**Animate()** adjusts the positions of the objects as described above, and calls the various subroutines (**AnimCRoutines** and **AnimORoutines**) that you have specified.

After the system has completed the **Animate()** routine, as some **GELs** may have been moved, the **GelsInfo** list order may possibly be incorrect. Therefore, the list must be re-sorted, as always, before passing it to a system routine.

If you are using collision detection, you then perform **DoCollision()**. Your collision routines may also have an effect on the relative position of the **GELs**. Therefore, you should again call **SortGList()** to assure that the system correctly orders the objects before you call **DrawGList()**. When you call **DrawGList()**, the system renders all the **GELs** it finds in the **GelsInfo** list, and the changes caused by the **Animate()** can be seen.

This is illustrated in the following typical call sequence:

```
/* ... setup of graphics elements and objects */

Animate(myAnimKey, rp);      /* "move" objects per instructions */
SortGList(rp);              /* put them in order */
DoCollision(rp);            /* software collision detect/action */
SortGList(rp);              /* put them back into right order */
DrawGList(vp, rp);          /* draw into current RastPort */
```

## THE ANIMKEY

The system uses one pointer, known as the **AnimKey**, to keep track of all the **AnimObs** via the **PrevOb** and **NextOb** pointer linkage. The **AnimKey** points to the **AnimOb** that is first in the **PrevOb/NextOb** linkage. As each new Object is added (via **AddAnimOb()**), it is linked in at the beginning of the list, so **AnimKey** will always point to the object most recently added to the list.

To search forward through the list, start with the AnimKey and move forward on the NextOB link. Continue to move forward until the NextOb is null, indicating the end of the list. The PrevOb link will allow you to move back to a previous object.

#### NOTE

It is important that the NextOb link of the last object is NULL, and that the PrevOb of the first object is NULL. In fact, the system expects the animation object lists to be EXACTLY the way that they are described above. If they are not, the system will have unexpected results.

### STANDARD GEL RULES STILL APPLY

Before you use the animation system, you must have called the routine **InitGels()**.

The section called "Getting the List of Bobs in Order" described how the system maintains the list of GELs to draw on the screen according to their various data fields. The animation system selectively adds GELs to and removes GELs from this list of screen objects during the **Animate()** routine. On the next call to **DrawGList()**, the system will draw the GELs in the list into the selected **RastPort**.

### ANIMATIONS SPECIAL NUMBERING SYSTEM

Velocities and accelerations can be either positive or negative. The system treats the velocity, acceleration and Ring values as fixed-point binary fractions, with the decimal point at position 6 in the word. That is:

vvvvvvvvvv.vvvvvv

where v stands for actual values that you add to the x or y (**AnX**, **AnY**) positions of the object for each call to **Animate()**, and f stands for the fractional part. By using a fractional part, you can specify the speed of an object in increments as precise as 1/64th of an interval.

If you set the value of **XVel** at 0x0001, it will take 64 calls to the **Animate()** routine before the system will modify the object's x coordinate position by a step of one. The system constant **ANFRACSIZE** can be used to shift values correctly. So if you set the value to (1 << **ANFRACSIZE**), it will be set to 0x0040, the value required to move the object one step per call to **Animate()**. The system constant **ANIMHALF** can be used if you want the object to move every other call to **Animate()**.

Each call you make to **Animate()** simply adds the value of **XAccel** to the current value of **XVel**, and **YAccel** to the current value of **YVel**, modifying these values accordingly.

## Complete Example Program

The following program produces a display with a single animation object. The animation object consists of four sequences: a boing ball which appears to rotate on its axis, and three orbiting satellites. The display is single-buffered, with a command line option to make it double buffered.

Here is the file *animtools.h*, which contains the typedefs required for this example program.



```

#ifndef GELTOOLS_H
#define GELTOOLS_H

/* these data structures are used by the functions in animtools.c to
** allow for an easier interface to the animation system.
*/

/* data structure to hold information for a new vsprite.
** note that:
**     NEWVSPRITE myNVS;
** is equivalent to:
**     struct newVSprite myNVS;
*/
typedef struct newVSprite
{
    WORD        *nvs_Image;      /* image data for the vsprite */
    WORD        *nvs_ColorSet;   /* color array for the vsprite */
    SHORT        nvs_WordWidth;  /* width in words */
    SHORT        nvs_LineHeight; /* height in lines */
    SHORT        nvs_ImageDepth; /* depth of the image */
    SHORT        nvs_X;          /* initial x position */
    SHORT        nvs_Y;          /* initial y position */
    SHORT        nvs_Flags;      /* vsprite flags */
} NEWVSPRITE;

/* data structure to hold information for a new bob.
** note that:
**     NEWBOB myNBob;
** is equivalent to:
**     struct newBob myNBob;
*/
typedef struct newBob
{
    WORD        *nb_Image;      /* image data for the bob */
    SHORT        nb_WordWidth;  /* width in words */
    SHORT        nb_LineHeight; /* height in lines */
    SHORT        nb_ImageDepth; /* depth of the image */
    SHORT        nb_PlanePick;  /* planes that get image data */
    SHORT        nb_PlaneOnOff; /* unused planes to turn on */
    SHORT        nb_BFlags;     /* bob flags */
    SHORT        nb_DBuf;       /* 1=double buf, 0=not */
    SHORT        nb_RasDepth;   /* depth of the raster */
    SHORT        nb_X;          /* initial x position */
    SHORT        nb_Y;          /* initial y position */
} NEWBOB ;

/* data structure to hold information for a new animation component.
** note that:
**     NEWANIMCOMP myNAC;
** is equivalent to:
**     struct newAnimComp myNAC;
*/
typedef struct newAnimComp
{
    WORD        (*nac_Routine)(); /* routine called when Comp is displayed. */
    SHORT        nac_Xt;          /* initial delta offset position. */
    SHORT        nac_Yt;          /* initial delta offset position. */
    SHORT        nac_Time;        /* Initial Timer value. */
    SHORT        nac_CFlags;      /* Flags for the Component. */
} NEWANIMCOMP;

/* data structure to hold information for a new animation sequence.
** note that:
**     NEWANIMSEQ myNAS;
** is equivalent to:
**     struct newAnimSeq myNAS;
*/
typedef struct newAnimSeq
{
    struct AnimOb *nas_HeadOb; /* common Head of Object. */
    WORD        *nas_Images;   /* array of Comp image data */
    SHORT        *nas_Xt;      /* arrays of initial offsets. */
}

```

```

    SHORT  *nas_Yt;           /* arrays of initial offsets.      */
    SHORT  *nas_Times;        /* array of Initial Timer value.    */
    WORD   (**nas_Routines)(); /* Array of fns called when comp drawn */
    SHORT  nas_CFlags;        /* Flags for the Component.          */
    SHORT  nas_Count;         /* Num Comps in seq (= arrays size)  */
    SHORT  nas_HitMask;       /* Hit mask.                         */
    SHORT  nas_MeMask;        /* Me mask.                          */
    SHORT  nas_SingleImage;   /* one (or count) images.            */
} NEWANIMSEQ;

```

```
#endif
```

Here is the file *animtools\_proto.h*, which contains prototypes for the file *animtools.c*.

```

struct GelsInfo *setupGelSys(struct RastPort *rPort, BYTE reserved);
VOID cleanupGelSys(struct GelsInfo *gInfo, struct RastPort *rPort);

struct VSprite *makeVSprite(NEWVSPRITE *nVSprite);
struct Bob *makeBob(NEWBOB *nBob);
struct AnimComp *makeComp(NEWBOB *nBob, NEWANIMCOMP *nAnimComp);
struct AnimComp *makeSeq(NEWBOB *nBob, NEWANIMSEQ *nAnimSeq);

VOID freeVSprite(struct VSprite *vsprite);
VOID freeBob(struct Bob *bob, LONG rasdepth);
VOID freeComp(struct AnimComp *myComp, LONG rasdepth);
VOID freeSeq(struct AnimComp *headComp, LONG rasdepth);
VOID freeOb(struct AnimOb *headOb, LONG rasdepth);

```

Here is the file *animtools.c*, which contains tools for the animation system.

```

/* animtools.c 19oct89
** original code by Dave Lucas.
** rework by CATS
**
** This file is a collection of tools which are used with the VSprite, Bob
** and Animation system software. It is intended as a useful EXAMPLE, and
** while it shows what must be done, it is not the only way to do it.
** If Not Enough Memory, or error return, each cleans up after itself
** before returning.
**
** NOTE: these routines assume a very specific structure to the
** gel lists. make sure that you use the correct pairs together
** (i.e. makeOb()/freeOb(), etc.)
**
** lattice c 5.04
** lc -bl -cfist -v -y animtools.c
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/gfx.h>
#include <graphics/gels.h>
#include <graphics/clip.h>
#include <graphics/rastport.h>
#include <graphics/view.h>
#include <graphics/gfxbase.h>

#include "animtools/animtools.h"
#include "animtools/animtools_proto.h"

#include <proto/all.h>

/*-----
** setup the gels system. After this call is made you can use
** vsprites, bobs, anim comps, and anim obs.
**
** note that this links the GelsInfo structure into the rast port,
** and calls InitGels().
**
** all resources are properly freed on failure.

```

```

**
** It uses information in your RastPort structure to establish
** boundary collision defaults at the outer edges of the raster.
**
** This routine sets up for everything - collision detection and all.
**
** You must already have run LoadView before ReadyGelsSys is called.
*/
struct GelsInfo *setupGelsSys(struct RastPort *rPort, BYTE reserved)
{
    struct GelsInfo *gInfo;
    struct VSprite *vsHead;
    struct VSprite *vsTail;

    if (NULL != (gInfo =
        (struct GelsInfo *)AllocMem((LONG)sizeof(struct GelsInfo), MEMF_CLEAR)))
    {
        if (NULL != (gInfo->nextLine =
            (WORD *)AllocMem((LONG)sizeof(WORD) * 8, MEMF_CLEAR)))
        {
            if (NULL != (gInfo->lastColor =
                (WORD **)AllocMem((LONG)sizeof(LONG) * 8, MEMF_CLEAR)))
            {
                if (NULL != (gInfo->collHandler =
                    (struct collTable *)AllocMem((LONG)sizeof(struct collTable),
                        MEMF_CLEAR)))
                {
                    if (NULL != (vsHead = (struct VSprite *)AllocMem(
                        (LONG)sizeof(struct VSprite), MEMF_CLEAR)))
                    {
                        if (NULL != (vsTail = (struct VSprite *)AllocMem(
                            (LONG)sizeof(struct VSprite), MEMF_CLEAR)))
                        {
                            gInfo->sprRsrvd = reserved;
                            gInfo->leftmost = 0;
                            gInfo->rightmost =
                                (rPort->BitMap->BytesPerRow << 3) - 1;
                            gInfo->topmost = 0;
                            gInfo->bottommost = rPort->BitMap->Rows - 1;

                            rPort->GelsInfo = gInfo;

                            InitGels(vsHead, vsTail, gInfo);

                            return(gInfo);
                        }
                        FreeMem(vsHead, (LONG)sizeof(*vsHead));
                    }
                    FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
                }
                FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
            }
            FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
        }
        FreeMem(gInfo, (LONG)sizeof(*gInfo));
    }
    return(NULL);
}

/*-----
** free all of the stuff allocated by setupGelsSys().
** only call this routine if setupGelsSys() returned successfully.
** the GelsInfo structure is the one returned by setupGelsSys().
**
** It also unlinks the GelsInfo from the RastPort.
*/
VOID cleanupGelsSys(struct GelsInfo *gInfo, struct RastPort *rPort)
{
    rPort->GelsInfo = NULL;

    FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
    FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
}

```

```

FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
FreeMem(gInfo->gelHead, (LONG)sizeof(struct VSprite));
FreeMem(gInfo->gelTail, (LONG)sizeof(struct VSprite));
FreeMem(gInfo, (LONG)sizeof(*gInfo));
}

/*-----
** create a VSprite from the information given in nVSprite.
** use freeVSprite() to free this gel.
*/
struct VSprite *makeVSprite(NEWVSPRITE *nVSprite)
{
    struct VSprite *vsprite;
    LONG          line_size;
    LONG          plane_size;

    line_size = (LONG)sizeof(WORD) * nVSprite->nvs_WordWidth;
    plane_size = line_size * nVSprite->nvs_LineHeight;

    if (NULL != (vsprite =
        (struct VSprite *)AllocMem((LONG)sizeof(struct VSprite), MEMF_CLEAR)))
    {
        if (NULL != (vsprite->BorderLine =
            (WORD *)AllocMem(line_size, MEMF_CHIP)))
        {
            if (NULL != (vsprite->CollMask =
                (WORD *)AllocMem(plane_size, MEMF_CHIP)))
            {
                {
                    vsprite->Y          = nVSprite->nvs_Y;
                    vsprite->X          = nVSprite->nvs_X;
                    vsprite->Flags      = nVSprite->nvs_Flags;
                    vsprite->Width      = nVSprite->nvs_WordWidth;
                    vsprite->Depth      = nVSprite->nvs_ImageDepth;
                    vsprite->Height     = nVSprite->nvs_LineHeight;
                    vsprite->MeMask     = 1;
                    vsprite->HitMask    = 1;
                    vsprite->ImageData = nVSprite->nvs_Image;
                    vsprite->SprColors  = nVSprite->nvs_ColorSet;
                    vsprite->PlanePick  = 0x00;
                    vsprite->PlaneOnOff = 0x00;

                    InitMasks(vsprite);
                    return(vsprite);
                }
                FreeMem(vsprite->BorderLine, line_size);
            }
            FreeMem(vsprite, (LONG)sizeof(*vsprite));
        }
    }
    return(NULL);
}

/*-----
** create a Bob from the information given in nBob.
** use freeBob() to free this gel.
**
** A VSprite is created for this bob.
** This routine properly allocates all double buffered information
** if it is required.
*/
struct Bob *makeBob(NEWBOB *nBob)
{
    struct Bob      *bob;
    struct VSprite  *vsprite;
    NEWVSPRITE      nVSprite ;
    LONG            rassize;

    rassize = (LONG)sizeof(WORD) *
        nBob->nb_WordWidth * nBob->nb_LineHeight * nBob->nb_RasDepth;

    if (NULL != (bob =
        (struct Bob *)AllocMem((LONG)sizeof(struct Bob), MEMF_CLEAR)))
    {

```

```

if (NULL != (bob->SaveBuffer = (WORD *)AllocMem(rassize, MEMF_CHIP)))
{
    nVSprite.nvs_WordWidth = nBob->nb_WordWidth;
    nVSprite.nvs_LineHeight = nBob->nb_LineHeight;
    nVSprite.nvs_ImageDepth = nBob->nb_ImageDepth;
    nVSprite.nvs_Image = nBob->nb_Image;
    nVSprite.nvs_X = nBob->nb_X;
    nVSprite.nvs_Y = nBob->nb_Y;
    nVSprite.nvs_ColorSet = NULL;
    nVSprite.nvs_Flags = nBob->nb_BFlags;

    if ((vsprite = makeVSprite(&nVSprite)) != NULL)
    {
        vsprite->PlanePick = nBob->nb_PlanePick;
        vsprite->PlaneOnOff = nBob->nb_PlaneOnOff;

        vsprite->VSBob = bob;
        bob->BobVSprite = vsprite;
        bob->ImageShadow = vsprite->CollMask;
        bob->Flags = 0;
        bob->Before = NULL;
        bob->After = NULL;
        bob->BobComp = NULL;

        if (nBob->nb_DBuf)
        {
            if (NULL != (bob->DBuffer = (struct DBufPacket *)AllocMem(
                (LONG)sizeof(struct DBufPacket), MEMF_CLEAR)))
            {
                if (NULL != (bob->DBuffer->BufBuffer =
                    (WORD *)AllocMem(rassize, MEMF_CHIP)))
                {
                    return(bob);
                }
                FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
            }
        }
        else
        {
            bob->DBuffer = NULL;
            return(bob);
        }

        freeVSprite(vsprite);
    }
    FreeMem(bob->SaveBuffer, rassize);
}
FreeMem(bob, (LONG)sizeof(*bob));
}
return(NULL);
}

/*-----
** create a Animation Component from the information given in nAnimComp
** and nBob.
** use freeComp() to free this gel.
**
** makeComp calls makeBob(), and links the bob into a AnimComp.
*/
struct AnimComp *makeComp(NEWBOB *nBob, NEWANIMCOMP *nAnimComp)
{
    struct Bob *compBob;
    struct AnimComp *aComp;

    if ((aComp = AllocMem((LONG)sizeof(struct AnimComp), MEMF_CLEAR)) != NULL)
    {
        if ((compBob = makeBob(nBob)) != NULL)
        {
            compBob->After = NULL; /* Caller can deal with these later. */
            compBob->Before = NULL;
            compBob->BobComp = aComp; /* Link 'em up. */

```

```

        aComp->AnimBob      = compBob;
        aComp->TimeSet      = nAnimComp->nac_Time; /* Num ticks active. */
        aComp->YTrans       = nAnimComp->nac_Yt; /* Offset rel to HeadOb */
        aComp->XTrans       = nAnimComp->nac_Xt;
        aComp->AnimCRoutine = nAnimComp->nac_Routine;
        aComp->Flags        = nAnimComp->nac_CFlags;
        aComp->Timer        = 0;
        aComp->NextSeq      = NULL;
        aComp->PrevSeq      = NULL;
        aComp->NextComp     = NULL;
        aComp->PrevComp     = NULL;
        aComp->HeadOb       = NULL;

        return(aComp);
    }
    FreeMem(aComp, (LONG)sizeof(struct AnimComp));
}
return(NULL);
}

/*-----
** create an Animation Sequence from the information given in nAnimSeq
** and nBob.
** use freeSeq() to free this gel.
**
** this routine creates a linked list of animation components which
** make up the animation sequence.
**
** It links them all up, making a circular list of the PrevSeq
** and NextSeq pointers. That is to say, the first component of the
** sequences' PrevSeq points to the last component; the last component of
** the sequences' NextSeq points back to the first component.
**
** If dbuf is on, the underlying Bobs'll be set up for double buffering.
** If singleImage is non-zero, the pImages pointer is assumed to point to
** an array of only one image, instead of an array of 'count' images, and
** all Bobs will use the same image.
*/
struct AnimComp *makeSeq(NEWBOB *nBob, NEWANIMSEQ *nAnimSeq)
{
    int seq;
    struct AnimComp *firstCompInSeq = NULL;
    struct AnimComp *seqComp = NULL;
    struct AnimComp *lastCompMade = NULL;
    LONG image_size;
    NEWANIMCOMP nAnimComp;

    /* get the initial image. this is the only image that is used
    ** if nAnimSeq->nas_SingleImage is non-zero.
    */
    nBob->nb_Image = nAnimSeq->nas_Images;
    image_size = nBob->nb_LineHeight * nBob->nb_ImageDepth * nBob->nb_WordWidth;

    /* for each comp in the sequence */
    for (seq = 0; seq < nAnimSeq->nas_Count; seq++)
    {
        nAnimComp.nac_Xt      = *(nAnimSeq->nas_Xt + seq);
        nAnimComp.nac_Yt      = *(nAnimSeq->nas_Yt + seq);
        nAnimComp.nac_Time    = *(nAnimSeq->nas_Times + seq);
        nAnimComp.nac_Routine = nAnimSeq->nas_Routines[seq];
        nAnimComp.nac_CFlags  = nAnimSeq->nas_CFlags;

        if ((seqComp = makeComp(nBob, &nAnimComp)) == NULL)
        {
            if (firstCompInSeq != NULL)
                freeSeq(firstCompInSeq, (LONG)nBob->nb_RasDepth);
            return(NULL);
        }

        seqComp->AnimBob->BobVSprite->HitMask = nAnimSeq->nas_HitMask;
        seqComp->AnimBob->BobVSprite->MeMask = nAnimSeq->nas_MeMask;
        seqComp->HeadOb = nAnimSeq->nas_HeadOb;
    }

```

```

/* Make a note of where the first component is. */
if (firstCompInSeq == NULL)
    firstCompInSeq = seqComp;

/* link the component into the list */
if (lastCompMade != NULL)
    lastCompMade->NextSeq = seqComp;

seqComp->NextSeq = NULL;
seqComp->PrevSeq = lastCompMade;
lastCompMade = seqComp;

/* If nAnimSeq->nas_SingleImage is zero,
** the image array has nAnimSeq->nas_Count images.
*/
if (!nAnimSeq->nas_SingleImage)
    nBob->nb_Image += image_size;
}

/* On The last component in the sequence, set Next/Prev to make
** the linked list a loop of components.
*/
lastCompMade->NextSeq = firstCompInSeq;
firstCompInSeq->PrevSeq = lastCompMade;

return(firstCompInSeq);
}

/*-----
** free the data created by makeVSprite()
**
** assumes images deallocated elsewhere.
*/
VOID freeVSprite(struct VSprite *vsprite)
{
    LONG    line_size;
    LONG    plane_size;

    line_size = (LONG)sizeof(WORD) * vsprite->Width;
    plane_size = line_size * vsprite->Height;

    FreeMem(vsprite->BorderLine, line_size);
    FreeMem(vsprite->CollMask, plane_size);

    FreeMem(vsprite, (LONG)sizeof(*vsprite));
}

/*-----
** free the data created by makeBob()
**
** it's important that rasdepth match the depth you
** passed to makeBob() when this gel was made.
** assumes images deallocated elsewhere.
*/
VOID freeBob(struct Bob *bob, LONG rasdepth)
{
    LONG    rassize;

    rassize = (LONG)sizeof(WORD) *
        bob->BobVSprite->Width * bob->BobVSprite->Height * rasdepth;

    if (bob->DBuffer != NULL)
    {
        FreeMem(bob->DBuffer->BufBuffer, rassize);
        FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
    }
    FreeMem(bob->SaveBuffer, rassize);
    freeVSprite(bob->BobVSprite);
    FreeMem(bob, (LONG)sizeof(*bob));
}

/*-----
** free the data created by makeComp()

```

```

**
** it's important that rasdepth match the depth you
** passed to makeComp() when this gel was made.
** assumes images deallocated elsewhere.
*/
VOID freeComp(struct AnimComp *myComp, LONG rasdepth)
{
    freeBob(myComp->AnimBob, rasdepth);
    FreeMem(myComp, (LONG)sizeof(struct AnimComp));
}

/*-----
** free the data created by makeSeq()
**
** Complimentary to makeSeq(), this routine goes through the NextSeq
** pointers and frees the Components
**
** This routine only goes forward through the list, and so
** it must be passed the first component in the sequence, or the sequence
** must be circular (which is guaranteed if you use makeSeq()).
**
** it's important that rasdepth match the depth you
** passed to makeSeq() when this gel was made.
** assumes images deallocated elsewhere.
*/
VOID freeSeq(struct AnimComp *headComp, LONG rasdepth)
{
    struct AnimComp *curComp;
    struct AnimComp *nextComp;

    /* this is freeing a loop of AnimComps, hooked together by the
    ** NextSeq and PrevSeq pointers.
    */

    /* break the NextSeq loop, so we get a NULL at the end of the list. */
    headComp->PrevSeq->NextSeq = NULL;

    curComp = headComp;          /* get the start of the list */
    while (curComp != NULL)
    {
        nextComp = curComp->NextSeq;
        freeComp(curComp, rasdepth);
        curComp = nextComp;
    }
}

/*-----
** free an animation object (list of sequences).
**
** freeOb() goes through the NextComp pointers, starting at the AnimObs'
** HeadComp, and frees every sequence.
** it only goes forward. It then frees the Object itself.
** assumes images deallocated elsewhere.
*/
VOID freeOb(struct AnimOb *headOb, LONG rasdepth)
{
    struct AnimComp *curSeq;
    struct AnimComp *nextSeq;

    curSeq = headOb->HeadComp;    /* get the start of the list */
    while (curSeq != NULL)
    {
        nextSeq = curSeq->NextComp;
        freeSeq(curSeq, rasdepth);
        curSeq = nextSeq;
    }

    FreeMem(headOb, (LONG)sizeof(struct AnimOb));
}

```



Here is the file *image\_boing.h*, which gives the gel data.

```
/* This include file has the needed constants for a three plane
** boing(tm) ball and a two plane satellite.
*/

SHORT boing3Times[BNG3COUNT] = { 1, 1, 1, 1, 1, 1 };
SHORT boing3YTranses[BNG3COUNT] = { 0, 0, 0, 0, 0, 0 };
SHORT boing3XTranses[BNG3COUNT] = { 0, 0, 0, 0, 0, 0 };

/* APTR boing3CRoutines[BNG3COUNT] */
WORD (*boing3CRoutines[BNG3COUNT])(struct AnimComp *) =
    { NULL, NULL, NULL, NULL, NULL, NULL };

/* NOTE the chip keyword causes this data to load into chip memory
** under lattice. Be sure that this initialized data resides in
** chip memory when the program executes.
*/
UWORD chip boing3Image[BNG3COUNT][BNG3WWIDTH * BNG3HEIGHT * BNG3DEPTH] =
{
    /*----- bitmap Boing-A w = 32, h = 25 ----- */
    {
        /*----- plane # 0: -----*/
        0x0023, 0x0000, 0x004E, 0x3000, 0x00E3, 0x3A00, 0x03C3, 0xC900,
        0x0787, 0x8780, 0x108F, 0x8700, 0x31F7, 0x8790, 0x61F0, 0x4790,
        0x63E0, 0xF890, 0x43E0, 0xF848, 0x3BC0, 0xF870, 0x3801, 0xF870,
        0x383D, 0xF070, 0x387E, 0x1070, 0x387C, 0x0EE0, 0xD87C, 0x1F10,
        0x467C, 0x1E10, 0x479C, 0x1E30, 0x6787, 0x3E20, 0x0787, 0xCC60,
        0x0F0F, 0x8700, 0x048F, 0x0E00, 0x0277, 0x1C00, 0x0161, 0xD800,
        0x0027, 0x2000,
    },
    /*----- bitmap Boing-B w = 32, h = 25 ----- */
    {
        /*----- plane # 0: -----*/
        0x0031, 0x8000, 0x0107, 0x1800, 0x00F0, 0x1900, 0x09E1, 0xEC80,
        0x13C1, 0xE340, 0x1803, 0xE380, 0x387B, 0xC390, 0x30F8, 0x01D0,
        0x70F8, 0x3DC0, 0xE1F0, 0x3E08, 0x9DF0, 0x7C30, 0x9E30, 0x7C30,
        0x9E1C, 0x7C30, 0x1C1F, 0x9C30, 0x1C1F, 0x0630, 0x7C1F, 0x0780,
        0x623F, 0x0798, 0x63DE, 0x0F10, 0x23C1, 0x0F20, 0x33C3, 0xEE20,
        0x0BC3, 0xC380, 0x0647, 0xC700, 0x023F, 0x8E00, 0x0130, 0xF800,
        0x0033, 0x8000,
    },
    /*----- bitmap Boing-C w = 32, h = 25 ----- */
    {
        /*----- plane # 0: -----*/
        0x0019, 0xC000, 0x0103, 0x8800, 0x0278, 0x8D00, 0x0CF0, 0xFE80,
        0x11F0, 0xF140, 0x0E60, 0xF1E0, 0x1C39, 0xF0C0, 0x1C3E, 0x30C0,
        0x387E, 0x0CE0, 0xF87C, 0x1F28, 0x8C7C, 0x1F18, 0x8F3C, 0x1F18,
        0x8F06, 0x1E18, 0x8F07, 0xDE18, 0x8F07, 0xC018, 0x6E0F, 0xC1C0,
        0x300F, 0x83C8, 0x31EF, 0x8390, 0x31F0, 0x8780, 0x11E0, 0xF720,
        0x11E1, 0xF1C0, 0x0B61, 0xE300, 0x071B, 0xC600, 0x0138, 0x6C00,
        0x0031, 0x8000,
    },
    /*----- bitmap : w = 32, h = 25 ----- */
    {
        /*----- plane Boing-D 0: -----*/
        0x001C, 0xE000, 0x01B1, 0xCC00, 0x031C, 0xC500, 0x0C3C, 0x3680,
        0x1878, 0x7840, 0x2F70, 0x78E0, 0x0E08, 0x7860, 0x1E0F, 0xB860,
        0x1C1F, 0x0460, 0xBC1F, 0x07B0, 0xC43F, 0x0788, 0xC7FE, 0x0788,
        0xC7C0, 0x0F88, 0xC781, 0xEF88, 0xC783, 0xF118, 0x2783, 0xE0E8,
        0x3983, 0xE1E0, 0x3863, 0xE1C0, 0x1878, 0xC1C0, 0x3878, 0x3380,
        0x10F0, 0x78C0, 0x0B70, 0xF180, 0x0588, 0xE200, 0x009E, 0x2400,
        0x0018, 0xC000,
    },
    /*----- bitmap : w = 32, h = 25 ----- */
    {
        /*----- plane Boing-E 0: -----*/
        0x000E, 0x6000, 0x00F8, 0xE400, 0x030F, 0xE600, 0x061E, 0x1300,
        0x0C3E, 0x1C80, 0x27FC, 0x1C60, 0x0784, 0x3C60, 0x4F07, 0xFE20,
        0x8F07, 0xC230, 0x1E0F, 0xC1F0, 0x620F, 0x83C8, 0x61CF, 0x83C8,
        0x61E1, 0x83C8, 0x63E0, 0x63C8, 0x63E0, 0xF9C8, 0x03E0, 0xF878,
    },
}
```

```

        0x1DC0, 0xF860, 0x1C21, 0xF0E0, 0x5C3E, 0xF0C0, 0x0C3C, 0x11C0,
        0x143C, 0x3C40, 0x09B8, 0x3880, 0x05C0, 0x7000, 0x00CF, 0x0400,
        0x000C, 0x6000,
    },
    /*----- bitmap : w = 32, h = 25 ----- */
    {
        /*----- plane Boing-F 0: -----*/
        0x0026, 0x2000, 0x00FC, 0x7400, 0x0187, 0x7200, 0x030F, 0x0100,
        0x0E0F, 0x0E80, 0x319F, 0x0E00, 0x23C6, 0x0F30, 0x63C1, 0xCF30,
        0x4781, 0xF310, 0x0783, 0xE0D0, 0x7383, 0xE0E0, 0x70C3, 0xE0E0,
        0x70F9, 0xE1E0, 0x70F8, 0x21E0, 0x70F8, 0x3FE0, 0x91F0, 0x3E38,
        0x4FF0, 0x7C30, 0x4E10, 0x7C60, 0x4E0F, 0x7860, 0x2E1F, 0x08C0,
        0x0E1E, 0x0E00, 0x049E, 0x1C80, 0x00E4, 0x3800, 0x00C7, 0x9000,
        0x000E, 0x6000,
    }
};

/*
** Orbit goes from far top -> mid l -> near bot -> mid right
*/

/*-----*/
SHORT satTimes[SATCOUNT] =
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

/* APTR satACRoutines[SATCOUNT] */
WORD (*satACRoutines[SATCOUNT])(struct AnimComp *) =
    {
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
    };

SHORT satAYTranes[SATCOUNT] =
    {
        ( 18<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 17<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 15<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 11<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 0<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-11<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        (-15<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-17<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        (-18<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-17<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        (-15<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-11<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 0<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 11<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 15<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 17<<BNFS)+((BNG3HEIGHT/2)<<ANFS)
    };

SHORT satAXTranes[SATCOUNT] =
    {
        ( 0<<BNFS)+((BNG3WIDTH/2)<<ANFS), ( 22<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        ( 45<<BNFS)+((BNG3WIDTH/2)<<ANFS), ( 66<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        ( 80<<BNFS)+((BNG3WIDTH/2)<<ANFS), ( 66<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        ( 45<<BNFS)+((BNG3WIDTH/2)<<ANFS), ( 22<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        ( 0<<BNFS)+((BNG3WIDTH/2)<<ANFS), (-22<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        (-45<<BNFS)+((BNG3WIDTH/2)<<ANFS), (-66<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        (-80<<BNFS)+((BNG3WIDTH/2)<<ANFS), (-66<<BNFS)+((BNG3WIDTH/2)<<ANFS),
        (-45<<BNFS)+((BNG3WIDTH/2)<<ANFS), (-22<<BNFS)+((BNG3WIDTH/2)<<ANFS)
    };

/* APTR satBCRoutines[SATCOUNT] */
WORD (*satBCRoutines[SATCOUNT])(struct AnimComp *) =
    {
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
    };

/*----- bitmap : w = 5, h = 5 ----- */
SHORT satBYTranes[SATCOUNT] =
    {
        (-57<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-40<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        (-23<<BNFS)+((BNG3HEIGHT/2)<<ANFS), (-6<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 13<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 27<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 41<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 53<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 57<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 40<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
        ( 22<<BNFS)+((BNG3HEIGHT/2)<<ANFS), ( 4<<BNFS)+((BNG3HEIGHT/2)<<ANFS),
    }

```

```

(-13<<BNFS)+(BNG3HEIGHT/2)<<ANFS), (-28<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
(-43<<BNFS)+(BNG3HEIGHT/2)<<ANFS), (-55<<BNFS)+(BNG3HEIGHT/2)<<ANFS)
);

SHORT satBXTranes[SATCOUNT] =
{
(-57<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-53<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-41<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-27<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-13<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 6<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 23<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 40<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 57<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 55<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 43<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 28<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 13<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( -4<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-22<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-40<<BNFS)+(BNG3WIDTH/2)<<ANFS)
);

/* APTX satCCRoutines[SATCOUNT] */
WORD (*satCCRoutines[SATCOUNT])(struct AnimComp *) =
{
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
};

/*----- bitmap : w = 5, h = 5 ----- */
SHORT satCYTranes[SATCOUNT] =
{
(-13<<BNFS)+(BNG3HEIGHT/2)<<ANFS), ( 4<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
( 22<<BNFS)+(BNG3HEIGHT/2)<<ANFS), ( 40<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
( 57<<BNFS)+(BNG3HEIGHT/2)<<ANFS), ( 53<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
( 41<<BNFS)+(BNG3HEIGHT/2)<<ANFS), ( 27<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
( 13<<BNFS)+(BNG3HEIGHT/2)<<ANFS), ( -6<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
(-23<<BNFS)+(BNG3HEIGHT/2)<<ANFS), (-40<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
(-57<<BNFS)+(BNG3HEIGHT/2)<<ANFS), (-55<<BNFS)+(BNG3HEIGHT/2)<<ANFS),
(-43<<BNFS)+(BNG3HEIGHT/2)<<ANFS), (-28<<BNFS)+(BNG3HEIGHT/2)<<ANFS)
);

SHORT satCXTranes[SATCOUNT] =
{
(-13<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-28<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-43<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-55<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-57<<BNFS)+(BNG3WIDTH/2)<<ANFS), (-40<<BNFS)+(BNG3WIDTH/2)<<ANFS),
(-23<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( -6<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 13<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 27<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 41<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 53<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 57<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 40<<BNFS)+(BNG3WIDTH/2)<<ANFS),
( 22<<BNFS)+(BNG3WIDTH/2)<<ANFS), ( 4<<BNFS)+(BNG3WIDTH/2)<<ANFS)
);

/*-----*/
/*----- bitmap : w = 5, h = 5 ----- */
/* NOTE the chip keyword causes this data to load into chip memory
** under lattice. Be sure that this initialized data resides in
** chip memory when the program executes.
*/
UWORD chip satImage[1][SATWIDTH * SATHEIGHT * SATDEPTH] =
{
{
0x5000, 0xE800, 0x4800, 0x9800, 0x7000,
0x6000, 0xF000, 0xF000, 0x6000, 0x0000
},
};

```

Here is the actual animation example.

```

/* anim_ex.c 19oct89
** original code by Dave Lucas.
** rework by CATS
**
** lattice c 5.04
** lc -bl -cfist -v -y anim_ex.c
** blink FROM LIB:c.o anim_ex.o /animtools/animtools.o LIB LIB:lc.lib TO anim
*/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gels.h>
#include <exec/memory.h>
#include <libraries/dos.h>

#include "/animtools/animtools.h"
#include "/animtools/animtools_proto.h"

#include <stdlib.h>
#include <stdio.h>
#include <proto/all.h>
int CXBRK(void) { return(0); } /* disable lattice CTRL-C handling */

/*-----*/
/* prototypes for the functions in this file. */
/*-----*/
struct AnimOb *setupBoing(SHORT dbufing);

WORD bounceORoutine(struct AnimOb *anOb);
WORD goInFrontOfHead(struct AnimComp *aComp);
WORD goBehindHead(struct AnimComp *aComp);

VOID runAnimation(struct Window *win, SHORT dbufing,
                  struct AnimOb **animKey, struct BitMap **myBitMaps);

LONG setupPlanes(struct BitMap *bitMap, LONG depth, LONG width, LONG height);
struct BitMap **setupBitMaps(LONG depth, LONG width, LONG height);
VOID freePlanes(struct BitMap *bitMap, LONG depth, LONG width, LONG height);
VOID freeBitMaps(struct BitMap **myBitMaps,
                  LONG depth, LONG width, LONG height);
struct GelsInfo *setDisplay(struct Window **win, SHORT dbufing,
                           struct BitMap **myBitMaps);

VOID DrawGels(struct Window *win, struct AnimOb **animKey,
              SHORT dbufing, WORD *toggleFrame, struct BitMap **myBitMaps);

/*-----*/
/* animation and screen constants. */
/*-----*/
#define ANFS      ANFRACSIZE
#define BNFS      (ANFRACSIZE-2)
#define CNFS      (ANFRACSIZE-2)

/* These are used for MeMask (1 << ID_xxx) */
#define ID_BORDER 0
#define ID_BNG    4

#define SBMWIDTH  320 /* My screen size constants. */
#define SBMHEIGHT 200
#define SBMDEPTH  4
#define SCRNMODE  NULL /* (HIRES | LACE) for NewScreen, ends up in view.*/

#define RBMWIDTH  320 /* My raster size constants. (These CAN differ) */
#define RBMHEIGHT 200
#define RBMDEPTH  4

/*-----*/
/* global data for libraries and display. */
/*-----*/
struct NewScreen ns =
{
    0, 0, SBMWIDTH, SBMHEIGHT, SBMDEPTH, 0, 0, SCRNMODE,
    CUSTOMSCREEN | SCREENQUIET, NULL, NULL, NULL, NULL

```

```

};

/* this is a little window that will let us get a CLOSEWINDOW message */
struct NewWindow nw =
{
    0, 0, 25, 12, 0, 0, CLOSEWINDOW,
    WINDOWCLOSE | BORDERLESS | RMBTRAP, NULL, NULL, NULL, NULL,
    NULL, 0, 0, SBMWIDTH, SBMHEIGHT-1, CUSTOMSCREEN
};

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

int return_code;

/*-----*/
/* constants for the boing ball and satellites. */
/*-----*/
#define BNG3RINGY 0
#define BNG3RINGX 0
#define BNG3COUNT 6
#define BNG3HEIGHT 25
#define BNG3WIDTH 32
#define BNG3DEPTH 1
#define BNG3WWIDTH ((BNG3WIDTH + 15) / 16)

#define SATCOUNT 16
#define SATHEIGHT 5
#define SATWIDTH 5
#define SATDEPTH 2
#define SATWWIDTH ((SATWIDTH + 15) / 16)

#include "image_boing.h"

NEWBOB newBoingBob =
{
    NULL, BNG3WWIDTH, BNG3HEIGHT, BNG3DEPTH, 0x2, 0xC,
    SAVEBACK | OVERLAY, 0, RBMDEPTH, 0,0,
};

NEWANIMSEQ newBoingSeq =
{
    NULL, (WORD *)boing3Image, boing3XTrances, boing3YTrances,
    boing3Times, boing3CRoutines, 0, BNG3COUNT,
    (1L<<ID_BORDER), (1L<<ID_BNG), 0
};

NEWBOB newSatABob =
{
    NULL, SATWWIDTH, SATHEIGHT, SATDEPTH, 0xC, 0x0,
    SAVEBACK | OVERLAY, 0, RBMDEPTH, 0,0,
};

NEWANIMSEQ newSatASeq =
{
    NULL, (WORD *)satImage, satAXTrances, satAYTrances,
    satTimes, satACRoutines, 0, SATCOUNT,
    (1L<<ID_BORDER), (1L<<ID_BNG), 1
};

NEWBOB newSatBBob =
{
    NULL, SATWWIDTH, SATHEIGHT, SATDEPTH, 0xC, 0x3,
    SAVEBACK | OVERLAY, 0, RBMDEPTH, 0,0,
};

NEWANIMSEQ newSatBSeq =
{
    NULL, (WORD *)satImage, satBXTrances, satBYTrances,
    satTimes, satBCRoutines, 0, SATCOUNT,
    (1L<<ID_BORDER), (1L<<ID_BNG), 1
};

NEWBOB newSatCBob =
{
    NULL, SATWWIDTH, SATHEIGHT, SATDEPTH, 0x3, 0x0,
    SAVEBACK | OVERLAY, 0, RBMDEPTH, 0,0,
};

```

```

    );
    NEWANIMSEQ newSatCSeq =
    {
        NULL, (WORD *)satImage, satCXTranes, satCYTranes,
        satTimes, satCCRoutines, 0, SATCOUNT,
        (1L<<ID_BORDER), (1L<<ID_BNG), 1
    };

/*-----*/
/*-----*/
/*          PROCEDURES          */
/*-----*/
/*-----*/

/*-----*/
** setupBoing() - allocate and initialize an object that will
** display as a boing ball with orbiting satellites.
**
** this is an animation object with four animation sequences.
** (boing and three satellites.)
** (note that the satellites all share the same single image data.)
**
** return NULL on failure.
*/
struct AnimOb *setupBoing(SHORT dbufing)
{
    struct AnimOb    *bngOb;
    struct AnimComp  *bngComp;
    struct AnimComp  *satAComp;
    struct AnimComp  *satBComp;
    struct AnimComp  *satCComp;

    if (NULL != (bngOb = AllocMem((LONG)sizeof(struct AnimOb), MEMF_CLEAR)))
    {
        bngOb->NextOb      = NULL;
        bngOb->PrevOb      = NULL;
        bngOb->Clock       = 0;
        bngOb->AnY         = 50;
        bngOb->AnX         = 50;
        bngOb->AnOldY      = bngOb->AnY;
        bngOb->AnOldX      = bngOb->AnX;
        bngOb->YVel        = 3 << ANFRACSIZE;
        bngOb->XVel        = 3 << ANFRACSIZE;
        bngOb->YAccel      = 0;
        bngOb->XAccel      = 0;
        bngOb->RingYTrans  = BNG3RINGY << ANFRACSIZE;
        bngOb->RingXTrans  = BNG3RINGX << ANFRACSIZE;
        bngOb->AnimORoutine = bounceORoutine;
        bngOb->AUserExt     = 0;

        newBoingBob.nb_DBuf = dbufing;
        newBoingSeq.nas_HeadOb = bngOb;

        /* these routines are called when a specific comp in a sequence is
        ** drawn.  satellite A will go in front when the 5th (counting from
        ** zero) comp is drawn.
        */
        satACRoutines[4]    = goInFrontOfHead;
        satACRoutines[12]   = goBehindHead;
        satBCRoutines[8]    = goInFrontOfHead;
        satBCRoutines[0]    = goBehindHead;
        satCCRoutines[4]    = goInFrontOfHead;
        satCCRoutines[12]   = goBehindHead;

        /* set up the double buf flag and pointer to the Head Object.
        ** these are not known until run time.
        */
        newSatABob.nb_DBuf  = dbufing;
        newSatBBob.nb_DBuf  = dbufing;
        newSatCBob.nb_DBuf  = dbufing;
        newSatASeq.nas_HeadOb = bngOb;
        newSatBSeq.nas_HeadOb = bngOb;
    }
}

```

```

newSatCSeq.nas_HeadOb = bngOb;

if (NULL != (bngComp = makeSeq(&newBoingBob, &newBoingSeq)))
{
    /* set RINGTRIGGER for the first comp in the sequence.
    ** this will cause RingYTrans and RingXTrans to be added to
    ** the object position when this comp is drawn.
    ** (Here they are zero, so this could be taken out.)
    */
    bngComp->Flags |= RINGTRIGGER;
    bngOb->HeadComp = bngComp;

    if (NULL != (satAComp = makeSeq(&newSatABob, &newSatASeq)))
    {
        /* set up the drawing precedence for the bobs */
        bngComp->AnimBob->Before = satAComp->AnimBob;
        satAComp->AnimBob->After = bngComp->AnimBob;

        if (NULL != (satBComp = makeSeq(&newSatBBob, &newSatBSeq)))
        {
            satAComp->AnimBob->Before = satBComp->AnimBob;
            satBComp->AnimBob->After = satAComp->AnimBob;

            if (NULL != (satCComp = makeSeq(&newSatCBob, &newSatCSeq)))
            {
                satBComp->AnimBob->Before = satCComp->AnimBob;
                satCComp->AnimBob->After = satBComp->AnimBob;

                /* connect all of the head comps (one for each sequence)
                ** together to form a single animation object.
                */
                bngComp->NextComp = satAComp;
                bngComp->PrevComp = NULL;

                satAComp->NextComp = satBComp;
                satAComp->PrevComp = bngComp;

                satBComp->NextComp = satCComp;
                satBComp->PrevComp = satAComp;

                satCComp->NextComp = NULL;
                satCComp->PrevComp = satBComp;

                return(bngOb);
            }
            /* if something failed, close everything */
            freeSeq(satBComp, RBMDEPTH);
        }
        freeSeq(satAComp, RBMDEPTH);
    }
    freeSeq(bngComp, RBMDEPTH);
}
FreeMem(bngOb, (LONG)sizeof(struct AnimOb));
}

return_code = RETURN_WARN;
return(NULL);
}

/*-----
** This ORoutine makes the Object Bounce off Borders.
*/
WORD bounceORoutine(struct AnimOb *anOb)
{
    SHORT Y;
    SHORT X;

    Y = anOb->AnY >> ANFRACSIZE;
    X = anOb->AnX >> ANFRACSIZE;

    if ((Y<0 && anOb->YVel < 0) ||
        ((Y+anOb->HeadComp->AnimBob->BobVSprite->Height > RBMHEIGHT) &&
         (anOb->YVel > 0)))

```

```

    {
        anOb->YVel = -(anOb->YVel);
    }

if (((X < 0) && (anOb->XVel < 0)) ||
    ((X + (anOb->HeadComp->AnimBob->BobVSprite->Width << 4) > RBMWIDTH) &&
    (anOb->XVel > 0)))
    {
        anOb->XVel = -(anOb->XVel);
    }
return(0) ;
}

/*-----
** This CRoutine rearranges Bob Before and After pointers in a
** way that makes the Component passed look like it is in front
** of its' head component.
**
** Used for Boing satellites.
**
** So that they go in front of AND behind the boing ball.
*/
WORD goInFrontOfHead(struct AnimComp *aComp)
{
    /* remove bob and close up hole */
    if (aComp->AnimBob->Before != NULL)
        aComp->AnimBob->Before->After = aComp->AnimBob->After;
    if (aComp->AnimBob->After != NULL)
        aComp->AnimBob->After->Before = aComp->AnimBob->Before;

    /* reinsert bob in front of HeadOb (it will be drawn after Head) */
    aComp->AnimBob->Before = aComp->HeadOb->HeadComp->AnimBob->Before;
    aComp->AnimBob->After = aComp->HeadOb->HeadComp->AnimBob;
    if (aComp->AnimBob->Before != NULL)
        aComp->AnimBob->Before->After = aComp->AnimBob;
    aComp->HeadOb->HeadComp->AnimBob->Before = aComp->AnimBob;

    return(0) ;
}

/*-----
** This CRoutine rearranges Bob Before and After pointers in a
** way that makes the Component passed look like it is behind
** its' head component.
**
*/
WORD goBehindHead(struct AnimComp *aComp)
{
    /* remove bob and close up hole */
    if (aComp->AnimBob->Before != NULL)
        aComp->AnimBob->Before->After = aComp->AnimBob->After;
    if (aComp->AnimBob->After != NULL)
        aComp->AnimBob->After->Before = aComp->AnimBob->Before;

    /* reinsert bob in behind of HeadOb (it will be drawn before Head) */
    aComp->AnimBob->After = aComp->HeadOb->HeadComp->AnimBob->After;
    aComp->AnimBob->Before = aComp->HeadOb->HeadComp->AnimBob;
    if (aComp->AnimBob->After != NULL)
        aComp->AnimBob->After->Before = aComp->AnimBob;
    aComp->HeadOb->HeadComp->AnimBob->After = aComp->AnimBob;

    return(0) ;
}

/*-----
**
**
*/
VOID runAnimation(struct Window *win,
                  SHORT dbufing,
                  struct AnimOb **animKey,
                  struct BitMap **myBitMaps)
{
    struct IntuiMessage *intuiMsg;

```



```

WORD toggleFrame;

toggleFrame = 0;

/* everything opened, and allocated, and initialized.
** hang out, move the gels, tell the system to redraw them,
** and let the collision and anim routines bounce them about.
**
** check after each draw for CLOSEWINDOW events.
** go away when we get one.
*/
for (;;)
{
    /* All the work done here */
    DrawGels(win, animKey, dbufing, &toggleFrame, myBitMaps);

    /* you MUST be sure to call:
    ** WaitTOF()
    ** after you call DrawGList(). Here the DrawGels() routine
    ** does it for you.
    */

    while (intuiMsg = (struct IntuiMessage *)GetMsg(win->UserPort))
    {
        if (intuiMsg->Class == CLOSEWINDOW)
        {
            ReplyMsg((struct Message *)intuiMsg);
            return;
        }
        ReplyMsg((struct Message *)intuiMsg);
    }
}

/*-----
** allocate the bit planes for a screen bit map.
*/
LONG setupPlanes(struct BitMap *bitMap,
                 LONG depth, LONG width, LONG height)
{
    SHORT plane_num ;

    for (plane_num = 0; plane_num < depth; plane_num++)
    {
        if (NULL != (bitMap->Planes[plane_num] =
                     (PLANEPTR)AllocRaster(width, height)))
            BltClear(bitMap->Planes[plane_num], (width / 8) * height, 1);
        else
        {
            freePlanes(bitMap, depth, width, height);
            return_code = RETURN_WARN;
            return(NULL);
        }
    }
    return(TRUE);
}

/*-----
** allocate the bit maps for a double buffered screen.
*/
struct BitMap **setupBitMaps(LONG depth, LONG width, LONG height)
{
    /* this must be static -- it cannot go away when the routine exits. */
    static struct BitMap *myBitMaps[2];

    if (NULL != (myBitMaps[0] =
                 (struct BitMap *)AllocMem((LONG)sizeof(struct BitMap), MEMF_CLEAR)))
    {
        if (NULL != (myBitMaps[1] =
                     (struct BitMap *)AllocMem((LONG)sizeof(struct BitMap), MEMF_CLEAR)))
        {
            InitBitMap(myBitMaps[0], depth, width, height);

```

```

        InitBitMap(myBitMaps[1], depth, width, height);

        if (NULL != setupPlanes(myBitMaps[0], depth, width, height))
        {
            if (NULL != setupPlanes(myBitMaps[1], depth, width, height))
                return(myBitMaps);

            freePlanes(myBitMaps[0], depth, width, height);
        }
        FreeMem(myBitMaps[1], (LONG)sizeof(struct BitMap));
    }
    FreeMem(myBitMaps[0], (LONG)sizeof(struct BitMap));
}

return_code = RETURN_WARN;
return(NULL);
}

/*-----
** free up the memory allocated by setupPlanes().
*/
VOID    freePlanes(struct BitMap *bitMap,
                    LONG depth, LONG width, LONG height)
{
    SHORT plane_num ;

    for (plane_num = 0; plane_num < depth; plane_num++)
    {
        if (NULL != bitMap->Planes[plane_num])
            FreeRaster(bitMap->Planes[plane_num], width, height);
    }
}

/*-----
** free up the memory allocated by setupBitMaps().
*/
VOID    freeBitMaps(struct BitMap **myBitMaps,
                    LONG depth, LONG width, LONG height)
{
    freePlanes(myBitMaps[0], depth, width, height);
    freePlanes(myBitMaps[1], depth, width, height);

    FreeMem(myBitMaps[0], (LONG)sizeof(struct BitMap));
    FreeMem(myBitMaps[1], (LONG)sizeof(struct BitMap));
}

/*-----
** setup the screen and window for the animation.
*/
struct GelsInfo *setupDisplay(struct Window **win,
                              SHORT dbufing,
                              struct BitMap **myBitMaps)
{
    struct GelsInfo    *gInfo;
    struct Screen      *screen;
    struct ViewPort    *vport;
    struct RastPort    *rport;

    if (dbufing)
    {
        /* Screen type. We alloc two BitMaps. See DBLBUF comments. */
        ns.Type |= CUSTOMBITMAP;
        ns.CustomBitMap = myBitMaps[0];
    }

    if ((screen = (struct Screen *)OpenScreen(&ns)) != NULL)
    {
        vport = &screen->ViewPort;
        rport = &screen->RastPort;

        SetRGB4(vport, 0x0, 0x0, 0x0, 0x0); /* Black          */
        SetRGB4(vport, 0x1, 0x0, 0x6, 0x0); /* dk green         */
        SetRGB4(vport, 0x2, 0x0, 0x9, 0x0); /* med green        */
    }
}

```

```

SetRGB4(vport, 0x3, 0x0, 0xC, 0x0); /* lt green      */
SetRGB4(vport, 0x4, 0x1, 0x1, 0x7); /* dk blue      */
SetRGB4(vport, 0x5, 0x7, 0x0, 0x8); /* dk violet    */
SetRGB4(vport, 0x6, 0x6, 0x6, 0x6); /* dk grey      */
SetRGB4(vport, 0x7, 0x7, 0x1, 0x0); /* dk red       */
SetRGB4(vport, 0x8, 0x3, 0x3, 0xB); /* med blue     */
SetRGB4(vport, 0x9, 0xB, 0x0, 0xC); /* med violet   */
SetRGB4(vport, 0xA, 0x9, 0x9, 0x9); /* med grey     */
SetRGB4(vport, 0xB, 0xB, 0x0, 0x0); /* med red      */
SetRGB4(vport, 0xC, 0x5, 0x5, 0xF); /* lt blue      */
SetRGB4(vport, 0xD, 0xE, 0x0, 0xF); /* lt violet    */
SetRGB4(vport, 0xE, 0xF, 0xF, 0xF); /* lt grey (white) */
SetRGB4(vport, 0xF, 0xF, 0x0, 0x0); /* lt red       */

/* put some stuff in the background, so we can
** see that it does not get destroyed.
*/
SetAPen(rport, 0xA);
SetDrMd(rport, JAM1);
Move(rport, 70, 105);
Text(rport, "Animation Example...", 20);

nw.Screen = screen;
if ((*win = (struct Window *)OpenWindow(&nw)) != NULL)
{
    if (dbufing)
    {
        (*win)->WScreen->RastPort.Flags = DBUFFER;

        /* copy the rast port data into the alternate rast port */
        (*win)->WScreen->RastPort.BitMap = myBitMaps[1];
        BltBitMapRastPort(myBitMaps[0], 0,0, &(*win)->WScreen->RastPort,
            0,0, RBMWIDTH, RBMHEIGHT, 0xC0);
        (*win)->WScreen->RastPort.BitMap = myBitMaps[0];
    }

    /* set up the gels system.
    ** 0xFC says: when you allocate sprites for me, don't ever use
    ** sprites zero or one. This guarantees that sprite zero, the
    ** intuition pointer, stays intact. Remember sprite one shares
    ** colors with sprite zero.
    */
    if (NULL != (gInfo = setupGelSys(&(*win)->WScreen->RastPort, 0xFC)))
        return(gInfo);

    CloseWindow(*win);
}
CloseScreen(screen);
}
return_code = RETURN_WARN;
return(NULL);
}

/*-----
** draw all of those animation objects.
*/
VOID DrawGels(struct Window *win,
    struct AnimOb **animKey,
    SHORT dbufing,
    WORD *toggleFrame,
    struct BitMap **myBitMaps)
{
    Animate(animKey, &win->WScreen->RastPort);

    SortGList(&win->WScreen->RastPort); /* Put the list in order. */
    DoCollision(&win->WScreen->RastPort); /* Collision routines may called now */
    SortGList(&win->WScreen->RastPort); /* Put the list in order. */

    if (dbufing)
        win->WScreen->ViewPort.RasInfo->BitMap = myBitMaps[*toggleFrame];

    /* Draw 'em. */

```

```

DrawGLList(&win->WScreen->RastPort, &win->WScreen->ViewPort);

/* if single buffered and using TRUE VSprites, you have to do a
** MrgCop() & LoadView(). Under Intuition use RethinkDisplay().
** if this is not done, then the vsprites will not be updated.
**
** Here, we are not using any TRUE VSprites, so we only do a
** WaitTOF(). Note that RethinkDisplay() does a WaitTOF()
** for you.
*/

if (dbufing)
{
    MakeScreen(win->WScreen); /* Tell intuition to do it's stuff. */
    RethinkDisplay(); /* Intuition compatible MrgCop & LoadView */
                        /* also does a WaitTOF() */

    *toggleFrame ^= 1;
    /* Flip to the next BitMap. */
    win->WScreen->RastPort.BitMap = myBitMaps[*toggleFrame];
}
else
    WaitTOF();
}

/*-----
** main routine. setup and run the animation.
** clean up all resources when done or on any error.
*/
VOID main(int argc, char **argv)
{
    struct BitMap    **myBitMaps;
    struct AnimOb    *boingOb;
    struct Window    *win;
    struct Screen    *screen;
    struct GelsInfo   *gInfo;
    struct AnimOb    *animKey;
    SHORT dbufing;

    return_code = RETURN_OK; /* a global variable, yech! */

    printf("Program will run double buffered if there are\n");
    printf("any command line arguments.\n");

    if (argc > 1)
        dbufing = 1; /* run double buffered when arguments */
    else
        dbufing = 0; /* not double buff */

    if (NULL == (IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 33L)))
        return_code = RETURN_FAIL;
    else
    {
        if (NULL == (GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library", 33L)))
            return_code = RETURN_FAIL;
        else
        {
            if ((!dbufing) ||
                (NULL != (myBitMaps=setupBitMaps(RBMDEPTH,RBMWIDTH,RBMHEIGHT))))
            {
                if (NULL != (gInfo = setupDisplay(&win,dbufing,myBitMaps)))
                {
                    InitAnimate(&animKey);
                    /* Simple sequenced animation. (Boing ball)
                    ** smaller components animated using XTrans and YTrans.
                    ** (tiny orbiting satellites)
                    */
                    if (NULL != (boingOb = setupBoing(dbufing)))
                    {
                        AddAnimOb(boingOb, &animKey, &win->WScreen->RastPort);

```

```

        runAnimation(win, dbufing, &animKey, myBitMaps);
        freeOb(boingOb, RBMDEPTH);
    }

    cleanupGelSys(gInfo, &win->WScreen->RastPort);
    screen = win->WScreen;
    CloseWindow(win);
    CloseScreen(screen);
}

if (dbufing)
    freeBitMaps(myBitMaps, RBMDEPTH, RBMWIDTH, RBMHEIGHT);
}
CloseLibrary((struct Library *)GfxBase);
}
CloseLibrary((struct Library *)IntuitionBase);
}
exit(return_code);
}

```

# Chapter 26

## Layers Library

This chapter describes the Layers Library, which provides routines that are used to manage overlapping rectangular drawing areas that share a common display. It also describes routines in the Graphics Library that manipulate **Regions**, which are used to mask off areas where drawing can take place. An example is provided that shows how to use the majority of the routines described.

### Introduction

The Layers Library contains routines that:

- Allow an application to share a displays' **BitMap** among various tasks by creating "layers" in the **BitMap**.
- Maintain each layer as a separate entity, which may optionally have its own **BitMap**.
- Manage the remapping of coordinates, so the programmer needn't track the offsets into layers.
- Move, size or depth-arrange the layers, which may obscure portions of some layers, or bring previously obscured portions into view.

- Automatically update newly visible portions (optionally).

When multiple tasks are outputting graphics to layers, a locking mechanism is provided to coordinate display updating.

The windowing environment provided by the Intuition library is largely based on Layers.

The Layers Library takes care of the mundane things, the low level, repetitive tasks that are needed to keep track of where to put which bits.

## DEFINITION OF LAYERS

The internal definition of the layers is essentially a set of *clipping rectangles*.

The Layers Library manages interactions between the various layers by using a data structure called **Layer\_Info**. Each common drawing area (shared by multiple layers) requires one **Layer\_Info** data structure.

## TYPES OF LAYERS SUPPORTED

The Layers Library supports three types of layers:

- *Simple Refresh*

All graphics rendering routines are “clipped”, so that only exposed sections of the layer are drawn into. No back-up of obscured areas is provided. When an obscured section of the layer is exposed to view, the routine using this layer may determine that a “refresh” of that section is in order.

- *Smart Refresh*

The system provides dynamic backup of obscured sections, into which the graphics routines will automatically draw. The backup sections will be used to automatically update the screen when the obscured sections later become exposed.

- *SuperBitMap*

The application provides a single back-up area to store what is not exposed to view in the layer. The back-up area may be as large or larger than the layer. Whenever an obscured area is made visible, the corresponding part of the backup area is copied to the screen automatically.

Any type of layer may also be a backdrop layer, which will always appear behind all other layers. They may not be moved, sized, or depth-arranged.

## Layers Library Routines

The Layers Library contains these routines:

<b>Purpose</b>	<b>Routine</b>
Allocating a <b>Layer_Info</b> structure	<b>NewLayerInfo( )</b>
Deallocating a <b>Layer_Info</b> structure	<b>DisposeLayerInfo( )</b>
Creating and deleting layers	<b>CreateBehindLayer( ),</b> <b>CreateUpfrontLayer( ),</b> <b>DeleteLayer( )</b>
Moving layers	<b>MoveLayer( )</b>
Sizing layers	<b>SizeLayer( )</b>
Changing a viewpoint	<b>ScrollLayer( )</b>
Reordering layers	<b>BehindLayer, UpfrontLayer( ),</b> <b>MoveLayerInFrontOf( )</b>
Determining layer position	<b>WhichLayer( )</b>
Sub-layer rectangle operations	<b>SwapBitsRastPortClipRect( )</b>
Intertask operations	<b>LockLayer( ), UnlockLayer( ),</b> <b>LockLayers( ), UnlockLayers( ),</b> <b>LockLayerInfo( ), UnlockLayerInfo( )</b>
Synchronization operations	<b>BeginUpdate( ), EndUpdate( )</b>

The following routines from **graphics.library** also allow access to layers functions:

<b>Purpose</b>	<b>Routine</b>
Intertask operations	<b>LockLayerRom( ), UnlockLayerRom( ),</b> <b>AttemptLockLayerRom( )</b>

These functions are similar to the layers **LockLayer( )** and **UnlockLayer( )** functions, but do not require the **layers.library** to be open. See the *Includes and Autodocs Manual* for more details.

## **ALLOCATING AND DEALLOCATING LAYER\_INFO**

The function **NewLayerInfo( )** allocates and initializes a **Layer\_Info** structure and some associated sub-structures. It must be called before attempting to use the other Layers functions described below.



The function **DisposeLayerInfo( )** deallocates a **Layer\_Info** and associated structures (as allocated by **NewLayerInfo( )**).

## ALLOCATING AND DEALLOCATING LAYERS

Layers are created using the routines **CreateUpfrontLayer( )** and **CreateBehindLayer( )**.

## CREATING AND DELETING LAYERS

**CreateUpfrontLayer( )** creates a layer that will appear in front of any existing layers.

**CreateBehindLayer( )** creates a layer that appears behind existing layers, but in front of backdrop layers.

Both of these routines return a pointer to a Layer data structure (as defined in the include file *graphics/layers.h*), or NULL if the operation was unsuccessful.

### NOTE

When a layer is created, the routine automatically creates a **RastPort** to go along with it. If this layer's **RastPort** is passed to the drawing routines, drawing is restricted to the layer. See also the topic called "The Layer's RastPort" below.

**DeleteLayer( )** is used to remove a layer from the layer list and free the memory allocated by the layer creation calls listed above.

## MOVING LAYERS

**MoveLayer( )** moves a layer to a new location.

## SIZING LAYERS

The **SizeLayer( )** command changes the size of a layer by modifying the coordinates of the lower right corner of the layer.

## CHANGING A VIEWPOINT

**ScrollLayer( )** is most useful with SuperBitMap layers. This command changes the portion of a SuperBitMap that is shown by a layer. It simulates this effect on the other layer types by adding the scroll offset to all future rendering.

## REORDERING LAYERS

**BehindLayer()** and **UpfrontLayer()** are used, respectively, to move a layer behind all other layers or in front of all other layers. **BehindLayer()** also considers any backdrop layers, moving a current layer behind all others except backdrop layers.

**MoveLayerInFrontOf()** is used to place a layer at a specific depth, just in front of a given layer.

## DETERMINING LAYER POSITION

If the viewing area has been separated into several layers, the application may need to find out which layer is topmost at a particular x,y coordinate.

To be sure that no task adds, deletes, or changes the sequence of layers before your task can use this information, call **LockLayerInfo()** before calling **WhichLayer()**, and call **UnlockLayerInfo()** when your operation is complete. In this way, you can be sure that you are acting on valid information.

## INTERTASK OPERATIONS

This section shows the use of the routines **LockLayerInfo()**, **UnlockLayerInfo()**, **LockLayer()**, **UnlockLayer()**, **LockLayers()**, and **UnlockLayers()**.

### **LockLayerInfo()** and **UnlockLayerInfo()**

If multiple tasks are manipulating layers on the same screen they will be sharing a **Layer\_Info** structure, and their use of it and its related data structures need to be coordinated. To ensure that a structure remains cohesive, it should be operated on by only one task at a time. The **Layer\_Info** encompasses all the layers existing on a single screen. **LockLayerInfo()** needs to be called whenever the visible portions of layers may be affected, or when the **Layer\_Info** structure is changed.

The lock should be obtained whenever the **CreateUpfrontLayer()**, **CreateBehindLayer()** and **DeleteLayer()** functions are called. And when a layer is moved, the list of layers that is being managed by the **Layer\_Info** data structure is affected.

It is not necessary to lock the **Layer\_Info** data structure while rendering, or when calling routines like **ScrollLayer()**, because layer sizes and on-screen positions are not being affected.

**LockLayerInfo()** grants the calling task exclusive access to the **Layer\_Info** structure. If some other task already has the **Layer\_Info** locked, this call will block until the other task calls **UnlockLayerInfo()**.

### **LockLayer()** and **UnlockLayer()**

If a task makes changes to a layer, other tasks should be prevented from rendering graphics into the layer. **LockLayer()** is used to block graphics output to this layer. If a graphics function is in process, the lock will return when the function is completed. Other tasks are blocked only if they attempt to draw graphics into this layer, or try to obtain a lock on this layer. The **MoveLayer()**, **SizeLayer()** and **ScrollLayer()** functions automatically lock and unlock the layer they operate on.

**UnlockLayer()** frees the locked layer for other operations.

If more than one layer must be locked, then the **LockLayer()** calls should be surrounded by **LockLayerInfo()** and **UnlockLayerInfo()** calls, to prevent deadlock situations.

### **LockLayers() and UnlockLayers()**

**LockLayers()** is used to lock all layers in a single command. **UnlockLayers()** releases the layers. The system calls these routines during the **BehindLayer()**, **UpfrontLayer()** and **MoveLayerInFrontOf()** operations.

As areas of Simple-Refresh layers become exposed, due to layer movement or sizing, for example, the newly exposed areas have not been drawn into, and need refreshing. The system keeps track of these areas by using a **DamageList**. To update only those areas that need it, **BeginUpdate()** is called. **BeginUpdate()** saves the pointer to the current clipping rectangles and installs a pointer to a set of **ClipRects** generated from the **DamageList** in the layer structure. To repair the layer, use the graphics rendering routines as if to redraw the entire layer, and the routines will automatically use the new clipping rectangle list. So, only the damaged areas are actually rendered into, saving time.

### **NOTE**

The program should never access the **DamageList**. The system generates and maintains the **DamageList** region. All user clipping is done through **ClipRects** (user clipping rectangles), and all access to this should be done through the function **InstallClipRegion()**.

To complete the update process call **EndUpdate()**, which will restore the original **ClipRect** list.

### **SUB-LAYER RECTANGLE OPERATIONS**

The **SwapBitsRastPortClipRect()** routine is for users who do not want to worry about clipping rectangles. If you wish to produce a menu, there are two ways to do it:

- Create an up-front layer with **CreateUpfrontLayer()**, then render the menu in it. This could use lots of memory and require a lot of (very temporary) “slice-and-dice” operations to create all of the clipping rectangles for the existing windows and so on.
- Use **SwapBitsRastPortClipRect()**, directly on the screen drawing area:
  - Render the menu in a back-up area off the screen, then lock all of the on-screen layers so that no task can use graphics routines to draw over your menu area on the screen.
  - Next, swap the on-screen bits with the off-screen bits, making the menu appear.
  - When you finish with the menu, swap again and unlock the layers.

The second method is faster and leaves the clipping rectangles and most of the rest of the window data structures untouched.

#### NOTE

All of the layers must be locked while the menu is visible. Any task that is using any of the layers for graphics output will be halted while the menu operations are taking place. If, on the other hand, the menu is rendered as a layer, no task need be halted while the menu is up because the lower layers need not be locked.

## The Layer's RastPort

#### NOTE

When a layer is created, the routine automatically creates a **RastPort** to go along with it. The pointer to the **RastPort** is contained in the layer data structure.

Using this **RastPort**, the application can draw anywhere into the layer's **bounds** rectangle. If the application tries to draw outside of this rectangle, the graphics routines will clip the graphics.

The type of layer specified by the **Flags** variable determines the other facilities the layer provides. The following paragraphs describe the layer types —simple refresh, smart refresh and **SuperBitMap**—and the **Layer Flags** that need to be set for each.

#### NOTE

The three layer-type **Flags** are mutually exclusive. That is, only one layer-type flag (**LAYERSIMPLE**, **LAYERSMART**, **LAYERSUPER**), should be specified.

### SIMPLE REFRESH LAYER

When an application draws into the layer, any portion of the layer that is visible (not obscured) will be rendered into the common **BitMap** of the viewing area.

If another layer operation is performed that causes part of a simple refresh layer to be obscured and then exposed, the application must restore the damaged part of the layer.

Simple refresh has two basic advantages:

- It does not require back-up area to save drawing sections that cannot be seen, saving memory.
- When an application restores the layer by performing a full-layer redraw, only the damaged areas are redrawn, making the operation time efficient.

The disadvantage is that the application needs to monitor to see if its layer needs refreshing. This is typically performed with statements like:

```
if (layer->Flags & LAYERREFRESH)
    refresh(layer);
```

## **SMART REFRESH LAYER**

If any portion of the layer is hidden by another layer, the bits for the obscured portions are rendered into back-up areas, which is provided automatically by the system. With smart refresh layers, the system handles all of the refresh requirements except when the layer is made larger. Its disadvantage is the additional memory needed to handle this automatic refresh.

## **SUPERBITMAP LAYER**

A SuperBitmap layer is similar to a smart refresh layer. It too has a back-up area into which drawings are rendered for currently obscured parts of the display. However, it differs from smart refresh in that:

- The back-up Bitmap is user-supplied, rather than being allocated dynamically by the system.
- The back-up Bitmap may be larger than the area of the Bitmap that is shown in the current size of this layer.

To see a larger portion of a SuperBitmap on-screen, use `SizeLayer( )`. To see a different portion of the SuperBitmap in the layer, use `ScrollLayer( )`.

When the graphics routines perform drawing commands, part of the drawing appears in the common Bitmap (the on-screen portion). Any drawing outside the layer itself is rendered into the SuperBitmap. When it is time to scroll or size the layer, the layer contents are copied into the SuperBitmap, the scroll or size positioning is modified, and the appropriate portions are then copied back into the layer.

## **BACKDROP LAYER**

A layer of any type may be designated a backdrop layer. The backdrop flag may be turned off to temporarily allow a layer to be depth-arranged. Then the backdrop flag can be restored to again inhibit depth-arrangement operations.

# **Using the Layers Library**

The following is a step-by-step example showing how the Layers Library can be used in your programs.

## **OPENING THE LAYERS LIBRARY**

Like all library routines, the Layers Library must be opened before it can be used. This is typically done by the following code:

```

struct LayersBase *LayersBase;

LayersBase = (struct LayersBase *)OpenLibrary("layers.library",0);
if(LayersBase == NULL)
{
    clean_exit(NO_LAYERS_LIBRARY_FOUND);
}

```

## OPENING THE GRAPHICS LIBRARY

Because the application probably uses various graphics library functions, it must also open the graphics library, as described elsewhere in this manual.

## CREATING A VIEWING WORKSPACE

You can create a viewing workspace by using the primitives **InitVPort()**, **InitView()**, **MakeVPort()**, **MrgCop()**, and **LoadView()**. Please reference the "Graphics Primitives" chapter.

To allocate and initialize a **Layer\_Info** data structure with which the system can keep track of layers that are created, use statements like:

```

struct Layer_Info *li;
li = NewLayerInfo( );
if(li == NULL)
{
    clean_exit(NEWLAYERINFO_FAILED);
}

```

## CREATING THE LAYERS

You can create layers in the common bit map by calling **CreateUpfrontLayer()** (or **CreateBehindLayer()**), with a sequence such as the following. This sequence requests construction of a smart refresh layer.

```

struct RastPort *rp;    /* allocate a RastPort pointer for each layer */
struct Layer *layer;    /* allocate a layer pointer for each layer */

/* Layer_Info, common BitMap, x1,y1,x2,y2,
** flags = 0 (smart refresh), null pointer to SuperBitMap
*/
layer = CreateUpfrontLayer(li,&b,20,20,100,80,LAYERSMART,NULL);

/* if not enough memory, can't continue the example */
if(layer==NULL)
    clean_exit(CANT_CREATE_LAYER);

```

## GETTING THE POINTERS TO THE RASTPORTS

Each layer pointer data structure contains a pointer to the **RastPort** that it uses. Here is the assignment from the layer structure to a local pointer:

```
rp = layer->rp;
```

## USING THE RASTPORTS FOR DISPLAY

Here are the rectangle-fill operations that create the display:

```
SetAPen(rp,1);  
SetDrMd(rp,JAM1);  
RectFill(rp,0,0,80,50);
```

## Regions

A clipping rectangle is a rectangular area into which the graphics routines will draw. All drawing that would fall outside of that rectangular area is clipped (not rendered). User clipping regions are linked lists of clipping rectangles created by an application program. Use `InstallClipRegion()` to make a user clipping region active. The *graphics.library* contains support routines for regions.

### NOTE

All of the following functions but `InstallClipRegion()` are in *graphics.library*. `InstallClipRegion()` is in *layers.library*.

Among these are routines for the following operations:

Operation	Routine
Creating and deleting regions	<code>NewRegion()</code> , <code>DisposeRegion()</code>
Installing a region	<code>InstallClipRegion()</code>
Changing a region	<code>AndRectRegion()</code> , <code>OrRectRegion()</code> , <code>XorRectRegion()</code> , <code>ClearRectRegion()</code> , <code>AndRegionRegion()</code> , <code>OrRegionRegion()</code> , <code>XorRegionRegion()</code>
Clearing a region	<code>ClearRegion()</code>

The region commands are used to construct a user clipping region, which can be used with the graphics rendering routines. With this list, the application can selectively update a custom-shaped part of a layer without disturbing any of the other layers that might be present.

### NOTE

Never access the `DamageList` directly. Use the routine `InstallClipRegion()` to add clipping to the layer. After a region has been added with `InstallClipRegion()`, you may not modify it unless it has been removed with another call to `InstallClipRegion()`.

## CREATING AND DELETING REGIONS

**NewRegion( )** allocates and initializes a new data structure that has no drawable areas defined in it.

If this new region is to be used as a user clipping region, and the application tries to draw something through it, nothing would be drawn, as there is nothing in the region.

Because regions are dynamically created using **NewRegion( )**, the procedure **DisposeRegion( )** is provided to return the memory to the system when the application has finished with it.

### NOTE

The region structure and all rectangles that have been linked to it are deallocated. All of the functions that add rectangles to the region make copies of the rectangles. If the program allocates a rectangle, then adds it to a region, it still must deallocate the rectangle. The call to **DisposeRegion( )** will not deallocate the rectangle.

## INSTALLING REGIONS

Use the function **InstallClipRegion( )** to install your region. This installs a transparent clipping region in the layer. All subsequent graphics calls will be clipped to this region. You must remove the region with a second call to **InstallClipRegion( )** before you remove the Layer.

For example:

```
register struct Region    *new_region ;
register struct Region    *old_region ;

old_region = InstallClipRegion(win->WLayer, new_region);

/* draw into the layer or window */

if (NULL != (old_region = InstallClipRegion(win->WLayer, old_region)))
    DisposeRegion(new_region) ;
```

### NOTE

You must be very careful not to call **InstallClipRegion( )** inside of a **BeginRefresh/EndRefresh( )** or **BeginUpdate/EndUpdate( )** pair. The following code segment shows how you may modify the user clipping region with these calls.

```
register struct Region    *new_region ;
register struct Region    *old_region ;

/* you have to have already setup the new_region and old_region */

BeginRefresh(window);
/* draw through the damage list */
/* into the layer or window */
EndRefresh(window, FALSE);           /* keep the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the damage list and the new_region */
/* into the layer or window */
```



```

EndRefresh(window, FALSE);          /* keep the damage list */

/* put back the old region */
new_region = InstallClipRegion(win->WLayer, old_region);

BeginRefresh(window);
EndRefresh(window, TRUE);           /* remove the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the new region only into the layer or window */
EndRefresh(window, FALSE);

/* finally get rid of the new region, old_region still installed */
if (NULL != (new_region = InstallClipRegion(win->WLayer, old_region)))
    DisposeRegion(new_region);

```

Here is some sample code for clipping windows:

```

/*-----
** UnclipWindow( )
**
** routine to remove a clipping region installed by ClipWindow( ) or
** ClipWindowToBorders( ), disposing of the installed region and
** reinstalling the region removed.
*/
void UnclipWindow(struct Window *win, struct Region *prev_region)
{
    register struct Region    *old_region ;

    /* remove any old region by installing a NULL,
    ** then dispose of the old region if one was there.
    */
    if (NULL != (old_region = InstallClipRegion(win->WLayer, prev_region)))
        DisposeRegion(old_region);
}

/*-----
** ClipWindow( )
** clip a window to a specified rectangle (given by upper left and
** lower right corner. the removed region is returned so that it
** can be re-installed later.
*/
struct Region *ClipWindow(struct Window *win,
                          LONG minX, LONG minY, LONG maxX, LONG maxY)
{
    register struct Region    *new_region ;
    register struct Region    *old_region ;

    struct Rectangle    my_rectangle ;

    /* set up the limits for the clip */
    my_rectangle.MinX = minX ;
    my_rectangle.MinY = minY ;
    my_rectangle.MaxX = maxX ;
    my_rectangle.MaxY = maxY ;

    /* get a new region and OR in the limits. */
    new_region = NewRegion( ) ;
    OrRectRegion(new_region, &my_rectangle);

    /* install the new region, and dispose of any existing region */
    return(InstallClipRegion(win->WLayer, new_region));
}

/*-----
** ClipWindowToBorders( )
** clip a window to its borders.
** the removed region is returned so that it can be re-installed later.
*/

```

```

struct Region *ClipWindowToBorders(struct Window *win)
{
    register struct Region    *new_region ;
    register struct Region    *old_region ;

    struct Rectangle          my_rectangle ;

    /* set up the limits for the clip */
    my_rectangle.MinX = win->BorderLeft - 1 ;
    my_rectangle.MinY = win->BorderTop - 1 ;
    my_rectangle.MaxX = win->Width - win->BorderRight ;
    my_rectangle.MaxY = win->Height - win->BorderBottom ;

    /* get a new region and OR in the limits. */
    new_region = NewRegion( ) ;
    OrRectRegion(new_region, &my_rectangle) ;

    /* install the new region, and dispose of any existing region */
    return(InstallClipRegion(win->WLayer, new_region));
}

```

## CHANGING A REGION

Regions may be modified by performing logical operations with rectangles, or with other regions.

## RECTANGLES AND REGIONS

### NOTE

In all of the RectRegion routines the clipping rectangle is *copied* into the region. This means that a single clipping rectangle (struct Rectangle) can be used many times by simply changing the x and y values.

For instance:

```

LONG ktr;
struct Rectangle rect;
struct Region *RowRegion = NULL;

for (ktr=1; ktr<6; ktr++)
{
    rect.MinX = 50;
    rect.MaxX = 315;
    rect.MinY = (ktr*10)-5;
    rect.MaxY = (ktr*10);

    if (!OrRectRegion(RowRegion, &rect))
        clean_exit(RETURN_WARN);
}

```

**OrRectRegion( )** modifies a region structure by *or*'ing a clipping rectangle into the region. If the application now tries to draw through this region (assuming that the region was originally empty), only the pixels within the clipping rectangle will be affected. If the region already has drawable areas, they will still exist, this rectangle is just added to the drawable area.

**AndRectRegion( )** modifies the region structure by *and*'ing a clipping rectangle into the region. Only those pixels that were already drawable and within the rectangle will remain drawable, any that are outside of it will be clipped in future.

**XorRectRegion( )** applies the rectangle to the region in an *exclusive-or* mode. Within the given rectangle, any areas that were drawable become clipped, any areas that were clipped become drawable. Areas outside of the rectangle are not affected.

**ClearRectRegion( )** clears the rectangle from the region. Within the given rectangle, any areas that were drawable become clipped. Areas outside of the rectangle are not affected.

## REGIONS AND REGIONS

**AndRegionRegion( )** performs a logical *and* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever the regions drawable areas overlap. That is, where there are drawable areas in both region 1 *and* region 2, there will be drawable areas left in the result region.

**OrRegionRegion( )** performs a logical *or* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever there are drawable areas in either region. That is, where there are drawable areas in either region 1 *or* region 2, there will be drawable areas left in the result region.

**XorRegionRegion( )** performs a logical *exclusive-or* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever there are drawable areas in either region but not both. That is, where there are drawable areas in either region 1 *or* region 2, there will be drawable areas left in the result region. But where there are drawable areas in both region 1 *and* region 2, there will **not** be drawable areas left in the result region.

## CLEARING A REGION

**ClearRegion( )** puts the region back to the same state it was in when the region was created with **NewRegion( )**, that is, no areas are drawable.

## LAYERS EXAMPLE

Here is the example code. Layers functions are exercised first, then regions functions are used. To see a window explaining the calls as they happen, leave **MESSAGES** defined. To slow the action down, increase the size of the **S\_DELAY** define.

### NOTE

For the sake of brevity, the example is a single task. So, no Layer locking is done. Also note that the routine *myLabelLayer* is ALWAYS used to redraw a given Layer. It is called only when a Layer needs refreshing or to show the affect of region manipulation.

```
/* Layers.c
 * Compiled with Lattice 5.04: LC -bl -cfist -L -v -y
 */

#include <exec/types.h>
#include <graphics/gfxbase.h>
#include <graphics/layers.h>
#include <proto/all.h>
#include <stdlib.h>
#include <string.h>

VOID clean_exit(LONG retc);
```

```

VOID cleanUp(VOID);
VOID myOrCols(struct Region *region);
VOID pMessage(UBYTE *string);
VOID myLabelRegion( struct Region *region, struct Layer *layer,
                    LONG color, UBYTE *string);
VOID myLabelAllRegions(LONG color);
VOID myLabelLayer( struct Layer *layer, LONG color, UBYTE *string);
VOID myResetRegions(VOID);

#define L_DELAY 100
#define S_DELAY 50

#define DUMMY 0L

#define CLR_RED 1
#define CLR_GRN 2
#define CLR_BLU 3

#define SCREEN_D 2
#define SCREEN_W 320
#define SCREEN_H 200

/* the starting size of example layers, offsets are used for placement */
#define W_H 50
#define W_T 5
#define W_B (W_T+W_H)-1
#define W_W 80
#define W_L (SCREEN_W/2) - (W_W/2)
#define W_R (W_L+W_W)-1

/* size of the superbitmap */
#define SUPER_H SCREEN_H
#define SUPER_W SCREEN_W

/* starting size of the message layer */
#define M_H 10
#define M_T SCREEN_H-M_H
#define M_B (M_T+M_H)-1
#define M_W SCREEN_W
#define M_L 0
#define M_R (M_L+M_W)-1

/* This example shows how to use the layers.library.
** This code may be freely utilized to develop programs for the Amiga.
*/
struct GfxBase *GfxBase;
struct Library *LayersBase;

/* global for FreeMem( ) */
struct View *oldview = NULL; /* save old view so can go back to sys */
struct View theView;
struct ViewPort theViewPort;
/* pointer to colormap struct, dynamic alloc */
struct ColorMap *theColorMap = NULL;
struct BitMap theBitMap, theSuperBitMap;
struct Layer *theLayers[3] = { NULL, NULL, NULL, };
struct Layer *msgLayer = NULL;
LONG theLayerFlags[3] = { LAYERSUPER, LAYERSMART, LAYERSIMPLE };
struct Layer_Info *theLayerInfo = NULL;

USHORT colortable[] = { 0x000, 0xf00, 0x0f0, 0x00f };
struct Region *ColRegion = NULL;
struct Region *RowRegion = NULL;
struct Region *theRegions[3] = { NULL, NULL, NULL };

VOID main(int argc, char **argv)
{
    struct RasInfo theRasInfo;
    short iii,jjj;
    UWORD *colorpalette;
    struct Rectangle rect; /* some rectangle structures */

```

```

if ((GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",33L)) == NULL)
    clean_exit(RETURN_WARN);

if((LayersBase = OpenLibrary("layers.library",33L)) == NULL)
    clean_exit(RETURN_WARN);

/* example steals screen from Intuition,
** this is just an example. In real life, open your own.
*/
oldview = GfxBase->ActiView;    /* save current view, go back later */

/* get a LayerInfo structure */
if((theLayerInfo = NewLayerInfo( )) == NULL)
    clean_exit(RETURN_WARN);

InitView(&theView);
theView.ViewPort = &theViewPort;
InitVPort(&theViewPort);

theViewPort.DWidth  = SCREEN_W;
theViewPort.DHeight = SCREEN_H;
theViewPort.RasInfo = &theRasInfo;

InitBitMap(&theBitMap, SCREEN_D, SCREEN_W, SCREEN_H);

theRasInfo.BitMap    = &theBitMap;
theRasInfo.RxOffset  = 0;
theRasInfo.RyOffset  = 0;
theRasInfo.Next      = NULL;

theColorMap = GetColorMap(4);

colorpalette = (UWORD *)theColorMap->ColorTable;
for(iii=0; iii<4; iii++)
    *colorpalette++ = colortable[iii];
theViewPort.ColorMap = theColorMap;    /* link it with the viewport */

for(iii=0; iii<SCREEN_D; iii++)
{
    if((theBitMap.Planes[iii] =
        (PLANEPTR)AllocRaster(SCREEN_W,SCREEN_H)) == NULL)
        clean_exit(RETURN_WARN);
    BltClear(theBitMap.Planes[iii], RASSIZE(SCREEN_W, SCREEN_H), 1);
}

MakeVPort( &theView, &theViewPort ); /* construct copper (prelim) list */
MrgCop( &theView );    /* merge copper lists in the view structure. */
LoadView(&theView);
WaitTOF( );

if ((msgLayer = CreateUpfrontLayer( theLayerInfo, &theBitMap,
    M_L, M_T, M_R, M_B, LAYERSMART, NULL)) == NULL)
    clean_exit(RETURN_WARN);
pMessage("Setting up Layers");

/* Layers stuff starts here *****/
InitBitMap(&theSuperBitMap, SCREEN_D, SUPER_W, SUPER_H);
for(iii=0; iii<SCREEN_D; iii++)
{
    if((theSuperBitMap.Planes[iii] =
        (PLANEPTR)AllocRaster(SUPER_W, SUPER_H)) == NULL)
        clean_exit(RETURN_WARN);
    BltClear(theSuperBitMap.Planes[iii], RASSIZE(SUPER_W, SUPER_H), 1);
}

for(iii=0; iii<3; iii++)
{
    pMessage("Create BehindLayer");
    if (iii == 0)
    {
        if((theLayers[iii] = CreateBehindLayer( theLayerInfo, &theBitMap,
            W_L+(iii*30), W_T+(iii*30), W_R+(iii*30), W_B+(iii*30),

```

```

        theLayerFlags[iii], &theSuperBitMap)) == NULL)
    clean_exit(RETURN_WARN);
}
else
{
    if((theLayers[iii] = CreateBehindLayer( theLayerInfo, &theBitMap,
        W_L+(iii*30), W_T+(iii*30), W_R+(iii*30), W_B+(iii*30),
        theLayerFlags[iii], NULL)) == NULL)
        clean_exit(RETURN_WARN);
}

pMessage("RectFill the RastPort");
SetAPen(theLayers[iii]->rp, iii+1);
SetDrMd(theLayers[iii]->rp, JAM1);
if (iii == 0) RectFill(theLayers[iii]->rp, 0, 0, SUPER_W-1, SUPER_H-1);
if (iii == 1) RectFill(theLayers[iii]->rp, 0, 0, W_W-1, W_H-1);
if (iii == 2) RectFill(theLayers[iii]->rp, 0, 0, W_W-1, W_H-1);
SetAPen(theLayers[iii]->rp, 0);
Move(theLayers[iii]->rp, 5, 7);
}

pMessage("Label all Layers");
Text(theLayers[0]->rp, "Super", 5);
Text(theLayers[1]->rp, "Smart", 5);
Text(theLayers[2]->rp, "Simple", 6);

pMessage("MoveLayer 1 InFrontOf 0");
if (!MoveLayerInFrontOf( theLayers[1], theLayers[0]))
    clean_exit(RETURN_WARN);

pMessage("MoveLayer 2 InFrontOf 1");
if (!MoveLayerInFrontOf( theLayers[2], theLayers[1]))
    clean_exit(RETURN_WARN);
myLabelLayer(theLayers[2], CLR_BLU, "Simple");

pMessage("Incrementally MoveLayers...");
for(iii=0; iii<30; iii++)
{
    if (!MoveLayer(DUMMY, theLayers[1], -1, 0))
        clean_exit(RETURN_WARN);
    if (!MoveLayer(DUMMY, theLayers[2], -2, 0))
        clean_exit(RETURN_WARN);
    myLabelLayer(theLayers[2], CLR_BLU, "Simple");
}

pMessage("make Layer 0 the UpfrontLayer");
if (!UpfrontLayer(DUMMY, theLayers[0]))
    clean_exit(RETURN_WARN);

pMessage("make Layer 2 the BehindLayer");
if (!BehindLayer(DUMMY, theLayers[2]))
    clean_exit(RETURN_WARN);

pMessage("Incrementally MoveLayers again...");
for(iii=0; iii<30; iii++)
{
    if (!MoveLayer(DUMMY, theLayers[1], 0, 1))
        clean_exit(RETURN_WARN);
    if (!MoveLayer(DUMMY, theLayers[2], 0, 2))
        clean_exit(RETURN_WARN);
    myLabelLayer(theLayers[2], CLR_BLU, "Simple");
}

pMessage("Big MoveLayer");
for(iii=0; iii<3; iii++)
{
    if (!MoveLayer(DUMMY, theLayers[iii], -theLayers[iii]->bounds.MinX, 0))
        clean_exit(RETURN_WARN);
}

pMessage("Incrementally increase size");
for(iii=0; iii<5; iii++)

```

```

{
    for(jjj=0; jjj<3; jjj++)
    {
        if (!SizeLayer(DUMMY, theLayers[jjj], 1, 1))
            clean_exit(RETURN_WARN);
    }
    myLabelLayer(theLayers[1], CLR_GRN, "Smart");
    myLabelLayer(theLayers[2], CLR_BLU, "Simple");
}

pMessage("Big SizeLayer");
for(iii=0; iii<3; iii++)
{
    if (!SizeLayer(DUMMY, theLayers[iii], SCREEN_W-(theLayers[iii]->bounds.MaxX)-1, 0))
        clean_exit(RETURN_WARN);
}
myLabelLayer(theLayers[1], CLR_GRN, "Smart");
myLabelLayer(theLayers[2], CLR_BLU, "Simple");

pMessage("ScrollLayer down");
for(iii=0; iii<30; iii++)
{
    for(jjj=0; jjj<3; jjj++)
    {
        ScrollLayer(DUMMY, theLayers[jjj], 0, -1);
    }
    myLabelLayer(theLayers[1], CLR_GRN, "Smart");
    myLabelLayer(theLayers[2], CLR_BLU, "Simple");
}

pMessage("ScrollLayer up");
for(iii=0; iii<30; iii++)
{
    for(jjj=0; jjj<3; jjj++)
    {
        ScrollLayer(DUMMY, theLayers[jjj], 0, 1);
    }
    myLabelLayer(theLayers[1], CLR_GRN, "Smart");
    myLabelLayer(theLayers[2], CLR_BLU, "Simple");
}

/* Regions stuff starts here *****/
pMessage("Create Regions");
if ((RowRegion = NewRegion( )) == NULL)
    clean_exit(RETURN_WARN);
if ((ColRegion = NewRegion( )) == NULL)
    clean_exit(RETURN_WARN);
for(iii=0; iii<3; iii++)
    if ((theRegions[iii] = NewRegion( )) == NULL) /* for each layer */
        clean_exit(RETURN_WARN);

pMessage("Build two tmp regions");
myOrCols(ColRegion); /* made into subroutine, used often */
for (iii=1; iii<6; iii++)
{
    rect.MinX = 50;
    rect.MaxX = 315;
    rect.MinY = (iii*10)-5;
    rect.MaxY = (iii*10);
    if (!OrRectRegion(RowRegion, &rect))
        clean_exit(RETURN_WARN);
}

rect.MinX = 5;
rect.MaxX = 315;
rect.MinY = 25;
rect.MaxY = 30;

myResetRegions( );
pMessage("OrRectRegion one row...");
for (iii=0; iii<3; iii++)
    if (!OrRectRegion(theRegions[iii], &rect))
        clean_exit(RETURN_WARN);

```

```

pMessage("OrRectRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("XorRectRegion one row...");
for (iii=0; iii<3; iii++)
    if (!XorRectRegion(theRegions[iii], &rect))
        clean_exit(RETURN_WARN);

pMessage("XorRectRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("AndRectRegion one row...");
for (iii=0; iii<3; iii++)
    AndRectRegion(theRegions[iii], &rect);

pMessage("AndRectRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("ClearRectRegion one row...");
for (iii=0; iii<3; iii++)
    if (!ClearRectRegion(theRegions[iii], &rect))
        clean_exit(RETURN_WARN);

pMessage("ClearRectRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("OrRegionRegion Rows...");
for (iii=0; iii<3; iii++)
    if (!OrRegionRegion(RowRegion, theRegions[iii]))
        clean_exit(RETURN_WARN);

pMessage("OrRegionRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("XorRegionRegion Rows...");
for (iii=0; iii<3; iii++)
    if (!XorRegionRegion(RowRegion, theRegions[iii]))
        clean_exit(RETURN_WARN);

pMessage("XorRegionRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

myResetRegions( );
pMessage("AndRegionRegion Rows...");
for (iii=0; iii<3; iii++)
    if (!AndRegionRegion(RowRegion, theRegions[iii]))
        clean_exit(RETURN_WARN);

pMessage("AndRegionRegion result (blue)");
myLabelAllRegions(CLR_BLU);
Delay(L_DELAY);

cleanUp( );
} /* end of main( ) */

VOID clean_exit(LONG retc)
{
    cleanUp( );
    exit(retc);
}

```



```

VOID cleanUp(VOID)
{
    short iii;

    if (oldview)
    {
        LoadView(oldview);          /* put back the old view */
        WaitTOF( );
    }

    if (msgLayer != NULL)
        if (!DeleteLayer(DUMMY, msgLayer))
            exit(RETURN_FAIL);

    if (ColRegion != NULL)
        DisposeRegion(ColRegion);
    if (RowRegion != NULL)
        DisposeRegion(RowRegion);
    for(iii=0; iii<3; iii++)
    {
        if (theRegions[iii] != NULL)
            DisposeRegion(theRegions[iii]);
        if (theLayers[iii] != NULL)
            if (!DeleteLayer(DUMMY, theLayers[iii]))
                exit(RETURN_FAIL);
    }

    /* !!! free superbitmap */
    for (iii=0; iii<SCREEN_D; iii++)
    {
        /* free the drawing area */
        if (theSuperBitMap.Planes[iii] != NULL)
            FreeRaster(theSuperBitMap.Planes[iii], SUPER_W, SUPER_H);
    }

    if (theLayerInfo != NULL)
        DisposeLayerInfo(theLayerInfo);
    if (theColorMap != NULL)
        FreeColorMap(theColorMap);    /* free the color map */
    for (iii=0; iii<SCREEN_D; iii++)
    {
        /* free the drawing area */
        if (theBitMap.Planes[iii] != NULL)
            FreeRaster(theBitMap.Planes[iii], SCREEN_W, SCREEN_H);
    }

    /* free dynamically created structures */
    FreeVPortCopLists(&theViewPort);
    FreeCprList(theView.LOFCprList);

    if (LayersBase != NULL)
        CloseLibrary((struct Library *)LayersBase);
    if (GfxBase != NULL)
        CloseLibrary((struct Library *)GfxBase);
}

VOID myOrCols(struct Region *region)
{
    short iii;
    struct Rectangle rect;
    for (iii=5; iii<10; iii++)
    {
        rect.MinX = (iii*10);
        rect.MaxX = (iii*10)+5;
        rect.MinY = 5;
        rect.MaxY = 50;
        if (!OrRectRegion(region, &rect))
            clean_exit(RETURN_WARN);
    }
}

VOID pMessage(UBYTE *string)
{
    Delay(S_DELAY);
}

```

```

myLabelLayer(msgLayer, CLR_GRN, string);
}

VOID myLabelRegion( struct Region *region, struct Layer *layer,
                   LONG color, UBYTE *string)
{
    struct Region *old_region;

    /* blow away the damage list. */
    if (BeginUpdate(layer))
        EndUpdate(layer, TRUE);

    /* install a user clipping region.
    ** draw into the layer
    ** then put back the old clipping region
    */
    old_region = InstallClipRegion(layer, region);
    myLabelLayer(layer, color, string);
    region = InstallClipRegion(layer, old_region);
}

VOID myLabelAllRegions(LONG color)
{
    myLabelRegion(theRegions[0], theLayers[0], color, "Super");
    myLabelRegion(theRegions[1], theLayers[1], color, "Smart");
    myLabelRegion(theRegions[2], theLayers[2], color, "Simple");
}

VOID myLabelLayer( struct Layer *layer, LONG color, UBYTE *string)
{
    SetAPen(layer->rp, color);
    SetDrMd(layer->rp, JAM1);
    RectFill(layer->rp, 0, 0, layer->bounds.MaxX - layer->bounds.MinX,
             layer->bounds.MaxY - layer->bounds.MinY);
    SetAPen(layer->rp, 0);
    Move(layer->rp, 5, 7);
    Text(layer->rp, string, strlen(string));
}

VOID myResetRegions(VOID)
{
    short iii;

    pMessage("Clear all Regions");
    myLabelLayer(theLayers[0], CLR_RED, "Super");
    myLabelLayer(theLayers[1], CLR_RED, "Smart");
    myLabelLayer(theLayers[2], CLR_RED, "Simple");
    for (iii=0; iii<3; iii++)
        ClearRegion(theRegions[iii]);

    /* put the col region into each layer */
    pMessage("OrRegionRect Columns...");
    for (iii=0; iii<3; iii++)
        myOrCols(theRegions[iii]);

    pMessage("ORed in Column Rects - in green");
    myLabelAllRegions(CLR_GRN);
}

```

## Chapter 27

### Expansion Library

Amiga RAM expansions and other expansion bus peripherals are designed to reside at dynamically assigned address spaces within the system. The configuration and initialization of these expansion peripherals is performed by the `expansion.library`.

#### **AUTOCONFIG™**

The Amiga AUTOCONFIG protocol is designed to allow the dynamic assignment of available address slots to expansion boards, eliminating the need for user configuration via jumpers. Upon reset, each board appears in turn at \$E80000, with readable identification information, most of which is in one's complement format, appearing in the high nibbles of the first \$40 words (\$80 bytes) of the board. This identification information includes the size of the board, its address space preferences, type of board (memory or other), and a unique Hardware Manufacturer Number assigned by Commodore Applications and Technical Support (CATS), West Chester, Pennsylvania.

#### **NOTE**

This unique number is not the same as a Developer number. All commercial expansion slot boards for the Amiga must implement the AUTOCONFIG protocol. More in-depth machine-specific information on the design and implementation of AUTOCONFIG boards is available from Commodore Applications and Technical Support.

## The Expansion Sequence

During system initialization, expansion.library configures each expansion peripheral in turn by examining its identification information and assigning it an appropriate address space. If the board is a RAM board, it is added to the system memory list, and the RAM becomes available for allocation by system tasks. Descriptions of all configured boards are kept in a private ExpansionBase list of ConfigDev structures. Applications can examine this list with the expansion.library function FindConfigDev().

The ConfigDev structure (*libraries/configvars.h* and *.i*) follows:

```
struct ConfigDev
{
    struct Node      cd_Node;
    UBYTE           cd_Flags;
    UBYTE           cd_Pad;
    struct ExpansionRom cd_Rom;      /* image of expansion rom area */
    APTR            cd_BoardAddr;    /* where in memory the board is */
    APTR            cd_BoardSize;    /* size in bytes */
    UWORD           cd_SlotAddr;     /* which slot number */
    UWORD           cd_SlotSize;     /* number of slots the board takes */
    APTR            cd_Driver;       /* pointer to node of driver */
    struct ConfigDev * cd_NextCD;    /* linked list of drivers to config */
    ULONG           cd_Unused[4];    /* for whatever the driver wants */
};

/* cd_Flags */
#define CDB_SHUTUP      0      /* this board has been shut up */
#define CDB_CONFIGME    1      /* this board needs a driver to claim it */

#define CDF_SHUTUP      0x01
#define CDF_CONFIGME    0x02
```

As shown above, the ConfigDev structure contains an ExpansionRom structure. The ExpansionRom structure, defined in *libraries/configregs.h* and *.i*, contains the board identification information which was read from the board's PAL or EPROM at expansion time. This information includes the board's unique Manufacturer and Product ID. The following example uses FindConfigDev() to print out information about your system's configured expansion peripherals.

```
/*
 * FindBoards.c - Examine all AUTOCONFIG(tm) boards in the system
 * Compiled with Lattice 5.02: LC -b1 -cfist -v -y
 * Linkage: c.o,findboards.o library LC.lib,amiga.lib
 */
#include <exec/types.h>
#include <libraries/configvars.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

struct Library *ExpansionBase = NULL;

void main(int argc, char **argv)
{
    struct ConfigDev *myCD;
    UWORD m,p;

    if((ExpansionBase=OpenLibrary("expansion.library",0L))==NULL)
```

```

    exit (RETURN_FAIL);

/*-----*/
/* FindConfigDev(oldConfigDev,manufacturer,product) */
/* oldConfigDev = NULL for the top of the list */
/* manufacturer = -1 for any manufacturer */
/* product      = -1 for any product */
/*-----*/

myCD = NULL;
while(myCD=FindConfigDev(myCD,-1L,-1L)) /* search for all ConfigDevs */
{
    printf("\n---ConfigDev structure found at location %lx---\n",myCD);

    /* These values were read directly from the board at expansion time */
    printf("Board ID (ExpansionRom) information:\n");

    m = myCD->cd_Rom.er_Manufacturer;
    printf("er_Manufacturer      =%d=%x=(~%4x)\n",m,m,~m);

    p = myCD->cd_Rom.er_Product;
    printf("er_Product              =%d=%x=(~%4x)\n",p,p,~p);

    printf("er_Type                  =%x",myCD->cd_Rom.er_Type);
    if(myCD->cd_Rom.er_Type & ERTF_MEMLIST)
        printf(" (Adds memory to free list)\n");
    else printf("\n");

    printf("er_Flags                  =");
    printf("%x\n",myCD->cd_Rom.er_Flags);
    printf("er_InitDiagVec            =");
    printf("%x\n",myCD->cd_Rom.er_InitDiagVec);

    /* These values are generated when the AUTOCONFIG software
     * relocates the board
     */
    printf("Configuration (ConfigDev) information:\n");
    printf("cd_BoardAddr              =%lx\n",myCD->cd_BoardAddr);
    printf("cd_BoardSize              =%lx (%ldK)\n",
        myCD->cd_BoardSize, ((ULONG)myCD->cd_BoardSize)/1024);

    printf("cd_Flags                  =%x",myCD->cd_Flags);
    if(myCD->cd_Flags & CDF_CONFIGME)
        printf("\n");
    else printf(" (driver clears CONFIGME bit)\n");
}
CloseLibrary(ExpansionBase);
}

```

## Expansion Board Drivers

The Amiga operating system contains support for matching up disk-based drivers with AUTOCONFIG boards. Though such drivers are commonly Exec Devices, this is not required. The driver may, for instance, be an Exec Library or Task. The 1.3 system software also supports the initialization of onboard ROM driver software.

### DISK BASED DRIVERS

Disk-based expansion board drivers and their icons are generally placed in the *SYS:Expansion* drawer of the user's *SYS:* disk or partition. The icon ToolTypes field (which may be viewed and edited using the Workbench Info selection) must contain the unique Hardware Manufacturer number, and the Product number of the expansion board(s) the driver is written for.

The **BindDrivers** command issued during the disk startup-sequence attempts to match disk-based drivers with their expansion boards. To do this, **BindDrivers** looks in the **ToolTypes** field of all icon files in *SYS:Expansion*. If the **ToolType** "PRODUCT" is found in the icon, then this is an icon file for a driver. **Binddrivers** will then attempt to match the the Manufacturer and Product number in this **PRODUCT** **ToolType** with those of a board that was configured at expansion time.

For example, suppose you are manufacturer #1019. You have two products, #1 and #2 which both use the same driver. The icon for your driver for these two products would have a **Tool Type** set to "PRODUCT=1019/1|1019/2". This means: I am an icon for a driver that works with product number 1 or 2 from manufacturer 1019, now bind me. Spaces are not legal. Here are two other examples:

```
PRODUCT=1208/11    is the Tool Type for a driver for product
                    11 from manufacturer number 1208.

PRODUCT=1017       is the Tool Type for a driver for any
                    product from manufacturer number 1017.
```

If a matching board is found for the disk-based driver, the driver code is loaded and then initialized with the **Exec InitResident()** function. From within its initialization code, the driver can get information about the board it is bound to by calling the *expansion.library* function **GetCurrentBinding()**. This function will provide the driver with a copy of a **CurrentBinding** structure, including a pointer to a **ConfigDev** structure (possibly linked to additional **ConfigDevs** via the **cd\_NextCD** field) of the expansion board(s) which matched the Manufacturer and Product IDs.

```
/* this structure is used by GetCurrentBinding() and SetCurrentBinding() */
struct CurrentBinding
{
    struct ConfigDev *  cb_ConfigDev;          /* first configdev in chain */
    UBYTE *            cb_FileName;           /* file name of driver */
    UBYTE *            cb_ProductString;      /* product # string */
    UBYTE **           cb_ToolTypes;          /* tooltypes from disk object */
};
```

**GetCurrentBinding()** allows the driver to find out the the base address and other information about its board(s). The driver must unset the **CONFIGME** bit in the **ConfigDev cd\_Flags** field of each board it intends to drive, and record the driver's **Exec** node pointer in the **cd\_Driver** structure. This node should contain the **LN\_NAME** and **LN\_TYPE** (ie. **NT\_DEVICE**, **NT\_TASK**, etc.) of the driver.

#### NOTE

The **GetCurrentBinding()** function, and driver binding in general, must be bracketed by an **ObtainConfigBinding()** and **ReleaseConfigBinding()** semaphore. The **BindDrivers** command obtains the semaphore and performs a **SetCurrentBinding** before calling **InitResident()**, allowing the driver to simply do a **GetCurrentBinding()**.

Full source code for a disk-based **Expansion** or **DEVS:** device driver, and autodocs for *expansion.library* functions can be found in the Addison-Wesley *Includes and Autodocs Manual*.

#### MakeDosNode AND AddDosNode

Two other *expansion.library* functions used by expansion board drivers are **MakeDosNode()** and **AddDosNode()**. These functions allow a driver to create and add a DOS device node (for example **DH0:**) to the system.

**MakeDosNode()** requires an initialized structure of environment information for creating a DOS device node. The format of the command is:

```
struct DeviceNode *deviceNode = MakeDosNode(parameterPkt);
```

where parameterPkt is a pointer (passed in A0 from assembler) to an initialized structure of the following form:

```

;-----
;
; Layout of parameter packet for MakeDosNode
;
;-----
* The packet for MakeDosNode starts with the following four
* longwords, directly followed by a DosEnvec structure.

    APTR dosName          ; Points to a DOS device name (ex. 'RAM1',0)
    APTR execName         ; Points to device driver name (ex. 'ram.device',0)
    ULONG unit            ; Unit number
    ULONG flags           ; OpenDevice flags

* The DosEnvec disk "environment" is a longword array that describes the
* disk geometry. It is variable sized, with the length at the beginning.
* Here are the constants for a standard geometry.
* See libraries/filehandler.i for additional notes.

STRUCTURE DosEnvec,0
    ULONG de_TableSize    ; Size of Environment vector
    ULONG de_SizeBlock    ; in longwords: standard value is 128
    ULONG de_SecOrg       ; not used; must be 0
    ULONG de_Surfaces     ; # of heads (surfaces). drive specific
    ULONG de_SectorPerBlock ; not used; must be 1
    ULONG de_BlocksPerTrack ; blocks per track. drive specific
    ULONG de_Reserved     ; DOS reserved blocks at start of partition.
    ULONG de_PreAlloc     ; DOS reserved blocks at end of partition
    ULONG de_Interleave   ; usually 0
    ULONG de_LowCyl       ; starting cylinder. typically 0
    ULONG de_HighCyl      ; max cylinder. drive specific
    ULONG de_NumBuffers    ; Initial # DOS of buffers.
    ULONG de_BufMemType    ; type of mem to allocate for buffers
    ULONG de_MaxTransfer   ; Max number of bytes to transfer at a time
    ULONG de_Mask          ; Address Mask to block out certain memory
    LONG de_BootPri        ; Boot priority for autoboot
    ULONG de_DosType       ; ASCII (HEX) string showing filesystem type;
                          ; 0X444F5300 is old filesystem,
                          ; 0X444F5301 is fast file system

    LABEL DosEnvec_SIZEOF

```

After making a DOS device node, drivers (except for autoboot drivers) use **AddDosNode(deviceNode)** to add their node to the system. Autoboot drivers will instead use the **Exec Enqueue()** function to add a **BootNode** to the 1.3 **ExpansionBase eb\_MountList**.

## ROM BASED AND AUTOBOOT DRIVERS

The 1.3 system software supports the initialization of ROM drivers residing on expansion peripherals, including the ability for drivers to provide a DOS node which the system can boot from. This feature is known as **Autoboot**.

Automatic boot from a ROM-equipped expansion board is accomplished before DOS is initialized. This facility makes it possible to automatically boot from a hard disk without any floppy disks inserted. Likewise, it is possible to automatically boot from any device which supports the ROM protocol, even allowing the initialization of a disk operating system other than the Amiga's dos.library. ROM-based drivers contain several special entry points which are called at different stages of system initialization.

## EVENTS AT DIAG TIME

When your AUTOCONFIG hardware board is configured by the expansion initialization routine, its ExpansionRom structure is copied into the ExpansionRom subfield of a ConfigDev structure. This ConfigDev structure will be linked to the expansion.library as part of the eb\_BoardList.

After the board is configured, the er\_Type field of its ExpansionRom structure is checked. The **DIAGVALID** bit set declares that there is a valid DiagArea (a rom/diagnostic area) on this board. If there is a valid DiagArea, expansion next tests the er\_InitDiagVec vector in its copy of the ExpansionRom structure. This offset is added to the base address of the configured board; the resulting address points to the start of this board's DiagArea.

```
struct ExpansionRom
{
    UBYTE   er_Type;           /* <-- if ERTB_DIAGVALID set */
    UBYTE   er_Product;
    UBYTE   er_Flags;
    UBYTE   er_Reserved03;
    UWORD   er_Manufacturer;
    ULONG   er_SerialNumber;
    UWORD   er_InitDiagVec;    /* <-- then er_InitDiagVec      */
    UBYTE   er_Reserved0c;    /* is added to cd_BoardAddr */
    UBYTE   er_Reserved0d;    /* and points to DiagArea   */
    UBYTE   er_Reserved0e;    /* ( see figure 3 )        */
    UBYTE   er_Reserved0f;
};
```

Now expansion knows that there is a DiagArea, and knows where it is.

```
struct DiagArea
{
    UBYTE   da_Config;        /* <-- if DAC_CONFIGTIME is set */
    UBYTE   da_Flags;
    UWORD   da_Size;          /* <-- then da_Size bytes will  */
    UWORD   da_DiagPoint;     /* be copied into RAM          */
    UWORD   da_BootPoint;
    UWORD   da_Name;
    UWORD   da_Reserved01;
    UWORD   da_Reserved02;
};

/* da_Config definitions */
#define DAC_BUSWIDTH      0xC0    /* two bits for bus width */
#define DAC_NIBBLEWIDE    0x00
#define DAC_BYTEWIDE      0x40    /* invalid for 1.3 - see note below */
#define DAC_WORDWIDE      0x80

#define DAC_BOOTTIME      0x30    /* two bits for when to boot */
#define DAC_NEVER          0x00    /* obvious */
#define DAC_CONFIGTIME    0x10    /* call da_BootPoint when first
                                   configging the device */
#define DAC_BINDTIME      0x20    /* run when binding drivers to boards */
```

Next, expansion tests the first byte of the DiagArea structure to determine if the CONFIGTIME bit is set. If this bit is set, it checks the da\_BootPoint offset vector to make sure that a valid bootstrap routine exists. If so, expansion copies da\_Size bytes into RAM memory, starting at beginning of the DiagArea structure.

The copy will include the DiagArea structure itself, and typically will also include the da\_DiagPoint rom/diagnostic routine, a Resident structure (romtag), a device driver (or at least the device initialization tables or structures which need patching), and the da\_BootPoint routine. In addition, the BootNode and parameter packet for MakeDosNode may be included in the copy area for Diag-time patching. Strings such as dos and exec device names, library names,



and the romtag id string may also be included in the copy area so that both position-independent ROM code and position-independent routines in the copy area may reference them PC relative.

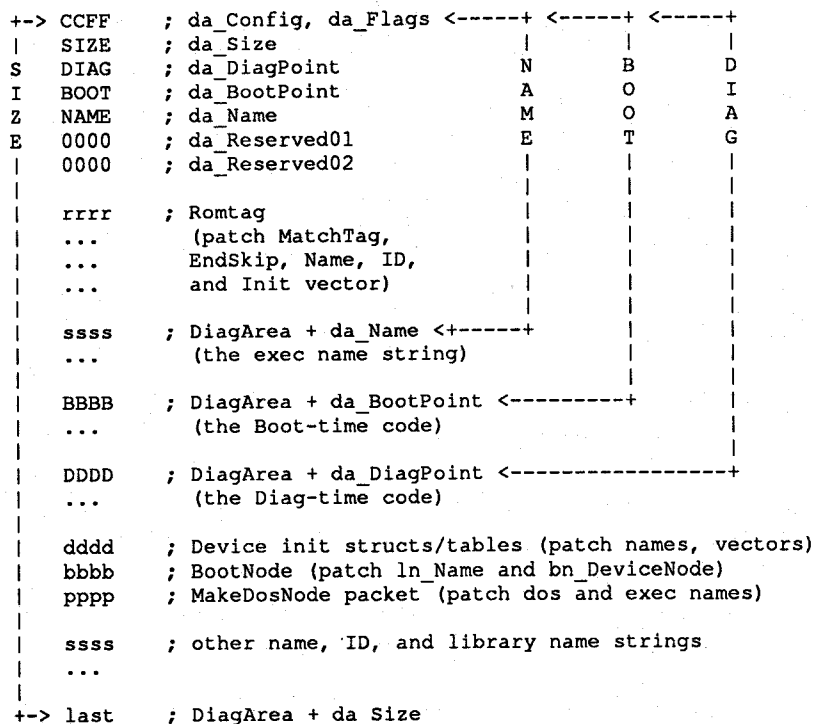
The copy will be made either nibblewise, or wordwise, according to the BUSWIDTH subfield of da\_Config.

#### NOTE

The da\_BootPoint offset MUST BE NON-NULL, OR ELSE NO COPY WILL OCCUR. (under 1.3, DAC\_BYTEWIDE is not supported. Byte wide roms must use DAC\_NIBBLEWIDE and drivers must supply additional code to re-copy their DiagArea)

The following diagram illustrates an example Diag copy area.

Example DiagArea Copy in RAM



Now the ROM "image" exists in RAM memory. Expansion stores the ULONG address of that "image" in the UBYTES er\_Reserved0c, 0d, 0e and 0f. The address is stored with the most significant byte in er\_Reserved0c, the next to most significant byte in er\_Reserved0d, the next to least significant byte in er\_Reserved0e, and the least significant byte in er\_Reserved0f - i.e. it is stored as a longword at the address er\_Reserved0c.

Expansion finally checks the da\_DiagPoint offset vector, and if valid executes the rom/diagnostic routine contained as part of the ROM "image". This diagnostic routine is responsible for "patching" the ROM image so that required absolute addresses are relocated to reflect the actual location of code and strings, as well as performing any diagnostic functions essential to the operation of its associated auto-config board. The structures which require patching are located within the copy area so that they can be patched at this time. Patching is required because many of the structures involved require absolute pointers to such things as name strings and code, but the absolute locations of the board and the RAM copy area are not known when code the structures.

The patching may be accomplished by coding pointers which require absolute addresses instead as relative offsets from either the start of the DiagArea structure, or the start of the board's ROM (depending on whether the final absolute pointer will point to a RAM or ROM location). The Diag routine is passed both the actual base address of

the board, and the address of the Diag copy area in RAM. The routine can then patch the structures in the Diag copy area by adding the appropriate address to resolve each pointer.

Example DiagArea and Diag patching routine:

```

**
** Sample autoboot code fragment
**
** These are the calling conventions for Diag or Boot area
**
** A7 -- points to at least 2K of stack
** A6 -- ExecBase
** A5 -- ExpansionBase
** A3 -- your board's ConfigDev structure
** A2 -- Base of diag/init area that was copied
** A0 -- Base of your board
**
** Your Diag routine should return a non-zero value in D0 for success.
** If this value is NULL, then the diag/init area that was copied
** will be returned to the free memory pool.
**

        INCLUDE "exec/types.i"
        INCLUDE "exec/nodes.i"
        INCLUDE "exec/resident.i"
        INCLUDE "libraries/configvars.i"

        ; LVO's resolved by linking with library amiga.lib
        XREF _LVOFindResident

VERSION    EQU    1
REVISION   EQU    4

ROMOFFS    EQU    $2000

* ROMOFFS is the offset from your board base where your ROMs appear.
* Your ROMs might appear at offset 0 and contain your autoconfig ID
* information in the high nibble of the first $40 words ($80 bytes).
* Or, your autoconfig ID information may be in a PAL, with your
* ROMs possibly being addressed at some offset (for example $2000)
* from your board base. This ROMOFFS constant will be used as an
* additional offset from your configured board address when patching
* structures which require absolute pointers to ROM code or data.

        CODE

***** RomStart *****
*****
RomStart:

* This is the start of your ROM.
* If your ROMOFFS is 0 and you have $40 words of autoconfig ID
* information here, it will look something like the this:
*
* MANUF_ID      EQU    2011    ; CBM assigned (2011 for hackers only)
* PRODUCT_ID    EQU    1      ; Manufacturer picks product ID
*
* SIZE_FLAG     EQU    3      ; Autoconfig 3-bit flag for BOARDSIZE
*                ; 0=$800000(8meg)  4=$800000(512K)
*                ; 1=$10000(64K)    5=$100000(1meg)
*                ; 2=$20000(128K)   6=$200000(2meg)
*                ; 3=$40000(256K)   7=$400000(4meg)
*
* Start of $40 words of autoconfig ID information:
*
*                ; High nibbles of first two words ($00,$02) are er_Type (not inverted)
*                DC.W    $D000    ; 11xx = normal type
*                                ; xx0x = not a memory board
*                                ; xxx1 = we have a ROM driver
*

```

```

*          DC.W    (SIZE_FLAG<<12)&$7000    ; 0xxx = not chained req
*                                           ; xNNN = size flag above
*
* ; High nibbles of next two words are er_Product
* ; These are inverted (~), as are all other words except $40 and $42
*          DC.W    (~(PRODUCT_ID<<8))&$f000
*          DC.W    (~(PRODUCT_ID<<12))&$f000
*
* etc. ($40 words at even addresses $00 through $7E)
* See the Addison-Wesley Amiga Hardware Manual for more info.

```

```

***** DiagStart *****
DiagStart: ; This is the DiagArea structure whose relative offset from
           ; your board base appears as the Init Diag vector in your
           ; autoconfig ID information. This structure is designed
           ; to use all relative pointers (no patching needed).
           dc.b    DAC_WORDWIDE+DAC_CONFIGTIME    ; da_Config
           dc.b    0                               ; da_Flags
           dc.w    EndCopy-DiagStart               ; da_Size
           dc.w    DiagEntry-DiagStart             ; da_DiagPoint
           dc.w    BootEntry-DiagStart             ; da_BootPoint
           dc.w    DevName-DiagStart               ; da_Name
           dc.w    0                               ; da_Reserved01
           dc.w    0                               ; da_Reserved02

```

```

***** Resident Structure *****
Romtag:
           dc.w    RTC_MATCHWORD    ; UWORD RT_MATCHWORD
rt_Match: dc.l    Romtag-DiagStart  ; APTR RT_MATCHTAG
rt_End:   dc.l    EndCopy-DiagStart ; APTR RT_ENDSKIP
           dc.b    RTW_COLDSTART    ; UBYTE RT_FLAGS
           dc.b    VERSION          ; UBYTE RT_VERSION
           dc.b    NT_DEVICE        ; UBYTE RT_TYPE
           dc.b    20               ; BYTE RT_PRI
rt_Name:  dc.l    DevName-DiagStart ; APTR RT_NAME
rt_Id:    dc.l    IdString-DiagStart ; APTR RT_IDSTRING
rt_Init:  dc.l    Init-RomStart     ; APTR RT_INIT

```

```

DevName:  dc.b    'abc.device',0          ; Name string
IdString  dc.b    'abc ',48+VERSION,'.',48+REVISION ; Id string

DosName:   dc.b    'dos.library',0        ; DOS library name

DosDevName: dc.b    'ABC',0              ; dos device name for MakeDosNode()
           ; (dos device will be ABC:)

           ds.w    0                    ; word align

```

```

***** DiagEntry *****
*****
* success = DiagEntry(BoardBase,DiagCopy, configDev)
* d0          a0          a2          a3
*
* Called by expansion architecture to relocate any pointers
* in the copied diagnostic area. We will patch the romtag.
* If you have pre-coded your MakeDosNode packet, BootNode,
* or device initialization structures, they would also need
* to be within this copy area, and patched by this routine.
*
*****

```

```

DiagEntry: lea     patchTable-RomStart(a0),a1    ; find patch table
           adda.l  #ROMOFFS,a1                  ; adjusting for ROMOFFS

```

```

* Patch relative pointers to labels within DiagCopy area
* by adding Diag RAM copy address. These pointers were coded as
* long relative offsets from base of the DiagArea structure.
*

```

```

dpatches:
    move.l    a2,d1            ;d1=base of ram Diag copy
dloop:
    move.w    (a1)+,d0         ;d0=word offs. into Diag needing patch
    bmi.s     bpatches        ;-1 is end of word patch offset table
    add.l     d1,0(a2,d0.w)    ;add DiagCopy addr to coded rel. offset
    bra.s     dloop

* Patches relative pointers to labels within the ROM by adding
* the board base address + ROMOFFS. These pointers were coded as
* long relative offsets from RomStart.
bpatches:
    move.l    a0,d1            ;d1 = board base address
    add.l     #ROMOFFS,d1      ;add offset to where your ROMs are
rloop:
    move.w    (a1)+,d0         ;d0=word offs. into Diag needing patch
    bmi.s     endpatches      ;-1 is end of patch offset table
    add.l     d1,0(a2,d0.w)    ;add ROM address to coded relative offset
    bra.s     rloop

endpatches:
    moveq.l   #1,d0            ; indicate "success"
    rts

***** BootEntry *****
*****

BootEntry:  lea     DosName(PC),a1        ; 'dos.library',0
            jsr     _LVOFindResident(a6)  ; find the DOS resident tag
            move.l  d0,a0                ; in order to bootstrap
            move.l  RT_INIT(A0),a0        ; set vector to DOS INIT
            jsr     (a0)                  ; and initialize DOS
            rts

*
* End of the Diag copy area which is copied to RAM
*
EndCopy:
*****

*****
*
* Beginning of rom driver code and data that is accessed only in
* the rom space. This must all be position-independent.
*

patchTable:
* Word offsets into Diag area where pointers need Diag copy address added
    dc.w     rt_Match-DiagStart
    dc.w     rt_End-DiagStart
    dc.w     rt_Name-DiagStart
    dc.w     rt_Id-DiagStart
    dc.w     -1

* Word offsets into Diag area where pointers need boardbase+ROMOFFS added
    dc.w     rt_Init-DiagStart
    dc.w     -1

***** Romtag InitEntry *****
*****

Init:
    ; After Diag patching, our romtag will point to this
    ; routine in ROM so that it can be called at Resident
    ; initialization time.
    ; This routine will be similar to a normal expansion device
    ; initialization routine, but will MakeDosNode then set up a
    ; BootNode, and Enqueue() on eb_MountList.
    ;
    rts

    ; Rest of your position-independent device code goes here.

```

END

Your `da_DiagPoint` rom/diagnostic routine should return a non-zero value to indicate success; otherwise the ROM "image" will be unloaded from memory, and its address will be replaced with NULL bytes in locations `er_Reserved0c`, `0d`, `0e` and `0f`.

Now that the ROM "image" has been copied into RAM, validated, and linked to board's `ConfigDev` structure, the expansion module is free to configure all other boards on the `eb_BoardList`.

## EVENTS AT ROMTAG INIT TIME

Next, most resident system modules (for example graphics) are initialized. As part of the system initialization procedure a search is made of the `eb_BoardList` (which contains a `ConfigDev` structure for each of the auto-config hardware boards). If the `cd_Flags` specify `CONFIGME` and the `er_Type` specifies `DIAGVALID`, the system initialization will do three things:

First, it will bind the address of the current `ConfigDev` to the `eb_CurrentBinding` structure (see `expansion.library/SetCurrentBinding`). Second, it will check the `DiagArea`'s `da_Config` flag to make sure that the `CONFIGTIME` bit is set. Third, it will search the ROM "image" associated with this hardware board for a valid Resident structure (`exec/resident.h`); and, if one is located, will call `InitResident()` on it, passing a NULL segment list pointer as part of the call.

Next, the board's device driver is initialized. The Resident structure associated with this board's device driver (which has now been "patched" by the rom/diagnostic routine) should follow standard system conventions in initializing the device driver provided in the bootroms. This driver should obtain the address of its associated `ConfigDev` structure via `GetCurrentBinding()`.

Once the driver is initialized, it is responsible for some further steps. It must clear the `CONFIGME` bit in the `cd_Flags` of its `ConfigDev` structure, so that the system knows not to configure this device again if `binddrivers` is run after bootstrap. Also, though it is not currently mandatory, the driver should place a pointer to its Exec node in the `cd_Driver` field of the `ConfigDev` structure. This will generally be a device (`NT_DEVICE`) node. And for this device to be bootable, the driver must create a `BootNode` structure, and link this `BootNode` onto the expansion `eb_MountList`.

The `BootNode` structure (see `libraries/romboot_base.h`) contains a Node of the "new" type `NT_BOOTNODE` (see `exec/nodes.h`). The driver MUST initialize the `bn_DeviceNode` field to point to the `ConfigDev` structure which it has obtained via the `GetCurrentBinding()` call. The `bn_Flags` subfield is currently unused and should be initialized to NULL.

When the DOS is initialized later, it will attempt to boot from the first `BootNode` on the `eb_MountList`. The `eb_MountList` is a priority sorted List, with nodes of the highest priority at the head of the List. For this reason, the device driver must `Enqueue()` a `BootNode` onto the List using the `exec.library/Enqueue`.

In the case of an autoboot of AmigaDOS, the `BootNode` must be linked to a `DeviceNode` of the AmigaDOS type (see `libraries/filehandler.h`), which the driver can create via the `expansion.library/MakeDosNode` call. When the DOS "wakes up", it will attempt to boot from this `DeviceNode`.

## EVENTS AT BOOT TIME

If there is no boot disk in the internal floppy drive, the system strap module will call a routine to perform autoboot. It will examine the `eb_MountList`; find the highest priority `BootNode` structure at the head of the List; validate the `BootNode`; determine which `ConfigDev` on the `eb_BootList` is associated with this `BootNode`; find its `DiagArea`; and call its `da_BootPoint` function in the ROM "image" to bootstrap the appropriate DOS. Generally, the `BootPoint` code of a ROM driver will perform the same function as the boot code installed on a floppy disk, ie. it will `FindResident()` the `dos.library`, and jump to its `RT_INIT` vector. The `da_BootPoint` call, if successful, should not return.

If a boot disk IS in the internal floppy drive, the strap will `Enqueue()` a `BootNode` on the `eb_MountList` for `DF0:` at the "suggested" priority (see autodoc for `expansion.library/AddDosNode`). Strap will then open AmigaDOS, overriding the autoboot. AmigaDOS will boot from the highest priority node on the `eb_MountList` which should, in this case, be `DF0:`. Thus, games and other "bootable" floppy disks will still be able to obtain the system for their own use.

In the event that there is no boot disk in the internal floppy drive AND there are no ROM bootable devices on the auto-configuration chain, the system does the normal thing, asking the user to insert a WorkBench disk, and waiting until its request is satisfied before proceeding.

## RigidDiskBlock and Alternate Filesystems

Through the use of `RigidDiskBlock` information and the `FileSysResource`, it is possible for an autoboot driver to have access to enough information to mount all of its device partitions and even load alternate filesystems for use with these partitions.

The `RigidDiskBlock` specification (also known as "hardblocks") defines blocks of data that exist on a hard disk to describe that disk. These blocks are created or modified with an installation (or "prep") utility provided by the disk controller manufacturer, and they are read and used by the device driver rom (or expansion) code. They are not generally accessible to the user as they do not appear on any DOS device. The blocks are tagged with a unique identifier, checksummed, and linked together.

The five block types currently defined are `RigidDiskBlock`, `BadBlockBlock`, `PartitionBlock`, `FileSysHeaderBlock`, and `LoadSegBlock`.

The root of these blocks is the `RigidDiskBlock`. The `RigidDiskBlock` must exist on the disk within the first `RDB_LOCATION_LIMIT` blocks. This inhibits the use of the first cylinder(s) in an AmigaDOS partition: although it is strictly possible to store the `RigidDiskBlock` data in the reserved area of a partition, this practice is discouraged since the reserved blocks of a partition are overwritten by `Format`, `Install`, `DiskCopy`, etc. The recommended disk layout, then, is to use the first cylinder(s) to store all the drive data specified by these blocks: i.e. partition descriptions, file system load images, drive bad block maps, spare blocks, etc. This allocation range is described in the `RigidDiskBlock`.

The `RigidDiskBlock` contains basic information about the configuration of the drive: number and size of blocks, tracks, and cylinders, as well as other relevant information. The `RigidDiskBlock` points to bad block, partition, file system and drive initialization description blocks.

The bad block list contains a series of bad-block/good-block pairs. Each block contains as many as will fit in a physical sector on the drive. These mappings are to be handled by the driver on read and write requests.

The drive initialization description blocks are LoadSegBlock blocks that are loaded at boot time to perform drive-specific initialization. They are called with both "C" style parameters on the stack, and assembler parameters in registers as follows:

```
d0 = DriveInit(lun, rdb, ior)(d0/a0/a1)
```

where lun is the SCSI logical unit number (needed to construct SCSI commands), rdb is a pointer to a memory copy of the RigidDiskBlock (which should not be altered), and ior is a standard IO request block that can be used to access the drive with synchronous DoIO() calls. The result of DriveInit is either -1, 0, or 1. A -1 signifies that an error occurred and drive initialization cannot continue. A 0 (zero) result reports success. In cases -1 and 0, the code is unloaded. A result of 1 reports success, and causes the code to be kept loaded. Furthermore, this resident code will be called whenever a reset is detected on the SCSI bus.

The FileSysHeaderBlock entries contain code for alternate file handlers to be used by partitions. There are several strategies that can be used to determine which of them to load. The most robust would scan all drives for those that are both required by partitions and have the highest fhb\_Version, and load those. Whatever method is used, the loaded file handlers are added to the exec resource FileSystem.resource, where they are used as needed to mount disk partitions.

The PartitionBlock entries contains most of the data necessary to add each partition to the system. They replace the Mount and DEVS:MountList mechanism for adding these partitions. The only items required by the expansion.library MakeDosNode function which are not in this partition block are the exec device name and unit, which is expected to be known by driver reading this information. The file system to be used is specified in the pb\_Environment. If it is not the default file system (i.e. specified in a FileSystem.resource's FileSysEntry before adding it to the dos list.

Though only 512 byte blocks are currently supported by the file system, this proposal tries to be forward-looking by making the block size explicit, and by using only the first 256 bytes for all blocks but the LoadSeg and BadBlock data. This would allow using drives formatted with sectors 256 bytes or larger. LoadSeg and BadBlock data use whatever space is available in a sector.

## RigidDiskBlock

This is the current specification for the RigidDiskBlock:

```
rdb_ID          == 'RDSK'
rdb_SummedLongs == 64
rdb_ChkSum      block checksum (longword sum to zero)

rdb_HostID      SCSI Target ID of host
                This is the initiator ID of the creator of this
                RigidDiskBlock. It is intended that
                modification of the RigidDiskBlock, or of any
                of the blocks pointed to by it, by another
                initiator (other than the one specified here)
                be allowed only after a suitable warning. The
                user is then expected to perform an audio
                lock out (''Hey, is anyone else setting up SCSI
                stuff on this bus?''). The rdb_HostID may
                become something other than the initiator ID
                when connected to a real network: that is an
                area for future standardization.

rdb_BlockBytes  size of disk blocks
```

Currently this must be 512 for a disk with any AmigaDOS partitions on it.

<b>rdb_Flags</b>	longword of flags:
<b>RDBF._LAST</b>	no disks exist to be configured after this one on this controller (SCSI bus).
<b>RDBF._LASTLUN</b>	no LUNs exist to be configured greater than this one at this SCSI Target ID
<b>RDBF._LASTTID</b>	no Target IDs exist to be configured greater than this one on this SCSI bus
<b>RDBF._NORESELECT</b>	don't bother trying to perform reselection when talking to this drive
<b>RDBF._DISKID</b>	rdb_Disk... identification variables below contain valid data.
<b>RDBF._CTRLRID</b>	rdb_Controller... identification variables below contain valid data.
<b>RDBF._SYNCH</b>	drive supports scsi synchronous mode CAN BE DANGEROUS TO USE IF IT DOESN'T!

These fields point to other blocks on the disk which are not a part of any filesystem. All block pointers referred to are block numbers on the drive.

<b>rdb_BadBlockList</b>	optional bad block list A singly linked list of blocks of type PartitionBlock
<b>rdb_PartitionList</b>	optional first partition block A singly linked list of blocks of type PartitionBlock
<b>rdb_FileSysHeaderList</b>	optional file system header block A singly linked list of blocks of type FileSysHeaderBlock
<b>rdb_DriveInit</b>	optional drive-specific init code A singly linked list of blocks of type LoadSegBlock containing initialization code. Called as DriveInit(lun,rdb,ior) (d0/a0/a1).
<b>rdb_Reserved1</b>	set to \$ffffffff These are reserved for future block lists. Since NULL for block lists is \$ffffffff, these reserved entries must be set to \$ffffffff.

These fields describe the physical layout of the drive.

<b>rdb_Cylinders</b>	number of drive cylinders
<b>rdb_Sectors</b>	sectors per track
<b>rdb_Heads</b>	number of drive heads
<b>rdb_Interleave</b>	interleave This drive interleave is independent from, and unknown to, the DOS's understanding of interleave as set in the partition's environment vector.
<b>rdb_Park</b>	landing zone cylinder
<b>rdb_Reserved2</b>	set to zeros



These fields are intended for ST506 disks. They are generally unused for SCSI devices and set to zero.

rdb_WritePreComp	starting cylinder: write precompensation
rdb_ReducedWrite	starting cylinder: reduced write current
rdb_StepRate	drive step rate
rdb_Reserved3	set to zeros

These fields are used while partitions are set up to constrain the partitionable area and help describe the relationship between the drive's logical and physical layout.

rdb_RDBlocksLo	low block of the range allocated for blocks described here. Replacement blocks for bad blocks may also live in this range.
rdb_RDBlocksHi	high block of this range (inclusive)
rdb_LoCylinder	low cylinder of partitionable disk area Blocks described by this include file will generally be found in cylinders below this one.
rdb_HiCylinder	high cylinder of partitionable data area Usually rdb_Cylinders-1.
rdb_CylBlocks	number of blocks available per cylinder This may be rdb_Sectors*rdb_Heads, but a SCSI disk that, for example, reserves one block per cylinder for bad block mapping would use rdb_Sectors*rdb_Heads-1.
rdb_AutoParkSeconds	number of seconds to wait before parking drive heads automatically. If zero, this feature is not desired.
rdb_HighRDSKBlock	highest block used by these drive definitions Must be less than or equal to rdb_RDBlocksHi. All replacements for bad blocks should be between rdb_HighRDSKBlock+1 and rdb_RDBlocksHi (inclusive).
rdb_Reserved4	set to zeros

These fields are of the form available from a SCSI Identify command. Their purpose is to help the user identify the disk during setup. Entries exist for both controller and disk for non-embedded SCSI disks.

rdb_DiskVendor	vendor name of the disk
rdb_DiskProduct	product name of the disk
rdb_DiskRevision	revision code of the disk
rdb_ControllerVendor	vendor name of the disk controller
rdb_ControllerProduct	product name of the disk controller
rdb_ControllerRevision	revision code of the disk controller
rdb_Reserved5	set to zeros

## BadBlockBlock

This is the current specification for the BadBlockBlock. The end of data occurs when bbb\_Next is null (\$ffffff), and the summed data is exhausted. Note that null for RigidDiskBlocks is equal to \$ffffff.

bbb_ID	== 'BADB'
bbb_SummedLongs	size of this checksummed structure Note that this is not 64 like most of the other structures. This is the number of valid longs in this image, and can be from 6 to rdb_BlockBytes/4. The latter is the best size for all blocks other than the last one.
bbb_ChkSum	block checksum (longword sum to zero)
bbb_HostID	SCSI Target ID of host This describes the initiator ID for the creator of these blocks. (see rdb_HostID discussion)
bbb_Next	block number of the next BadBlockBlock
bbb_Reserved	set to zeros
bbb_BlockPairs	pairs of block remapping information The data starts here and continues as long as indicated by bbb_SummedLongs-6: e.g. if bbb_SummedLongs is 128 (512 bytes), 61 pairs are described here.

## PartitionBlock

This is the current specification for the PartitionBlock.

### NOTE

While reading these blocks you may encounter partitions that are not to be mounted because the pb\_HostID does not match, or because the pb\_DriveName is in use and no fallback strategy exists, or because PBF.\_NOMOUNT is set, or because the proper filesystem cannot be found. Some partitions may be mounted but not be bootable because PBF.\_BOOTABLE is not set.

pb_ID	== 'PART'
pb_SummedLongs	== 64
pb_ChkSum	block checksum (longword sum to zero)
pb_HostID	SCSI Target ID of host This describes the initiator ID for the owner of this partition. (see rdb_HostID discussion)
pb_Next	block number of the next PartitionBlock
pb_Flags	see below for defines
PBF._BOOTABLE	this partition is intended to be bootable (e.g. expected directories and files exist)
PBF._NOMOUNT	this partition description is to reserve space on the disk without mounting it. It may be manually mounted later.
pb_Reserved1	set to zeros
pb_DevFlags	preferred flags for OpenDevice
pb_DriveName	preferred DOS device name: BSTR form This name is not to be used if it is already in use.

### NOTE

pb\_Reserved2 will always be at least 4 longwords so that the ram image of this record may be converted to the parameter packet to the expansion.library function MakeDosNode

pb\_Reserved2

filler to 32 longwords

The specification of the location of the partition is one of the components of the environment, below. If possible, describe the partition in a manner that tells the DOS about the physical layout of the partition: specifically, where the cylinder boundaries are. This allows the filesystem's smart block allocation strategy to work.

pb_Environment	environment vector for this partition
de_TableSize	size of Environment vector
de_SizeBlock	== 128 (for 512 bytes/logical block)
de_SecOrg	== 0
de_Surfaces	number of heads (see layout discussion)
de_SectorPerBlock	== 1
de_BlocksPerTrack	blocks per track (see layout discussion)
de_Reserved	DOS reserved blocks at start of partition. Must be >= 1. 2 is recommended.
de_PreAlloc	DOS reserved blocks at end of partition Valid only for filesystem type DOS^A (the fast file system). Zero otherwise.
de_Interleave	DOS interleave Valid only for filesystem type DOS^@ (the old file system). Zero otherwise.
de_LowCyl	starting cylinder
de_HighCyl	max cylinder
de_NumBuffers	initial # DOS of buffers.
de_BufMemType	type of mem to allocate for buffers The second argument to AllocMem().
de_MaxTransfer	max number of bytes to transfer at a time. Drivers should be written to handle requests of any length.
de_Mask	address mask to block out certain memory Normally \$00ffffff for DMA devices.
de_BootPri	Boot priority for autoboot Suggested value: zero. Keep less than five, so it won't override a boot floppy.
de_DosType	ASCII string showing filesystem type; DOS^@ (\$444F5300) is old filesystem, DOS^A (\$444F5301) is fast file system. UNI<anything> is a Unix partition.
pb_EReserved	reserved for future environment vector

## FileSysHeaderBlock

The current specification for the FileSysHeaderBlock follows.

fhb_ID	== 'FSHD'
fhb_SummedLongs	== 64
fhb_ChkSum	block checksum (longword sum to zero)
fhb_HostID	SCSI Target ID of host This describes the initiator ID for the creator of this block. (see rdb_HostID discussion)
fhb_Next	block number of next FileSysHeaderBlock
fhb_Flags	see below for defines

fhb\_Reserved1                    set to zero

The following information is used to construct a FileSysEntry node in the FileSystem.resource.

fhb_DosType	file system description This is matched with a partition environment's de_DosType entry.
fhb_Version	release version of this load image Usually MSW is version, LSW is revision.
fhb_PatchFlags	patch flags These are bits set for those of the following that need to be substituted into a standard device node for this file system, lsb first: e.g. 0x180 to substitute SegList & GlobalVec
fhb_Type	device node type: zero
fhb_Task	standard dos 'task' field: zero
fhb_Lock	not used for devices: zero
fhb_Handler	filename to loadseg: zero placeholder
fhb_StackSize	stacksize to use when starting task
fhb_Priority	task priority when starting task
fhb_Startup	startup msg: zero placeholder
fhb_SegListBlocks	first of linked list of LoadSegBlocks: Note that if the fhb_PatchFlags bit for this entry is set (bit 7), the blocks pointed to by this entry must be LoadSeg'd and the resulting BPTR put in the FileSysEntry node.
fhb_GlobalVec	BCPL global vector when starting task Zero or -1.
fhb_Reserved2	(those reserved by PatchFlags)
fhb_Reserved3	set to zero

## LoadSegBlock

This is the current specification of the LoadSegBlock. The end of data occurs when lsb\_Next is null (\$ffffff), and the summed data is exhausted. Note that null for RigidDiskBlocks is equal to \$ffffff.

lsb_ID	== 'LSEG'
lsb_SummedLongs	size of this checksummed structure Note that this is not 64 like most of the other structures. This is the number of valid longs in this image, like bbb_SummedLongs.
lsb_ChkSum	block checksum (longword sum to zero)
lsb_HostID	SCSI Target ID of host This describes the initiator ID for the creator of these blocks. (see rdb_HostID discussion)
lsb_Next	block number of the next LoadSegBlock
lsb_LoadData	data for 'loadseg' The data starts here and continues as long as indicated by lsb_SummedLongs-5: e.g. if lsb_SummedLongs is 128 (512 bytes), 123 longs of data are valid here.

## filesysres.h and i

The FileSysResource is created by the first code that needs to use it. It is added to the resource list for others to use. (Checking and creation should be performed while Forbid()).

### FileSysResource

fsr_Node	on resource list with the name FileSystem.resource
fsr_Creator	name of creator of this resource
fsr_FileSysEntries	list of FileSysEntry structs

### FileSysEntry

fse_Node	on fsr_FileSysEntries list ln_Name is of creator of this entry
fse_DosType	DosType of this FileSys
fse_Version	release version of this FileSys Usually MSW is version, LSW is revision.
fse_PatchFlags	bits set for those of the following that need to be substituted into a standard device node for this file system: e.g. \$180 for substitute SegList & GlobalVec
fse_Type	device node type: zero
fse_Task	standard dos ``task'' field
fse_Lock	not used for devices: zero
fse_Handler	filename to loadseg (if SegList is null)
fse_StackSize	stacksize to use when starting task
fse_Priority	task priority when starting task
fse_Startup	startup msg: FileSysStartupMsg for disks
fse_SegList	segment of code to run to start new task
fse_GlobalVec	BCPL global vector when starting task

No more entries need exist than those implied by fse\_PatchFlags, so entries do not have a fixed size.

### NOTE

Also see the following include files (.h and .i) for additional notes and related structures: *devices/hardblocks*, *resources/filesysres*, and *libraries/filehandler*.

## Chapter 28

# Math Libraries

This chapter describes the structure and calling sequences required to access the Motorola Fast Floating Point and IEEE Double-Precision math libraries via the Amiga-supplied interfaces.

### Introduction

In its present state, the FFP library consists of three separate entities: the basic math library, the transcendental math library, and C and assembly-language interfaces to the basic math library plus FFP conversion functions. The IEEE Double-Precision library presently consists of two entities: the basic math library and the transcendental math library.

#### NOTE

Each **Task** using the an IEEE math library must open the library itself. Library base pointers to these libraries may *not* be shared, as the libraries are context sensitive and use the **Task** structure to keep track of the current context. This is for any of the IEEE math libraries.

## FFP Floating Point Data Format

FFP floating-point variables are defined within C by the float or FLOAT directive. In assembly language they are simply defined by a DC.L/DS.L statement. All FFP floating-point variables are defined as 32-bit entities (longwords) with the following format:

MMMMMMMM	MMMMMMMM	MMMMMMMM	SEEEEEEE
31	23	15	7

where

M = 24-bit mantissa

S = Sign of FFP number

E = Exponent in excess-64 notation

The mantissa is considered to be a binary fixed-point fraction; except for 0, it is always normalized (has a 1 bit in its highest position). Thus, it represents a value of less than 1 but greater than or equal to 1/2.

The sign bit is reset (0) for a positive value and set (1) for a negative value.

The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. It is held in excess-64 notation, which means that the two's-complement values are adjusted upward by 64, thus changing \$40 (-64) through \$3F (+63) to \$00 through \$7F. This facilitates comparisons among floating-point values.

The value of 0 is defined as all 32 bits being 0's. The sign, exponent, and mantissa are entirely cleared. Thus, 0's are always treated as positive.

The range allowed by this format is as follows:

DECIMAL:

$$9.22337177 * 10^{18} > +VALUE > 5.42101070 * 10^{-20}$$

$$-9.22337177 * 10^{18} < -VALUE < -2.71050535 * 10^{-20}$$

BINARY (HEXADECIMAL):

$$FFFFFF * 2^{63} > +VALUE > .800000 * 2^{-63}$$

$$-FFFFFF * 2^{63} < -VALUE < -.800000 * 2^{-64}$$

Remember that you cannot perform *any* arithmetic on these variables without using the fast floating-point libraries. The formats of the variables are *incompatible* with the arithmetic format of C-generated code; hence, all floating-point operations are performed through function calls.

## FFP Basic Mathematics Library

The FFP basic math library resides in ROM and is opened by making a call to the `OpenLibrary()` function with `mathffp.library` as the argument. In C, this might be implemented as shown below:

```
#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;

VOID main()
{
    if((MathBase = OpenLibrary("mathffp.library", 0)) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }
    .
    .
    .
    CloseLibrary(MathBase);
}
```

The global variable `MathBase` is used internally for all future library references.

This library contains entries for the basic mathematics functions such as add, subtract, and so on. The C-called entry points are accessed via code generated by the C compiler when standard numerical operators are given within the source code, *not* by user calls. Note that to use either the C or assembly language interfaces to the basic math library all user code must be linked with the library *amiga.lib*. The C entry points defined for the basic math functions are as follows:

### SPFix (ffixi)

Convert FFP variable to integer

Usage: `i1 = SPFix(f1); (i1 = (LONG)f1;)`

### SPFlt (fflti) Convert integer variable to FFP

Usage: `f1 = SPFlt(i1); (f1 = (FLOAT)i1;)`

### SPCmp (fcmpi)

Compare two FFP variables

Usage: `if (SPCmp(f1, f2)) {}; (if (f1 <> f2) {});`

### SPTst (fsti)

Test an FFP variable against zero

Usage: `if (!(SPTst(f1))) {}; (if (!f1) {});`

### SPAbs (fabsi)

Take absolute value of FFP variable

Usage: `f1 = SPAbs(f2); (f1 = abs(f2);)`



**SPNeg (fnegi)**

Take two's complement of FFP variable

Usage: `f1 = SPNeg(f2); (f1 = -f2;)`**SPAdd (faddi)**

Add two FFP variables

Usage: `f1 = SPAdd(f2, f3); (f1 = f2 + f3;)`**SPSub (fsubi)**

Subtract two FFP variables

Usage: `f1 = SPSub(f2, f3); (f1 = f2 - f3;)`**SPMul (fmuli)**

Multiply two FFP variables

Usage: `f1 = SPMul(f2, f3); (f1 = f2 * f3;)`**SPDiv (fdivi)**

Divide two FFP variables

Usage: `f1 = SPDiv(f2, f3); (f1 = f2 / f3;)`**SPCeil** Compute least integer greater than or equal to variable.Usage: `f1 = SPCeil(f2);`**SPFloor** Computer largest integer less than or equal to variable.Usage: `f1 = SPFloor(f2);`

Be sure to include proper data type definitions as shown in the example below.

```

#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;

VOID main()
{
    FLOAT f1, f2, f3;
    LONG i1;

    if ((MathBase = OpenLibrary("mathffp.library", 0) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }

    i1 = SPFix(f1);           /* Call SPFix entry */
    f1 = SPFlt(i1);           /* Call SPFlt entry */

    if (SPCmp(f1, f2)) {};    /* Call SPCmp entry */
    if (!(SPTst(f1))) {};     /* Call SPTst entry */

    f1 = SPAbs(f2);           /* Call SPAbs entry */
    f1 = SPNeg(f2);           /* Call SPNeg entry */
    f1 = SPAdd(f2, f3);       /* Call SPAdd entry */
    f1 = SPSub(f2, f3);       /* Call SPSub entry */
    f1 = SPMul(f2, f3);       /* Call SPMul entry */
    f1 = SPDiv(f2, f3);       /* Call SPDiv entry */
    f1 = SPCeil(f2);          /* Call SPCeil entry */
    f1 = SPFloor(f2);         /* Call SPFloor entry */

```

```

    CloseLibrary(MathBase);
}

```

The Amiga assembly language interface to the Motorola Fast Floating Point basic math routines is shown below, including some details about how the system flags are affected by each operation. This interface resides in *amiga.lib* and must be linked with the user code. Note that the access mechanism from assembly language is as follows:

```

MOVEA.L _MathBase,A6
JSR _LVOSPFix(A6)

```

#### **\_LVOSPFix - Convert FFP to integer**

Inputs:	D0 = FFP argument
Outputs:	D0 = Integer (two's complement) result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 1 if overflow occurred
	C = undefined
	X = undefined

#### **\_LVOSPFlt - Convert integer to FFP**

Inputs:	D0 = Integer (two's complement) argument
Outputs:	D0 = FFP result
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined

#### **\_LVOSPCmp - Compare**

Inputs:	D1 = FFP argument 1
	D0 = FFP argument 2
Outputs:	D0 = +1 if arg1 < arg2
	D0 = -1 if arg1 > arg2
	D0 = 0 if arg1 = arg2
Condition codes:	N = 0
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	GT = arg2 > arg1
	GE = arg2 >= arg1
	EQ = arg2 = arg1
	NE = arg2 != arg1
	LT = arg2 < arg1
	LE = arg2 <= arg1

#### **\_LVOSPSt - Test**

Inputs:	D1 = FFP argument
---------	-------------------

Outputs:	D0 = +1 if arg > 0.0 D0 = -1 if arg < 0.0 D0 = 0 if arg = 0.0
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined EQ = arg = 0.0 NE = arg <> 0.0 PL = arg >= 0.0 MI = arg < 0.0

*Note:* This routine trashes the argument in D1.

#### **\_LVOSPabs -**

**Absolute value**

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP absolute value result
Condition codes:	N = 0 Z = 1 if result is zero V = 0 C = undefined X = undefined

#### **\_LVOSPNeg -**

**Negate**

Inputs:	D0 = FFP argument
Outputs:	D0 = FFP negated result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined

#### **\_LVOSPAAdd -**

**Addition**

Inputs:	D1 = FFP argument 1 D0 = FFP argument 2
Outputs:	D0 = FFP addition of arg1+arg2 result
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

#### **\_LVOSPSub -**

**Subtraction**

Inputs:	D1 = FFP argument 1 D0 = FFP argument 2
---------	--

Outputs:  
 Condition codes:

D0 = FFP subtraction of arg2-arg1 result  
 N = 1 if result is negative  
 Z = 1 if result is zero  
 V = 1 if result overflowed  
 C = undefined  
 Z = undefined

\_LVOSPMul - Multiply

Inputs:

Outputs:  
 Condition codes:

D0 = FFP argument 1  
 D1 = FFP argument 2  
 D0 = FFP multiplication of arg1\*arg2 result  
 N = 1 if result is negative  
 Z = 1 if result is zero  
 V = 1 if result overflowed  
 C = undefined  
 Z = undefined

\_LVOSPDIV - Divide

Inputs:

Outputs:  
 Condition codes:

D1 = FFP argument 1  
 D0 = FFP argument 2  
 D0 = FFP division of arg2/arg1 result  
 N = 1 if result is negative  
 Z = 1 if result is zero  
 V = 1 if result overflowed  
 C = undefined  
 Z = undefined

\_LVOSPCeil Ceil

Inputs:  
 Outputs:  
 Condition codes:

D0 = FFP argument  
 D0 = least integer  $\geq$  argument  
 N = 1 if result is negative  
 Z = 1 if result is zero  
 V = undefined  
 C = undefined  
 Z = undefined

\_LVOSPFloor Floor

Inputs:  
 Outputs:  
 Condition codes:

D0 = FFP argument  
 D0 = largest integer  $\leq$  argument  
 N = 1 if result is negative  
 Z = 1 if result is zero  
 V = undefined  
 C = undefined  
 Z = undefined

# FFP Transcendental Mathematics Library

The FFP transcendental math library resides on disk and must be accessed in the same way as the basic math library after it is loaded into system RAM. The name to be included in the `OpenLibrary()` function is `mathtrans.library`. In C, this might be implemented as follows:

```
#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;
struct Library *MathTransBase;

VOID main()
{
    if((MathBase = OpenLibrary("mathffp.library", 0)) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }
    if((MathTransBase = OpenLibrary("mathtrans.library",0)) == 0)
    {
        printf("Can't open mathtrans.library\n");
        CloseLibrary(MathBase);
        exit(21);
    }
    .
    .
    .

    CloseLibrary(MathTransBase);
    CloseLibrary(MathBase);
}
```

The global variable `MathTransBase` is used internally for all future library references. Note that the transcendental math library is dependent upon the basic math library, which it will open if it is not open already. If you want to use the basic math functions in conjunction with the transcendental math functions however, you have to specifically open the basic math library yourself.

This library contains entries for the transcendental math functions sine, cosine, and so on. The C-called entry points are accessed via code generated by the C compiler when the actual function names are given within the source code. The C entry points defined for the transcendental math functions are as follows:

**SPAsin**     Return arcsine of FFP variable.

Usage: `f1 = SPAsin(f2);`

**SPAcos**     Return arccosine of FFP variable.

Usage: `f1 = SPAcos(f2);`

**SPAtan**     Return arctangent of FFP variable.

Usage: `f1 = SPAtan(f2);`

**SPSin**     Return sine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric sine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made.

Usage: `f1 = SPSin(f2);`

<b>SPCos</b>	Return cosine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric cosine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made. Usage: <code>f1 = SPCos(f2);</code>
<b>SPTan</b>	Return tangent of FFP variable. This function accepts an FFP radian argument and returns the trigonometric tangent value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made. Usage: <code>f1 = SPTan(f2);</code>
<b>SPSincos</b>	Return sine and cosine of FFP variable. This function accepts an FFP radian argument and returns both the trigonometric sine and cosine values. If both the sine and cosine are required for a single radian value of interest, this function will result in almost twice the execution speed of calling the sin and cos functions independently. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made. Usage: <code>f1 = SPSincos(&amp;f3, f2);</code>
<b>SPSinh</b>	Return hyperbolic sine of FFP variable. Usage: <code>f1 = SPSinh(f2);</code>
<b>SPCosh</b>	Return hyperbolic cosine of FFP variable. Usage: <code>f1 = SPCosh(f2);</code>
<b>SPTanh</b>	Return hyperbolic tangent of FFP variable. Usage: <code>f1 = SPTanh(f2);</code>
<b>SPExp</b>	Return $e$ to the FFP variable power. This function accepts an FFP argument and returns the result representing the value of $e$ (2.71828...) raised to that power. Usage: <code>f1 = SPExp(f2);</code>
<b>SPLog</b>	Return natural log (base $e$ ) of FFP variable. Usage: <code>f1 = SPLog(f2);</code>
<b>SPLog10</b>	Return naparian log (base 10) of FFP variable. Usage: <code>f1 = SPLog10(f2);</code>
<b>SPPow</b>	Return FFP arg2 to FFP arg1. Usage: <code>f1 = SPPow(f3, f2);</code>
<b>SPSqrt</b>	Return square root of FFP variable. Usage: <code>f1 = SPSqrt(f2);</code>
<b>SPTieee</b>	Convert FFP variable to IEEE format Usage: <code>i1 = SPTieee(f1);</code>

**SPFieee** Convert IEEE variable to FFP format.

Usage: `f1 = SPFieee(i1);`

Be sure to include proper data type definitions, as shown in the example below.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;
struct Library *MathTransBase;

VOID main()
{
    FLOAT f1, f2, f3;
    FLOAT i1;

    if((MathBase = OpenLibrary("mathffp.library",0)) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }

    if((MathTransBase = OpenLibrary("mathtrans.library",0)) == 0)
    {
        printf("Can't open mathtrans.library\n");
        CloseLibrary(MathBase);
        exit(21);
    }

    f1 = SPAsin(f2);          /* Call SPAsin entry */
    f1 = SPACos(f2);          /* Call SPACos entry */
    f1 = SPAtan(f2);          /* Call SPAtan entry */

    f1 = SPSin(f2);           /* Call SPSin entry */
    f1 = SPCos(f2);           /* Call SPCos entry */
    f1 = SPTan(f2);           /* Call SPTan entry */
    f1 = SPSincos(&f3, f2);   /* Call SPSincos entry */

    f1 = SPSinh(f2);          /* Call SPSinh entry */
    f1 = SPCosh(f2);          /* Call SPCosh entry */
    f1 = SPTanh(f2);          /* Call SPTanh entry */

    f1 = SPExp(f2);           /* Call SPExp entry */
    f1 = SPLog(f2);           /* Call SPLog entry */
    f1 = SPLog10(f2);         /* Call SPLog10 entry */
    f1 = SPPow(f2);           /* Call SPPow entry */
    f1 = SPSqrt(f2);          /* Call SPSqrt entry */

    i1 = SPTieee(f2);         /* Call SPTieee entry */
    f1 = SPFieee(i1);         /* Call SPFieee entry */

    CloseLibrary(MathTransBase);
    CloseLibrary(MathBase);
}
```

The section below describes the Amiga assembly language interface to the Motorola Fast Floating Point transcendental math routines and includes some details about how the system flags are affected by the operation. Again, this interface resides in the library file *amiga.lib* and must be linked with the user code. Note that the access mechanism from assembly language is as shown below:

```
MOVEA.L _MathTransBase, A6
JSR     _LVOSPAsin(A6)
```

#### **\_LVOSPAsin - Arcsine**

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP arctangent radian result  
N = 0  
Z = 1 if result is zero  
V = 0  
C = undefined  
X = undefined

#### **\_LVOSPACos - Arccosine**

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP arctangent radian result  
N = 0  
Z = 1 if result is zero  
V = 1 if overflow occurred  
C = undefined  
X = undefined

#### **\_LVOSPAtan - Arctangent**

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP arctangent radian result  
N = 0  
Z = 1 if result is zero  
V = 0  
C = undefined  
X = undefined

#### **\_LVOSPSin - Sine**

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument in radians  
D0 = FFP sine result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = 1 if result is meaningless  
(that is, input magnitude too large)  
C = undefined  
X = undefined

#### **\_LVOSPCos - Cosine**

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument in radian  
D0 = FFP cosine result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = 1 if result is meaningless



(that is, input magnitude too large)

C = undefined

X = undefined

**\_LVOSPtan -**

Tangent

Inputs:

Outputs:

Condition codes:

D0 = FFP argument in radians

D0 = FFP tangent result

N = 1 if result is negative

Z = 1 if result is zero

V = 1 if result is meaningless

(that is, input magnitude too large)

C = undefined

X = undefined

**\_LVOSPSincos -**

Sine and cosine

Inputs:

Outputs:

Condition codes:

D0 = FFP argument in radians

D1 = Address to store cosine result

D0 = FFP sine result

(D1) = FFP cosine result

N = 1 if result is negative

Z = 1 if result is zero

V = 1 if result is meaningless

(that is, input magnitude too large)

C = undefined

X = undefined

**\_LVOSPSinh -**

Hyperbolic sine

Inputs:

Outputs:

Condition codes:

D0 = FFP argument in radians

D0 = FFP hyperbolic sine result

N = 1 if result is negative

Z = 1 if result is zero

V = 1 if overflow occurred

C = undefined

X = undefined

**\_LVOSPCosh -**

Hyperbolic cosine

Inputs:

Outputs:

Condition codes:

D0 = FFP argument in radians

D0 = FFP hyperbolic cosine result

N = 1 if result is negative

Z = 1 if result is zero

V = 1 if overflow occurred

C = undefined

X = undefined

**\_LVOSPtanh -**

Hyperbolic tangent

Inputs:

D0 = FFP argument in radians

Outputs:  
Condition codes:

D0 = FFP hyperbolic tangent result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = 1 if overflow occurred  
C = undefined  
X = undefined

**\_LVOSPExp -**

Exponential

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP exponential result  
N = 0  
Z = 1 if result is zero  
V = 1 if overflow occurred  
C = undefined  
Z = undefined

**\_LVOSPLog -**

Natural logarithm

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP natural logarithm result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = 1 if argument negative or zero  
C = undefined  
Z = undefined

**\_LVOSPLog10 -**

Naparian (base 10) logarithm

Inputs:  
Outputs:  
Condition codes:

D0 = FFP argument  
D0 = FFP natural logarithm result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = 1 if argument negative or zero  
C = undefined  
Z = undefined

**\_LVOSPPow -**

Power

Inputs:  
Outputs:  
Condition codes:

D1 = FFP argument value  
D0 = FFP exponent value  
D0 = FFP result of arg taken to exp power  
N = 0  
Z = 1 if result is zero  
V = 1 if result overflowed or arg < 0  
C = undefined  
Z = undefined

**\_LVOSPSqrt -**

Square root

Inputs:

D0 = FFP argument

Outputs:  
Condition codes:

D0 = FFP square root result  
N = 0  
Z = 1 if result is zero  
V = 1 if argument was negative  
C = undefined  
Z = undefined

**\_LVOSP ieee -** Convert to IEEE format

Inputs:  
Outputs:  
Condition codes:

D0 = FFP format argument  
D0 = IEEE floating-point format result  
N = 1 if result is negative  
Z = 1 if result is zero  
V = undefined  
C = undefined  
Z = undefined

**\_LVOSPF ieee -** Convert from IEEE format

Inputs:  
Outputs:  
Condition codes:

D0 = IEEE floating-point format argument  
D0 = FFP format result  
N = undefined  
Z = 1 if result is zero  
V = 1 if result overflowed FFP format  
C = undefined  
Z = undefined

## FFP Mathematics Conversion Library

The FFP mathematics conversion library is accessed by linking code into the executable file being created. The name of the file to include in the library description of the link command line is *amiga.lib*. When this is included, direct calls are made to the conversion functions. Only a C interface exists for the conversion functions; there is no assembly language interface. The basic math library is required in order to access these functions and might be opened as shown below.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;

VOID main()
{
    if((MathBase = OpenLibrary("mathffp.library", 0)) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }
    .
    .
    .
    CloseLibrary(MathBase);
}
```

The C-called entry points are accessed via code generated by the C compiler when the actual function names are given within the source code. The C entry points defined for the math conversion functions are as follows:

<b>afp</b>	Convert ASCII string into FFP equivalent. Usage: fnum = afp(&string[0]);
<b>fpa</b>	Convert FFP variable into ASCII equivalent. Usage: exp = fpa(fnum, &string[0]);
<b>arnd</b>	Round ASCII representation of FFP number. Usage: arnd(place, exp, &string[0]);
<b>dbf</b>	Convert FFP dual-binary number to FFP equivalent. Usage: fnum = dbf(exp, mant);
<b>fpbcd</b>	Convert FFP variable to BCD equivalent. Usage: fpbcd(fnum, &string[0]);

### WARNING

The **fpbcd()** function does not work correctly in the **amiga.lib** supplied for V1.3 and earlier versions. Avoid using this function at this time.

Be sure to include proper data type definitions, as shown in the example below. Print statements have been included to help clarify the format of the math conversion function calls.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

struct Library *MathBase;

UBYTE st1[80] = "3.1415926535897";
UBYTE st2[80] = "2.718281828459045";
UBYTE st3[80], st4[80];

VOID main()
{
    FLOAT num1, num2;
    FLOAT n1, n2, n3, n4;
    LONG exp1, exp2, exp3, exp4;
    LONG mant1, mant2, mant3, mant4;
    LONG place1, place2;

    if((MathBase = OpenLibrary("mathffp.library", 0)) == 0)
    {
        printf("Can't open mathffp.library\n");
        exit(20);
    }

    n1 = afp(st1);          /* Call afp entry */
    n2 = afp(st2);          /* Call afp entry */
    printf("\n\nASCII %s converts to floating point %f",
        st1, n1);
    printf("\n\nASCII %s converts to floating point %f",
        st2, n2);

    num1 = 3.1415926535897;
    num2 = 2.718281828459045;
```

```

exp1 = fpa(num1, st3);    /* Call fpa entry */
exp2 = fpa(num2, st4);    /* Call fpa entry */
printf("\n\nfloating point %f converts to ASCII %s", num1, st3);
printf("\n\nfloating point %f converts to ASCII %s",
      num2, st4);

place1 = -2;
place2 = -1;
arnd(place1, exp1, st3);  /* Call arnd entry */
arnd(place2, exp2, st4);  /* Call arnd entry */
printf("\n\nASCII round of %f to %d places yields %s",
      num1, place1, st3);
printf("\n\nASCII round of %f to %d places yields %s",
      num2, place2, st4);

exp1 = -3;  exp2 = 3;  exp3 = -3;  exp4 = 3;
mant1 = 12345; mant2 = -54321; mant3 = -12345; mant4 = 54321;

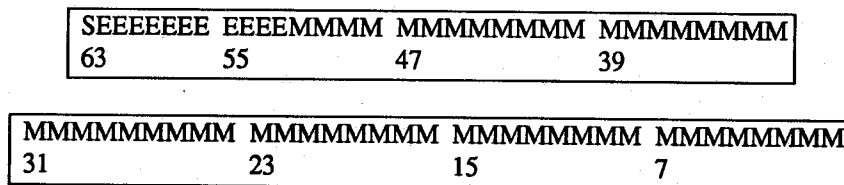
n1 = dbf(exp1, mant1);    /* Call dbf entry */
n2 = dbf(exp2, mant2);    /* Call dbf entry */
n3 = dbf(exp3, mant3);    /* Call dbf entry */
n4 = dbf(exp4, mant4);    /* Call dbf entry */
printf("\n\nndbf of exp = %d and mant = %d yields FFP number of %f",
      exp1, mant1, n1);
printf("\n\nndbf of exp = %d and mant = %d yields FFP number of %f",
      exp2, mant2, n2);
printf("\n\nndbf of exp = %d and mant = %d yields FFP number of %f",
      exp3, mant3, n3);
printf("\n\nndbf of exp = %d and mant = %d yields FFP number of %f",
      exp4, mant4, n4);

CloseLibrary(MathBase);
)

```

## IEEE Double-Precision Data Format

The IEEE double-precision variables are defined as 64-bit entities with the following format:



where

**S** = Sign of IEEE number

**M** = 52-bit(+1) mantissa

**E** = 11-bit exponent

## NOTE

There is a "hidden" bit in the mantissa part of the IEEE numbers. Since all numbers are normalized, the leading 1 is dropped off. The IEEE double-precision range is 2.2E-308 (4.9E-324 de-normalized) to 1.8E+307.

## IEEE Double-Precision Basic Math Library

The IEEE double-precision basic math library resides on disk and is opened by making a call to the `OpenLibrary()` function with `mathieeedoubbas.library` as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter for details. In C, this might be implemented as shown below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

struct Library *MathIeeeDoubBasBase;

VOID main()
{
    /* do not share base pointer between tasks. */
    if ((MathIeeeDoubBasBase =
        OpenLibrary("mathieeedoubbas.library", 0)) == 0)
    {
        printf("Can't open mathieeedoubbas.library\n");
        exit(20);
    }

    .
    .
    .

    CloseLibrary(MathIeeeDoubBasBase);
}
```

The global variable `MathIeeeDoubBasBase` is used internally for all future library references.

This library contains entries for the basic IEEE mathematics functions, such as add, subtract, and so on.

If an 68020/68881 processor combination is available, it will be used by the IEEE basic library instead of the software emulation. Also, if an auto-configured math resource is available, that will be used. Typically this is a 68881 designed as a 16 bit I/O port, but it could be another device as well.

The IEEE double-precision basic math library is accessed by linking code into the executable file being created. The name of the file to include in the library description of the link command is `mathieeedoubbas_lib.lib`. The C entry points defined for the IEEE double-precision basic math functions are listed below:

### IEEEDPFix

Convert IEEE double-precision variable to integer

Usage: `i1 = IEEEDPFix(d1);`

### IEEEDPFlt

Convert integer variable to IEEE double-precision

Usage: `d1 = IEEEDPFlt(i1);`

### IEEEDPComp

Compare two IEEE double-precision variables

Usage: `switch (IEEEDPComp(d1, d2)) {};`

**IEEEDPTst** Test an IEEE double-precision variable against zero

Usage:   switch (IEEEDPTst(d1)) {};

**IEEEDPAbs**

Take absolute value of IEEE double-precision variable

Usage:   d1 = IEEEDPAbs(d2);

**IEEEDPNeg**

Take two's complement of IEEE double-precision variable

Usage:   d1 = IEEEDPNeg(d2);

**IEEEDPAdd**

Add two IEEE double-precision variables

Usage:   d1 = IEEEDPAdd(d2, d3);

**IEEEDPSub**

Subtract two IEEEDPSub variables

Usage:   d1 = IEEEDPSub(d2, d3);

**IEEEDPMul**

Multiply two IEEE double-precision variables

Usage:   d1 = IEEEDPMul(d2, d3);

**IEEEDPDiv**

Divide two IEEE double-precision variables

Usage:   d1 = IEEEDPDiv(d2, d3);

**IEEEDPCeil**

Compute least integer greater than or equal to variable

Usage:   d1 = IEEEDPCeil(d2);

**IEEEDPFloor**

Compute largest integer less than or equal to variable

Usage:   d1 = IEEEDPFloor(d2);

Be sure to include proper data type definitions, as shown in the example below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

struct Library *MathIeeeDoubBasBase;

VOID main()
{
    DOUBLE d1, d2, d3;
    LONG   i1;

    if((MathIeeeDoubBasBase =
        OpenLibrary("mathieeedoubbas.library", 0)) == 0)
    {
        printf("Can't open mathieeedoubbas.library0);
        exit(20);
    }

    i1 = IEEEDPFix(d1);                /* Call IEEEDPFix entry */
```

```

fi = IEEEEDPflt(i1);          /* Call IEEEEDPflt entry */
switch (IEEEEDPCmp(d1, d2)) {}; /* Call IEEEEDPCmp entry */
switch (IEEEEDPTst(d1)) {};   /* Call IEEEEDPTst entry */
d1 = IEEEEDPAbs(d2);          /* Call IEEEEDPAbs entry */
d1 = IEEEEDPNeg(d2);          /* Call IEEEEDPNeg entry */
d1 = IEEEEDPAdd(d2, d3);      /* Call IEEEEDPAdd entry */
d1 = IEEEEDPSub(d2, d3);      /* Call IEEEEDPSub entry */
d1 = IEEEEDPMul(d2, d3);      /* Call IEEEEDPMul entry */
d1 = IEEEEDPDiv(d2, d3);      /* Call IEEEEDPDiv entry */
d1 = IEEEEDPCeil(d2);         /* Call IEEEEDPCeil entry */
d1 = IEEEEDPFloor(d2);        /* Call IEEEEDPFloor entry */

```

```
CloseLibrary(MathIeeeDoubBasBase);
```

The Amiga assembly language interface to the IEEE double-precision floating-point basic math routines is shown below, including some details about how the system flags are affected by each operation. Note that the access mechanism from assembly language is as shown below:

```

MOVEA.L _MathIeeeDoubBasBase, A6
JSR     _LVOIEEEDPFix(A6)

```

**\_LVOIEEEDPFix - Convert IEEE double-precision to integer**

Inputs:	D0/D1 = IEEE double-precision argument
Outputs:	D0 = Integer (two's complement) result
Condition codes:	N = undefined
	Z = undefined
	V = undefined
	C = undefined
	X = undefined

**\_LVOIEEEDPflt - Convert integer to IEEE double-precision**

Inputs:	D0 = Integer (two's complement) argument
Outputs:	D0/D1 = IEEE double-precision result
Condition codes:	N = undefined
	Z = undefined
	V = undefined
	C = undefined
	X = undefined

**\_LVOIEEEEDPCmp - Compare two IEEE double-precision values**

Inputs:	D0/D1 = IEEE double-precision argument 1
	D2/D3 = IEEE double-precision argument 2
Outputs:	D0 = +1 if arg1 < arg2
	D0 = -1 if arg1 > arg2
	D0 = 0 if arg1 = arg2
Condition codes:	N = 1 if result is negative
	Z = 1 if result is zero
	V = 0
	C = undefined
	X = undefined
	GT = arg2 > arg1



GE = arg2 >= arg1  
 EQ = arg2 = arg1  
 NE = arg2 <> arg1  
 LT = arg2 < arg1  
 LE = arg2 <= arg1

**\_LVOIEEEEDPTst - Test an IEEE double-precision value against zero**

Inputs:	D0/D1 = IEEE double-precision argument
Outputs:	D0 = +1 if arg > 0.0 D0 = -1 if arg < 0.0 D0 = 0 if arg = 0.0
Condition codes:	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined EQ = arg = 0.0 NE = arg <> 0.0 PL = arg >= 0.0 MI = arg < 0.0

**\_LVOIEEEEDPAbs - Absolute value**

Inputs:	D0/D1 = IEEE double-precision argument
Outputs:	D0/D1 = IEEE double-precision absolute value result
Condition codes:	N = undefined Z = undefined V = undefined C = undefined X = undefined

**\_LVOIEEEEDPNeg - Negate**

Inputs:	D0/D1 = IEEE double-precision argument
Outputs:	D0/D1 = IEEE double-precision negated result
Condition codes:	N = undefined Z = undefined V = undefined C = undefined X = undefined

**\_LVOIEEEEDPAdd - Addition**

Inputs:	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2
Outputs:	D0/D1 = IEEE double-precision addition of arg1+arg2 result
Condition codes:	N = undefined

Z = undefined  
V = undefined  
C = undefined  
X = undefined

#### **\_LVOIEEEEDPSub - Subtraction**

Inputs: D0/D1 = IEEE double-precision argument 1  
D2/D3 = IEEE double-precision argument 2  
Outputs: D0/D1 = IEEE double-precision subtraction  
of arg1-arg2 result  
Condition codes: N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

#### **\_LVOIEEEEDPMul - Multiply**

Inputs: D0/D1 = IEEE double-precision argument 1  
D2/D3 = IEEE double-precision argument 2  
Outputs: D0/D1 = IEEE double-precision multiplication  
of arg1\*arg2 result  
Condition codes: N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

#### **\_LVOIEEEEDPDiv - Divide**

Inputs: D0/D1 = IEEE double-precision argument 1  
D2/D3 = IEEE double-precision argument 2  
Outputs: D0/D1 = IEEE double-precision division  
of arg1/arg2 result  
Condition codes: N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

#### **\_LVOIEEEEDPCeil - Ceil**

Inputs: D0/D1 = IEEE double-precision variable  
Outputs: D0/D1 = least integer >= variable  
Condition codes: N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

**\_LVOIEEEDPFloor - Floor**

Inputs:  
Outputs:  
Condition codes:

D0/D1 = IEEE double-precision variable  
D0/D1 = largest integer <= argument  
N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

## IEEE Double-Precision Transcendental Math Library

The IEEE double-precision transcendental math library resides on disk and is opened by making a call to the **OpenLibrary()** function with **mathieeedoubtrans.library** as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter for details. In C, this might be implemented as shown below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

struct Library *MathIeeeDoubBasBase;
struct Library *MathIeeeDoubTransBase;

VOID main()
{
    if((MathIeeeDoubBasBase =
        OpenLibrary("mathieeedoubbas.library",0)) == 0)
    {
        printf("Can't open mathieeedoubbas.library0);
        exit(20);
    }

    if((MathIeeeDoubTransBase =
        OpenLibrary("mathieeedoubtrans.library",0)) == 0)
    {
        printf("Can't open mathieeedoubtrans.library\n");
        CloseLibrary(MathIeeeDoubBasBase);
        exit(21);
    }

    .
    .
    .
    CloseLibrary(MathIeeeDoubTransBase);
}
```

The global variable **MathIeeeDoubTransBase** is used internally for all future library references.

### NOTE

The IEEE transcendental math library is dependent upon the IEEE basic math library, which it will open if it is not open already. If you want to use the IEEE basic math functions in conjunction with the transcendental math functions however, you have to specifically open the basic math library yourself.

This library contains entries for the transcendental math functions sine, cosine, and so on. Just as the IEEE basic math library, the IEEE transcendental math library will take advantage of a 68020/68881 combination or another math resource, if present.

The IEEE double-precision transcendental math library is accessed by linking code into the executable file being created. The name of the file to include in the library description of the link command is *mathieeedoubtrans\_lib.lib*. The C entry points defined for the IEEE double-precision basic math functions are listed below:

**IEEEDPASin**

Return arcsine of IEEE variable.

Usage: `d1 = IEEEDPASin(d2);`

**IEEEDPACos**

Return arccosine of IEEE variable.

Usage: `d1 = IEEEDPACos(d2);`

**IEEEDPATan**

Return arctangent of IEEE variable.

Usage: `d1 = IEEEDPATan(d2);`

**IEEEDPSin**

Return sine of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric sine value.

Usage: `d1 = IEEEDPSin(d2);`

**IEEEDPCos**

Return cosine of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric cosine value.

Usage: `d1 = IEEEDPCos(d2);`

**IEEEDPTan**

Return tangent of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric tangent value.

Usage: `d1 = IEEEDPTan(d2);`

**IEEEDPSincos**

Return sine and cosine of IEEE variable. This function accepts an IEEE radian argument and returns both the trigonometric sine and cosine values.

Usage: `d1 = IEEEDPSincos(&d3, d2);`

**IEEEDPSinh**

Return hyperbolic sine of IEEE variable.

Usage: `d1 = IEEEDPSinh(d2);`

**IEEEDPCosh**

Return hyperbolic cosine of IEEE variable.

Usage: `d1 = IEEEDPCosh(d2);`

**IEEEDPTanh**

Return hyperbolic tangent of IEEE variable.

Usage: `d1 = IEEEDPTanh(d2);`

**IEEEDPExp**

Return  $e$  to the IEEE variable power. This function accept an IEEE argument and returns the result representing the value of  $e$  (2.712828...) raised to that power.

Usage: `d1 = IEEEDPExp(d2);`

**IEEEDPLog**

Return natural log (base  $e$  of IEEE variable.

Usage: `d1 = IEEEDPLog(d2);`

**IEEEDPLog10**

Return naperian log (base 10) of IEEE variable.

Usage: `d1 = IEEEDPLog10(d2);`

**IEEEDPPow**

Return IEEE arg2 to IEEE arg1.

Usage: `d1 = IEEEDPPow(d2);`

**IEEEDPSqrt**

Return square root of IEEE variable.

Usage: `d1 = IEEEDPSqrt(d2);`

**IEEEDPTieee**

Convert IEEE double-precision number to IEEE single-precision number.

Usage: `s1 = IEEEDPTieee(d2);`

Be sure to include proper data type definitions as shown below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

struct Library *MathIeeeDoubBasBase;
struct Library *MathIeeeDoubTransBase;

VOID main()
{
    DOUBLE d1, d2, d3;
    FLOAT f1;

    if ((MathIeeeDoubBasBase =
        OpenLibrary("mathieeedoubbas.library", 0)) == 0)
    {
        printf("Can't open mathieeedoubbas.library\n");
        exit(20);
    }

    if ((MathIeeeDoubTransBase =
        OpenLibrary("mathieeedoubtrans.library", 0)) == 0)
    {
        printf("Can't open mathieeedoubtrans.library\n");
        CloseLibrary(MathIeeeDoubBasBase);
        exit(21);
    }

    d1 = IEEEDPAsin(d2);      /* Call IEEEDPAsin entry */
    d1 = IEEEDPAcos(d2);      /* Call IEEEDPAcos entry */
    d1 = IEEEDPATan(d2);      /* Call IEEEDPATan entry */
    d1 = IEEEDPSin(d2);       /* Call IEEEDPSin entry */
    d1 = IEEEDPCos(d2);       /* Call IEEEDPCos entry */
    d1 = IEEEDPTan(d2);       /* Call IEEEDPTan entry */
}
```

```

d1 = IEEEEDPSincos(&d3, d2); /* Call IEEEEDPSincos entry */
d1 = IEEEEDPSinh(d2);        /* Call IEEEEDPSinh entry */
d1 = IEEEEDPCosh(d2);        /* Call IEEEEDPCosh entry */
d1 = IEEEEDPTanh(d2);        /* Call IEEEEDPTanh entry */
d1 = IEEEEDPExp(d2);         /* Call IEEEEDPExp entry */
d1 = IEEEEDPLog(d2);         /* Call IEEEEDPLog entry */
d1 = IEEEEDPLog10(d2);       /* Call IEEEEDPLog10 entry */
d1 = IEEEEDPPow(d2, d3);     /* Call IEEEEDPPow entry */
d1 = IEEEEDPSqrt(d2);        /* Call IEEEEDPSqrt entry */
f1 = IEEEEDPTieee(d2);       /* Call IEEEEDPTieee entry */
d1 = IEEEEDPFieee(f1);       /* Call IEEEEDPFieee entry */

```

```

CloseLibrary(MathIeeeDoubTransBase);
CloseLibrary(MathIeeeDoubBasBase);
)

```

The section below describes the Amiga assembly interface to the IEEE double-precision transcendental math library. Again this interface resides in the library file *mathieeedoubtrans\_Lib.lib* and must be linked with the user code. The access mechanism from assembly language is as show below:

```

MOVEA.L _MathIeeeDoubTransBase,A6
JSR     _LVOSPAsin(A6)

```

#### \_LVOIEEEDPAsin - Arcsine

Inputs:	D0/D1 = IEEE argument
Outputs:	D0/D1 = IEEE arctangent radian result
Condition codes:	N = undefined
	Z = undefined
	V = undefined
	C = undefined
	X = undefined

#### \_LVOIEEEDPacos - Arccosine

Inputs:	D0/D1 = IEEE argument
Outputs:	D0/D1 = IEEE arctangent radian result
Condition codes:	N = undefined
	Z = undefined
	V = undefined
	C = undefined
	X = undefined

#### \_LVOIEEEDPatan - Arctangent

Inputs:	D0/D1 = IEEE argument
Outputs:	D0/D1 = IEEE arctangent radian result
Condition codes:	N = undefined
	Z = undefined
	V = undefined
	C = undefined
	X = undefined

**\_LVOIEEEDPSin - Sine**

Inputs:  
Outputs:  
Condition codes:

D0/D1 = IEEE argument in radians  
D0/D1 = IEEE sine result  
N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

**\_LVOIEEEDPCos - Cosine**

Inputs:  
Outputs:  
Condition codes:

D0/D1 = IEEE argument in radian  
D0/D1 = IEEE cosine result  
N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

**\_LVOIEEEDPTan - Tangent**

Inputs:  
Outputs:  
Condition codes:

D0/D1 = IEEE argument in radians  
D0/D1 = IEEE tangent result  
N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

**\_LVOIEEEDPSincos -**

Sine and cosine

Inputs: A0 = Address to store cosine result

D0/D1 = IEEE argument in radians

Outputs:  
Condition codes:

D0/D1 = IEEE sine result  
(A0) = IEEE cosine result  
N = undefined  
Z = undefined  
V = undefined  
C = undefined  
X = undefined

**\_LVOIEEEDPSinh - Hyperbolic sine**

Inputs:  
Outputs:  
Condition codes:

D0/D1 = IEEE argument in radians  
D0/D1 = IEEE hyperbolic sine result  
N = undefined  
Z = undefined  
V = undefined  
C = undefined

X = undefined

**\_LVOIEEEDPCosh - Hyperbolic cosine**

Inputs:

Outputs:

Condition codes:

D0/D1 = IEEE argument in radians

D0/D1 = IEEE hyperbolic cosine result

N = undefined

Z = undefined

V = undefined

C = undefined

X = undefined

**\_LVOIEEEDPTanh - Hyperbolic tangent**

Inputs:

Outputs:

Condition codes:

D0/D1 = IEEE argument in radians

D0/D1 = IEEE hyperbolic tangent result

N = undefined

Z = undefined

V = undefined

C = undefined

X = undefined

**\_LVOIEEEDPExp - Exponential**

Inputs:

Outputs:

Condition codes:

D0/D1 = IEEE argument

D0/D1 = IEEE exponential result

N = undefined

Z = undefined

V = undefined

C = undefined

X = undefined

**\_LVOIEEEDPLog - Natural logarithm**

Inputs:

Outputs:

Condition codes:

D0/D1 = IEEE argument

D0/D1 = IEEE natural logarithm result

N = undefined

Z = undefined

V = undefined

C = undefined

X = undefined

**\_LVOIEEEDPLog10 -**

Inputs:

Outputs:

Condition codes:

Naparian (base 10) logarithm

D0/D1 = IEEE argument

D0/D1 = IEEE natural logarithm result

N = undefined

Z = undefined

V = undefined

C = undefined

X = undefined



**\_LVOIEEEEDPPow - Power**

**Inputs:**

**Outputs:**

**Condition codes:**

**D0/D1 = IEEE exponent value**

**D2/D3 = IEEE argument value**

**D0/D1 = IEEE result of arg taken to exp power**

**N = undefined**

**Z = undefined**

**V = undefined**

**C = undefined**

**X = undefined**

**\_LVOIEEEEDPSqrt - Square root**

**Inputs:**

**Outputs:**

**Condition codes:**

**D0/D1 = IEEE argument**

**D0/D1 = IEEE square root result**

**N = undefined**

**Z = undefined**

**V = undefined**

**C = undefined**

**X = undefined**

**\_LVOIEEEEDPTieee - Convert to single-precision IEEE format**

**Inputs:**

**Outputs:**

**Condition codes:**

**D0/D1 = IEEE format argument**

**D0 = single-precision IEEE floating-point format result**

**N = undefined**

**Z = undefined**

**V = undefined**

**C = undefined**

**X = undefined**

## Chapter 29

# Translator Library

This chapter describes the translator library which, together with the narrator device, provides the Amiga's text-to-speech capability. To fully understand how speech is produced on the Amiga, you should also read the "Narrator Device" chapter. The translator library provides a single function, `Translate()`, that converts an English language string into a phonetic string. You may then pass this phonetic string to the narrator device which will say the string using the Amiga's audio hardware. The two subsystems may also be used individually. You don't have to use the narrator to say the phonetic strings; you could use them instead for phonetic analysis or some other special purpose.

### OPENING THE TRANSLATOR LIBRARY

To use the `Translate()` function, you must first open the translator library. Setting a global variable, `TranslatorBase`, to the value returned from the call to `OpenLibrary()` enables the Amiga linker to correctly locate the translator library:

```
struct Library *TranslatorBase;

TranslatorBase = OpenLibrary("translator.library", REVISION);
if (TranslatorBase != NULL)
{
    /* use translator here -- library open */
}
```

## NOTE

Since translator is a disk-based library, the call to `OpenLibrary()` will work only if the *LIBS:* directory contains *translator.library*.

## USING THE TRANSLATE FUNCTION

Once the library is open, you can call the translate function:

```
#define BUFLen 500

APTR EnglStr;           /* pointer to sample input string */
LONG EnglLen;           /* input length */
UBYTE PhonBuffer[BUFLen]; /* place to put the translation */
LONG rtnCode;           /* return code from function */

EnglStr = "This is Amiga speaking."; /* a test string */
EnglLen = strlen(EnglStr);
rtnCode = Translate(EnglStr, EnglLen, (APTR)&PhonBuffer[0], BUFLen);
```

The input string will be translated into its phonetic equivalent and can be used to feed the narrator device. If you receive a non-zero return code, you haven't provided enough output buffer space to hold the entire translation. In this case, the `Translate()` function breaks the translation at the end of a word in the input stream and returns the position in the input stream at which the translation ended. You can use the output buffer, then call the `Translate()` function again, starting at this original ending position, to continue the translation where you left off. This method will sound smoothest if the ending position ends on sentence boundaries.

## NOTE

The value returned is *negative*. Therefore, you must use `-(rtnCode)` as the starting point for a new translation.

## CLOSING THE TRANSLATOR LIBRARY

As with all other libraries of functions, if you have successfully opened the translator library for use, be sure to close it before your program exits. If the system needs memory resources, it can then expunge closed libraries to gain additional memory space:

```
struct Library *TranslatorBase;

if(TranslatorBase) CloseLibrary(TranslatorBase);
```

## ADDITIONAL NOTES ABOUT TRANSLATE

The English language has many words that do not sound the same as they are spelled. The translator library has exception rules that it consults as the translation progresses. It also provides for common abbreviations such as Dr., Prof., LB., etc. Words that are not in the exception table are translated literally. This translation allows unrestricted English text as input, and uses over four hundred and fifty context sensitive rules. It automatically accents content words, and leaves function words (e.g. of, by, the, and at) unaccented. It is possible, however, that certain words will not translate well. You can improve the quality of the translation by handling those words on your own.

The phoneme table that the narrator uses is listed in the “Narrator Device” chapter. You will also find other important information on the Amiga’s speech capability in the narrator chapter including a working example which shows how to use the translator library together with the narrator device.

# Chapter 30

## Workbench

Workbench provides the iconic user interface on the Amiga. This chapter shows how to use the Workbench and `icon.library` in your applications.

### Introduction

Workbench is both an application program and a screen in which other applications can run. Workbench allows users to interact with the Amiga file system by using graphic file representations known as icons. The `icon.library` gives the programmer access to a body of library functions for manipulating these icons.

Here are definitions of some terms that may be unfamiliar or used in unfamiliar ways in this chapter.

#### *Workbench object*

A Workbench object contains all the information that Workbench needs to display and use a project, tool, drawer, etc. Workbench objects are used at the program level as `DiskObjects`, allowing an application program to read, update, or create icons.

#### *icon*

Icon is a shorthand name for a Workbench object. Icons allow for the graphical representation of a file, directory or disk. On disk, the information for the icon is currently stored in a “.info” file. An icon may be on disk or in memory or both.

### *“.info” file*

The current disk representation of an icon. The format of an icon on disk is slightly different from an icon in memory, i.e. from a DiskObject, but one is obtainable from the other through icon.library functions.

### *strings*

A null-terminated sequence of bytes which are interpreted as ASCII text.

### *activating*

The act of starting a tool, or opening a drawer or disk. Tools may be activated either by directly accessing the tool, or by accessing projects associated with the tool (see default tool mechanism.)

### *tool*

An application program or system utility as used from the Workbench. The tool is the executable program and its associated “.info” file. Examples of tools would be a word processor or a paint program. The data file used by the tool is a project.

### *project*

A data file and its associated “.info” file, produced by an executable program (a tool), and associated with a specific tool. The default tool does not have to be the tool which created the project. Example of projects would be a text file or a drawing.

### *drawer*

A file system directory and its associated “.info” file. A drawer may be on disk, RAM, or over a network, etc.

### *default tool mechanism*

Method of starting a tool by activating a project. When a project is activated, the tool given in the “default tool” field is activated and the project is sent to the tool as an argument.

### *extend select mechanism*

Method of starting a tool and passing one or more projects as arguments. Currently performed by holding the shift key to enable selection of one or more icons to be passed to the tool as Workbench arguments.

## **The Icon Library**

The icon library, *icon.library*, has the following routines, which allow high-level access to icons.

```
LONG      AddFreeList(struct FreeList *free, UBYTE *mem, LONG len);
VOID      FreeFreeList(struct FreeList *free);

LONG      BumpRevision(UBYTE *newbuf, UBYTE *oldname);

UBYTE     *FindToolType(UBYTE **toolTypeArray, UBYTE *typeName);
LONG      MatchToolValue(UBYTE *typeString, UBYTE *value);

LONG      PutDiskObject(UBYTE *name, struct DiskObject *diskObj);
struct DiskObject *GetDiskObject(UBYTE *name);
VOID      FreeDiskObject(struct DiskObject *diskObj);
```

See the *Includes & Autodocs Manual* for the reference pages for this library.

## The Info File

The “.info” file is the center of interaction between applications and Workbench. This file stores all the necessary information to display an icon and to start up an application. The “.info” file can describe one of several different types of icons, as shown in the next table.

Table 30-1: Workbench Object Types

Icon Type	Object
WBDISK	The root of a disk
WBDRAWER	A directory on the disk
WBTOOL	A directly runnable program
WBPROJECT	A data file of some sort
WBGARBAGE	The Trashcan directory
WBKICK	A A Kickstart disk

Icons are associated with a particular file or drawer by being in the same directory and having the same name with the added extension “.info”. For example, an icon file with the name “myprogram.info” is the icon for the file named “myprogram” in the same directory.

The actual data present in the “.info” file depends on the icon type.

### NOTE

Any graphical image can be used for any icon type in the “.info” file. In fact, the graphical image need not be unique for each type of icon.

The icon.library functions do all the work needed to read and write “.info” files. The **GetDiskObject( )**, **PutDiskObject( )**, and **FreeDiskObject( )** routines are especially helpful. The “RKM\_Icon\_Example.c” code at the end of this chapter demonstrates the use of these routines in reading and creating icons.

## THE DISKOBJECT STRUCTURE

A **DiskObject** structure, and its associated strings and images, are stored in a “.info” file. The icon.library routines **GetDiskObject( )**, **FreeDiskObject( )**, and **PutDiskObject( )** are used to respectively to read, close after reading, and create new icon files. When writing an icon, the **PutDiskObject( )** function will automatically take along the images, strings, and arrays pointed to by a **DiskObject** structure, and store them in a relative format which can later be retrieved by **GetDiskObject( )**. The **DiskObject** structure is defined in *workbench/workbench.h* and contains the following elements:

```

struct DrawerData
{
    struct NewWindow dd_NewWindow;    /* structure to open window */
    LONG             dd_CurrentX;     /* current x coordinate of origin */
    LONG             dd_CurrentY;     /* current y coordinate of origin */
};

struct DiskObject
{
    UWORD            do_Magic;         /* magic number at start of file */
    UWORD            do_Version;       /* so we can change structure */
    struct Gadget    do_Gadget;        /* a copy of in core gadget */
    UBYTE            do_Type;
    char             *do_DefaultTool;
    char             **do_ToolTypes;
    LONG             do_CurrentX;
    LONG             do_CurrentY;
    struct DrawerData *do_DrawerData;
    char             do_ToolWindow;    /* only applies to tools */
    LONG             do_StackSize;     /* only applies to tools */
};

```

### do\_Magic

A magic number that the icon library looks for to make sure that the file it is reading really contains an icon. It should be the manifest constant `WB_DISKMAGIC`. `PutDiskObject()` will put this value in the structure, and `GetDiskObject` will not believe that a file is really an icon unless this value is correct.

### do\_Version

This provides a way to enhance the “.info” file in an upwardly-compatible way. It should be `WB_DISKVERSION`. The icon library will set this value for you and will not believe weird values.

### do\_Gadget

This contains all the imagery for the icon. See the “Gadget Structure” section for more details.

### do\_Type

The type of the icon (`WBTOOL`, `WBPROJECT`, and so on). See the table of Workbench Object Types.

### do\_DefaultTool

Default tools are used for project and disk icons. For projects (data files), the default tool is the program invoked when the project is activated. This tool may be absolute as in “`DISK:file`” or “`DISK:dir/file`”, relative to the root of this disk as in “`:file`” or “`:dir/file`”, or relative to the project as in “`file`” or “`dir/file`”. These path examples are only examples. Any valid path may be entered for the tool, including the use of assigns such as “`T:`”, or “`SYS:`”.

If the icon is of type `WBDISK`, the default tool is the diskcopy program (“`SYS:System/DiskCopy`”) that will be used when *this disk is the source* of a copy.

### NOTE

If the tool is run *via the default tool mechanism* (for example, a project was activated, not a tool), all the information in the project’s “.info” file is used, and the tool’s “.info” file is ignored. This is especially important for variables like `StackSize`.

### do\_ToolTypes

This is an array of free-format strings. Workbench does not enforce any rules on these strings, but they are useful for passing environment information. See the “ToolTypes” section for more information.



#### **do\_CurrentX, do\_CurrentY**

Drawers have a virtual coordinate system. The user can scroll around in this system using the scroll gadgets on the "drawers" window. Each icon in the drawer has a position in the coordinate system. **CurrentX** and **CurrentY** contain the icon's current position in the drawer. The constant **NO\_ICON\_POSITION** indicates that the icon should be placed wherever there is room.

#### **do\_DrawerData**

If the icon is capable of being opened as a drawer (**WBDISK**, **WBDRAWER**, **WBGARBAGE**), it needs a **DrawerData** structure to go with it. This structure contains an Intuition **NewWindow** structure. See the section on Intuition for more information about windows. Workbench uses this to hold the current window position and size of the window so it will reopen in the same place. The **CurrentX** and **CurrentY** of the origin of the window is also stored.

#### **do\_ToolWindow**

This field is reserved for future use.

#### **do\_StackSize**

This is the size of the stack (in bytes) used for running the tool. If this is null, then Workbench will use a reasonable default stack size (currently 4K bytes).

### **THE GADGET STRUCTURE**

To hold the icon's image, Workbench uses an Intuition **Gadget** structure, defined in *intuition/intuition.h*. Workbench restricts some of the values of the gadget. All unused fields should be set to 0 or NULL.

#### **NOTE**

The assembly version of the Gadget structure has leading "gg\_" for each variable name.

The Intuition gadget structure members that Workbench uses are listed below:

#### **Width**

This is the width (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

#### **Height**

This is the height (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

#### **Flags**

The gadget *must* be of type **GADGIMAGE**. Three highlight modes are supported: **GADGHCOMP**, **GADGHIMAGE**, and **GADGBACKFILL**. **GADGHCOMP** complements everything within the area defined by **CurrentX**, **CurrentY**, **Width**, **Height**. **GADGHIMAGE** uses an alternate selection image. **GADGBACKFILL** is similar to **GADGHCOMP**, but ensures that there is no "ring" around the selected image. It does this by first complementing the image, and then flooding all color 3 pixels that are on the border of the image to color 0. All other flag bits should be 0.

#### **Activation**

The activation should have only **RELVERIFY** and **GADGIMMEDIATE** set.

#### **Type**

The gadget type should be **BOOLGADGET**.

### **GadgetRender**

Set this to an appropriate **Image** structure.

### **SelectRender**

Set this to an appropriate alternate **Image** structure if and only if the highlight mode is GADGHIMAGE.

The **Image** structure is typically the same size as the gadget, except that **Height** is often one pixel less than the gadget height. This allows a blank line between the icon image and the icon name. The image depth *must* be 2; **PlanePick** *must* be 3; and **PlaneOnOff** should be 0. The **NextImage** field should be null.

## **ICONS WITH NO POSITION**

Picking a position for a newly created icon can be tricky. **NO\_ICON\_POSITION** is a magic value for **do\_CurrentX** and **do\_CurrentY** that instructs Workbench to pick a reasonable place for the icon. Workbench will place the icon in an unused region of the drawer. If there is no space in the drawers window, the icon will be placed just to the right of the visible region.

## **Workbench Environment**

When a user activates a tool or project, Workbench runs a program. This program is a separate process and runs asynchronously to Workbench. This allows the user to take advantage of the multi-tasking features of the Amiga.

The environment for a tool under the Workbench is quite different from the environment when a tool is run from the CLI. The CLI does not create a new process for a program; it jumps to the program's code and the program shares the process with the CLI. Programs run under the CLI have access to all the CLI's environment, including the ability to modify that environment. Programs run under the CLI should be careful to restore all values that existed on startup. Workbench starts a tool as a new DOS process, explicitly passing the environment to the tool.

By default, a Workbench program does not have a window to which its output will go. Therefore, **stdin** and **stdout** do not point to legal file handles. **Note:** If your program attempts to read from **stdin** or write to **stdout** (including calls such as **printf( )**), without first setting them up it may crash the system. Some compilers have options or defaults to provide a **stdio** window for programs started from Workbench. The 1.3 Amiga startup.asm code can also provide a **stdio** window for Workbench programs, for use with the **amiga.lib** **stdio** functions. As always, remember to close or deallocate any resources that are opened by your program.

## **WBSTARTUP MESSAGE**

Right after Workbench loads and starts a tool, Workbench sends the tool a **WBStartup** message which is posted to the message port in the tool's process structure.

### **NOTE**

This is the only valid external use of a process's message port.

The process message port is for the exclusive use of DOS, and this message must be removed from the port by the tool's startup code prior to opening **dos.library**. The **WBStartup** message contains the environment and the Workbench arguments for the tool.

Workbench arguments are passed as the **sm\_ArgList** array of pointers to **WBArg** structures. The first **WBArg** in the list is always the tool itself. If multiple icons have been selected when a tool is activated, the selected icons are passed to the tool as additional **WBArgs**. If the tool was derived from a default tool, the project will be the second **WBArg**. Arguments other than the tool are passed in order of selection; the first icon selected will be first (after the tool), and so on.

Applications that are started from Workbench (as signified by a null **pr\_CLI** pointer in their **Process** structure) must remove (**GetMsg( )**) the **WBStartup** message from their **Process** message port before opening **dos.library**. This action is generally performed by the startup code which is linked as the first module of an application. Generally, your compiler will provide suitable startup code (for example **Lattice c.o.**).

Startup code for C code generally passes the **WBStartup** message pointer in **argv**, and 0 (zero) in **argc**, when the program has been started from Workbench. Startup code usually calls your application code as a function. When your application returns or exits to the startup code, the startup module will **Forbid( )**, and **ReplyMsg( )** the **WBStartup** message, notifying Workbench that the application process may be terminated, and its code unloaded from memory.

### NOTE

The **DOS Exit( )** function will NOT return you to the startup code. If you wish to exit your application, use the exit function provided by your startup code (usually lower-case **exit**, or **\_exit** for assembler), passing it a valid dos return code (*libraries/dos.h*).

The **WBStartup** message, whose structure is outlined in *workbench/startup.h*, has the following structure elements:

```
struct WBStartup
{
    struct Message      sm_Message;      /* a standard message structure */
    struct MsgPort *    sm_Process;      /* the process descriptor for you */
    BPTR               sm_Segment;      /* a descriptor for your code */
    LONG               sm_NumArgs;      /* the number of elements in ArgList */
    char *              sm_ToolWindow;  /* reserved for future use */
    struct WBArg *      sm_ArgList;     /* the arguments themselves */
};

struct WBArg
{
    BPTR               wa_Lock;          /* a lock descriptor */
    BYTE *             wa_Name;          /* a string relative to that lock */
};
```

#### **sm\_Message**

A standard Exec message. The reply port is set to the Workbench.

#### **sm\_Process**

The process descriptor for the tool (as returned by **CreateProcess( )**)

#### **sm\_Segment**

The loaded code for the tool (returned by **LoadSeg( )**)

#### **sm\_NumArgs**

The number of arguments in **sm\_ArgList**

#### **sm\_ToolWindow**

Reserved (not currently passed in startup message)

### **sm\_ArgList**

This is the argument list itself. It is a pointer to an array of **WBArg** structures with **sm\_NumArgs** elements.

Each argument is a struct **WBArg** and has two parts: **wa\_Name** and **wa\_Lock**.

The **wa\_Name** element is the name of the argument. If this is not a directory object (drawer, disk, or Trashcan) or a default tool, the **wa\_Name** will be the same as the string displayed under the icon. A default tool will instead have the text of the **do\_DefaultTool** pointer; a directory object will have a null name passed. The default tool argument will only appear as the first argument, and only when the project was activated (not the tool).

The **wa\_Lock** is always a lock on a directory, or is NULL if that object type does not support locks.

NOTE - You must never **UnLock** a **wa\_Lock**. These locks belong to Workbench, and Workbench will **UnLock** them when the **WBStartup** message is replied. You also must never **UnLock** your program's initial current directory lock (ie. the lock returned by an initial **CurrentDir( )** call). The classic symptom caused by unlocking Workbench locks is a system hang after your Workbench program exits, even though the same program exits with no problems when started from CLI. You should save the lock returned from an initial **CurrentDir( )**, and **CurrentDir( )** back to it before exiting. In the Workbench environment, depending on your startup module, the current directory will generally be set to one of the **wa\_Locks**. By using **CurrentDir(wa\_Lock)** and then referencing **wa\_Name**, you can find, read, and modify the files which have been passed as **WBArgs**. The example code "PrArgs.c" at the end of this chapter demonstrates the handling of Workbench and CLI arguments.

## **The ToolTypes Array**

This section shows how the **ToolTypes** array should be formatted, and describes the standard entries in the **ToolTypes** array. In brief, **ToolTypes** is an array of pointers to strings. These strings can be used to encode information about the icon that will be available to all who wish to use it. The formats are user-definable and user-extensible.

Workbench does not place many restrictions on the **ToolTypes** array, but some conventions are strongly encouraged. A string may be up to 128 bytes long. The alphabet is 8-bit ANSI (for example, normal ASCII with foreign-language extensions). This means that users may enter **ToolType** strings containing international characters. Avoid special or nonprinting characters. The case of the characters is currently significant, so "Window" is not equal to "WINDOW".

The general format is

**<name>=<value>[<value>]**

where **<name>** is the field name and **<value>** is the text to associate with that name. If the ID has multiple values, the values may be separated by a vertical bar. The values may be the type of the file, programs that can access the data, parameters to be passed to an application, etc. For example, a paint program might set:

**FILETYPE=PaintProgram.file|ILBM**

This notifies the world that this file is acceptable to either a program that is expecting a generic ILBM IFF file, or to a program that understands the format of PaintProgram files.

Two routines are provided to help you deal with the **Tooltype** array. **FindToolType( )** returns the value of a **Tooltype** element. Using the above example, if you are looking for FILETYPE, the string "PaintProgram.file|ILBM" will be returned.

**MatchToolValue( )** returns nonzero if the specified string is in the reference value string. This routine knows how to parse vertical bars. For example, using the reference value string of "PaintProgram.file|ILBM", **MatchToolValue( )** will return TRUE for "ILBM" and "PaintProgram.file" and FALSE for everything else.

## Example Code

### PrArgs.c

The following example will display all WBArgs if started from Workbench, and all CLI arguments if started from CLI.

```
/* PrArgs.c - This program prints its Workbench or CLI arguments.
** Compiled with lattice c 5.04. Works under workbench and CLI.
** 'tinymain' statement turns off default stdin/stdout handling.
**
** lc -bl -cfist -v -y prargs.c
** blink FROM LIB:c.o prargs.o
**      LIB LIB:lc.lib LIB:amiga.lib
**      TO prargs
**      DEFINE __main=__tinymain
**
** NOTE: main and tinymain are prepended with two underscores.
*/
#include <workbench/startup.h>
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
/*-----
** disable lattice CTRL-C handling
*/
int CXBRK(VOID)
{
    return(0);
}
/*-----
** program to print arguments,
** works if run from the CLI or WORKBENCH.
*/
void main(int argc, char **argv)
{
    struct WBStartup *argmsg;
    struct WBArg *wb_arg;
    LONG ktr;
    BPTR olddir;
    FILE *outFile;

    /* argc is zero when run from the Workbench,
    **      positive when run from the CLI.
    */
    if (argc == 0)
    {
```

```

if (NULL != (outFile = fopen("CON:0/0/640/200/Print Args","r+")))
{
    /* in lattice, argv is a pointer to the WBStartup message
    ** when argc is zero. (run under the Workbench.)
    */
    argmsg = (struct WBStartup *)argv ;
    wb_arg = argmsg->sm_ArgList ;      /* head of the arg list */

    fprintf(outFile, "Run from the workbench, %ld args.\n",
            argmsg->sm_NumArgs);

    for (ktr = 0; ktr < argmsg->sm_NumArgs; ktr++, wb_arg++)
    {
        if (NULL != wb_arg->wa_Lock)
        {
            /* locks supported, change to the proper directory */
            olddir = CurrentDir(wb_arg->wa_Lock) ;

            /* process the file.
            ** if you have done the CurrentDir( ) above,
            ** then you can access the file by its name.
            ** otherwise, you have to look at the lock to get
            ** a complete path to the file.
            */
            fprintf(outFile, "\tArg %2.2ld (w/ lock): '%s'.\n",
                    ktr, wb_arg->wa_Name);

            /* change back to the original directory when done.
            ** be sure to change back before you exit.
            */
            CurrentDir(olddir) ;
        }
        else
        {
            /* something that does not support locks */
            fprintf(outFile, "\tArg %2.2ld (no lock): '%s'.\n",
                    ktr, wb_arg->wa_Name);
        }
    }

    /* wait before closing down */
    Delay(500L);
    fclose(outFile);
}

else
{
    /* using 'tinymain' from lattice c.
    ** define a place to send the output (originating CLI window = "")
    ** Note - if you open "" and your program is RUN, the user will not
    ** be able to close the CLI window until you close the "" file.
    */
    if (NULL != (outFile = fopen("", "r+")))
    {
        fprintf(outFile, "Run from the CLI, %d args.\n", argc);

        for ( ktr = 0; ktr < argc; ktr++)
        {
            /* print an arg, and its number */
            fprintf(outFile, "\tArg %2.2ld: '%s'.\n", ktr, argv[ktr]);
        }
        fclose(outFile);
    }
}
}

```

## RKM\_Icon\_Example.c

The following example demonstrates icon creation, icon reading and ToolType parsing, and the Workbench environment. When called from CLI, the example creates a small data file in RAM: and creates or updates a Project icon for the data file. The created Project icon points to this example as its default tool. When the new Project icon is double-clicked, Workbench will invoke the default tool (this example) as a Workbench process, and pass it a description of the Project data file as a Workbench argument (WBArg) in the WBStartup message.

```
/* RKM_Icon_Example.c - Workbench icon startup, creation, and parsing example
 *
 * Compiled with Lattice 5.02: LC -bl -cfist -v -y
 * Linkage: c.o,RKM_Icon_Example.o library LC.lib,amiga.lib
 */
```

```
#include <exec/types.h>
#include <libraries/dos.h>
#include <workbench/workbench.h>
#include <workbench/startup.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif
```

```
/* our functions */
void cleanexit(UBYTE *,LONG);
void cleanup(void);
void message(UBYTE *);
BOOL makeIcon(UBYTE *, char **, char *);
BOOL showToolTypes(struct WBArg *);
```

```
UBYTE *projname = "RAM:Example_Project";
UBYTE *conwinname = "CON:10/10/620/180/RKM_Icon_Example";
```

```
char deftoolname[] = "RKM_Icon_Example";
```

```
USHORT IconImageData1[] =
```

```
{
/* Plane 0 */
0x0000,0x0000,0x0000,0x3fff,0xffcc,0x0000,0x3fff,0xffcf,
0x0000,0x3fff,0xffcf,0xc000,0x3803,0xffcf,0xf000,0x3fff,
0xffc0,0x0000,0x3803,0xffff,0xfc00,0x3fff,0xffff,0xfc00,
0x3fff,0xffff,0xfc00,0x3f84,0x00c0,0x7c00,0x3fff,0xffff,
0xfc00,0x3900,0x8000,0x7c00,0x3fff,0xffff,0xfc00,0x3800,
0x0040,0x7c00,0x3fff,0xffff,0xfc00,0x3fff,0xffff,0xfc00,
0x3fff,0xfe00,0x7c00,0x3fff,0xffff,0xfc00,0x3fff,0xffff,
0xfc00,0x0000,0x0000,0x0000,
/* Plane 1 */
0xffff,0xfffc,0x0000,0xc000,0x0033,0x0000,0xc000,0x0030,
0xc000,0xc000,0x0030,0x3000,0xc7fc,0x0030,0x0c00,0xc000,
0x003f,0xff00,0xc7fc,0x0000,0x0300,0xc000,0x0000,0x0300,
0xc000,0x0000,0x0300,0xc07b,0xff3f,0x8300,0xc000,0x0000,
0x0300,0xc6ff,0x7fff,0x8300,0xc000,0x0000,0x0300,0xc7ff,
0xffbf,0x8300,0xc000,0x0000,0x0300,0xc000,0x0000,0x0300,
0xc000,0x01ff,0x8300,0xc000,0x0000,0x0300,0xc000,0x0000,
0x0300,0xffff,0xffff,0xff00,
};
```

```
struct Image iconImage1 =
{
0, 0, /* Top Corner */
40, 20, 2, /* Width, Height, Depth */
};
```

```

&IconImageData1[0], /* Image Data */
0x003, 0x000, /* PlanePick,PlaneOnOff */
NULL /* Next Image */
};

UBYTE *toolTypes[] =
{
"FILETYPE=text",
"FLAGS=BOLD|ITALICS",
NULL
};

struct DiskObject projIcon =
{
WB_DISKMAGIC, /* Magic Number */
WB_DISKVERSION, /* Version */
{
/* Embedded Gadget Structure */
NULL, /* Next Gadget Pointer */
97,12,40,21, /* Left,Top,Width,Height */
GADGIMAGE|GADGHBOX, /* Flags */
GADGIMMEDIATE|RELVERIFY, /* Activation Flags */
BOOLGADGET, /* Gadget Type */
(APTR)&iconImage1, /* Render Image */
NULL, /* Select Image */
NULL, /* Gadget Text */
NULL, /* Mutual Exclude */
NULL, /* Special Info */
0, /* Gadget ID */
NULL /* User Data */
},
WBPROJECT, /* Icon Type */
deftoolname, /* Default Tool */
toolTypes, /* Tool Type Array */
NO_ICON_POSITION, /* Current X */
NO_ICON_POSITION, /* Current Y */
NULL, /* Drawer Structure */
NULL, /* Tool Window */
4000 /* Stack Size */
};

/* Opens and allocations we must clean up */
struct Library *IconBase = NULL;
FILE *conwin = NULL;
LONG olddir = -1;

BOOL FromWb;

void main(argc, argv)
int argc;
char **argv;
{
struct WBStartup *WBenchMsg;
struct WBArg *wbarg;
FILE *file;
LONG wlen;
SHORT i;

FromWb = (argc==0) ? TRUE : FALSE;

/* Open icon.library */
if(!IconBase = OpenLibrary("icon.library",33))
cleanexit("Can't open icon.library\n",RETURN_FAIL);

/* If started from cli, this example will create
* a small text file RAM:Example_Project, and
* create an icon for the file which points
* to this program as its default tool.
*/
if(!FromWb)
{
/* Make a sample project (data) file */

```



```

wLen = -1;
if(file=fopen(projname,"w"))
{
    wLen = fprintf(file,"Have a nice day\n");
    fclose(file);
}
if(wLen < 0) cleanexit("Error writing data file\n",RETURN_FAIL);

/* Now save/update icon for this data file */
if(makeIcon(projname, toolTypes, deftoolname))
{
    printf("%s data file and icon saved.\n",projname);
    printf("Use Workbench Info to examine the icon.\n");
    printf("Then copy this example (RKM_Icon_Example) to RAM:\n");
    printf("and double-click the %s project icon\n",projname);
}
else cleanexit("Error writing icon\n",RETURN_FAIL);
}

else /* Else we are FromWb - ie. we were either
    * started by a tool icon, or as in this case,
    * by being the default tool of a project icon.
    */
{
    if(!(conwin = fopen(conwinname,"r+")))
        cleanexit("Can't open output window\n",RETURN_FAIL);

    WBenchMsg = (struct WBStartup *)argv;

    /* Note wbarg++ at end of FOR statement steps through wbargs.
    * First arg is our executable (tool). Any additional args
    * are projects/icons passed to us via either extend select
    * or default tool method.
    */
    for(i=0, wbarg=WBenchMsg->sm_ArgList;
        i < WBenchMsg->sm_NumArgs;
        i++, wbarg++)
    {
        /* if there's a directory lock for this wbarg, CD there */
        olddir = -1;
        if((wbarg->wa_Lock)&&(*wbarg->wa_Name))
            olddir = CurrentDir(wbarg->wa_Lock);

        showToolTypes(wbarg);

        if((i>0)&&(*wbarg->wa_Name))
            fprintf(conwin,"In Main. We could open the %s file here\n",
                wbarg->wa_Name);
        if(olddir != -1) CurrentDir(olddir); /* CD back where we were */
    }
    Delay(500);
}
cleanup();
exit(RETURN_OK);
}

BOOL makeIcon(UBYTE *name, char **newtooltypes, char *newdeftool)
{
    struct DiskObject *dobj;
    char *olddeftool;
    char **oldtooltypes;
    BOOL success = FALSE;

    if(dobj=GetDiskObject(name))
    {
        /* If file already has an icon, we will save off any fields we
        * need to update, update those fields, put the object, restore
        * the old field pointers and then free the object. This will
        * preserve any custom imagery the user has, and the user's
        * current placement of the icon. If your application does
        * not know where the user currently keeps your application,
        * you should not update his dobj->do_DefaultTool.

```

```

    */
    oldtooltypes = dobj->do_ToolTypes;
    olddeftool = dobj->do_DefaultTool;

    dobj->do_ToolTypes = newtooltypes;
    dobj->do_DefaultTool = newdeftool;

    success = PutDiskObject(name,dobj);

    /* we must restore the original pointers before freeing */
    dobj->do_ToolTypes = oldtooltypes;
    dobj->do_DefaultTool = olddeftool;
    FreeDiskObject(dobj);
}
/* Else, put our default icon */
if(!success) success = PutDiskObject(name,&projIcon);
return(success);
}

BOOL showToolTypes(struct WBArg *wbarg)
{
    struct DiskObject *dobj;
    char **toolarray;
    char *s;
    BOOL success = FALSE;

    fprintf(conwin,"\nWBArg Lock=0x%lx, Name=%s\n",
            wbarg->wa_Lock,wbarg->wa_Name);

    if((*wbarg->wa_Name) && (dobj=GetDiskObject(wbarg->wa_Name)))
    {
        fprintf(conwin," We have read the DiskObject (icon) for this arg\n");
        toolarray = (char **)dobj->do_ToolTypes;

        if(s=(char *)FindToolType(toolarray,"FILETYPE"))
        {
            fprintf(conwin," Found tooltype FILETYPE with value %s\n",s);
        }
        if(s=(char *)FindToolType(toolarray,"FLAGS"))
        {
            fprintf(conwin," Found tooltype FLAGS with value %s\n",s);
            if(MatchToolValue(s,"BOLD"))
                fprintf(conwin," BOLD flag requested\n");
            if(MatchToolValue(s,"ITALICS"))
                fprintf(conwin," ITALICS flag requested\n");
        }
        /* Free the diskobject we got */
        FreeDiskObject(dobj);
        success = TRUE;
    }
    else if(!(*wbarg->wa_Name))
        fprintf(conwin," Must be a disk or drawer icon\n");
    else
        fprintf(conwin," Can't find DiskObject (icon) for this WBArg\n");
    return(success);
}

/* Workbench-started programs with no output window
 * will want to display messages in a different manner
 * (requester, window title, etc) if FromWb is TRUE.
 */
void message(UBYTE *s)
{
    if(FromWb && conwin) fprintf(conwin,s,strlen(s));
    else if (!FromWb) printf(s);
}

void cleanexit(UBYTE *s, LONG n)
{
    if(*s) message(s);
    cleanup();
}

```

```

    exit(n);
}

void cleanup()
{
    if(conwin)      fclose(conwin);
    if(IconBase)   CloseLibrary(IconBase);
}

```

## PROGRAM STARTUP CODE

Standard start-up code handles the detail work of interfacing with the arguments and environment of CLI and Workbench. When a program is started from CLI (or a script), standard startups will parse the command line arguments (received in A0, with length in D0) to properly split the line into an array of pointers to individual command line arguments (argv), and an argc argument count. Argc will equal at least one if a program is started from CLI because the first argv element will be a pointer to the typed command name. For example, if the command line was:

```
df0:myprogram "my file1" file2 ;this is a comment
```

then argc will be 3, argv[0] will be "df0:myprogram", argv[1] will be "my file1", and argv[2] will be "file2". Correct startup code will strip spaces between arguments and trailing spaces from the last argument (file2), and will also properly deal with quoted arguments and embedded spaces. Standard CLI startup will usually also set up SysBase, DOSBase, and stdio file handles (\_stdin, \_stdout, etc.) for the application. Argv, then argc, will be pushed on the stack and the application will be called via a JSR. When the application returns or exits back to the startup code, the startup closes or frees all opens and allocations it has made for the application, and then returns to the system with the program exit(n) value.

When a program is started from Workbench, a standard startup will wait for, and get the WBStartup message. The startup code will usually set up SysBase and DOSBase for the application, and some special startups may open a stdio window for the application, or NIL: input and output streams. A pointer to the WBStartup message (argv) and an argc of 0 are pushed onto the stack, and the application is called via a JSR. When the application returns or exits back to the startup code from Workbench, the startup closes or frees all opens and allocations it has made for the application, calls Forbid( ), replies to the WBStartup message, and returns to the system with the program exit(n) value. Workbench then terminates the application process and unloads its code.

It is strongly suggested that all programmers use standard tested startup code of some type. Even assembler programs can use a startup such as the standard Amiga startup, receiving argc and argv on their stack as described above, and exiting through the startup's \_exit label with their error code on the stack. Startup code can provide your programs with correct consistent handling of CLI and Workbench arguments, and will perform some initializations and cleanups which would otherwise need to be handled by your own code. Very small startups can be used for programs that require no command line arguments.

A few words of warning for those of you who do not use standard startup code:

You *must* GetMsg( ) the WBStartup message before opening dos.library.

You *must* turn off task switching (with Forbid( )) before replying the message to Workbench. This will prevent Workbench from unloading your code before you can tell the DOS that you want to exit.

If you do your own command line parsing, you *must* provide the user with consistent and correct handling of command line arguments.

## Standard Amiga Startup Source

The following code is the standard 1.3 Amiga startup code module. By using the appropriate .i file (printed at end of the module) this startup.asm can be assembled into a variety of startups including reentrant versions, and versions which can provide an application-defined amiga.lib stdio window when started from Workbench.

### NOTE

This startup is Amiga-specific and uses direct AmigaDOS file handles for stdio (amiga.lib printf, getchar, etc.). It is designed for use with assemblers and compilers when linking with amiga.lib as the first linker lib, and using only amiga.lib direct AmigaDOS stdio and fileio.

```
*----- startup.asm v 34.12 Copyright 1988 Commodore-Amiga, Inc.
*-----
*----- Conditional assembly flags
*----- ASTART: 1=Standard Globals Defined      0=Reentrant Only
*----- WINDOW: 1=AppWindow for WB startup      0=No AppWindow code
*----- XNIL: 1=Remove startup NIL: init        0=Default Nil: WB Output
*----- NARGS: 1=Argv[0] only                  0=Normal cmd line arg parse
*----- DEBUG: 1=Set up old statics for Wack    0=No extra statics
*----- QARG: 1=No argv                        0=Passes argc,argv
```

```
* Include the appropriate .i file to set the flags
```

```
INCLUDE "astartup.i"
```

* Flags for	[A]start	AWstart	Rstart	RWstart	RXstart	QStart
* ASTART	1	1	0	0	0	0
* WINDOW	0	1	0	1	0	0
* XNIL	0	0	0	0	1	1
* NARGS	0	0	0	0	0	0
* DEBUG	0	0	0	0	0	0
* QARG	0	0	0	0	0	1

```
;----- Flag WB output initialization
WBOUT SET (ASTART!WINDOW!(1-XNIL))
```

```
*****
```

```
*
* startup.asm --- Reentrant C Program Startup/Exit (CLI and WB)
* v34.12 07/25/88
*
* Copyright (c) 1988 Commodore-Amiga, Inc.
*
* Title to this software and all copies thereof remain vested in the
* authors indicated in the above copyright notice. The object version
* of this code may be used in software for Commodore Amiga computers.
* All other rights are reserved.
*
* NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE
* ACCURACY, RELIABILITY, PERFORMANCE OR OPERATION OF THIS SOFTWARE,
* AND ALL SUCH USE IS AT YOUR OWN RISK. NEITHER COMMODORE NOR THE
* AUTHORS ASSUME ANY RESPONSIBILITY OR LIABILITY WHATSOEVER WITH
* RESPECT TO YOUR USE OF THIS SOFTWARE.
*
```

```
*
* RSTARTUP.ASM
*
```

```
* This startup dynamically allocates a structure which includes
* the argv buffers. If you use this startup, your code must return
* to this startup when it exits. Use exit(n) or final curly brace
* (rts) to return here. Do not use AmigaDOS Exit( ) function.
* Due to this dynamic allocation and some code consolidation, this
* startup can make executables several hundred bytes smaller.
*
```

```

*      Because a static initialSP variable can not be used, this
*      code depends on the fact that AmigaDOS places the address of
*      the top of our stack in SP and proc->pr_ReturnAddr right before
*      JSR'ing to us. This code uses pr_ReturnAddr when restoring SP.
*
*      Most versions of startup will initialize a Workbench process's
*      input and output streams (and stdio globals if present) to NIL:
*      if no other form of Workbench output (like WINDOW) is provided.
*      This should help prevent crashes if a user puts an icon on a CLI
*      program, and will also protect against careless stdio debugging
*      or error messages left in a Workbench program. The code for
*      initializing Workbench IO streams only be removed by assembling
*      startup with ASTART and WINDOW set to 0, and XNIL set to 1.
*
*
*      Some startups which can be conditionally assembled:
*
*      1. Standard Astartup for non-reentrant code
*      2. Reentrant Rstartup (no unshareable globals)
*      3. Smaller reentrant-only RXstartup (no NIL: WB init code)
*      4. Standard AWstartup (WB output window) for non-reentrant code
*      5. Reentrant RWstartup (WB output window, no unshareable globals)
*      6. Smallest Qstartup (No argv - argv is ptr to NULL string)
*
*      Explanation of conditional assembly flags:
*
*      ASTART (ASTART SET 1) startups will set up and XDEF the
*      global variables _stdin, _stdout, _stderr, _errno and _WBenchMsg.
*      These startups can be used as smaller replacements for startups
*      like (A)startup.obj and TWstartup.obj. Startups with ASTART
*      would generally be used for non-reentrant programs, although the
*      startup code itself is still reentrant if the globals are not
*      referenced.
*
*      Reentrant (ASTART SET 0) startups will NOT set up or
*      XDEF the stdio and WBenchMsg globals. This not only makes the
*      startup slightly smaller, but also lets you know if your code
*      is referencing these non-reentrant globals (you will get an
*      unresolved external reference when you link). Programs
*      get their input and output handles from Input( ) and Output( ),
*      and the WBenchMsg is passed in argv on Workbench startup.
*
*      WINDOW (WINDOW SET 1) startups use an XREF'd CON: string
*      named AppWindow, defined in your application, to open a stdio
*      console window when your application is started from Workbench.
*      For non-reentrant programs, this window can be used for normal
*      stdio (printf, getchar, etc). For reentrant programs the window
*      is Input( ) and Output( ). WINDOW is useful when adding Workbench
*      capability to a stdio application, and also for debugging other
*      Workbench applications. To insure that applications requiring
*      a window startup are linked with a window startup, the label
*      _NeedWStartup can be externed and referenced in the application
*      so that a linker error will occur if linked with a standard
*      startup.
*
*      example:  /* Optional safety reference to NeedWStartup */
*                extern UBYTE NeedWStartup;
*                UBYTE *HaveWStartup = &NeedWStartup;
*                /* Required window specification */
*                char AppWindow[] = "CON:30/30/200/150/MyProgram";
*                ( OR char AppWindow[] = "\0"; for no window )
*
*      XNIL (XNIL SET 1) allows the creation of a smaller startup
*      by removing the code that initializes a Workbench process's
*      output streams to NIL:. This flag can only remove the code
*      if it is not required for ASTART or WINDOW.
*
*      NARGS (NARGS SET 1) removes the code used to parse command line
*      arguments. The command name is still passed to _main as argv[0].
*      This option can take about 120 bytes off the size of any program that
*      does not use command line args.

```

```

*
*   DEBUG (DEBUG SET 1) will cause the old startup.asm statics
*   initialSP, dosCmdLen and dosCmdBuf to be defined and initialized
*   by the startup code, for use as debugging symbols when using Wack.
*
*   QARG (QARG SET TO 1) will bypass all argument parsing. A CLI
*   startup is passed argc == 1, and a Workbench startup is passed
*   argc == 0. Argv[0] will be a pointer to a NULL string rather than
*   a pointer to the command name. This option creates a very small
*   startup with no sVar structure allocation, and therefore must be used
*   with XNIL (it is incompatible with default or AWindow output options).
*
*
*   RULES FOR REENTRANT CODE
*
*   - Make no direct or indirect (printf, etc) references to the
*     globals _stdin, _stdout, _stderr, _errno, or _WBenchMsg.
*
*   - For stdio use either special versions of printf and getchar
*     that use Input( ) and Output( ) rather than _stdin and _stdout,
*     or use fprintf and fgetc with Input( ) and Output( ) file handles.
*
*   - Workbench applications must get the pointer to the WBenchMsg
*     from argv rather than from a global extern WBenchMsg.
*
*   - Use no global or static variables within your code. Instead,
*     put all former globals in a dynamically allocated structure, and
*     pass around a pointer to that structure. The only acceptable
*     globals are constants (message strings, etc) and global copies
*     of Library Bases to resolve Amiga.lib references. Your code
*     must return all OpenLibrary's into non-global variables,
*     copy the result to the global library base only if successful,
*     and use the non-globals when deciding whether to Close any
*     opened libraries.
*
*****
***** Included Files *****
*****
INCLUDE "exec/types.i"
INCLUDE "exec/alerts.i"
INCLUDE "exec/memory.i"
INCLUDE "libraries/dos.i"
INCLUDE "libraries/dosextens.i"
INCLUDE "workbench/startup.i"
*****
***** Macros *****
*****
xlib    macro
xref    _LVO\1
endm

callsys macro
CALLLIB _LVO\1
endm

***** Imported *****
*****
ABSEXECSBASE    EQU    4

xref    _main          ; C code entry point

IFGT     WINDOW
xref    _AppWindow     ; CON: spec in application for WB stdio window
xdef     _NeedWStartup  ; May be externed and referenced in application
ENDC     WINDOW

xlib     Alert
xlib     AllocMem
xlib     FindTask

```

```

xlib    Forbid
xlib    FreeMem
xlib    GetMsg
xlib    OpenLibrary
xlib    CloseLibrary
xlib    ReplyMsg
xlib    Wait
xlib    WaitPort

xlib    CurrentDir
xlib    Open
xlib    Close
xlib    Input
xlib    Output

***** Exported *****

*----- These globals are set up for standard startup code only
IFGT    ASTART
xdef    _stdin
xdef    _stdout
xdef    _stderr
xdef    _errno
xdef    _WBenchMsg
ENDC    ASTART

*----- These globals available to normal and reentrant code

xdef    _SysBase
xdef    _DOSBase
xdef    _exit          ; standard C exit function

***** Startup Variables structure *****

IFEQ    QARG
ARGVSLOTS EQU    32

STRUCTURE SVar,0
LONG     sv_WbOutput
STRUCT   sv_argvArray,ARGVSLOTS*4
STRUCT   sv_argvBuffer,256
LABEL    SV_SIZEOF
ENDC     QARG

*****
*   Standard Program Entry Point
*****
*
*   Entered with
*       d0 dosCmdLen
*       a0 dosCmdBuf
*   Any registers (except sp) are allowed to be modified
*
*   Calls
*       main (argc, argv)
*       int  argc;
*       char *argv[];
*
*   For Workbench startup, argc=0, argv=WBenchMsg
*
*****
startup:
IFGT    DEBUG
move.l  sp,initialSP
move.l  d0,dosCmdLen
move.l  a0,dosCmdBuf
ENDC    DEBUG

IFEQ    QARG
move.l  d0,d2
move.l  a0,a2

```

```

ENDC    QARG

;----- get Exec library base pointer
movea.l ABSEXECSBASE,a6
move.l  a6,_SysBase

;----- get the address of our task
suba.l  a1,a1          ; clear a1
callsys FindTask
move.l  d0,a4          ; keep task address in a4

;----- get DOS library base pointer
moveq   #0,d0
lea     DOSName(pc),A1 ; dos.library
callsys OpenLibrary

tst.l   d0
beq     alertDOS        ; fail on null with alert
move.l  d0,_DOSBase     ; Else set the global

IFEQ    QARG
;----- alloc the argument structure
move.l  #SV_SIZEOF,d0
move.l  #(MEMF_PUBLIC!MEMF_CLEAR),d1
callsys AllocMem
tst.l   d0
beq     alertMem        ; fail on null with alert
move.l  d0,-(sp)        ; save sVar ptr on stack
move.l  d0,a5           ; sVar ptr to a5
ENDC
IFGT    QARG
clr.l   -(sp)
ENDC
QARG

clr.l   -(sp)          ; reserve space for WBenchMsg if any

;----- branch to Workbench startup code if not a CLI process
move.l  pr_CLI(A4),d0
beq     fromWorkbench

;=====
;===== CLI Startup Code =====
;=====
;
; d0 process CLI BPTR (passed in), then temporary
; d2 dos command length (passed in)
; d3 argument count
; a0 temporary
; a1 argv buffer
; a2 dos command buffer (passed in)
; a3 argv array
; a4 Task (passed in)
; a5 SVar structure if not QARG (passed in)
; a6 AbsExecBase (passed in)
; sp WBenchMsg (still 0), sVar or 0, then RetAddr (passed in)
; sp argc, argv, WBenchMsg, sVar or 0, RetAddr (at bra domain)

IFEQ    QARG
;----- find command name
lsl.l   #2,d0           ; pr_CLI bcpl pointer conversion
move.l  d0,a0
move.l  cli_CommandName(a0),d0
lsl.l   #2,d0           ; bcpl pointer conversion

;-- start argv array
lea     sv_argvBuffer(a5),a1
lea     sv_argvArray(a5),a3

;-- copy command name
move.l  d0,a0
moveq.l #0,d0
move.b  (a0)+,d0        ; size of command name

```



```

        clr.b    0(a0,d0.l)      ; terminate the command name
        move.l   a0,(a3)+
        moveq    #1,d3           ; start counting arguments

IFEQ    NARGS
;----- null terminate the arguments, eat trailing control characters
        lea      0(a2,d2.l),a0
stripjunk:
        cmp.b    #' ',-(a0)
        dbhi     d2,stripjunk

        clr.b    1(a0)

;----- start gathering arguments into buffer
newarg:
        ;-- skip spaces
        move.b    (a2)+,d1
        beq.s     parmExit
        cmp.b     #' ',d1
        beq.s     newarg
        cmp.b     #9,d1          ; tab
        beq.s     newarg

        ;-- check for argument count overflow
        cmp.w     #ARGVSLOTS-1,d3
        beq.s     parmExit

        ;-- push address of the next parameter
        move.l    a1,(a3)+
        addq.w     #1,d3

        ;-- process quotes
        cmp.b     #'"',d1
        beq.s     doquote

        ;-- copy the parameter in
        move.b     d1,(a1)+

nextchar:
        ;----- null termination check
        move.b     (a2)+,d1
        beq.s     parmExit
        cmp.b     #' ',d1
        beq.s     endarg

        move.b     d1,(a1)+
        bra.s     nextchar

endarg:
        clr.b     (a1)+
        bra.s     newarg

doquote:
;----- process quoted strings
        move.b     (a2)+,d1
        beq.s     parmExit
        cmp.b     #'"',d1
        beq.s     endarg

        ;-- '*' is the BCPL escape character
        cmp.b     #'*',d1
        bne.s     addquotechar

        move.b     (a2)+,d1
        move.b     d1,d2
        and.b     #$df,d2        ;d2 is temp toupper'd d1

        cmp.b     #'N',d2        ;check for dos newline char
        bne.s     checkEscape

        ;-- got a *N -- turn into a newline
        moveq     #10,d1

```

```

        bra.s    addquotechar

checkEscape:
        cmp.b    #'E',d2
        bne.s    addquotechar

        ;--      got a *E -- turn into a escape
        moveq     #27,d1

addquotechar:
        move.b    d1,(a1)+
        bra.s     doquote

parmExit:
        ;----- all done -- null terminate the arguments
        clr.b     (a1)
        clr.l     (a3)
        ENDC      NARGS

        pea       sv_argvArray(a5) ; argv
        move.l     d3,-(sp)          ; argc
        ENDC      QARG

        IFGT      QARG
        pea       nullArgV(pc)      ; pointer to pointer to null string
        pea       1                  ; only one pointer
        ENDC

        IFGT      ASTART
        movea.l     _DOSBase,a6
        ;----- get standard input handle:
        callsys     Input
        move.l     d0,_stdin

        ;----- get standard output handle:
        callsys     Output
        move.l     d0,_stdout
        move.l     d0,_stderr
        movea.l     ABSEXECSBASE,a6
        ENDC      ASTART

        bra       domain

;=====
;===== Workbench Startup Code =====
;=====
;
; a2 WBenchMsg
; a4 Task (passed in)
; a5 SVar structure if not QARG (passed in)
; a6 AbsExecBase (passed in)
; sp WBenchMsg (still 0), sVar or 0, then RetAddr (passed in)
; sp argc=0,argv=WBenchMsg,WBenchMsg,sVar or 0,RetAddr (at domain)

fromWorkbench:
        ;----- get the startup message that workbench will send to us.
        ; must get this message before doing any DOS calls
        bsr.s     getWbMsg

        ;----- save the message so we can return it later
        move.l     d0,(sp)
        IFGT      ASTART
        move.l     d0,_WBenchMsg
        ENDC      ASTART

        ;----- push the message on the stack for wbmain (as argv)
        move.l     d0,-(sp)
        clr.l     -(sp)              ; indicate run from Workbench (argc=0)

        IFNE      (1-QARG)+WBOUT
        ;----- put DOSBase in a6 for next few calls
        move.l     _DOSBase,a6

```

```

ENDC      (1-QARG)+WBOUT

IFEQ      QARG
;----- get the first argument
move.l    d0,a2
move.l    sm_ArgList(a2),d0
beq.s     doCons

;----- and set the current directory to the same directory
move.l    d0,a0
move.l    wa_Lock(a0),d1
;should be a beq.s doCons here
callsys CurrentDir

doCons:
ENDC      QARG

IFGT      WBOUT

;----- Open NIL: or AppWindow for WB Input()/Output() handle
; Also for possible initialization of stdio globals
; Stdio used to be initialized to -1

IFGT      WINDOW
;----- Get AppWindow defined in application
lea        _AppWindow,a0
cmp.b      #0,(a0)
bne.s      doOpen          ; Open if not null string
ENDC      WINDOW

;----- Open NIL: if no window provided
lea        NilName(PC),a0

doOpen:
;----- Open up the file whose name is in a0
; DOSBase still in a6
move.l     a0,d1
move.l     #MODE_OLDFILE,d2
callsys Open
;----- d0 now contains handle for Workbench Output
;----- save handle for closing on exit
move.l     d0,sv_WbOutput(a5)
bne.s      gotOpen
moveq.l     #RETURN_FAIL,d2
bra        exit2

gotOpen:
IFGT      ASTART
;----- set the C input and output descriptors
move.l     d0,_stdin
move.l     d0,_stdout
move.l     d0,_stderr
ENDC      ASTART

;----- set the console task (so Open( "*", mode ) will work
; task pointer still in A4
move.l     d0,pr_CIS(A4)
move.l     d0,pr_COS(A4)
lsl.l      #2,d0
move.l     d0,a0
move.l     fh_Type(a0),d0
beq.s      noConTask
move.l     d0,pr_ConsoleTask(A4)

noConTask:
ENDC      WBOUT

;----- Fall though to common WB/CLI code

```

```

*****
** This code now used by both CLI and WB startup **
*****

```

```

domain:
    jsr      _main
    ;----- main didn't use exit(n) so provide success return code
    moveq.l  #RETURN_OK,d2
    bra.s   exit2

*****
**    subroutines here to allow short branches    **
*****

getWbMsg:
    ;----- a6 = ExecBase
    lea      pr_MsgPort(A4),a0      ; our process base
    callsys  WaitPort
    lea      pr_MsgPort(A4),a0      ; our process base
    callsys  GetMsg
    rts

*****

alertDOS:
    ;----- do recoverable alert for no DOS and exit
    ALERT    (AG_OpenLib!AO_DOSLib)

    ;----- do recoverable alert for no memory and exit
    ;----- If we got this far, DOS is open, so close it
    IFEQ QARG
    bra.s    failExit

alertMem:
    movea.l  _DOSBase,a1
    callsys  CloseLibrary
    ALERT    AG_NoMemory
    ENDC QARG

failExit:
    tst.l    pr_CLI(a4)
    bne.s    fail2
    bsr.s    getWbMsg
    movea.l  d0,a2
    bsr.s    repWbMsg

fail2:
    moveq.l  #RETURN_FAIL,d0
    rts

*****

repWbMsg:
    ;----- return the startup message to our parent
    ; a6 = ExecBase (passed)
    ; a2 = WBenchMsg (passed)
    ; we forbid so workbench can't UnLoadSeg() us before we are done
    callsys  Forbid
    move.l    a2,a1
    callsys  ReplyMsg
    rts

*****
** C Program exit() Function, return code on stack **
** **
** pr_ReturnAddr points to our RTS addr on stack **
** and we use this to calculate our stack ptr: **
** **
** SP -> WBenchMsg or 0 (CLI) **
** sVar ptr or 0 (QARG) **
** Address for RTS to DOS **
*****

_exit:
    move.l    4(sp),d2      ; exit(n) return code to d2

exit2:
    ;exit code in d2

```

```

;----- restore initial stack ptr
;-- FindTask
movea.l ABSEXECSBASE,a6
suba.l a1,a1
callsys FindTask
;-- get SP as it was prior to DOS's jsr to us
move.l d0,a4
move.l pr_ReturnAddr(a4),a5
;-- subtract 4 for return address, 4 for SVar, 4 for WBenchMsg
suba.w #12,a5

;-- restore sp
move.l a5,sp

;-- recover WBenchMsg
move.l (sp)+,a2
;-- recover SVar
move.l (sp)+,a5

IFGT WBOUT
;----- Close any WbOutput file before closing dos.library
move.l sv_WbOutput(a5),d1
beq.s noWbOut
move.l _DOSBase,a6
callsys Close
noWbOut:
;----- Restore a6 = ExecBase
movea.l ABSEXECSBASE,a6
ENDC WBOUT

;----- Close DOS library, if we got here it was opened
; SysBase still in a6
movea.l _DOSBase,a1
callsys CloseLibrary

;----- if we ran from CLI, skip workbench reply
checkWB:
move.l a2,d0
beq.s deallocSV

bsr.s repWbMsg

deallocSV:
IFEQ QARG
;----- deallocate the SVar structure
move.l a5,a1
move.l #SV_SIZEOF,d0
callsys FreeMem
ENDC QARG

;----- this rts sends us back to DOS:
move.l d2,d0
rts

```

\*\*\*\*\*

;----- PC relative data

```

DOSName      DOSNAME
NilName      dc.b  'NIL:',0
             IFGT  QARG
nullArgV     dc.l  nullArg
nullArg      dc.l  0          ; "" & the null entry after nullArgV
             ENDC

```

\*\*\*\*\*

DATA

\*\*\*\*\*

```

_SysBase     dc.l  0

```

```

_DOSBase      dc.l    0

        IFGT      ASTART
_WBenchMsg    dc.l    0
_stdin        dc.l    0
_stdout       dc.l    0
_stderr       dc.l    0
_errno        dc.l    0
        ENDC      ASTART

        IFGT      DEBUG
initialSP     dc.l    0
dosCmdLen     dc.l    0
dosCmdBuf     dc.l    0
        ENDC      DEBUG

VerRev        dc.w    34,12
        IFGT      ASTART
                dc.b    'A'
        ENDC      ASTART
        IFEQ      ASTART
                dc.b    'R'
        ENDC      ASTART
        IFGT      WINDOW
_NeedWStartup:
                dc.b    'W'
        ENDC      WINDOW
        IFEQ      WBOUT
                dc.b    'X'
        ENDC      WBOUT
        IFGT      NARGS
                dc.b    'N'
        ENDC      NARGS
        IFGT      DEBUG
                dc.b    'D'
        ENDC      DEBUG
        IFGT      QARG
                dc.b    'Q'
        ENDC      QARG

END

```

;==== The .i flag files for assembling the various versions of startup1.3 ====

```

*----- astartup.i
ASTART      SET    1
WINDOW      SET    0
XNIL        SET    0
NARGS       SET    0
DEBUG       SET    0
QARG        SET    0

```

```

*----- awstartup.i
ASTART      SET    1
WINDOW      SET    1
XNIL        SET    0
NARGS       SET    0
DEBUG       SET    0
QARG        SET    0

```

```

*----- qstartup.i
ASTART      SET    0
WINDOW      SET    0
XNIL        SET    1
NARGS       SET    0
DEBUG       SET    0
QARG        SET    1

```

```

*----- rstartup.i
ASTART      SET    0
WINDOW      SET    0
XNIL        SET    0

```

NARGS      SET    0  
DEBUG      SET    0  
QARG       SET    0

\*----- rwstartup.i

ASTART     SET    0  
WINDOW     SET    1  
XNIL       SET    0  
NARGS      SET    0  
DEBUG      SET    0  
QARG       SET    0

\*----- rxstartup.i

ASTART     SET    0  
WINDOW     SET    0  
XNIL       SET    1  
NARGS      SET    0  
DEBUG      SET    0  
QARG       SET    0





# **Chapter 31**

## **Audio Device**

### **Introduction**

The Amiga has four hardware audio channels—two of the channels produce audio output from the left audio connector, and two from the right. These channels can be used in many ways. You can combine a right and a left channel for stereo sound, use a single channel, or play a different sound through each of the channels to create four-part harmony.

The audio software is implemented as a standard Amiga input/output device with commands that allocate audio channels and control the sound output. To make sounds, you open the audio device, send IO requests to it, and then close it.

The audio device commands help isolate the programmer from the idiosyncrasies of the custom chip hardware and make it easier to use. But you can also produce sound on the Amiga by directly accessing the hardware registers if you temporarily lock out other users first. For certain types of sound synthesis, this is more CPU-efficient.

Some commands enable your program to co-reside with other programs using the audio device at the same time. Programs can co-reside because the audio device handles allocation of audio channels and arbitrates among programs competing for the same resources. When properly used, this allows many programs to use the audio device simultaneously.

Most personal computers that produce sound have hardware designed for one *specific* synthesis technique. The Amiga uses a very general method of digital sound synthesis that is quite similar to the method used in digital hi-fi components and state-of-the-art keyboard and drum synthesizers.

For programs that can afford the memory, playing sampled sounds gives you a simple and very CPU-efficient method of sound synthesis. A sampled sound is a table of numbers which represents a sound digitally. When the sound is played back by the Amiga, the table is fed by a DMA channel into one of the four digital-to-analog converters in the custom chips. The digital-to-analog converter converts the samples into voltages that can be played through amplifiers and loudspeakers, reproducing the sound.

On the Amiga you can create sound data in many other ways. For instance, you can use trigonometric functions in your programs to create the more traditional sounds—sine waves, square waves, or triangle waves—by using tables that describe their shapes. Then you can combine these waves for richer sound effects by adding the tables together. Once the data is entered, you can modify it with techniques described in “Audio Functions and Commands” below. For information about the limitations of the audio hardware and suggestions for improving system efficiency and sound quality, refer to the *Amiga Hardware Reference Manual*.

## DEFINITIONS

Terms used in the following discussions may be unfamiliar. Some of the more important terms are defined below.

### Amplitude

The height of a waveform, which corresponds to the amount of voltage or current in the electronic circuit.

### Amplitude modulation

A means of producing special audio effects by using one channel to alter the amplitude of another.

### Channel

One “unit” of the audio device.

### Cycle

One repetition of a waveform.

### Frequency

The number of times per second a cycle repeats.

### Frequency modulation

A means of producing special audio effects by using one channel to affect the period of the waveform produced by another channel.

### Period

The time elapsed between the output of successive sound samples, in units of system clock ticks.

### Precedence

Priority of the user of a sound channel.

### Sample

Byte of audio data, one of the fixed-interval points on the waveform.

### Waveform

Graph that shows a model of how the amplitude of a sound varies over time—usually over one cycle.

## Audio Functions and Commands

The audio device is similar to the other Amiga IO devices. To make sound, you first open the audio device, then send IO requests to it, and then close it when finished.

Audio device commands use an extended IO block named **IOAudio** to send commands to the audio device. This is the standard **IORequest** block with some extra fields added at the end.

```
struct IOAudio
{
    struct IORequest ioa_Request; /* IO request block. See exec/io.h. */
    WORD ioa_AllocKey;           /* Alloc. key filled in by audio device */
    UBYTE *ioa_Data;             /* Pointer to a sample or allocation mask */
    ULONG ioa_Length;            /* Length of sample or allocation mask. */
    UWORD ioa_Period;             /* Sample playback speed */
    UWORD ioa_Volume;            /* Volume of sound */
    UWORD ioa_Cycles;            /* # of times to play sample. 0=forever. */
    struct Message ioa_WriteMsg; /* Filled in by device - usually not used */
};
```

By filling in the appropriate fields with command data and sending the **IOAudio** block to the audio device, you can generate sound. For more details, you should see the command and reference section and the header files *devices/audio.h* and *devices/audio.i* in the *Amiga ROM Kernel Manual: Includes and Autodocs*. For general information on how IO devices work on the Amiga, refer to the *Exec* chapter.

### COMMAND TYPES

Commands and functions for audio use can be divided into three categories: system functions, allocation/arbitration commands, and hardware control commands. The system functions are:

- **OpenDevice("audio.device",0L,struct IORequest \*,0L)**
- **CloseDevice(struct IORequest \*)**
- **BeginIO(struct IORequest \*)**
- **Wait(ULONG) and WaitPort(struct MsgPort \*)**
- **AbortIO(struct IORequest \*)**

There are four allocation/arbitration commands. These do not actually produce any sound. Instead they manage and arbitrate the audio resources for the many tasks that may be using audio in the Amiga's multi-tasking environment.

- **ADCMD\_ALLOCATE** - Reserves an audio channel for your program to use.
- **ADCMD\_FREE** - Frees an audio channel.
- **ADCMD\_SETPREC** - Changes the precedence of a sound in progress.

- **ADCMD\_LOCK** - Tells you if a channel has been stolen from you.

The hardware control commands are used to set up, start, and stop sounds on the audio device:

- **CMD\_WRITE** - The main command. Starts a sound playing.
- **ADCMD\_FINISH** - Aborts a sound in progress.
- **ADCMD\_PERVOL** - Changes the period (speed) and volume of a sound in progress.
- **CMD\_FLUSH** - Clears the audio channels.
- **CMD\_RESET** - Resets and initializes the audio device.
- **ADCMD\_WAITCYCLE** - Signals you when a cycle finishes.
- **CMD\_STOP** - Temporarily stops a channel from playing.
- **CMD\_START** - Restarts an audio channel that was stopped.
- **CMD\_READ** - Returns a pointer to the current **IOAudio** request.

## SCOPE OF COMMANDS

Most audio commands can operate on multiple channels. The exceptions are **CMD\_WRITE**, **ADCMD\_WAITCYCLE**, and **CMD\_READ**, which can only operate on one channel at a time. You specify the channel that you want to use by setting the appropriate bits in the **ioa\_Request.io\_Unit** field of the **IOAudio** block. If you send a command for a channel that you do not own, your command will be ignored. For more details, see the section on “Allocation and Arbitration” below.

## SYSTEM FUNCTIONS

### **OpenDevice()**

Before you can use the audio device, you must first open it with a call to **OpenDevice()**. One nice feature of this function is that you can also automatically allocate channels for your program to use when you call **OpenDevice()**. To do this, you use a non-zero **ioa\_Request.io\_Length** field. The audio device will attempt to allocate channels just as if you had sent the **ADCMD\_ALLOCATE** command. If the allocation fails, the **OpenDevice()** call will return immediately.

If you want to allocate channels at some later time, then set the **ioa\_Request.io\_Length** field of the **IOAudio** block to zero when you call **OpenDevice()**. For more on channel allocation and the **ADCMD\_ALLOCATE** command, see the section on “Allocation and Arbitration” below.

## **CloseDevice()**

When you have finished with the audio device, you must close it with a call to the **CloseDevice()** function. **CloseDevice()** performs an **ADCMD\_FREE** command on any channels selected by the **ioa\_Request.io\_Unit** field of the **IOAudio** request. This means that if you close the device with the same **IOAudio** block that you used to allocate your channels (or a copy of it), the channels will be automatically freed.

If you allocated channels with multiple allocation commands, you cannot use this function to close all of them at once. Instead, you will have to issue one **ADCMD\_FREE** command for each allocation that you made. After issuing the **ADCMD\_FREE** commands for each of the allocations, you can call **CloseDevice()**.

## **BeginIO()**

All the commands that you can give to the audio device should be sent by calling the **BeginIO()** function. This differs from other Amiga devices which generally use **SendIO()** or **DoIO()**. You should not use **SendIO()** or **DoIO()** with the audio device because these functions clear some special flags used by the audio device; this might cause audio to work incorrectly under certain circumstances. To be safe, you should always use **BeginIO()** with the audio device.

## **Wait() and WaitPort()**

These functions can be used to put your task to sleep while a sound plays. **Wait()** takes a wake-up mask as its argument. The wake-up mask is usually the **mp\_SigBit** of a **MsgPort** that you have set up to get replies back from the audio device. You can also use **WaitPort()** to put your task to sleep while a sound plays. The argument to **WaitPort()** is a pointer to a **MsgPort** that you have set up to get replies back from the audio device. You must always use **Wait()** or **WaitPort()** to wait for IO to finish with the audio device. **WaitIO()** does not work correctly under all circumstances. Avoid using **WaitIO()** with V1.3 and earlier versions of the Amiga system software.

## **AbortIO()**

This function can be used to cancel requests for **ADCMD\_ALLOCATE**, **ADCMD\_LOCK**, **CMD\_WRITE**, or **ADCMD\_WAITCYCLE**. When used with the audio device, **AbortIO()** always succeeds.

## **A Simple Audio Example**

The Amiga's audio software has a complex allocation and arbitration system which is described in detail in the sections below. At this point, though, it may be helpful to look at a simple audio example:

```

/* Lattice use lc -bl -cfist -v -y. Link with lc.lib and amiga.lib */
#include <exec/types.h>          /* Some header files for system calls */
#include <exec/memory.h>
#include <devices/audio.h>
#include <graphics/gfxbase.h>
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) {return(0);} /* Disable Lattice Ctrl-C handling */
#endif

struct GfxBase *GfxBase;
/*-----*/
/* The whichchannel array is used when we allocate a channel. */
/* It tells the audio device which channel we want. The code */
/* is 1 =channel0, 2 =channel1, 4 =channel2, 8 =channel3. */
/* If you want more than one channel, add the codes up. */
/* This array says "Give me channel 0. If it's not available */
/* then try channel 1; then try channel 2 and then channel 3 */
/*-----*/
UBYTE          whichchannel[] = { 1,2,4,8 };

void main(int argc, char **argv)
{
    struct IOAudio *AIOptr;      /* Pointer to the IO block for IO commands */
    struct MsgPort *port;        /* Pointer to a port so the device can reply */
    struct Message *msg;         /* Pointer for the reply message */
    ULONG          device;
    BYTE           *waveptr;      /* Pointer to the sample bytes */
    LONG            frequency=440; /* Frequency of the tone desired */
    LONG            duration =3;   /* Duration in seconds */
    LONG            clock      =3579545; /* Clock constant, 3546895 for PAL */
    LONG            samples    =2;  /* Number of sample bytes */
    LONG            samcyc     =1;  /* Number of cycles in the sample */
    /*-----*/
    /* Ask the system if we are PAL or NTSC and set clock constant accordingly */
    /*-----*/
    GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",0L);
    if(GfxBase==0L)
        goto killaudio;
    if(GfxBase->DisplayFlags & PAL) clock=3546895; /* PAL clock */
    else                          clock=3579545; /* NTSC clock */

    if(GfxBase)
        CloseLibrary((struct Library *) GfxBase);
    /*-----*/
    /* Create an audio I/O block so we can send commands to the audio device */
    /*-----*/
    AIOptr=(struct IOAudio *)
        AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC | MEMF_CLEAR);
    if(AIOptr==0)
        goto killaudio;
    printf("IO block created...\n");
    /*-----*/
    /* Create a reply port so the audio device can reply to our commands */
    /*-----*/
    port=CreatePort(0,0);
    if(port==0)
        goto killaudio;
    printf("Port created...\n");
    /*-----*/
    /* Set up the audio I/O block for channel allocation: */
    /* ioa_Request.io_Message.mn_ReplyPort is the address of a reply port. */
    /* ioa_Request.io_Message.mn_Node.ln_Pri sets the precedence (priority) */
    /* of our use of the audio device. Any tasks asking to use the audio */
    /* device that have a higher precedence will steal the channel from us. */
    /* ioa_Request.io_Command is the command field for IO. */
    /* ioa_Request.io_Flags is used for the IO flags. */
    /* ioa_AllocKey will be filled in by the audio device if the allocation */
    /* succeeds. We must use the key it gives for all other commands sent.*/

```

```

/* ioa_Data is a pointer to the array listing the channels we want. */
/* ioa_Length tells how long our list of channels is. */
/*-----*/
AIOptr->ioa_Request.io_Message.mn_ReplyPort = port;
AIOptr->ioa_Request.io_Message.mn_Node.ln_Pri= 0;
AIOptr->ioa_Request.io_Command = ADCMD_ALLOCATE;
AIOptr->ioa_Request.io_Flags = ADIOF_NOWAIT;
AIOptr->ioa_AllocKey = 0;
AIOptr->ioa_Data = whichchannel;
AIOptr->ioa_Length = sizeof(whichchannel);
printf("IO block initialized for channel allocation...\n");
/*-----*/
/* Open the audio device and allocate a channel */
/*-----*/
device=OpenDevice("audio.device",0L, (struct IORequest *) AIOptr ,0L);
if(device!=0)
    goto killaudio;
printf("Audio device opened, channel allocated...\n");
/*-----*/
/* Create a very simple audio sample in memory. */
/*-----*/
waveptr=(BYTE *)AllocMem( samples , MEMF_CHIP|MEMF_PUBLIC);
if(waveptr==0)
    goto killaudio;
waveptr[0]= 127;
waveptr[1]= -127;
printf("Wave data ready...\n");

/*-----*/
/* Set up audio I/O block to play a sample using CMD_WRITE. */
/* The io_Flags are set to ADIOF_PERVOL so we can set the */
/* period (speed) and volume with the our sample; */
/* ioa_Data points to the sample; ioa_Length gives the length */
/* ioa_Cycles tells how many times to repeat the sample */
/* If you want to play the sample at a given sampling rate, */
/* set ioa_Period = clock/(given sampling rate) */
/*-----*/
AIOptr->ioa_Request.io_Message.mn_ReplyPort=port;
AIOptr->ioa_Request.io_Command =CMD_WRITE;
AIOptr->ioa_Request.io_Flags =ADIOF_PERVOL;
AIOptr->ioa_Data = (BYTE *)waveptr;
AIOptr->ioa_Length =samples;
AIOptr->ioa_Period =clock*samcyc/(samples*frequency);
AIOptr->ioa_Volume =64;
AIOptr->ioa_Cycles =frequency*duration/samcyc;
printf("IO block initialized to play tone...\n");

/*-----*/
/* Send the command to start a sound using BeginIO() */
/* Go to sleep and wait for the sound to finish with */
/* Wait(). When we wake-up we have to get the reply */
/*-----*/
printf("Starting tone now...\n");
BeginIO((struct IORequest *) AIOptr );
Wait(1L << port->mp_SigBit);
msg=GetMsg(port);

printf("Sound finished...\n");

killaudio:

printf("Killing audio device...\n");
if(waveptr!=0)
    FreeMem(waveptr, 2);
if(port!=0)
    DeletePort(port);
if(device==0)
    CloseDevice( (struct IORequest *) AIOptr );
if(AIOptr!=0)
    FreeMem( AIOptr,sizeof(struct IOAudio) );
}

```

## ALLOCATION AND ARBITRATION

The first command you send to the audio device should always be `ADCMD_ALLOCATE`. You can do this when you open the device, or at a later time. You specify the channels you want in the `ioa_Data` field of the `IOAudio` block. If the allocation succeeds, the audio device will return the channels that you now own in the lower four bits of the `ioa_Request.io_Unit` field of your `IOAudio` block. For instance, if the `io_Unit` field equals 5 (binary 0101) then you own channels 2 and 0. If the `io_Unit` field equals 15 (binary 1111) then you own all the channels.

When you send the `ADCMD_ALLOCATE` command, the audio device will also return a unique allocation key in the `ioa_AllocKey` of the `IOAudio` block. You must use this allocation key for all subsequent commands that you send to the audio device. The audio device uses this unique key to identify which task issued the command. If you do not use the right allocation key assigned to you by the audio device when you send a command, your command will be ignored.

When you request a channel with `ADCMD_ALLOCATE`, you specify a precedence number from -128 to 127 in the `ioa_Request.io_Message.mn_Node.In_Pri` field of the `IOAudio` block. If a channel you want is being used and you have specified a higher precedence than the current user, `ADCMD_ALLOCATE` will “steal” the channel from the other user. Later on, if your precedence is lower than that of another user who is performing an allocation, the channel may be stolen from you.

If you set the precedence to 127 when you open the device or raise the precedence to 127 with the `ADCMD_SETPREC` command, then no other tasks can steal a channel from you. When you have finished with a channel, you must relinquish it with the `ADCMD_FREE` command to make it available for other users. Table 5-1 shows suggested precedence values.

Table 31-1: Suggested Precedences for Channel Allocation

Precedence	Type of Sound
127	<i>Unstoppable.</i> Sounds first allocated at lower precedence, then set to this highest level.
90 - 100	<i>Emergencies.</i> Alert, urgent situation that requires immediate action.
80 - 90	<i>Annunciators.</i> Attention, bell (CTRL-G).
75	<i>Speech.</i> Synthesized or recorded speech (narrator.device).
50 - 70	<i>Sonic cues.</i> Sounds that provide information that is not provided by graphics. Only the beginning of each sound (enough to recognize it) should be at this level; the rest should be set to sound effects level.
-50 - 50	<i>Music program.</i> Musical notes in music-oriented program. The higher levels should be used for the attack portions of each note.
-70 - -50	<i>Sound effects.</i> Sounds used in conjunction with graphics. More important sounds should use higher levels.
-100 - -80	<i>Background.</i> Theme music and restartable background sounds.
-128	<i>Silence.</i> Lowest level (freeing the channel completely is preferred).



If you attempt to perform a command on a channel that has been stolen from you by a higher priority task, an **AUDIO\_NOALLOCATION** error is returned and the bit in the **ioa\_Request.io\_Unit** field corresponding to the stolen channel is cleared so you know which channel was stolen.

If you want to be warned before a channel is stolen so that you have a chance to stop your sound gracefully, then you should use the **ADCMD\_LOCK** command after you open the device. This command is also useful for programs which write directly to the audio hardware. For more on **ADCMD\_LOCK**, see the section below.

## ALLOCATION/ARBITRATION COMMANDS

These commands allow the audio channels to be shared among different tasks and programs. None of these commands can be called from interrupt code.

### ADCMD\_ALLOCATE

This command gives your program a channel to use and should be the first command you send to the audio device. You specify the channels you want by giving a pointer to an array in the **ioa\_Data** field of the **IOAudio** structure. This array uses a value of 1 to allocate channel 0, 2 for channel 1, 4 for channel 2, and 8 for channel 3. For multiple channels, add the values together. For example, if you want to allocate all channels, use a value of 15.

If you want a pair of stereo channels and you have no preference about which of the left and right channels the system will choose for the allocation, you can pass a pointer to an array containing 3, 5, 10, and 12. Channels 0 and 3 produce sound on the left side, and channels 1 and 2 on the right side. The table below shows how this array corresponds to all the possible combinations of a right and a left channel.

Table 31-2: Possible Channel Combinations

Channel 3 left	Channel 2 right	Channel 1 right	Channel 0 left	Decimal Value of Allocation Mask
0	0	1	1	3
0	1	0	1	5
1	0	1	0	10
1	1	0	0	12

**How ADCMD\_ALLOCATE Operates.** The **ADCMD\_ALLOCATE** command tries the first combination, 3, to see if channels 0 and 1 are not being used. If they are available, the 3 is copied into the **io\_Unit** field and you get an allocation key for these channels in the **ioa\_AllocKey** field. You copy the key into other IO blocks for any other commands you may want to perform on these channels.

If channels 0 and 1 are being used, **ADCMD\_ALLOCATE** tries the other combinations in turn. If all the combinations are in use, **ADCMD\_ALLOCATE** checks the precedence number of the users of the channels and finds the combination that requires it to steal the channel or channels of the lowest precedence. If all the combinations require stealing a channel or channels of equal or higher precedence, the IO request **ADCMD\_ALLOCATE** fails. Precedence is in the **ln\_Pri** field of the **io\_Message** in the **IORequest** block you pass to **ADCMD\_ALLOCATE**; it has a value from -128 to 127.

**The ADIOF\_NOWAIT Flag.** If you need to produce a sound right now and otherwise you don't want to allocate, set the ADIOF\_NOWAIT flag to 1. This will cause the command to return an IOERR\_ALLOCFAILED error if it cannot allocate any of the channels. If you are producing a non-urgent sound and you can wait, set the ADIOF\_NOWAIT flag to 0. Then, the IORequest block returns only when you gets the allocation. If ADIOF\_NOWAIT is set to 0, the audio device will continue to retry the allocation request whenever channels are freed until it is successful. If the program decides to cancel the request, AbortIO() can be used.

**ADCMD\_ALLOCATE Examples.** The following are some more examples of how to tell ADCMD\_ALLOCATE your channel preferences. If you want any channel, but want to try to get a left channel first, use an array containing 1, 8, 2, and 4:

```
0001
1000
0010
0100
```

If you want only a left channel, use 1 and 8 (channels 0 and 3):

```
0001
1000
```

For a right channel, use 2 and 4 (channels 1 and 2):

```
0010
0100
```

If you want to allocate a channel and keep it for a sound that can be interrupted and restarted, allocate it at a certain precedence. If it is stolen, allocate it again with the ADIOF\_NOWAIT flag set to 0. When the channel is relinquished, you will get it again.

**The Allocation Key.** If you want to perform multi-channel commands, all the channels must have the same key since the IORequest block has only one allocation key field. The channels must all have that same key even when they were not allocated simultaneously. If you want to use a key you already have, you can pass that key in the ioa\_AllocKey field and ADCMD\_ALLOCATE can allocate other channels with that existing key. The ADCMD\_ALLOCATE command returns a new and unique key only if you pass it a zero in the allocation key field.

## **ADCMD\_FREE**

ADCMD\_FREE is the opposite of ADCMD\_ALLOCATE. When you perform ADCMD\_FREE on a channel, it does a CMD\_RESET command on the hardware and "unlocks" the channel. It also checks to see if there are other pending allocation requests. You do not need to perform ADCMD\_FREE on channels stolen from you. If you want channels back after they have been stolen, you must reallocate them with the same allocation key.

## **ADCMD\_SETPREC**

This command changes the precedence of an allocated channel. As an example of the use of ADCMD\_SETPREC, assume that you are making sound of a chime that takes a long time to decay. It is important that user hears the chime but not so important that he hears it decay all the way. You could lower precedence after the initial attack portion of the sound to let another program steal the channel. You can also set the precedence to maximum (127) if you do not want the channel(s) stolen from you.

## ADCMD\_LOCK

The ADCMD\_LOCK command performs the “steal verify” function. When another application is attempting to steal a channel or channels, ADCMD\_LOCK gives you a chance to clean up before the channel is stolen. You perform a ADCMD\_LOCK command right after the ADCMD\_ALLOCATE command. ADCMD\_LOCK does not return until a higher-priority user attempts to steal the channel(s) or you perform an ADCMD\_FREE command. If someone is attempting to steal, you must finish up and ADCMD\_FREE the channel as quickly as possible.

You must use ADCMD\_LOCK if you want to write directly to the hardware registers instead of using the device commands. If your channel is stolen, you are not notified unless the ADCMD\_LOCK command is present. This could cause problems for the task that has stolen the channel and is now using it at the same time as your task. ADCMD\_LOCK sets a switch that is not cleared until you perform an ADCMD\_FREE command on the channel. Canceling an ADCMD\_LOCK request with AbortIO() will not free the channel.

The following outline describes how ADCMD\_LOCK works when a channel is stolen and when it is not stolen.

1. User A allocates a channel.
2. User A locks the channel.

If User B allocates the channel with a higher precedence:

3. User B's ADCMD\_ALLOCATE command is suspended (regardless of the setting of the ADIOF\_NOWAIT flag).
4. User A's lock command is replied to with an error (ADIOERR\_CHANNELSTOLEN).
5. User A does whatever is needed to finish up when a channel is stolen.
6. User A frees the channel with ADCMD\_FREE.
7. User B's ADCMD\_ALLOCATE command is replied to. Now user B has the channel.

If the channel is not allocated by another user:

3. User A finishes the sound.
4. User A performs the ADCMD\_FREE command.
5. User A's ADCMD\_LOCK command is replied to.

Never make the freeing of a channel (if the channel is stolen) dependent on allocating another channel. This may cause a deadlock. If you want channels back after they have been stolen, you must reallocate them with the same allocation key. To keep a channel and never let it be stolen, set precedence to maximum (127). Do not use a lock for this purpose.

## HARDWARE CONTROL COMMANDS

The following commands change hardware registers and affect the actual sound output.

### CMD\_WRITE

This is a single-channel command and is the main command for making sounds. You pass the following to CMD\_WRITE:

- A pointer to the waveform to be played (must start on a word boundary and must be in memory accessible by the custom chips, MEMF\_CHIP)
- The length of the waveform in bytes (must be an even number)
- A count of how many times you want to play the waveform

If the count is 0, CMD\_WRITE will play the waveform from beginning to end, then repeat the waveform continuously until something aborts it.

If you want period and volume to be set at the start of the sound, you set the WRITE command's ADIOF\_PERVOL flag. If you do not do this, the previous volume and period for that channel will be used. This is one of the flags that is cleared by DoIO() and SendIO(). The ioa\_WriteMsg field in the IORequest block is an extra message field that can be replied to at the start of the CMD\_WRITE. This second message is used only to tell you when the CMD\_WRITE command *starts* processing, and it is used only when the ADIOF\_WRITEMESSAGE flag is set to 1.

If a CMD\_STOP has been performed, the CMD\_WRITE requests are queued up. The CMD\_WRITE command does not make its own copy of the waveform, so any modification of the waveform before the CMD\_WRITE command is finished may affect the sound. This is sometimes desirable for special effects. To splice together two waveforms without clicks or pops, you must send a separate, second CMD\_WRITE command while the first is still in progress. This technique is used in double-buffering, which is described below.

**Double-buffering.** By using two waveform buffers and two CMD\_WRITE requests you can compute a waveform continuously. This is called double-buffering. The following describes how you use double-buffering.

1. Compute a waveform in memory buffer A.
2. Issue CMD\_WRITE A with io\_Data pointing to buffer A.
3. Continue the waveform in memory buffer B.
4. Issue CMD\_WRITE B with io\_Data pointing to Buffer B.
5. Wait for CMD\_WRITE A to finish.
6. Continue the waveform in memory buffer A.
7. Issue CMD\_WRITE A with io\_Data pointing to Buffer A.
8. Wait for CMD\_WRITE B to finish.
9. Loop back to step 3 until the waveform is finished.
10. At the end, remember to wait until both CMD\_WRITE A and B are finished.

## **ADCMD\_FINISH**

The ADCMD\_FINISH command aborts (calls **AbortIO()**) the current write request on a channel or channels. This is useful if you have something playing, such as a long buffer or some repetitions of a buffer, and you want to stop it.

ADCMD\_FINISH has a flag you can set (ADIOF\_SYNCCYCLE) that allows the waveform to finish the current cycle before aborting it. This is useful for splicing together sounds at zero crossings or some other place in the waveform where the amplitude at the end of one waveform matches the amplitude at the beginning of the next. Zero crossings are positions within the waveform at which the amplitude is zero. Splicing at zero crossings gives you fewer clicks and pops when the audio channel is turned off or the volume is changed.

## **ADCMD\_PERVOL**

ADCMD\_PERVOL lets you change the volume and period of a CMD\_WRITE that is in progress. The change can take place immediately or you can set the ADIOF\_SYNCCYCLE flag to have the change occur at the end of the cycle. This is useful to produce vibratos, glissandos, tremolos, and volume envelopes in music or to change the volume of a sound.

## **CMD\_FLUSH**

CMD\_FLUSH aborts (calls **AbortIO()**) all CMD\_WRITEs and all ADCMD\_WAITCYCLEs that are queued up for the channel or channels. It does not abort ADCMD\_LOCKs (only ADCMD\_FREE clears locks).

## **CMD\_RESET**

CMD\_RESET restores all the audio hardware registers. It clears the attach bits, restores the audio interrupt vectors if the programmer has changed them, and performs the CMD\_FLUSH command to cancel all requests to the channels. CMD\_RESET also unstops channels that have had a CMD\_STOP performed on them. CMD\_RESET does not unlock channels that have been locked by ADCMD\_LOCK.

## **ADCMD\_WAITCYCLE**

This is a single-channel command. ADCMD\_WAITCYCLE is replied to when the current cycle has completed, that is, after the current CMD\_WRITE command has reached the end of the current waveform it is playing. If there is no CMD\_WRITE in progress, it returns immediately.

## **CMD\_STOP**

This command stops the current write cycle immediately. If there are no CMD\_WRITEs in progress, it sets a flag so any future CMD\_WRITEs are queued up and do not begin processing (playing).

## CMD\_START

CMD\_START undoes the CMD\_STOP command. Any cycles that were stopped by the CMD\_STOP command are actually lost because of the impossibility of determining exactly where the DMA ceased. If the CMD\_WRITE command was playing two cycles and the first one was playing when CMD\_STOP was issued, the first one is lost and the second one will be played.

This command is also useful when you are playing the same wave form with the same period out of multiple channels. If the channels are stopped, when the CMD\_WRITE commands are issued, CMD\_START exactly synchronizes them, avoiding cancellation and distortion. When channels are allocated, they are effectively started by the CMD\_START command.

## CMD\_READ

CMD\_READ is a single-channel command. Its only function is to return a pointer to the current CMD\_WRITE command. It enables you to determine which request is being processed.

## DOUBLE BUFFERED SOUND EXAMPLE

The program listed below demonstrates double buffering with the audio device. Run the program from the CLI. It takes one parameter - the name of an IFF 8SVX sample file to play on the Amiga's audio device. The maximum size for a sample on the Amiga is 128K. However, by using double-buffering and queueing up requests to the audio device, you can play longer samples smoothly and without breaks.

```
/* Lattice use lc -bl -cfist -v -y. Link with lc.lib and amiga.lib */
/*-----*/
/* INCLUDES */
/*-----*/
#include <exec/types.h>
#include <exec/memory.h>
#include <devices/audio.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <graphics/gfxbase.h>
#include <iff/iff.h>
#include <iff/8svx.h>
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>

#define VHDR MakeID('V','H','D','R')
#define BODY MakeID('B','O','D','Y')
#define MY8S MakeID('8','S','V','X')

#ifdef LATTICE
int CXBRK(void){return(0);} /* Lattice Ctrl-C Handling */
#endif

void kill8svx(char *);
void kill8(void);

/*-----*/
/* GLOBALS */
/*-----*/
struct IOAudio *AIOptr1, /* Pointers to Audio IOBs */
```

```

        *AIOptr2,
        *Aptr;
struct Message      *msg;           /* Msg, port and device for */
struct MsgPort      *port;          /* driving audio */
        ULONG
        device;
        UBYTE      *sbase,*fbase;   /* For sample memory allocation */
        ULONG      fsize,ssize;     /* and freeing */
struct FileHandle   *v8handle;
        UBYTE      chan1[] = { 1 }; /* Audio channel allocation arrays */
        UBYTE      chan2[] = { 2 };
        UBYTE      chan3[] = { 4 };
        UBYTE      chan4[] = { 8 };
        UBYTE      *chans[] = {chan1,chan2,chan3,chan4};

/*-----*/
/*  M A I N  */
/*-----*/
void main(int argc,char **argv)
{
/*-----*/
/*  L O C A L S  */
/*-----*/
        char      *fname;           /* File name and data pointer*/
        UBYTE      *p8data;         /* for file read. */
        ULONG      clock;           /* Clock constant */
        ULONG      length[2];       /* Sample lengths */
        BYTE       iobuffer[8],     /* Buffer for 8SVX header */
        *psample[2];               /* Sample pointers */
        Chunk      *p8Chunk;        /* Pointers for 8SVX parsing */
        Voice8Header *pVoice8Header;
        ULONG      y,rd8count,speed; /* Counters, sampling speed */
        ULONG      wakebit;         /* A wakeup mask */
        BYTE       oldpri,c;        /* Stuff for bumping priority */
        struct Task *mt;

/*-----*/
/*  C O D E  */
/*-----*/

/*-----*/
/* Check Arguments, Initialize */
/*-----*/
fbase=0L;
sbase=0L;
AIOptr1=0L;
AIOptr2=0L;
port=0L;
v8handle=0L;
device=1L;

if (argc < 2)
{
        kill8svx("No file name given.0);
        exit(1L);
}
fname=argv[1];

/*-----*/
/* Initialize Clock Constant */
/*-----*/
GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",0L);
if(GfxBase==0L)
{
        puts("Can't open graphics library0);
        exit(1L);
}

if(GfxBase->DisplayFlags & PAL) clock=3546895L; /* PAL clock */
else clock=3579545L; /* NTSC clock */

if(GfxBase)
        CloseLibrary( (struct Library *) GfxBase);

```

```

/*-----*/
/* Open the File */
/*-----*/
v8handle= (struct FileHandle *) Open(fname,MODE_OLDFILE);
if(v8handle==0)
{
    kill8svx("Can't open 8SVX file.0);
    exit(1L);
}

/*-----*/
/* Read the 1st 8 Bytes of the File for Size */
/*-----*/
rd8count=Read((BPTR)v8handle,iobuffer,8L);
if(rd8count==-1)
{
    kill8svx ("Read error.0);
    exit(1L);
}
if(rd8count<8)
{
    kill8svx ("Not an IFF 8SVX file, too short0);
    exit(1L);
}

/*-----*/
/* Evaluate Header */
/*-----*/
p8Chunk=(Chunk *)iobuffer;
if( p8Chunk->ckID != FORM )
{
    kill8svx("Not an IFF FORM.0);
    exit(1L);
}

/*-----*/
/* Allocate Memory for File and Read it in. */
/*-----*/
fbase= (UBYTE *)AllocMem(fsize=p8Chunk->ckSize , MEMF_PUBLIC|MEMF_CLEAR);
if(fbase==0)
{
    kill8svx("No memory for read.0);
    exit(1L);
}
p8data=fbase;

rd8count=Read((BPTR)v8handle,p8data,p8Chunk->ckSize);
if(rd8count==-1)
{
    kill8svx ("Read error.0);
    exit(1L);
}
if(rd8count<p8Chunk->ckSize)
{
    kill8svx ("Malformed IFF, too short.0);
    exit(1L);
}

/*-----*/
/* Evaluate IFF Type */
/*-----*/
if(MakeID( *p8data, *(p8data+1) , *(p8data+2) , *(p8data+3) ) != MY8S )
{
    kill8svx("Not an IFF 8SVX file.0);
    exit(1L);
}

/*-----*/
/* Evaluate 8SVX Chunks */
/*-----*/

p8data=p8data+4;

```



```

while( p8data < fbase+fsize )
{
    p8Chunk=(Chunk *)p8data;

    switch(p8Chunk->ckID)
    {
        case VHDR:
            /*-----*/
            /* Get a pointer to the 8SVX header for later use */
            /*-----*/
            pVoice8Header=(Voice8Header *) (p8data+8L);
            break;
        case BODY:

            /*-----*/
            /* Create pointers to 1-shot and continuous parts */
            /* for the top octave and get length. Store them. */
            /*-----*/
            psample[0] = (BYTE *) (p8data + 8L);
            psample[1] = psample[0] + pVoice8Header->oneShotHiSamples;
            length[0] = (ULONG)pVoice8Header->oneShotHiSamples;
            length[1] = (ULONG)pVoice8Header->repeatHiSamples;
            break;

        default:
            break;
    }

    /* end switch */

    p8data = p8data + 8L + p8Chunk->ckSize;

    if(p8Chunk->ckSize%1L == 1)
        p8data++;
}

/* Play either the one-shot or continuous, not both */
if (length[0]==0)
    y=1;
else
    y=0;

/*-----*/
/* Allocate chip memory for samples and */
/* copy from read buffer to chip memory. */
/*-----*/
if(length[y]<=102400) ssize=length[y];
else                ssize=102400;

sbase=(UBYTE *)AllocMem( ssize , MEMF_CHIP | MEMF_CLEAR);
if(sbase==0)
{
    kill8svx("No chip memory.0);
    exit(1L);
}
CopyMem(psample[y],sbase,ssize);
psample[y]+=ssize;

/*-----*/
/* Calculate playback sampling rate */
/*-----*/
speed = clock / pVoice8Header->samplesPerSec;

/*-----*/
/* Bump our priority */
/*-----*/
mt=FindTask(NULL);
oldpri=SetTaskPri(mt,21);

/*-----*/
/* Allocate audio I/O blocks and make a port */
/*-----*/

```

```

AIOptr1=(struct IOAudio *)
    AllocMem( sizeof(struct IOAudio),MEMF_CHIP|MEMF_PUBLIC|MEMF_CLEAR);
if(AIOptr1==0)
{
    kill8svx("No IO memory0);
    exit(1L);
}

AIOptr2=(struct IOAudio *)
    AllocMem( sizeof(struct IOAudio),MEMF_CHIP|MEMF_PUBLIC|MEMF_CLEAR);
if(AIOptr2==0)
{
    kill8svx("No IO memory0);
    exit(1L);
}

port=CreatePort(0,0);
if(port==0)
{
    kill8svx("No port0);
    exit(1L);
}
c=0;

while(device!=0)
{
    /*-----*/
    /* Set up audio I/O block for channel */
    /* allocation and Open the audio device */
    /*-----*/
    AIOptr1->ioa_Request.io_Message.mn_ReplyPort = port;
    AIOptr1->ioa_Request.io_Message.mn_Node.ln_Pri = 128; /* No stealing! */
    AIOptr1->ioa_AllocKey = 0;
    AIOptr1->ioa_Data = chans[c];
    AIOptr1->ioa_Length = 1;

    device=OpenDevice("audio.device",0L,(struct IORequest *)AIOptr1,0L);
    c++;
}
if(device!=0)
{
    kill8svx("No channel0);
    exit(1L);
}

/*-----*/
/* Set Up Audio IO Blocks for Sample Playing */
/*-----*/
AIOptr1->ioa_Request.io_Command =CMD_WRITE;
AIOptr1->ioa_Request.io_Flags =ADIOF_PERVOL;

/*-----*/
/* Volume */
/*-----*/
AIOptr1->ioa_Volume=60;
/*-----*/
/* Period/Cycles */
/*-----*/
AIOptr1->ioa_Period =(UWORD) speed;
AIOptr1->ioa_Cycles =1;
*AIOptr2 = *AIOptr1; /* Make sure we have the same allocation keys, */
/* same channels selected and same flags */

/*-----*/
/* Data */
/*-----*/
AIOptr1->ioa_Data =(UBYTE *) sbase;
AIOptr2->ioa_Data =(UBYTE *) sbase + 51200;

Aptr=AIOptr2;
/*-----*/
/* Run the sample */

```

```

/*-----*/
if(length[y]<=102400)
{
    AIOptr1->ioa_Length=length[y];          /* No double buffering needed */
    BeginIO((struct IOREquest *)AIOptr1);  /* Begin the sample, wait for */
    wakebit=0L;                             /* it to finish, then quit. */
    wakebit=Wait(1 << port->mp_SigBit);
    msg=GetMsg(port);
}
else
{
    length[y]=102400;                       /* It's a real long sample so */
    AIOptr1->ioa_Length=51200L;             /* double buffering is needed */
    AIOptr2->ioa_Length=51200L;
    BeginIO((struct IOREquest *)AIOptr1);  /* Queue up two samples and Wait */
    BeginIO((struct IOREquest *)AIOptr2);  /* for the first to finish. */
    while(length[y]>0)                      /* Reuse the Audio IOB, queue it */
    {                                       /* up again and wait for the 2nd */
        wakebit=Wait(1 << port->mp_SigBit); /* Audio IOB to finish. Reuse */
        msg=GetMsg(port);                 /* the 2nd, queue it up, repeat. */
    }

    if(Aptr==AIOptr1) Aptr=AIOptr2;
    else               Aptr=AIOptr1;

    if(length[y]<=51200) Aptr->ioa_Length=length[y];
    else                Aptr->ioa_Length=51200L;

    CopyMem(psampl[y],Aptr->ioa_Data,Aptr->ioa_Length);

    length[y]=Aptr->ioa_Length;
    psampl[y]+=51200;
    BeginIO((struct IOREquest *)Aptr);
}
wakebit=Wait(1 << port->mp_SigBit); /* Finish off the last two requests */
msg=GetMsg(port);
wakebit=Wait(1 << port->mp_SigBit);
msg=GetMsg(port);
}
kill8();
exit(0L);
}

/*-----*/
/* Abort the Read */
/*-----*/
void
kill8svx(kill8svxstring)
char *kill8svxstring;
{
    puts(kill8svxstring);
    kill8();
}

/*-----*/
/* Return system resources */
/*-----*/
void
kill8()
{
    if(v8handle!=0) Close((BPTR)v8handle);
    if(fbase !=0)   FreeMem(fbase,fsize);
    if(sbase !=0)   FreeMem(sbase,ssize);

    if(device ==0)  CloseDevice((struct IOREquest *)AIOptr1);
    if(port !=0)    DeletePort(port);
    if(AIOptr1!=0)  FreeMem(AIOptr1,sizeof(struct IOAudio));
    if(AIOptr2!=0)  FreeMem(AIOptr2,sizeof(struct IOAudio));
}

```

# Chapter 32

## Clipboard Device

### Introduction

The clipboard device allows the exchange of data dynamically between one application and another. It is responsible for caching data that has been “cut” and providing data to “paste” in an application. The clipboard will cache the data in ram and will automatically spool the data to disk if necessary. A special “post” mode allows application to inform the clipboard.device that the application has data available. The clipboard device will request this data only if the data is actually needed. The clipboard device is implemented as an Exec-style device, and supports random access reads and writes on data within the clipboard.

### Clipboard Commands

The clipboard responds to the following Exec functions:

**OpenDevice()**    Open the clipboard device.

<b>CloseDevice()</b>	Close the clipboard device.
<b>SendIO()</b>	Initiate a command and return immediately.
<b>DoIO()</b>	Initiate a command and wait for it to complete.

The I/O commands and their implementations are as follows:

<b>CMD_READ</b>	Read data from the clipboard for a paste. <b>io_Offset</b> and <b>io_ClipID</b> must be set to zero for the first read of a paste sequence. An <b>io_Actual</b> that is less than the <b>io_Length</b> indicates that all the data has been read. After all the data has been read, a subsequent read must be performed (one whose <b>io_Actual</b> returns zero) to indicate to the clipboard device that all the data has been read. This allows random access of the clip while reading. Providing only valid reads are performed, your program can seek/read anywhere within the clip by setting the <b>io_Offset</b> field of the IO request appropriately.
<b>CMD_WRITE</b>	Write data to the clipboard as a cut. <b>io_Offset</b> and <b>io_ClipID</b> must be set to zero for the first write of a cut sequence. An update command indicates that all the data has been written.
<b>CMD_UPDATE</b>	Indicate that the data provided with a write command is complete and available for subsequent read/pastes.
<b>CMD_CLEAR</b>	Clear any cut from this unit. Subsequent read/pastes will have no data available.
<b>CMD_STOP</b>	Service no commands except invalid, start, flush.
<b>CMD_START</b>	Resume command servicing.
<b>CMD_FLUSH</b>	Abort all pending commands.
<b>CBD_POST</b>	Post the availability of clip data. <b>io_ClipID</b> must be set to zero. A subsequent write of this data does not have <b>io_ClipID</b> set to zero as described above, but preserves the value in <b>io_ClipID</b> set by this call. This command allows the application to inform the clipboard that data is available, and request that it be notified by the clipboard if and when the data is needed.
<b>CBD_CURRENTREADID</b>	Return the <b>io_ClipID</b> of the current clip to read. This is used to determine if a clip posting is still the latest cut.
<b>CBD_CURRENTWRITEID</b>	Return the <b>io_ClipID</b> of the latest clip written. This is used to determine if the clip posting data will never be requested by others.
<b>CMD_INVALID</b>	Always an invalid command.

## Clipboard Data

Data on the clipboard resides in one of three places. When an application posts a cut, the data resides in that private memory space of that application. When an application writes to the clipboard, either of its own volition or in response to a message from the clipboard requesting that it satisfy a post, the data is copied to the clipboard, either to memory or to a special disk file. When the clipboard is not open, the data resides in the special disk file located in the directory specified by the CLIPS: logical AmigaDOS assign.

Data on the clipboard is self-identifying. It must be a correct IFF (Interchange File Format) file; the rest of this section refers to IFF concepts. See the *Includes and Autodocs Manual* for a complete description of IFF. If the top-level chunk is of type CAT with an identifier of CLIP, that indicates that the contained chunks are different representations of the same data, in decreasing order of preference on the part of the producer of the clip. Any other data is as defined elsewhere (probably a single representation of the cut data produced by an application).

A clipboard tool, which is an application that allows a Workbench user to view a clip, should understand the text (FTXT) and graphics (ILBM) form types. Applications using the clipboard to export data should include at least one of these types in a CLIP CAT so that their data can be represented on the clipboard in some form for user feedback.

The clipboard device nonstandard I/O request is called an **IOClipReq** and looks like a standard request except for the addition of the **io\_ClripID** field, which is assigned by the device to identify clips. It must be set to zero by the application for a post or an initial write or read, but preserved for subsequent writes or reads, as the clipboard.device uses this field internally for bookkeeping purposes.

The **io\_Offset** field should also be set to zero by the application for an initial read or write.

```
struct IOClipReq
{
    struct Message io_Message;
    struct Device *io_Device;      /* device node pointer */
    struct Unit *io_Unit;          /* unit (driver private) */
    UWORD io_Command;              /* device command */
    UBYTE io_Flags;                /* including QUICK and SATISFY */
    BYTE io_Error;                 /* error or warning num */
    ULONG io_Actual;                /* number of bytes transferred */
    ULONG io_Length;               /* number of bytes requested */
    SPTR io_Data;                  /* either clip stream or post port */
    ULONG io_Offset;               /* offset in clip stream */
    LONG io_ClripID;               /* ordinal clip identifier */
}
```

This structure is defined in the include file *devices/clipboard.h* and *devices/clipboard.i*.

## Clipboard Messages

When an application performs a post, it must specify a message port for the clipboard to send a message to if it needs the application to satisfy the post with a write called the **SatisfyMsg**.

```

struct SatisfyMsg
{
    struct Message sm_Message; /* the length will be 6 */
    UWORD sm_Unit; /* 0 for the primary clip unit */
    LONG sm_ClipID; /* the clip identifier of the post */
}

```

This structure is defined in the include file *devices/clipboard.h* and *devices/clipboard.i*.

If the application wishes to determine if a post it has recently performed is still the current clip, it should check the **io\_ClipID** found in the post request upon return with that returned by the **CBD\_CURRENTREADID** command.

If an application has a pending post and wishes to determine if it should satisfy it (for example, before it exits), it should check the **io\_ClipID** of the post I/O request with that of the **CBD\_CURRENTWRITEID** command. If the application receives a satisfy message from the clipboard device (format described below), it must immediately perform the write with the **io\_ClipID** of the post. The satisfy message from the clipboard may be removed from the application message port by the clipboard device at any time (because it is re-used by the clipboard device). It is not dangerous to spuriously satisfy a post, however, because it is identified by the **io\_ClipID**.

The cut data is provided to the clipboard device via either a write or a post of the cut data. The write command accepts the data immediately and copies it onto the clipboard. The post command allows an application to inform the clipboard of a cut, but defers the write until the data is actually required for a paste. In the preceding discussion, references to the read and write commands of the clipboard device actually refer to a sequence of read or write commands, where the clip data is acquired and provided in pieces instead of all at once. The clipboard has an end-of-clip concept that is somewhat analogous to end-of-file for both read and write. The read end-of-file must be triggered by the user of the clipboard in order for the clipboard to move on to service other users' requests, and consists of reading data past the end of file. The write end-of-file is indicated by use of the update command, which indicates to the clipboard that the previous write commands are completed. See the description of the commands above for more information.

## Multiple Clips

The clipboard also supports multiple clips. This is not to be confused with the multiple IFF CLIP chunks in a clip, which allow for different representation of the same data. Multiple clips store different data. Applications performing cut and paste operations generally specify the primary clip. The alternate clips are provided to aid applications in the maintenance of a set of clips (like a scrapbook). The multiple clips are implemented as different units in the clipboard device, and are thus accessed at open time:

```
OpenDevice("clipboard.device", unit, &IOClipReq, 0);
```

The primary clip unit used by applications to share data is unit 0; use of alternate clip units is by private convention.

## Example Program

```
;/*
lc -d -j73 -O -oclip.o -i/include -v -cfirstq -y -v clip
blink LIB:c.o+clip.o+cbio.o to clip LIB LIB:lc.lib LIB:amiga.lib sc sd nd
quit
*/

/*****
/* Program name: clip
/* Purpose: Demonstrate the use of the clipboard
/* This program can be run by itself, or two or more
/* copies can be run, demonstrating how one program
/* can send data to another. If the POST option is used,
/* two programs must be used together.
*****/

#include <exec/types.h>
#include <libraries/dos.h>
#include <devices/clipboard.h>

#ifdef LATTICE
#include <proto/all.h>
#include <lattice/stdlib.h>
#include <lattice/stdio.h>
#include <lattice/string.h>
int CXBRK(void) { return(0); }
#endif

extern long CBOpen(long);
extern void CBClose(void);
extern void writeLong(long *);
extern void CBCutS(UBYTE *);
extern long CBPasteS(UBYTE *);
extern long CBPost(void);
extern long CBCurrentReadID(void);
extern long CBCurrentWriteID(void);
extern BOOL CBCheckSatisfy(long *);
extern void CBSatisfyPost(UBYTE *);
extern void CBCut(UBYTE *,long);

void main(int,char **);
void cleanExit(long);
void readS(UBYTE *);
void print(UBYTE *,long);

LONG con;

void main(argc,argv)
int argc;
char **argv;
{
    UBYTE c;
    ULONG postID;
    UBYTE cbuf[80];
    UBYTE buffer[80];

    con = Open("RAW:25/25/300/120/clipboard.device test",MODE_OLDFILE);
    if( !con || CBOpen(PRIMARY_CLIP))cleanExit(10);

    print(cbuf,sprintf(cbuf,"\033[20hclipboard.device is open.\n"));

    c = 0;
    postID = 0;
    while (c != '\34')
```



```

{ /* while not EOF */
while((postID) && (!WaitForChar(con, 1000000)))
{
    if (CBCheckSatisfy(&postID))
    {
        if (postID)
        {
            print("Please satisfy request for post:\n",0);
            readS(buffer);
            print(cbuf,
                (sprintf(cbuf, "\nsatisfying with \"%s\"\n", buffer)));
            CBSatisfyPost(buffer);
            postID = 0;
        }
    }
    Read(con, &c, 1);
    switch ((int)c)
    {
        case 'w':
            print("Enter cut data\n",0);
            readS(buffer);
            CBCutS(buffer);
            print(cbuf,
                (sprintf(cbuf, "\n\"%s\" sent to clipboard\n",buffer)));
            break;
        case 'r':
            CBPasteS(buffer);
            print(cbuf, (sprintf(cbuf, "paste is \"%s\"\n",buffer)));
            break;
        case 'p': /* This function will wait for another program */
            /* to request data from the clipboard. */
            /* Running two copies of this program is the */
            /* easiest way to do this. */
            print("Posting, waiting for data request\n",0);
            postID = CBPost();
            break;
        default:;
    }
}
print("Exiting...\n",0);
cleanExit(0);
}

void cleanExit(error)
long error;
{
    CBClose();
    Close(con);
    exit(error);
}

void readS(buf)
UBYTE *buf;
{
    UBYTE c;
    long count=0;

    while (Read(con, &c, 1), ((c != '\34') && (c != '\r')))
    {
        *buf++ = c;
        Write(con, &c, 1);
        if(++count > 79) return;
    }
    *buf = '\0';
}

void print(string, length)
UBYTE *string;
long length;
{

```

```

if(!length) length=strlen(string);
Write(con,string,length);
}

```

## Support Functions Called from Example Program

```

; /*
lc -d -j73 -O -ocbio.o -i/include -v -cfirstq -y -v cbio
blink LIB:c.o+clip.o+cbio.o to clip LIB LIB:lc.lib LIB:amiga.lib sc sd nd
quit
*/

/*****
/* Program name: cbio
/* Purpose: Provide standard clipboard device interface routines
/*          such as Open, Post, Read, Write, etc.
*****/

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/io.h>
#include <devices/clipboard.h>

#ifdef LATTICE
#include <proto/all.h>
#include <lattice/stdlib.h>
#include <lattice/stdio.h>
#include <lattice/string.h>
#endif

struct IOclipReq *clipboardIO = 0;
struct MsgPort *clipboardMsgPort = 0;
struct MsgPort *satisfyMsgPort = 0;

long CBOpen(long);
void CBClose(void);
void writeLong(long *);
void CBCutS(UBYTE *);
long CBPasteS(UBYTE *);
long CBPost(void);
long CBCurrentReadID(void);
long CBCurrentWriteID(void);
BOOL CBCheckSatisfy(long *);
void CBSatisfyPost(UBYTE *);
void CBCut(UBYTE *,long);

long CBOpen(unit)
long unit;
{
    long error;

    /* open the clipboard device */
    clipboardMsgPort = CreatePort(0L,0L);
    satisfyMsgPort = CreatePort(0L,0L);

    clipboardIO=(struct IOclipReq *)
    CreateExtIO(clipboardMsgPort,sizeof(struct IOclipReq));

    if ((error = OpenDevice("clipboard.device", unit, clipboardIO, 0)))
        return(error);
    return(0);
}

void CBClose()
{
    CloseDevice(clipboardIO);
    DeletePort(satisfyMsgPort);
}

```

```

    DeletePort(clipboardMsgPort);
    DeleteExtIO(clipboardIO);
}

void CBCut(stream, length)
UBYTE *stream;
long length;
{
    clipboardIO->io_Command = CMD_WRITE;
    clipboardIO->io_Data = stream;
    clipboardIO->io_Length = length;
    clipboardIO->io_Offset = 0;
    clipboardIO->io_ClipID = 0;
    DoIO(clipboardIO);
    clipboardIO->io_Command = CMD_UPDATE;
    DoIO(clipboardIO);
}

void CBCutS(string)
UBYTE *string;
{
    clipboardIO->io_ClipID = 0;
    CBSatisfyPost(string);
}

void writeLong(ldata)
long *ldata;
{
    clipboardIO->io_Command = CMD_WRITE;
    clipboardIO->io_Data = (char *)ldata;
    clipboardIO->io_Length = 4;
    DoIO(clipboardIO);
}

void CBSatisfyPost(string)
UBYTE *string;
{
    long length, slen=strlen(string);
    BOOL odd = (slen & 1); /* pad byte flag */

    length= (odd) ? slen+1 : slen;
    clipboardIO->io_Offset = 0;

    writeLong((long *)"FORM");          /* "FORM" */
    length += 12;
    writeLong(&length);                 /* # */
    writeLong((long *)"FTXT");           /* "FTXT" for example */
    writeLong((long *)"CHRS");           /* "CHRS" for example */
    writeLong(&slen);                    /* # (length of string) */

    clipboardIO->io_Command = CMD_WRITE;
    clipboardIO->io_Data = (char *)string;
    clipboardIO->io_Length = slen;       /* length of string */
    DoIO(clipboardIO);                 /* text string */

    if(odd)
    {
        clipboardIO->io_Command = CMD_WRITE;
        clipboardIO->io_Data = "";
        clipboardIO->io_Length = 1;
        DoIO(clipboardIO);             /* pad byte */
    }
    clipboardIO->io_Command = CMD_UPDATE;
    DoIO(clipboardIO);
}

long CBPasteS(string)
UBYTE *string;
{
    long length, slen;

```

```

long len[5];

clipboardIO->io_Command = CMD_READ; /* assume FORM # FTXTCHRS # */
clipboardIO->io_ClipID = 0;
clipboardIO->io_Offset = 0; /* offset must be 0 on initial read */
clipboardIO->io_Data = (char *)len;
clipboardIO->io_Length = 20;
DoIO(clipboardIO);

length=len[1]; /* the length of the cut */
slen=len[4]; /* the length of string */

clipboardIO->io_Data = (char *)string; /* read the string */
clipboardIO->io_Length = slen;
DoIO(clipboardIO);

clipboardIO->io_Offset += length; /* read past end of current clip to */
clipboardIO->io_Length = 1; /* close clip for reading */
clipboardIO->io_Data = 0;
DoIO(clipboardIO);

string[slen] = '\0'; /* NULL terminate the string */
return(slen);
}

long CBPost()
{
    clipboardIO->io_Command = CBD_POST;
    clipboardIO->io_Data = (char *)satisfyMsgPort;
    clipboardIO->io_ClipID = 0;
    DoIO(clipboardIO);
    return(clipboardIO->io_ClipID);
}

long CBCurrentReadID()
{
    clipboardIO->io_Command = CBD_CURRENTREADID;
    DoIO(clipboardIO);
    return(clipboardIO->io_ClipID);
}

long CBCurrentWriteID()
{
    clipboardIO->io_Command = CBD_CURRENTWRITEID;
    DoIO(clipboardIO);
    return(clipboardIO->io_ClipID);
}

BOOL CBCheckSatisfy(idVar)
long *idVar;
{
    struct SatisfyMsg *sm;

    if (*idVar == 0)
        return(TRUE);
    if (*idVar < CBCurrentWriteID())
    {
        *idVar = 0;
        return(TRUE);
    }
    if (sm = (struct SatisfyMsg *)GetMsg(satisfyMsgPort))
    {
        if (*idVar == sm->sm_ClipID)
            return(TRUE);
    }
    return(FALSE);
}

```

# Chapter 33

## Console Device

This chapter describes how to do console keyboard input and console (window) output in Amiga Intuition windows. The console device acts like an enhanced ASCII terminal. It obeys many of the standard ANSI sequences as well as additional special sequences unique to the Amiga.

### Introduction

Console I/O is closely associated with the Amiga Intuition interface; a console must be tied to a window that is already opened. From the **Window** data structure, the console device determines how many characters it can display on a line and how many lines of text it can display in a window without clipping at any edge.

You can open the console device many times, if you wish. The result of each open call is a new console unit. AmigaDOS and Intuition see to it that only one window is currently active and its console, if any, is the only one (with a few exceptions) that receives notification of input events, such as keystrokes. Later in this chapter you will see that other Intuition events can be sensed by the console device as well.

#### NOTE

For this entire chapter the characters “<CSI>” represent the *control sequence introducer*. For output you may use either the two-character sequence “<Esc>[” (0x1B 0x5B) or the one-byte value \$9B (hex). For input you will receive \$9B’s unless the sequence has been typed by the user.

## System Functions

The various system device functions such as **DoIO()**, **SendIO()**, **AbortIO()** and **CheckIO()** operate normally. The only caveat is that **CMD\_WRITE** may cause the caller to wait internally, even with **SendIO()**. And a task using **CMD\_READ** waiting on a response from a console is at the user's whim. If a user never reselects that window, and the console response provides the only wake-up call, that task may well sleep indefinitely.

## Console I/O

The console device may be thought of as a kind of terminal. You send character streams to the console device; you also receive them from the console device. These streams may be characters or special sequences.

### GENERAL CONSOLE SCREEN OUTPUT

Console character screen output (as compared to console command sequence transmission) outputs all standard printable characters (character values hex 20 through 7F and A0 through FF) normally.

Many control characters such as **BACKSPACE** and **RETURN** are translated into their exact ANSI equivalent actions. The line-feed character is a bit different, in that it can be translated into a new-line character. The net effect is that the cursor moves to the first column of the next line whenever a **<LF>** is displayed. This option is set via the mode control sequences discussed under "Control Sequences for Window Output."

### CONSOLE KEYBOARD INPUT

If you read from the console device, the keyboard inputs are preprocessed for you and you will get ASCII characters, such as "B." Most normal text-gathering programs will read from the console device in this manner. Some programs may also ask to receive raw events in their console stream. Keypresses are converted to ASCII characters or CSI sequences via the keymap associated with the unit.

The sections below deal with the following topics:

- Setting up for console I/O (creating an I/O request structure)
- Writing to the console to control its behavior
- Reading from the console
- Closing down a console device

The Amiga uses the ECMA-94 Latin1 International 8-bit character set.

				b.	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
				b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
					00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
b.	b.	b.	b.		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	à	P	`	p			NBSP	°	À	Đ	à	ò	
0	0	0	1	01			!	1	A	Q	a	q				ı	±	Á	Ñ	á	ñ
0	0	1	0	02			"	2	B	R	b	r				¢	²	Â	Ò	â	ò
0	0	1	1	03			#	3	C	S	c	s				£	³	Ã	Ó	ã	ó
0	1	0	0	04			\$	4	D	T	d	t				¤	´	Ä	Ô	ä	ô
0	1	0	1	05			%	5	E	U	e	u				¥	µ	Å	Õ	å	õ
0	1	1	0	06			&	6	F	V	f	v				¦	¶	Æ	Ö	æ	ö
0	1	1	1	07			'	7	G	W	g	w				§	·	Ç	×	ç	÷
1	0	0	0	08			(	8	H	X	h	x				"	,	È	Ø	è	ø
1	0	0	1	09			)	9	I	Y	i	y				©	¹	É	Ù	é	ù
1	0	1	0	10			*	:	J	Z	j	z				º	º	Ê	Ú	ê	ú
1	0	1	1	11			+	;	K	Ł	k	ł				«	»	Ë	Û	ë	û
1	1	0	0	12			,	<	L	\	l					¬	¼	Ì	Ü	ì	ü
1	1	0	1	13			-	=	M	Ÿ	m	ÿ				SHY	½	Í	Ý	í	ý
1	1	1	0	14			.	>	N	ˆ	n	˜				®	¾	Î	Ï	î	ï
1	1	1	1	15			/	?	O	_	o					™	¿	İ	ß	ï	ÿ

Figure 33-1: Amiga Character Set

## Creating an I/O Request

This section shows you how to set up for console I/O. Console I/O, like that used with other devices, requires that you create an I/O request message that you pass to the console device for processing. The message contains the command as well as a data area. In the data area, for a write, there will be a pointer to the stream of information you wish to write to the console. For a read, this data pointer shows where the console is to copy the data it has for you. There is also a length field that says how many characters (maximum) are to be copied either from or to the console device.

Here are program fragments that can be used to create the message blocks and ports that you need for console communications.

For *writing* to the console:

```
struct IOStdReq *writeReq = NULL; /* I/O request block pointer */
struct MsgPort *writePort = NULL; /* a port to receive replies */

/* Create reply port and io block for writing to console */
if(!(writePort = CreatePort("RKM.console.write",0)))
    cleanexit("Can't create write port\n",RETURN_FAIL);

if(!(writeReq = CreateExtIO(writePort, (LONG)sizeof(struct IOStdReq))))
    cleanexit("Can't create write request\n",RETURN_FAIL);
```

For *reading* from the console:

```
struct IOStdReq *readReq = NULL; /* I/O request block pointer */
struct MsgPort *readPort = NULL; /* a port to receive replies */

/* Create reply port and io block for reading from console */
if(!(readPort = CreatePort("RKM.console.read",0)))
    cleanexit("Can't create read port\n",RETURN_FAIL);

if(!(readReq = CreateExtIO(readPort, (LONG)sizeof(struct IOStdReq))))
    cleanexit("Can't create read request\n",RETURN_FAIL);
```

These fragments show two messages and two ports being set up. You would use this set-up if you want to have a read command continuously queued up while using a separate message with its associated port to send control command sequences to the console. In addition, if you want to queue up multiple commands to the console, you may wish to create multiple messages (but probably just one port for receiving replied messages from the device).

## Opening a Console Device

To attach a console device to an open Intuition window, you must place your window pointer in the `io_Data` field of your IO request block, then use the request to `OpenDevice()` console.device unit 0. Here is a function that can be used to attach a console device to an existing window. It assumes that `intuition.library` is already open, an Intuition window has been opened, and this new console is to be attached to the open window.



```

/* Attach console device to an open Intuition window.
 * This function returns a value of 0 if the console
 * device opened correctly and a nonzero value (the error
 * returned from OpenDevice) if there was an error.
 */
BYTE OpenConsole(writereq, readreq, window)
struct IOStdReq *writereq;
struct IOStdReq *readreq;
struct Window *window;
{
    BYTE error;

    writereq->io_Data = (APTR) window;
    writereq->io_Length = sizeof(struct Window);
    error = OpenDevice("console.device", 0, writereq, 0);
    readreq->io_Device = writereq->io_Device; /* clone required parts */
    readreq->io_Unit = writereq->io_Unit;
    return(error);
}

```

Notice that this routine opens the console using one I/O request (write), then copies the write request values into the read request. This assures that both input and output go to the same console device.

## THE CONUNIT STRUCTURE

When you successfully attach a console to an Intuition window via **OpenDevice** (as shown in the **OpenConsole()** fragment above), the **io\_Unit** field of your IO request block will be initialized with a pointer to the **ConUnit** structure of your console. Some fields in the **ConUnit** structure can provide useful information when performing console IO. The **cu\_XCP** and **cu\_YCP** fields contain the current column and line position of your cursor converted to an origin of (0,0) from the origin of (1,1) that the console sequences use. The **cu\_XMax** and **cu\_YMax** fields contain the current maximum positions which are 1 less than the number of columns and lines which can be printed in your window. This information is useful when moving the cursor or reformatting your output. The **ConUnit** structure is defined in *devices/conunit.h* and *.i*.

## SENDING A CHARACTER STREAM TO THE CONSOLE DEVICE

To perform console I/O, you fill in fields of the console I/O standard request and pass this block to the console device using one of the normal I/O functions. When the console device has completed the action, the device returns the message block to the port you have designated within the message itself. The function **CreateExtIO()** initializes the message to contain the address of the **ReplyPort**.

To write characters to the console you use the **CMD\_WRITE** command. The **io\_Data** field must point to the character(s) you wish to output. The **io\_Length** field may specify the number of characters to output, or -1 may be used if the character string is null terminated. For high-speed console output, print large numbers of characters with each **CMD\_WRITE**. Turning off the cursor (ESC[0 p) will enhance output speed even further.

### NOTE

If your console is attached to a 1.2/1.3 SuperBitmap window, you will not see a cursor rendered. For output speed and compatibility with future OS versions which may visibly render the cursor, you should send the cursor-off sequence (ESC[0 p) whenever you open or reset (ESCc) a SuperBitmap window's console.

The following console output function uses the `IOStdReq` created above to output a null terminated string.

#### NOTE

The `OpenDevice()` call in `OpenConsole()` initialized the io request block with a pointer to the console which was opened. Thus, a single function such as this one can be used to communicate with multiple consoles.

```
/* Output a NULL-terminated string of characters to a console
*/
void ConPuts(struct IOStdReq *writereq, UBYTE *string)
{
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)string;
    writereq->io_Length = -1; /* means print till terminating null */
    DoIO(writereq);
}
```

## Control Sequences for Window Output

The following table lists functions that the console device supports, along with the character stream that you must send to the console to produce the effect. For more information on the control sequences, consult the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. Where the function table indicates multiple characters, it is more efficient to use the `ConWrite()` function rather than `ConPutChar()` because it avoids the overhead of transferring the message block multiple times. The table uses the second form of <CSI>, that is, the hex value 9B, to minimize the number of characters to be transmitted to produce a function.

In the table, if an item is enclosed in square brackets, it is optional and may be omitted. For example, for INSERT [N] CHARACTERS the value for N or M is shown as optional. The console device responds to such optional items by treating the value of N as if it is not specified. The value of N or M is always a decimal number, having one or more ASCII digits to express its value.

Table 33-1: Console Control Sequences

Command	Sequence of Characters (in Hexadecimal Form)
BACKSPACE (move left one column)	08
LINE FEED (move down one text line as specified by the mode function below)	0A
VERTICAL TAB (move up one text line)	0B
FORM FEED (clear the console's window)	0C
CARRIAGE RETURN (move to first column)	0D
SHIFT IN (undo SHIFT OUT)	0E
SHIFT OUT (set MSB of each character before displaying)	0F
ESC (escape; can be part of the control sequence introducer)	1B
CSI (control sequence introducer)	9B

RESET TO INITIAL STATE	1B 63
INSERT [N] CHARACTERS (Inserts one or more spaces, shifting the remainder of the line to the right.)	9B [N] 40
CURSOR UP [N] CHARACTER POSITIONS (default = 1)	9B [N] 41
CURSOR DOWN [N] CHARACTER POSITIONS (default = 1)	9B [N] 42
CURSOR FORWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 43
CURSOR BACKWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 44
CURSOR NEXT LINE [N] (to column 1)	9B [N] 45
CURSOR PRECEDING LINE [N] (to column 1)	9B [N] 46
MOVE CURSOR TO ROW; COLUMN where N is row, M is column, and semicolon (hex 3B) must be present as a separator, or if row is left out, so the console device can tell that the number after the semicolon actually represents the column number.	9B [N] [3B N] 48
ERASE TO END OF DISPLAY	9B 4A
ERASE TO END OF LINE	9B 4B
INSERT LINE (above the line containing the cursor)	9B 4C
DELETE LINE (remove current line, move all lines up one position to fill gap, blank bottom line)	9B 4D
DELETE CHARACTER [N] (that cursor is sitting on and to the right if [N] is specified)	9B [N] 50
SCROLL UP [N] LINES (Remove line(s) from top of window, move all other lines up, blanks [N] bottom lines)	9B [N] 53
SCROLL DOWN [N] LINES (Remove line(s) from bottom of window, move all other lines down, blanks [N] top lines)	9B [N] 54
SET LINEFEED MODE (cause LINEFEED to respond as RETURN-LINEFEED)	9B 32 30 68
RESET NEWLINE MODE (cause LINEFEED to respond only as LINEFEED)	9B 32 30 6C
DEVICE STATUS REPORT (cause console to insert into your read-stream a CURSOR POSITION REPORT; see "Reading from the Console" for more information)	9B 36 6E
SELECT GRAPHIC RENDITION <style>;<fg>;<bg>6D (select text style foreground color, background color)	See note below.

## NOTE

For SELECT GRAPHIC RENDITION, any number of parameters, in any order, are valid. They are separated by semicolons.

The parameters follow:

<style> =

- 0 Plain text
- 1 Bold-face
- 3 Italic
- 4 Underscore
- 7 Inverse-video

<fg> =

- 30 - 37 Selecting system colors 0-7 for foreground.  
Transmitted as two ASCII characters.

<bg> =

- 40 - 47 selecting system colors 0-7 for background.  
Transmitted as two ASCII characters.

For example, to select bold face, with color 3 as foreground and color 0 as background, send the sequence:

9B 31 3B 33 33 3B 34 30 6D

representing the ASCII sequence:

<CSI>1;33;40m

where <CSI> is the control sequence introducer, here used as the single-character value 9B hex.

The sequences in the next table are not ANSI standard sequences; they are private Amiga sequences. In these command descriptions, length, width, and offset are comprised of one or more ASCII digits, defining a decimal value.

Table 33-2: Amiga Console-control Sequences

| Command  | Sequence of Characters<br>(in Hexadecimal Form) |
|--|---|
| ENABLE SCROLL (this is the default)  | 9B 3E 31 68                                     |
| DISABLE SCROLL   | 9B 3E 31 6C                                     |
| AUTOWRAP ON (the default)  | 9B 3F 37 68                                     |
| AUTOWRAP OFF   | 9B 3F 37 6C                                     |
| SET PAGE LENGTH (in character raster lines,<br>causes console to recalculate,<br>using current font, how many text<br>lines will fit on the page.)   | 9B <length> 74                                  |
| SET LINE LENGTH (in character positions,<br>using current font, how many characters<br>should be placed on each line).   | 9B <width> 75                                   |
| SET LEFT OFFSET (in raster columns, how far<br>from the left of the window<br>should the text begin).  | 9B <offset> 78                                  |
| SET TOP OFFSET (in raster lines, how far<br>from the top of the window's<br>RastPort should the topmost<br>line of the character begin).   | 9B <offset> 79                                  |
| SET RAW EVENTS—see the separate<br>topic “Selecting Raw Input Events”<br>below for more details.   |   |
| RESET RAW EVENTS—see<br>“Selecting Raw Input Events” below.  |   |
| SET CURSOR RENDITION - make the<br>cursor visible or invisible:<br>(Note - turning off the cursor<br>increases text output speed)  |   |
| Invisible:   | 9B 30 20 70                                     |
| Visible:   | 9B 20 70  |
| WINDOW STATUS REQUEST - ask the<br>console device to tell you the<br>current bounds of the window,<br>in upper and lower row and<br>column character positions.<br>(User may have resized or<br>repositioned it.) See<br>“Window Bounds Report” below. | 9B 30 20 71                                     |

## NOTE

The console device normally handles the SET PAGE LENGTH, SET LINE LENGTH, SET LEFT OFFSET, and SET TOP OFFSET functions automatically. To allow it to do so again after setting your own values, you can send the function without a parameter.

## Examples

Move cursor right by 1:

Character string equivalents:

<CSI>C or  
<CSI>1C

Numeric (hex) equivalents:

9B 43  
9B 31 43

Move cursor right by 20:

Character string equivalent: <CSI>20C  
Numeric (hex) equivalent: 9B 32 30 43

Move cursor to upper-left corner (home):

Character string equivalents:

<CSI>H or  
<CSI>1;1H or  
<CSI>;1H or  
<CSI>1;H

Numeric (hex) equivalents:

9B 48  
9B 31 3B 31 48  
9B 3B 31 48  
9B 31 3B 48

Move cursor to the fourth column of the first line of the window:

Character string equivalents:

<CSI>1;4H or  
<CSI>;4H

Numeric (hex) equivalents:

9B 31 3B 34 48  
9B 3B 34 48

Clear the window:

Character string equivalents:

<FF> or CTRL-L {clear window} or  
<CSI>H<CSI>J {home and clear to end of window} or

Numeric (hex) equivalents:

0C  
9B 48 9B 4A

## READING FROM THE CONSOLE

Reading input from the console device returns an ANSI 3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS and RESET RAW EVENTS control sequences discussed below. See "Selection of Raw Input Events."

Generally, console reads are performed asynchronously so that your program can respond to other events and other user input (such as menu selections) when the user is not typing on the keyboard. To perform asynchronous IO, a **CMD\_READ** request is sent to the console using the **SendIO()** function (rather than a synchronous **DoIO()** which would wait until the read request returned with a character).

### NOTE

A request for more than one character can be satisfied by only one. This example function requests one character at a time. If you request more than one character, you will have to examine the **io\_Actual** field of the request when it returns to determine how many characters you have actually received.

```
/* Send a read request to the console, passing it a pointer
 * to a buffer into which it can read the character(s)
 */
void QueueRead(struct IOStdReq *readreq, UBYTE *whereto)
{
    readreq->io_Command = CMD_READ;
    readreq->io_Data = (APTR)whereto;
    readreq->io_Length = 1;
    SendIO(readreq);
}
```

After sending the read request, your program can wait on a combination of signal bits including that of the reply port you passed to **CreateExtIO()** when creating the read request block. The following fragment demonstrates waiting on both a queued console read request, and Window IDCMP messages:

```
conreadsig = 1 << readPort->mp_SigBit;
windowSIG = 1 << win->UserPort->mp_SigBit;

/* A character, or an IDCMP msg, or both will wake us up */
signals = Wait(conreadsig|windowSIG);

if (signals & conreadsig) { /* Then check for a character */ };
if (signals & windowSIG) { /* Then check window messages */ };
```

The following function will get a character if the read request has returned, and then queue the next read request:

```
/* Check if a character has been received.
 * If none, return -1
 */
LONG ConMayGetChar(struct MsgPort *msgport, UBYTE *wheretoe)
{
    register temp;
    struct IOStdReq *readreq;

    if (!(readreq = (struct IOStdReq *)GetMsg(msgport))) return(-1);
    temp = *wheretoe;          /* get the character */
    QueueRead(readreq, wheretoe); /* then re-use the request block */
    return(temp);
}
```

## Closing a Console Device

When you have finished using a console, it must be closed so that the memory areas it utilized can be returned to the system memory manager. You should abort and wait for any pending requests, and then close the device. Here is a sequence that you could use to close a console device opened for read and write, with a pending queued read request:

```
/* If console read request is still out, abort it */
if(!(CheckIO(readReq))) AbortIO(readReq);
WaitIO(readReq); /* clear it from our replyport */
CloseDevice(writeReq);
```

### NOTE

You also need to delete the messages and ports associated with this console after the console has been closed:

```
if(readReq) DeleteExtIO(readReq);
if(readPort) DeletePort(readPort);
if(writeReq) DeleteExtIO(writeReq);
if(writePort) DeletePort(writePort);
```

If you have finished with the window used for the console device, you can now close it.

## Console Device Example Code

The following is a console device demonstration program with supporting routines:

```
/* Console.c - console.device example
 * Compiled with Lattice 5.02: LC -bl -cfist -v -y
 * Linkage: c.o,console.o library LC.lib,amiga.lib
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#include <devices/console.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
```



```

#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

/* Note - using two character <CSI> ESC[. Hex 9B could be used instead */
#define RESETCON "\033c"
#define CURSOFF "\033[0 p"
#define CURSON "\033[ p"
#define DELCHAR "\033[P"

/* SGR (set graphic rendition) */
#define COLOR02 "\033[32m"
#define COLOR03 "\033[33m"
#define ITALICS "\033[3m"
#define BOLD "\033[1m"
#define UNDERLINE "\033[4m"
#define NORMAL "\033[0m"

/* our functions */
void cleanexit(UBYTE *,LONG);
void cleanup(void);
BYTE OpenConsole(struct IOStdReq *,struct IOStdReq *, struct Window *);
void CloseConsole(struct IOStdReq *);
void QueueRead(struct IOStdReq *, UBYTE *);
UBYTE ConGetChar(struct MsgPort *, UBYTE *);
LONG ConMayGetChar(struct MsgPort *, UBYTE *);
void ConPuts(struct IOStdReq *, UBYTE *);
void ConWrite(struct IOStdReq *, UBYTE *, LONG);
void ConPutChar(struct IOStdReq *, UBYTE);

struct NewWindow nw =
{
    10, 10, /* starting position (left,top) */
    620,180, /* width, height */
    -1,-1, /* detailpen, blockpen */
    CLOSEWINDOW, /* flags for idcmp */
    WINDOWDEPTH|WINDOWSIZING|
    WINDOWDRAG|WINDOWCLOSE|
    SMART_REFRESH|ACTIVATE, /* window flags */
    NULL, /* no user gadgets */
    NULL, /* no user checkmark */
    "Console Test", /* title */
    NULL, /* pointer to window screen */
    NULL, /* pointer to super bitmap */
    100,45, /* min width, height */
    640,200, /* max width, height */
    WBENCHSCREEN /* open on workbench screen */
};

/* Opens/allocations we'll need to clean up */
struct Library *IntuitionBase = NULL;
struct Window *win = NULL;
struct IOStdReq *writeReq = NULL; /* I/O request block pointer */
struct MsgPort *writePort = NULL; /* replyport for writes */
struct IOStdReq *readReq = NULL; /* I/O request block pointer */
struct MsgPort *readPort = NULL; /* replyport for reads */
BOOL OpenedConsole = FALSE;

BOOL FromWb;

void main(argc, argv)
int argc;
char **argv;
{
    struct IntuiMessage *winmsg;
    ULONG signals, conreadsig, windowmsg;
    LONG lch;
    SHORT InControl = 0;
    BOOL Done = FALSE;
    UBYTE ch, ibuf;

```

```

UBYTE obuf[200];
BYTE error;

FromWb = (argc==0L) ? TRUE : FALSE;

if(!(IntuitionBase=OpenLibrary("intuition.library",0)))
    cleanexit("Can't open intuition\n",RETURN_FAIL);

/* Create reply port and io block for writing to console */
if(!(writePort = CreatePort("RKM.console.write",0)))
    cleanexit("Can't create write port\n",RETURN_FAIL);

if(!(writeReq = CreateExtIO(writePort, (LONG)sizeof(struct IOStdReq))))
    cleanexit("Can't create write request\n",RETURN_FAIL);

/* Create reply port and io block for reading from console */
if(!(readPort = CreatePort("RKM.console.read",0)))
    cleanexit("Can't create read port\n",RETURN_FAIL);

if(!(readReq = CreateExtIO(readPort, (LONG)sizeof(struct IOStdReq))))
    cleanexit("Can't create read request\n",RETURN_FAIL);

/* Open a window */
if(!(win = OpenWindow(&nw)))
    cleanexit("Can't open window\n",RETURN_FAIL);

/* Now, attach a console to the window */
if(error = OpenConsole(writeReq,readReq,win))
    cleanexit("Can't open console.device\n",RETURN_FAIL);
else OpenedConsole = TRUE;

/* Demonstrate some console escape sequences */
ConPuts(writeReq,"Here's some normal text\n");
sprintf(obuf,"%s%sHere's text in color 3 and italics\n",COLOR03,ITALICS);
ConPuts(writeReq,obuf);
ConPuts(writeReq,NORMAL);
Delay(50); /* Delay for dramatic demo effect */
ConPuts(writeReq,"We will now delete this asterisk ===");
Delay(50);
ConPuts(writeReq,"\b\b"); /* backspace twice */
Delay(50);
ConPuts(writeReq,DELCHAR); /* delete the character */
Delay(50);

QueueRead(readReq,&ibuf); /* send the first console read request */

ConPuts(writeReq,"\n\nNow reading console\n");
ConPuts(writeReq,"Type some keys. Close window when done.\n\n");

conreadsig = 1 << readPort->mp_SigBit;
windowsig = 1 << win->UserPort->mp_SigBit;

while(!Done)
{
    /* A character, or an IDCMP msg, or both could wake us up */
    signals = Wait(conreadsig|windowsig);

    /* If a console signal was received, get the character */
    if (signals & conreadsig)
    {
        if((lch = ConMayGetChar(readPort,&ibuf)) != -1)
        {
            ch = lch;
            /* Show hex and ascii (if printable) for char we got.
             * If you want to parse received control sequences, such as
             * function or Help keys, you would buffer control sequences
             * as you receive them, starting to buffer whenever you
             * receive 0x9B (or 0x1B[ for user-typed sequences) and
             * ending when you receive a valid terminating character
             * for the type of control sequence you are receiving.
             * For CSI sequences, valid terminating characters
             * are generally 0x40 through 0x7E.
            */

```

```

* In our example, InControl has the following values:
* 0 = no, 1 = have 0x1B, 2 = have 0x9B OR 0x1B and [,
* 3 = now inside control sequence, -1 = normal end esc,
* -2 = non-CSI(no []) 0x1B end esc
* NOTE - a more complex parser is required to recognize
* other types of control sequences.
*/

/* 0x1B ESC not followed by '[', is not CSI seq */
if (InControl==1)
{
    if(ch=='[') InControl = 2;
    else InControl = -2;
}

if ((ch==0x9B)|| (ch==0x1B)) /* Control seq starting */
{
    InControl = (ch==0x1B) ? 1 : 2;
    ConPuts(writeReq, "=== Control Seq ===\n");
}

/* We'll show value of this char we received */
if (((ch >= 0x1F) && (ch <= 0x7E)) || (ch >= 0xA0))
    sprintf(obuf, "Received: hex %02x = %c\n", ch, ch);
else sprintf(obuf, "Received: hex %02x\n", ch);
ConPuts(writeReq, obuf);

/* Valid ESC sequence terminator ends an ESC seq */
if ((InControl==3) && ((ch >= 0x40) && (ch <= 0x7E)))
{
    InControl = -1;
}
if (InControl==2) InControl = 3;
/* ESC sequence finished (-1 if OK, -2 if bogus) */
if (InControl < 0)
{
    InControl = 0;
    ConPuts(writeReq, "=== End Control ===\n");
}
}

/* If IDCMP messages received, handle them */
if (signals & windowSIG)
{
    /* We have to ReplyMsg these when done with them */
    while (winmsg = (struct IntuiMessage *)GetMsg(win->UserPort))
    {
        switch(winmsg->Class)
        {
            case CLOSEWINDOW:
                Done = TRUE;
                break;
            default:
                break;
        }
        ReplyMsg((struct Message *)winmsg);
    }
}

/* We always have an outstanding queued read request
* so we must abort it if it hasn't completed,
* and we must remove it.
*/
if(!(CheckIO(readReq))) AbortIO(readReq);
WaitIO(readReq); /* clear it from our replyport */

cleanup();
exit(RETURN_OK);
}

```

```

void cleanexit(UBYTE *s, LONG n)
{
    if(*s & (!FromWb)) printf(s);
    cleanup();
    exit(n);
}

void cleanup()
{
    if(OpenedConsole) CloseConsole(writeReq);
    if(readReq) DeleteExtIO(readReq);
    if(readPort) DeletePort(readPort);
    if(writeReq) DeleteExtIO(writeReq);
    if(writePort) DeletePort(writePort);
    if(win) CloseWindow(win);
    if(IntuitionBase) CloseLibrary(IntuitionBase);
}

/* Attach console device to an open Intuition window.
 * This function returns a value of 0 if the console
 * device opened correctly and a nonzero value (the error
 * returned from OpenDevice) if there was an error.
 */
BYTE OpenConsole(writereq, readreq, window)
struct IOStdReq *writereq;
struct IOStdReq *readreq;
struct Window *window;
{
    BYTE error;

    writereq->io_Data = (APTR) window;
    writereq->io_Length = sizeof(struct Window);
    error = OpenDevice("console.device", 0, writereq, 0);
    readreq->io_Device = writereq->io_Device; /* clone required parts */
    readreq->io_Unit = writereq->io_Unit;
    return(error);
}

void CloseConsole(struct IOStdReq *writereq)
{
    CloseDevice(writereq);
}

/* Output a single character to a specified console
 */
void ConPutChar(struct IOStdReq *writereq, UBYTE character)
{
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)&character;
    writereq->io_Length = 1;
    DoIO(writereq);
    /* command works because DoIO blocks until command is done
     * (otherwise ptr to the character could become invalid)
     */
}

/* Output a stream of known length to a console
 */
void ConWrite(struct IOStdReq *writereq, UBYTE *string, LONG length)
{
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)string;
    writereq->io_Length = length;
    DoIO(writereq);
    /* command works because DoIO blocks until command is done
     * (otherwise ptr to string could become invalid in the meantime)
     */
}

```

```

/* Output a NULL-terminated string of characters to a console
*/
void ConPuts(struct IOStdReq *writereq, UBYTE *string)
{
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)string;
    writereq->io_Length = -1; /* means print till terminating null */
    DoIO(writereq);
}

/* Queue up a read request to console, passing it pointer
 * to a buffer into which it can read the character
 */
void QueueRead(struct IOStdReq *readreq, UBYTE *whereteto)
{
    readreq->io_Command = CMD_READ;
    readreq->io_Data = (APTR)whereteto;
    readreq->io_Length = 1;
    SendIO(readreq);
}

/* Check if a character has been received.
 * If none, return -1
 */
LONG ConMayGetChar(struct MsgPort *msgport, UBYTE *whereteto)
{
    register temp;
    struct IOStdReq *readreq;

    if (!(readreq = (struct IOStdReq *)GetMsg(msgport))) return(-1);
    temp = *whereteto; /* get the character */
    QueueRead(readreq, whereteto); /* then re-use the request block */
    return(temp);
}

/* Wait for a character
 */
UBYTE ConGetChar(struct MsgPort *msgport, UBYTE *whereteto)
{
    register temp;
    struct IOStdReq *readreq;

    WaitPort(msgport);
    readreq = (struct IOStdReq *)GetMsg(msgport);
    temp = *whereteto; /* get the character */
    QueueRead(readreq, whereteto); /* then re-use the request block */
    return((UBYTE)temp);
}

```

## INFORMATION ABOUT THE READ-STREAM

For the most part, keys whose keycaps are labeled with ANSI-standard characters will ordinarily be translated into their ASCII-equivalent character by the console device through the use of its keymap. A separate section in this chapter has been dedicated to the method used to establish a keymap and the internal organization of the keymap.

For keys other than those with normal ASCII equivalents, an escape sequence is generated and inserted into your input stream. For example, in the default state (no raw input events selected) the function and arrow keys will cause the sequences shown in the next table to be inserted in the input stream.

Table 33-3: Special Key Report Sequences

| Key         | Unshifted Sends | Shifted Sends |                   |
|-------------|-----------------|---------------|-------------------|
| F1          | <CSI>0~         | <CSI>10~      |                   |
| F2          | <CSI>1~         | <CSI>11~      |                   |
| F3          | <CSI>2~         | <CSI>12~      |                   |
| F4          | <CSI>3~         | <CSI>13~      |                   |
| F5          | <CSI>4~         | <CSI>14~      |                   |
| F6          | <CSI>5~         | <CSI>15~      |                   |
| F7          | <CSI>6~         | <CSI>16~      |                   |
| F8          | <CSI>7~         | <CSI>17~      |                   |
| F9          | <CSI>8~         | <CSI>18~      |                   |
| F10         | <CSI>9~         | <CSI>19~      |                   |
| HELP        | <CSI>?~         | <CSI>?~       | (same)            |
| Arrow keys: |                 |               |                   |
| Up          | <CSI>A          | <CSI>T        |                   |
| Down        | <CSI>B          | <CSI>S        |                   |
| Left        | <CSI>D          | <CSI> A       | (notice the space |
| Right       | <CSI>C          | <CSI> @       | after <CSI>)      |

## CURSOR POSITION REPORT

If you have sent the DEVICE STATUS REPORT command sequence, the console device returns a cursor position report into your input stream. It takes the form:

<CSI><row>;<column>R

For example, if the cursor is at column 40 and row 12, here are the ASCII values you receive in a stream:

9B 34 30 3B 31 32 52

## WINDOW BOUNDS REPORT

A user may have either moved or resized the window to which your console is bound. By issuing a WINDOW STATUS REPORT to the console, you can read the current position and size in the input stream. This window bounds report takes the following form:

<CSI>1;1;<bottom margin>;<right margin>r

The bottom and right margins give you the window row and column dimensions as well. For a window that holds 20 lines with 60 characters per line, you will receive the following in the input stream:

9B 31 3B 31 3B 32 30 3B 36 30 20 72

## SELECTING RAW INPUT EVENTS

If the keyboard information—including “cooked” keystrokes—does not give you enough information about input events, you can request additional information from the console driver.

The command to SET RAW EVENTS is formatted as:

```
<CSI>[event-types-separated-by-semicolons]{
```

If, for example, you need to know when each key is pressed and released, you would request “RAW keyboard input.” This is done by writing “<CSI>1{” to the console. In a single SET RAW EVENTS request, you can ask the console to set up for multiple event types at one time. You must send multiple numeric parameters, separating them by semicolons (;). For example, to ask for gadget pressed, gadget released, and close gadget events, write “<CSI>7;8;11{” (all as ASCII characters, without the quotes).

You can reset, that is, delete from reporting, one or more of the raw input event types by using the RESET RAW EVENTS command, in the same manner as the SET RAW EVENTS was used to establish them in the first place. This command stream is formatted as:

```
<CSI>[event-types-separated-by-semicolons]}
```

So, for example, you could reset all of the events set in the above example by transmitting the command sequence: “<CSI>7;8;11}.” The following table lists the valid raw input event types.

Table 33-4: Raw Input Event Types

| Request Number | Description         |   |
|----------------|---------------------|---|
| 0              | No-op               | Used internally                                 |
| 1              | RAW keyboard input  | Intuition swallows all except the select button |
| 2              | RAW mouse input     |   |
| 3              | Event               | Sent whenever your window is made active        |
| 4              | Pointer position    |   |
| 5              | (unused)            |   |
| 6              | Timer               |   |
| 7              | Gadget pressed      |   |
| 8              | Gadget released     |   |
| 9              | Requester activity  |   |
| 10             | Menu numbers        |   |
| 11             | Close Gadget        |   |
| 12             | Window resized      |   |
| 13             | Window refreshed    |   |
| 14             | Preferences changed |   |
| 15             | Disk removed        |   |
| 16             | Disk inserted       |   |

# Complex Input Event Reports

If you select any of these events you will start to get information about the events in the following form:

`<CSI><class>;<subclass>;<keycode>;<qualifiers>;<x>;<y>;<seconds>;<microseconds>|`

## **<CSI>**

is a one-byte field. It is the “control sequence introducer,” 9B in hex.

## **<class>**

is the RAW input event type, from the above table.

## **<subclass>**

is usually 0. If the mouse is moved to the right controller, this would be 1.

## **<keycode>**

indicates which raw key number was pressed. This field can also be used for mouse information.

## **NOTE**

National keyboards often have different keyboard arrangements. This means that a particular raw key number may represent different characters on different national keyboards. The normal console read stream (as opposed to raw events) will contain the proper ASCII character for the keypress as translated according to the user’s keymap.

## **<qualifiers>**

indicates the state of the keyboard and system. The qualifiers are defined as follows:



Table 33-5: Input Event Qualifiers

| Bit | Mask | Key                     |   |
|-----|------|-------------------------|---|
| 0   | 0001 | Left shift              |   |
| 1   | 0002 | Right shift             |   |
| 2   | 0004 | Caps Lock               | Associated keycode is special; see below.               |
| 3   | 0008 | Ctrl                    |   |
| 4   | 0010 | Left Alt                |   |
| 5   | 0020 | Right Alt               |   |
| 6   | 0040 | Left Amiga key pressed  |   |
| 7   | 0080 | Right Amiga key pressed |   |
| 8   | 0100 | Numeric pad             |   |
| 9   | 0200 | Repeat                  |   |
| 10  | 0400 | Interrupt               | Not currently used.                                     |
| 11  | 0800 | Multi-broadcast         | This window (active one) or all windows.                |
| 12  | 1000 | Left mouse button       |   |
| 13  | 2000 | Right mouse button      |   |
| 14  | 4000 | Middle mouse button     | (Not available on standard mouse)                       |
| 15  | 8000 | Relative mouse          | Indicates mouse coordinates are relative, not absolute. |

The Caps Lock key is handled in a special manner. It generates a keycode only when it is pressed, not when it is released. However, the up/down bit (80 hex) is still used and reported. If pressing the Caps Lock key causes the LED to light, keycode 62 (Caps Lock pressed) is sent. If pressing the Caps Lock key extinguishes the LED, keycode 190 (Caps Lock released) is sent. In effect, the keyboard reports this key as held down until it is struck again.

The <x> and <y> fields are filled by some classes with an Intuition address:  $x \ll 16 + y$ .

The <seconds> and <microseconds> fields contain the system time stamp taken at the time the event occurred. These values are stored as long-words by the system.

With RAW keyboard input selected, keys will no longer return a simple one-character “a” to “z” but will instead return raw keycode reports of the form:

```
<CSI>1;0;<keycode>;<qualifiers>;<prev1>;<prev2>;<seconds>;<microseconds>|
```

For example, if the user pressed and released the ‘B’ key with the left Shift and right Amiga keys also pressed, you might receive the following data:

```
<CSI>1;0;35;129;0;0;23987;99|
<CSI>1;0;163;129;0;0;24003;18|
```

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code will result in the released keycode.

The <prev1> and <prev2> fields are relevant for the interpretation of keys which are modifiable by dead-keys (see “Dead-Class Keys” section). The <prev1> field shows the previous key pressed. The lower byte shows the qualifier, the upper byte shows the key code. The <prev2> field shows the key pressed before the previous key. The

lower byte shows the qualifier, the upper byte shows the key code.

### NOTE

The Amiga, Alt, Shift, CTRL and CAPS Lock keys are excluded from previous key reporting.

The keys with keycodes \$2B and \$30 in the following keyboard diagrams are keys which are present on some national Amiga keyboards.

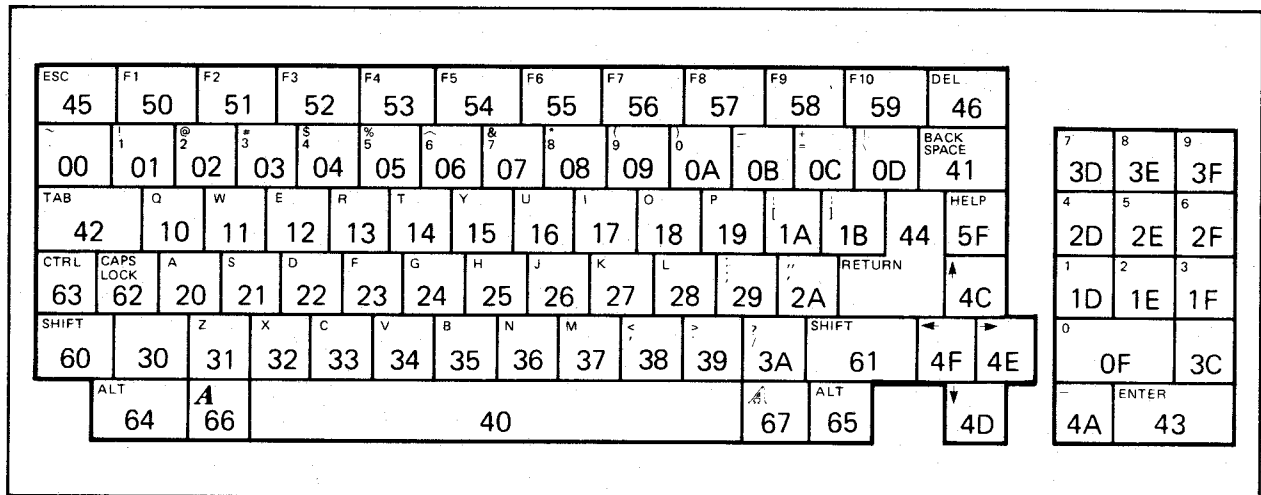


Figure 33-2: Amiga 1000 Keyboard Showing Keycodes in Hex

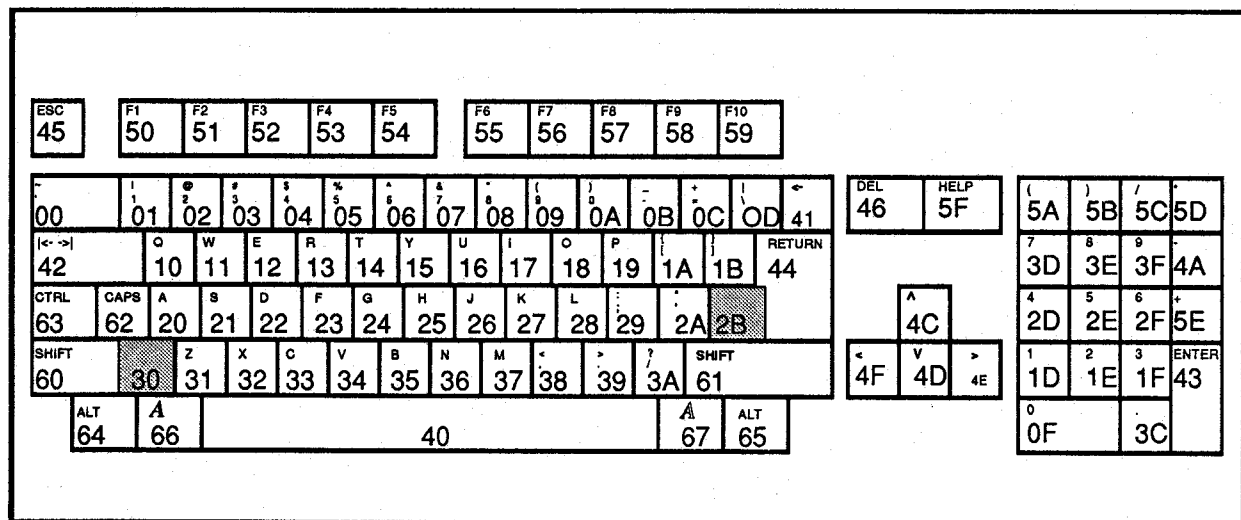


Figure 33-3: Amiga 500/2000 Keyboard Showing Keycodes in Hex

The default values given correspond to the values the console device will return when these keys are pressed with the keycaps as shipped with the standard American keyboard.

Table 33-6: ROM Default (USA0) and USA1 Console Key Mapping

| Raw Key Number | Keycap Legend | Unshifted Default Value | Shifted Default Value |
|----------------|---------------|-------------------------|-----------------------|
| 00             | ``            | ‘ (Accent grave)        | ~ (tilde)             |
| 01             | 1!            | 1                       | !                     |
| 02             | 2@            | 2                       | @                     |
| 03             | 3#            | 3                       | #                     |
| 04             | 4\$           | 4                       | \$                    |
| 05             | 5%            | 5                       | %                     |
| 06             | 6^            | 6                       | ^                     |
| 07             | 7&            | 7                       | &                     |
| 08             | 8*            | 8                       | *                     |
| 09             | 9(            | 9                       | (                     |
| 0A             | 0)            | 0                       | )                     |
| 0B             | - _           | - (Hyphen)              | _ (Underscore)        |
| 0C             | = +           | =                       | +                     |
| 0D             | \             | \                       |                       |
| 0E             |               | (undefined)             |                       |
| 0F             | 0             | 0                       | 0 (Numeric pad)       |
| 10             | Q             | q                       | Q                     |
| 11             | W             | w                       | W                     |
| 12             | E             | e                       | E                     |
| 13             | R             | r                       | R                     |
| 14             | T             | t                       | T                     |
| 15             | Y             | y                       | Y                     |
| 16             | U             | u                       | U                     |
| 17             | I             | i                       | I                     |
| 18             | O             | o                       | O                     |
| 19             | P             | p                       | P                     |
| 1A             | [ {           | [                       | {                     |
| 1B             | ] }           | ]                       | }                     |
| 1C             |               | (undefined)             |                       |
| 1D             | 1             | 1                       | 1 (Numeric pad)       |
| 1E             | 2             | 2                       | 2 (Numeric pad)       |
| 1F             | 3             | 3                       | 3 (Numeric pad)       |
| 20             | A             | a                       | A                     |
| 21             | S             | s                       | S                     |
| 22             | D             | d                       | D                     |
| 23             | F             | f                       | F                     |
| 24             | G             | g                       | G                     |
| 25             | H             | h                       | H                     |
| 26             | J             | j                       | J                     |
| 27             | K             | k                       | K                     |
| 28             | L             | l                       | L                     |
| 29             | ::            | :                       | :                     |
| 2A             | ;"            | ' (single quote)        | "                     |
| 2B             |               | (not on US keyboards)   |                       |
| 2C             |               | (undefined)             |                       |

| Raw Key Number | Keycap Legend  | Unshifted Default Value | Shifted Default Value                     |
|----------------|----------------|-------------------------|---|
| 2D             | 4              | 4                       | 4 (Numeric pad)                           |
| 2E             | 5              | 5                       | 5 (Numeric pad)                           |
| 2F             | 6              | 6                       | 6 (Numeric pad)                           |
| 30             |                | (not on US keyboards)   |   |
| 31             | Z              | z                       | Z   |
| 32             | X              | x                       | X   |
| 33             | C              | c                       | C   |
| 34             | V              | v                       | V   |
| 35             | B              | b                       | B   |
| 36             | N              | n                       | N   |
| 37             | M              | m                       | M   |
| 38             | , <            | , (comma)               | <   |
| 39             | . >            | . (period)              | >   |
| 3A             | / ?            | /                       | ?   |
| 3B             |                | (undefined)             |   |
| 3C             | .              | .                       | . (Numeric pad)                           |
| 3D             | 7              | 7                       | 7 (Numeric pad)                           |
| 3E             | 8              | 8                       | 8 (Numeric pad)                           |
| 3F             | 9              | 9                       | 9 (Numeric pad)                           |
| 40             | (Space bar)    | 20                      | 20  |
| 41             | Back Space     | 08                      | 08  |
| 42             | Tab            | 09                      | 09  |
| 43             | Enter          | 0D                      | 0D (Numeric pad)                          |
| 44             | Return         | 0D                      | 0D  |
| 45             | Esc            | 1B                      | 1B  |
| 46             | Del            | 7F                      | 7F  |
| 47             |                | (undefined)             |   |
| 48             |                | (undefined)             |   |
| 49             |                | (undefined)             |   |
| 4A             |                | -                       | - (Numeric Pad)                           |
| 4B             |                | (undefined)             |   |
| 4C             | Up arrow       | <CSI>A                  | <CSI>T                                    |
| 4D             | Down arrow     | <CSI>B                  | <CSI>S                                    |
| 4E             | Forward arrow  | <CSI>C                  | <CSI> A<br>(note blank space after <CSI>) |
| 4F             | Backward arrow | <CSI>D                  | <CSI> @<br>(note blank space after <CSI>) |
| 50             | F1             | <CSI>0~                 | <CSI>10~                                  |
| 51             | F2             | <CSI>1~                 | <CSI>11~                                  |
| 52             | F3             | <CSI>2~                 | <CSI>12~                                  |
| 53             | F4             | <CSI>3~                 | <CSI>13~                                  |
| 54             | F5             | <CSI>4~                 | <CSI>14~                                  |
| 55             | F6             | <CSI>5~                 | <CSI>15~                                  |
| 56             | F7             | <CSI>6~                 | <CSI>16~                                  |
| 57             | F8             | <CSI>7~                 | <CSI>17~                                  |

| Raw<br>Key<br>Number | Keycap<br>Legend | Unshifted<br>Default<br>Value | Shifted<br>Default<br>Value |
|----------------------|------------------|-------------------------------|-----------------------------|
| 58                   | F9               | <CSI>8~                       | <CSI>18~                    |
| 59                   | F10              | <CSI>9~                       | <CSI>19~                    |
| 5A                   | (                | (                             | ( (usa1 Numeric pad)        |
| 5B                   | )                | )                             | ) (usa1 Numeric pad)        |
| 5C                   | /                | /                             | / (usa1 Numeric pad)        |
| 5D                   | *                | *                             | * (usa1 Numeric pad)        |
| 5E                   | +                | +                             | + (usa1 Numeric pad)        |
| 5F                   | HELP             | <CSI>?~                       | <CSI>?~                     |

| Raw<br>Key<br>Number | Function or<br>Keycap<br>Legend        |   |
|----------------------|--|---|
| 60                   | Shift (left of space bar)              |   |
| 61                   | Shift (right of space bar)             |   |
| 62                   | Caps Lock                              |   |
| 63                   | Ctrl                                   |   |
| 64                   | (Left) Alt                             |   |
| 65                   | (Right) Alt                            |   |
| 66                   | Amiga (left of space bar)              | Left Amiga  |
| 67                   | Amiga (right of space bar)             | Right Amiga   |
| 68                   | Left mouse button<br>(not converted)   | Inputs are only for the<br>mouse connected to Intuition,<br>currently "gameport" one. |
| 69                   | Right mouse button<br>(not converted)  |   |
| 6A                   | Middle mouse button<br>(not converted) |   |
| 6B                   | (undefined)                            |   |
| 6C                   | (undefined)                            |   |
| 6D                   | (undefined)                            |   |
| 6E                   | (undefined)                            |   |
| 6F                   | (undefined)                            |   |

| <b>Raw<br/>Key<br/>Number</b> | <b>Function or<br/>Keycap<br/>Legend</b>   |
|-------------------------------|--|
| 70-7F                         | (undefined)  |
| 80-F8                         | Up transition (release or unpress key of one of the above keys) (80 for 00, F8 for 7F)             |
| F9                            | Last keycode was bad<br>(was sent in order to resynchronize)                                       |
| FA                            | Keyboard buffer overflow   |
| FB                            | (undefined, reserved for<br>keyboard processor catastrophe)  |
| FC                            | Keyboard selftest failed   |
| FD                            | Power-up key stream start.<br>Keys pressed or stuck at power-up<br>will be sent between FD and FE. |
| FE                            | Power-up key stream end  |
| FF                            | (undefined, reserved)  |
| FF                            | Mouse event, movement only,<br>no button change (not converted)                                    |

Notes about the preceding table:

- 1) “(undefined)” indicates that the current keyboard design should not generate this number. If you are using `SetKeyMap()` to change the key map, the entries for these numbers must still be included.
- 2) “(not converted)” refers to mouse button events. You must use the sequence “<CSI>2{” to inform the console driver that you wish to receive mouse events; otherwise these will not be transmitted.
- 3) “(RESERVED)” indicates that these keycodes have been reserved for national keyboards. The “2B” code key will be between the double-quote(”) and Return keys. The “30” code key will be between the Shift and “Z” keys.

## Using the Console Device Without a Window

Although most console device commands require a window, there are some special commands and functions which may be used without a window, by opening console.device unit -1. The `CD_ASKDEFAULTKEYMAP` and `CD_SETDEFAULTKEYMAP` commands, and the console.device functions `CDInputHandler()` and `RawKeyConvert()` may be used this way. To use the console device functions, you can `OpenDevice()` console unit -1, then use the `io_Device` field as the library base for the calls. C programmers would do this by copying `io_Device` to an externally visible variable named `ConsoleDevice`. See the Intuition chapters for an example of `RawKeyConvert()`.

# Keymapping

The Amiga has the capability of mapping the physical keyboard keys to any ECMA-94 Latin 1 character or string. The Amiga computers are sold in many countries with different national keyboards. The national keyboards differ in the positions of (and therefore raw key values of) many alphanumeric and special characters. For example, the “Y” on the German and Italian keyboards is positioned where U.S. keyboards have a “Z”. A startup command (SetMap) uses the console device commands CD\_SETKEYMAP and CD\_SETDEFAULTKEYMAP to set both the initial window and console device default to the correct national keymap. The console.device translates raw key events into the correct ASCII characters based on the installed keymap. Since V1.2 this includes the translation of special “deadkey” sequential key combinations to produce accented international characters. Programs which perform keyboard input using the console.device, CON:, RAW:, or Intuition VANILLAKEY, will receive the correct ASCII value for a user’s keypress, regardless of where the key is positioned on that user’s keyboard.

A particular console’s current keymap can be examined or replaced using the console device CD\_ASKKEYMAP and CD\_SETKEYMAP commands. The default console device keymap can be similarly examined or replaced using the CD\_ASKDEFAULTKEYMAP and CD\_SETDEFAULTKEYMAP commands. These commands each deal with a set of eight longword pointers, known as the **KeyMap** data structure. The **KeyMap** data structure is shown below.

```
struct KeyMap
{
    UBYTE *km_LoKeyMapTypes;
    ULONG *km_LoKeyMap;
    UBYTE *km_LoCapsable;
    UBYTE *km_LoRepeatable;
    UBYTE *km_HiKeyMapTypes;
    ULONG *km_HiKeyMap;
    UBYTE *km_HiCapsable;
    UBYTE *km_HiRepeatable;
};
```

The function **AskKeyMap()** shown below does not return a pointer to a table of pointers to currently assigned key mapping. Instead, it *copies* the current set of pointers to a user-designated area of memory. **AskKeyMap()** returns a TRUE/FALSE value that says whether or not the function succeeded.

The function **SetKeyMap()**, also shown below, copies the designated key map data structure to the console device. Thus this routine is complementary to **AskKeymap()** in that it can restore an original key mapping as well as establish a new one.

```
/* These functions require that you have set up a port and
 * io request, and have opened the console device as shown
 * earlier in the chapter.
 * They also require this additional include file.
 */
include <devices/keymap.h>

BOOL AskKeyMap(struct IOStdReq *request, struct KeyMap *keymap)
{
    request->io_Command = CD_ASKKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap; /* where to put it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else return(TRUE); /* if no error, it worked. */
}
```

```

BOOL SetKeyMap(struct IOStdReq *request, struct KeyMap *keymap)
{
    request->io_Command = CD_SETKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap;      /* where to get it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else return(TRUE);      /* if no error, it worked. */
}

```

As a prelude to the following material, note that the Amiga keyboard transmits raw key information to the computer in the form of a key position and a transition. When the key is released, its raw key value, plus hexadecimal 80, is transmitted to the computer. The key mapping described herein refers to the translation from this raw key transmission into console device output to the user.

The low key map provides translation of the key values from hex 00-3F; the high key map provides translation of key values from hex 40-67. Raw output from the keyboard for the low key map does not include the space bar, Tab, Alt, Ctrl, arrow keys, and several other keys.

Table 33-7: High Key Map Hex Values

| Key Number | Function or Keycap Legend             |
|------------|---------------------------------------|
| 40         | Space                                 |
| 41         | Backspace                             |
| 42         | Tab                                   |
| 43         | Enter                                 |
| 44         | Return                                |
| 45         | Escape                                |
| 46         | Delete                                |
| 4A         | Numeric Pad - character               |
| 4C         | Cursor Up                             |
| 4D         | Cursor Down                           |
| 4E         | Cursor Forward                        |
| 4F         | Cursor Backward                       |
| 50-59      | Function keys F1-F10                  |
| 5A-5E      | Numeric Pad characters (except A1000) |
| 5F         | Help                                  |
| 60         | Left Shift                            |
| 61         | Right Shift                           |
| 62         | Caps Lock                             |
| 63         | Control                               |
| 64         | Left Alt                              |
| 65         | Right Alt                             |
| 66         | Left Amiga                            |
| 67         | Right Amiga                           |

The keymap table for the low and high keymaps consists of 4-byte entries, one per hex keycode. These entries are interpreted in one of two possible ways:

- As four separate bytes, specifying how the key is to be interpreted when pressed alone, with one qualifier, with another qualifier, or with both qualifiers (where a qualifier is one of three possible keys: Ctrl, Alt, or Shift).



- As a longword containing the address of a string descriptor, where a string of hex digits is to be output when this key is pressed. If a string is to be output, any combination of qualifiers can affect the string that may be transmitted.
- As a longword containing the address of a dead-key descriptor, where additional data describe the character to be output when this key is pressed alone or with another dead key.

#### NOTE

The keymap table *must* begin aligned on a word boundary. Each entry is four bytes long, thereby maintaining word alignment throughout the table. This is necessary because some of the entries may be longword addresses and *must* be aligned properly for the 68000.

## ABOUT QUALIFIERS

As you may have noticed, there are three possible qualifiers, but only a 4-byte space in the table for each key. This does not allow space to describe what the computer should output for all possible combinations of qualifiers. A solution exists, however, for "vanilla" keys, such as the alphabetic keys. Here is how that works.

Keys of type KC\_VANILLA use the 4 bytes to represent the data output for the key alone, Shifted key, Alt'ed key, and Shifted-and-Alt'ed key. Then for the Ctrl-key-plus-vanilla-key, use the code for the key alone with bits 6 and 5 set to 0.

For other keys, such as the Return key or Esc key, the qualifiers specified in the keytypes table (up to two) are the qualifiers used to establish the response to the key. This is done as follows. In the keytypes table, the values listed for the key types are those listed for the qualifiers in *devices/keymap.h* and *devices/keymap.i*. Specifically, these qualifier equates are:

|             |      |
|-------------|------|
| KC_NOQUAL   | 0x00 |
| KCF_SHIFT   | 0x01 |
| KCF_ALT     | 0x02 |
| KCF_CONTROL | 0x04 |
| KC_VANILLA  | 0x07 |
| KCF_DOWNUP  | 0x08 |
| KCF_STRING  | 0x40 |

As shown above, the qualifiers for the various types of keys occupy specific bit positions in the key types control byte.

#### NOTE

The qualifier KC\_VANILLA is equivalent to KCF\_SHIFT+KCF\_ALT+KCF\_CONTROL.

In assembly code, a keymap table entry looks like this:

```
SOME_KEY:
    DC.B    VALUE_1, VALUE_2, VALUE_3, VALUE_4
```

This table shows how to interpret the keymap for various combinations of the qualifier bits:

Table 33-8: Keymap Qualifier Bits

| If Keytype is:        | Then value in this position in the keytable is output when the key is pressed along with: |      |       |        |
|-----------------------|---|------|-------|--------|
| KC_NOQUAL             | -   | -    | -     | alone  |
| KCF_SHIFT             | -   | -    | Shift | alone  |
| KCF_ALT               | -   | -    | Alt   | alone  |
| KCF_CONTROL           | -   | -    | Ctrl  | alone  |
| KCF_ALT+KCF_SHIFT     | Shift+Alt   | Alt  | Shift | alone  |
| KCF_CONTROL+KCF_ALT   | Ctrl+Alt  | Ctrl | Alt   | alone  |
| KCF_CONTROL+KCF_SHIFT | Ctrl+Shift  | Ctrl | Shift | alone  |
| KC_VANILLA            | Shift+Alt   | Alt  | Shift | alone* |

- \* Special case—Ctrl key, when pressed with one of the alphabet keys and certain others, is to output key-alone value with the bits 6 and 5 set to zero.

## KEYTYPE TABLE ENTRIES

The vectors named **km\_LoKeyTypes** and **km\_HiKeyTypes** contain one byte per raw key code. This byte defines the entry type that is made in the key table by a set of bit positions.

Possible key types are:

- Any of the qualifier groupings noted above
- KCF\_STRING + any combination of KCF\_SHIFT, KCF\_ALT, KCF\_CONTROL (or KC\_NOQUAL) if the result of pressing the key is to be a stream of bytes (and key-with-one-or-more-qualifiers is to be one or more alternate streams of bytes).

Any key can be made to output up to eight unique byte streams if KCF\_STRING is set in its keytype. The only limitation is that the total length of all of the strings assigned to a key must be within the “jump range” of a single byte increment. See the “String-Output Keys” section below for more information.

- KCF\_DEAD + any combination of KCF\_SHIFT, KCF\_ALT, KCF\_CONTROL (or KC\_NOQUAL) if the key is a dead-class key and can thus modify or be modified by another dead-class key. See the “Dead-Class Keys” section below for more information.

The low keytype table covers the raw keycodes from hex 00-3F and contains one byte per keycode. Therefore this table contains 64 (decimal) bytes. The high keytype table covers the raw keycodes from hex 40-67 and contains 38 (decimal) bytes.

## STRING-OUTPUT KEYS

When a key is to output a string, the keymap table contains the address of a string descriptor in place of a 4-byte mapping of a key as shown above. Here is a partial table for a new high keymap table that contains only three entries thus far. The first two are for the space bar and the backspace key; the third is for the tab key, which is to

output a string that says "[TAB]." An alternate string, "[SHIFTED-TAB]," is also to be output when a shifted TAB key is pressed.

```
newHiMapTypes:
  DC.B      KCF_ALT,KC_NOQUAL,      ;key 41
  DC.B      KCF_STRING+KCF_SHIFT,   ;key 42
  ...      ;(more)

newHiMap:
  DC.B      0,0,$A0,$20             ;key 40: space bar, and Alt-space bar
  DC.B      0,0,0,$08               ;key 41: Back Space key only
  DC.L      newkey42                 ;key 42: new definition for string to output for Tab key
  ...      ;(more)

newkey42:
  DC.B      new42ue - new42us        ;length of the unshifted string
  DC.B      new42us - newkey42       ;number of bytes from start of
                                      ;string descriptor to start of this string
  DC.B      new42se - new42ss        ;length of the shifted string
  DC.B      new42ss - newkey42       ;number of bytes from start of
                                      ;string descriptor to start of this string

new42us:
  DC.B      '[TAB]'

new42ue:
new42ss:
  DC.B      '[SHIFTED-TAB]'

new42se:
```

The new high map table points to the string descriptor at address newkey42. The new high map types table says that there is one qualifier, which means that there are two strings in the key string descriptor.

Each string in the descriptor takes two bytes in this part of the table: the first byte is the length of the string, and the second byte is the distance from the start of the descriptor to the start of the string. Therefore, a single string (KCF\_STRING + KC\_NOQUAL) takes 2 bytes of string descriptor. If there is one qualifier, 4 bytes of descriptor are used. If there are two qualifiers, 8 bytes of descriptor are used. If there are 3 qualifiers, 16 bytes of descriptor are used. All strings start immediately following the string descriptor in that they are accessed as single-byte offsets from the start of the descriptor itself. Therefore, the distance from the start of the descriptor to the last string in the set (the one that uses the entire set of specified qualifiers) must start within 255 bytes of the descriptor address.

Because the length of the string is contained in a single byte, the length of any single string must be 255 bytes or less while also meeting the "reach" requirement. However, the console input buffer size limits the string output from any individual key to 32 bytes maximum.

The length of a keymap containing string descriptors and strings is variable and depends on the number and size of the strings that you provide.

## CAPSABLE BIT TABLE

The vectors called **km\_LoCapsable** and **km\_HiCapsable** point to the first byte in an 8-byte table that contains more information about the keytable entries. Specifically, if the Caps Lock key has been pressed (the Caps Lock LED is on) and if there is a bit *on* in that position in the capsable map, then this key will be treated as though the Shift key is now currently pressed. For example, in the default key mapping, the alphabetic keys are "capsable" but the punctuation keys are not. This allows you to set the Caps Lock key, just as on a normal typewriter, and get all capital letters. However, unlike a normal typewriter, you need not go out of Caps Lock to correctly type the punctuation symbols or numeric keys.

In the table, the bits that control this feature are numbered from the lowest bit in the byte, and from the lowest memory byte address to the highest. For example, the bit representing capsable status for the key that transmits raw code 00 is bit 0 in byte 0; for the key that transmits raw code 08 it is bit 0 in byte 1, and so on.

There are 64 bits (8-bytes) in each of the two capsable tables.

## **REPEATABLE BIT TABLE**

For both the low and high key maps there is an 8-byte table that provides one bit per possible raw key code. This bit indicates whether or not the specified key should repeat at the rate set by the Preferences program. The bit positions correspond to those specified in the capsable bit table.

If there is a 1 in a specific position, the key can repeat. The vectors that point to these tables are called **km\_LoRepeatable** and **km\_HiRepeatable**.

## **KEY MAP STANDARDS**

Users and programs depend on certain predictable behaviors from all keyboards and keymaps. With the exception of dead-class keys (see “Dead-Class Keys” section), mapping of keys in the low key map should follow these general rules:

- When pressed alone, keys should transmit the ASCII equivalent of the unshifted letter or lower symbol on the keycap.
- When Shifted, keys should transmit the ASCII equivalent of the shifted letter or upper symbol printed on the keycap.
- When Alt’ed, keys should generally transmit the same character (or act as the same deadkey) as the Alt’ed key in the `usa1` keymap.
- When pressed with CTRL alone, alphabetic keys should generally transmit their unshifted value but with bits 5 and 6 cleared. This allows keyboard typing of “control characters.” For example, the ‘c’ key (normally value \$63) should transmit value \$03 (CTRL-C) when CTRL and ‘c’ are pressed.

The keys in the high key map (keys with raw key values \$40 and higher) are generally non-alphanumeric keys such as those used for editing (backspace, delete, cursor keys, etc.), and special Amiga keys such as the function and help keys. Keymaps should translate these keys as to the same values or strings as those shown in the ROM default keymapping table.

In addition to their normal unshifted and shifted values, the following translations are standard for particular qualified high keymap keys:

| Key    | Generates Value: | If Used with Qualifier,<br>Generates Value: |
|--------|------------------|---|
| SPACE  | \$20             | \$A0 with qualifier ALT                     |
| RETURN | \$0D             | \$0A with qualifier CONTROL                 |
| ESC    | \$1B             | \$9B with qualifier ALT                     |

## DEAD-CLASS KEYS

All of the national keymaps, including USA, contain *dead-class* keys. This term refers to keys that either modify or can themselves be modified by other dead-class keys. There are two types of dead-class keys: dead and deadable. A dead key is one which can modify certain keys pressed immediately following. For example, on the German keyboard there is a dead key marked with the accent “ ` ”. The dead key produces no console output, but when followed by (for instance) the ‘A’ key, the combination will produce the character “ à ” (National Character Code \$E0). On the U.S. keyboard, ALT/G is the deadkey used to add the same accent (“ ` ”) to the next appropriate character typed. A deadable key is one that can be prefixed by a dead key. The ‘A’ key in the previous example is a deadable key. Thus, a dead key can only affect the output of a deadable key.

For any key that is to have a dead-class function, whether dead or deadable, the qualifier KCF\_DEAD flag must be included in the entry for the key in the KeyMapTypes table. The KCF\_DEAD type may also be used in conjunction with the other qualifiers. Furthermore, the key’s KeyMap table entry must contain the longword address of the key’s dead-key descriptor data area in place of the usual 4 ASCII character mapping.

Below is an excerpt from the AMIGA 1000 German key map. It will be referenced in the following discussion.

```
KMLowMapType:
  DC.B    KCF_DEAD+KC_VANILLA    ; aA (Key 20)
        ...                      ; (more...)
  DC.B    KCF_DEAD+KC_VANILLA    ; hH (Key 20)
        ...                      ; (more...)

KMLowMap:
  DC.L    key20                  ; a, A, ae, AE
        ...                      ; (more...)
  DC.L    key25                  ; h, H, dead ^
        ...                      ; (more...)

;----- possible dead keys
key25:
  DC.B    0,'h',0,'H'           ; h, H
  DC.B    DPF_DEAD,3,DPF_DEAD,3 ; dead ^, dead ^
  DC.B    0,$08,0,$08,0,$88,0,$88 ; control translation
        ...                      ; (more...)

;----- deadable keys (modified by dead keys)
key20:
        ; a, A, ae, AE
  DC.B    DPF_MOD,key20u-key20   ; deadable flag, number of
        ; bytes from start of key20
        ; descriptor to start of un-
        ; shifted data

  DC.B    DPF_MOD,key20s-key20   ; deadable flag, number of
        ; bytes from start of key20
        ; descriptor to start of shift-
        ; ed data

  DC.B    0,$E6,0,$C6           ; null flags followed by rest
  DC.B    0,$01,0,$01,0,$81,0,$81 ; of values (ALT, CTRL...)

key20u:
```

```

DC.B      'a', $E1, $E0, $E2, $E3, $E4 ; 'a' alone and characters to
          ; output when key alone is
          ; prefixed by a dead key
DC.B      $E1, $E1, $E2, $E1, $E1, $E1 ; most recent is '
DC.B      $E0, $E2, $E0, $E0, $E0, $E0 ; most recent is '
key20s:
DC.B      'A', $C1, $C0, $C2, $C3, $C4 ; SHIFTEd 'a' and characters to
          ; output when SHIFTEd key is
          ; prefixed by a dead key
DC.B      $C1, $C1, $C2, $C1, $C1, $C1 ; most recent is '
DC.B      $C0, $C2, $C0, $C0, $C0, $C0 ; most recent is '

```

In the example, key 25 (the 'H' key) is a dead key and key 20 (the 'A' key) is a deadable key. Both keys use the addresses of their descriptor data areas as entries in the LoKeyMap table. The LoKeyMapTypes table says that there are four qualifiers for both: the requisite KCF\_DEAD, as well as KCF\_SHIFT, KCF\_ALT, and KCF\_CONTROL. The number of qualifiers determine length and arrangement of the descriptor data areas for each key. The next table shows how to interpret the KeyMapTypes for various combinations of the qualifier bits. For each possible position a pair of bytes is needed. The first byte in each pair tells how to interpret the second byte (more about this below).

#### Dead Key Qualifier Bits

| If type is:     | Then the pair of bytes in this position in the dead-class key descriptor data is output when the key is pressed along with: |   |   |     |   |     |     |       |   |
|-----------------|---|---|---|-----|---|-----|-----|-------|---|
| NOQUAL          | alone   | - | - | -   | - | -   | -   | -     | - |
| A               | alone   | A | - | -   | - | -   | -   | -     | - |
| C               | alone   | C | - | -   | - | -   | -   | -     | - |
| S               | alone   | S | - | -   | - | -   | -   | -     | - |
| A+C             | alone   | A | C | A+C | - | -   | -   | -     | - |
| A+S             | alone   | S | A | A+S | - | -   | -   | -     | - |
| C+S             | alone   | S | C | C+S | - | -   | -   | -     | - |
| S+A+C (VANILLA) | alone   | S | A | S+A | C | C+S | C+A | C+S+A |   |

#### NOTE

The abbreviations A, C, S stand for ALT, control, and SHIFT, respectively. Also note that the ordering is reversed from that in the normal KeyMap table.

Because keys 20 and 25 each use three qualifier bits (not including KCF\_DEAD), according to the table there must be 8 pairs of data, arranged as shown. Had only KCF\_ALT been set, for instance, (not including KCF\_DEAD), just two pairs would have been needed.

As mentioned earlier, the first byte of each data pair in the descriptor data area specifies how to interpret the second byte. There are three possible values: 0, DPF\_DEAD and DPF\_MOD. In Example 4-2 DPF\_DEAD appears in the data for key 25, while DPF\_MOD is used for key 20. It is the use of these flags which determines whether a dead-class key has dead or deadable function. A value of zero causes the unrestricted output of the following byte.

If the flag byte is DPF\_DEAD, then that particular key combination (determined by the placement of the pair of bytes in the data table) is dead and will modify the output of the next key pressed (if deadable). How it modifies is controlled by the second byte of the pair which is used as an index into part(s) of the data area for ALL the deadable (DPF\_MOD set) keys.

Before going further, an understanding of the structure of a descriptor data area wherein DPF\_MOD is set for one (or more) of its members is necessary. Referring to the example, we see that DPF\_MOD is set for the first and second pairs of bytes. According to its LoKeyMapTypes entry, and using the Dead-Key Qualifier Bits table as a guide, these pairs represent the alone and SHIFTEd values for the key. When DPF\_MOD is set, the byte immediately following the flag must be the offset from the start of the key's descriptor data area to the start of a table

of bytes describing the characters to output when this key combination is preceded by any dead keys. This is where the index mentioned above comes in. The value of the index from a prefixing dead key is used to determine which of the bytes from the deadable keys special table to output. The byte in the index+1 position is sent out. (The very first byte is the value to output if the key was not prefixed by a dead key.) Thus, if ALT'ed 'H' is pressed (dead) and then SHIFTed 'A', an "a" with a circumflex (") accent will be output. This is because:

- The byte pair for the ALT position of the 'H' key (key 25) is DPF\_DEAD,3 so the index is 3.
- The byte pair for the SHIFT position of the 'A' key (key 20) is DPF\_MOD,key20s-key20, so we refer to the table-of-bytes at key20s.
- The third+1 byte of the table-of-bytes is \$C2, an "a" character.

#### NOTE

The number of bytes in the table-of-bytes for all deadable keys must be equal to the highest index value of all dead keys plus 1.

### Double-Dead Keys

Double-dead keys are an extension of the dead-class keys explained above. Unlike normal dead keys wherein one dead key of type DPF\_DEAD can modify a second of type DPF\_MOD, double-dead keys employ two consecutive keys of type DPF\_DEAD to together modify a third of type DPF\_MOD.

For example, the key on the German keyboard labeled " " is a double-dead key. When this key is pressed alone and then pressed again shifted, there is no output. But when followed by an appropriate third key, for example 'A', the three keypresses combine to produce an "a" with a circumflex (") accent (character code \$E2). Thus the double-dead pair qualify the output of the 'A' key.

The system always keeps the last two down keycodes for possible further translation. If they are both of type DPF\_DEAD and the key immediately following is DPF\_MOD then the two are used to form an index into the (third) key's translation table as follows:

In addition to the index found after the DPF\_DEAD qualifier in a normal dead key, a second factor is included in the high nibble of double-dead keys (it is shifted into place with DP\_2DFACSHIFT). Its value equals the total number of dead key types + 1 in the keymap. This second index also serves as an identifying flag to the system that two dead keys can be significant.

When a key of type DPF\_MOD is pressed, the system checks the two keycodes which preceded the current one. If they were both DPF\_DEAD then the most recent of the two is checked for the double-dead index/flag. If it is found then a new index is formed by multiplying the value in lower nibble with that in the upper. Then, the lower nibble of the least recent DPF\_DEAD key is added in to form the final offset.

Finally, this last value is used as an index into the translation table of the current, DPF\_MOD, key.

The translation table of all deadable (DPF\_MOD) keys has <number of dead key types + 1> times <number of double dead key types + 1> entries, arranged in <number of double dead key types + 1> rows of <number of dead key types + 1> entries. This is because as indices are assigned for dead keys in the keymap, those that are double dead keys are assigned the lower numbers.

Following is a code fragment from the German (d) keymap source:

```
key0C:
    DC.B    DPF_DEAD,1+(6<<DP_2DFACSHIFT)    ; dead '
    DC.B    DPF_DEAD,2+(6<<DP_2DFACSHIFT)    ; dead `
    DC.B    0,'=',0,'+'                      ; =, +

key20:
                                ; a, A, ae, AE
    DC.B    DPF_MOD,key20u-key20,DPF_MOD,key20s-key20
    DC.B    0,$E6,0,$C6
    DC.B    0,$01,0,$01,0,$81,0,$81 ; control translation

key20u:
    DC.B    'a',$E1,$E0,$E2,$E3,$E4
    DC.B    $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
    DC.B    $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is `

key20s:
    DC.B    'A',$C1,$C0,$C2,$C3,$C4
    DC.B    $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
    DC.B    $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is `
```

Raw key0C is a double dead key. Pressing this key alone, then again while the shift key is down will produce no output but will form a double-dead qualifier. The output of key20 ('A'), a deadable key, will consequently be modified. The " ' " and the " ` " of the double-dead combination produce an "a" with a circumflex (ˆ) accent. The mechanics are as follows:

When key0C is pressed alone the DPF\_DEAD of the first byte pair in the key's table flags the key as dead. The second byte is then held by the system.

Next, when key0C is pressed again, this time with the shift key down, the DPF\_DEAD of the second byte pair (recall that the second pair is used because of the shift qualifier) again flags the key as dead. The second byte of this pair is also held by the system.

Finally, when the 'A' key is pressed the system recalls the latter of the two bytes it has saved. The upper nibble, 0x06, is multiplied by the lower nibble, 0x02. The result, 0x0C, is then added to the lower nibble of the earlier of the two saved bytes, 0x01. This new value, 0x0D, is used as an index into the (unshifted) translation table of key20. The character at position 0x0D is character \$E2, an "a" with a circumflex (ˆ) accent.

#### NOTE

If only one double-dead key is pressed before a deadable key then the output is the same as if the double-dead were a normal dead key. If shifted key0C were pressed on the German keyboard and then immediately followed by key20, the output produced would be character \$E0, 'à'. As before, the upper nibble is multiplied with the lower, resulting in 0x0C. But because there was no second dead-key, this product is used as the final index.

## Complete Keymap Source Example

The following example is the complete Amiga assembler source for the *d* (German) keymap. Comments in the source code illustrate the key layout of the German keyboard, and the standard U.S. keyboard layout for comparison.



```

*****
*
* d (GERMAN) A2000 key map
* Assemble and then link without startup code or linker libs
*
* Copyright (c) 1988 Commodore-Amiga, Inc. All Rights Reserved
*
*****

```

```

----- Included Files -----

```

```

INCLUDE      "exec/types.i"
INCLUDE      "devices/keymap.i"

```

```

-----

```

```

DC.L  0,0          ; ln_Succ, ln_Pred
DC.B  0,0          ; ln_Type, ln_Pri
DC.L  KMName       ; ln_Name
DC.L  KMLowMapType
DC.L  KMLowMap
DC.L  KMLCapsable
DC.L  KMLRepeatable
DC.L  KMHighMapType
DC.L  KMHighMap
DC.L  KMHCapsable
DC.L  KMHRepeatable

```

```

----- Key Translation Table -----

```

```

* Raw key codes

```

```

*
* 45  50  51  52  53  54  55  56  57  58  59
* 00  01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 41 46 5F 5A 5B 5C 5D
* 42  10 11 12 13 14 15 16 17 18 19 1A 1B 44 3D 3E 3F 4A
* 63 62 20 21 22 23 24 25 26 27 28 29 2A 2B 4C 2D 2E 2F 5E
* 60 30 31 32 33 34 35 36 37 38 39 3A 61 4F 4D 4E 1D 1E 1F 43
* 64 66 40 67 65 0F 3C

```

```

* German (D) mapping

```

```

* ESC F1 F2 F3 F4 F5 F6 F7 F8 F9 F10
* ` 1! 2" 3$ 4% 5& 7/ 8( 9) 0= B? '` BS DEL HELP [{ ]} / *
* TAB qQ wW eE rR tT zZ uU iI oO pP uU +* RET 7 8 9 -
* CT CL aA sS dD fF gG hH jJ kK lL oO aA #^ UC 4 5 6 +
* SH <> yY xX cC vV bB nN mM ,; .: -_ SH LC DC RC 1 2 3 ENT
* ALT AM SPACE AM ALT 0 .

```

```

* For comparison, here's the USA1 mapping

```

```

* ESC F1 F2 F3 F4 F5 F6 F7 F8 F9 F10
* ` 1! 2@ 3# 4$ 5% 6^ 7& 8* 9( 0) - _ =+ BS DEL HELP (( )) // **
* TAB qQ wW eE rR tT yY uU iI oO pP [{ ]} RET 77 88 99 --
* CT CL aA sS dD fF gG hH jJ kK lL ;: '" [2B] UC 44 55 66 ++
* SH [30] zZ xX cC vV bB nN mM ,< .> /? SH LC DC RC 11 22 33 ENT
* ALT AM SPACE AM ALT 00 ..

```

```

*
* 40 Space
* 41 Backspace
* 42 Tab
* 43 Enter
* 44 Return
* 45 Escape
* 46 Delete
* 4A Numeric Pad -
* 4C Cursor Up
* 4D Cursor Down
* 4E Cursor Forward

```

```

*      4F      Cursor Backward
*
*      50-59   Function keys F1-F10
*      5A      Numeric Pad [{ (A2000)
*      5B      Numeric Pad ]} (A2000)
*      5C      Numeric Pad / (A2000)
*      5D      Numeric Pad * (A2000)
*      5E      Numeric Pad + (A2000)
*      5F      Help
*
*      60      Left Shift
*      61      Right Shift
*      62      Caps Lock
*      63      Control
*      64      Left Alt
*      65      Right Alt
*      66      Left Amiga
*      67      Right Amiga
*
*      68      Left Mouse Button (not converted)
*      69      Right Mouse Button (not converted)
*      6A      Middle Mouse Button (not converted)
*
*-----

```

KMLCapsable: ; NL means NULL (undefined) and RE means RESERVED

```

DC.B      %000000000      ; 7 6& 5% 4$ 3s 2" 1! [] 07 - 00
DC.B      %000000000      ; 0N NL ' ' B? 0= 9) 8( 0F - 08
DC.B      %111111111      ; iI uU zZ tT rR eE wW qQ 17 - 10
DC.B      %000001111      ; 3N 2N 1N NL +* UU pP oO 1F - 18
DC.B      %111111111      ; kK jJ hH gG fF dD sS aA 27 - 20
DC.B      %000001111      ; 6N 5N 4N NL #' AA OO 1L 2F - 28
DC.B      %111111110      ; mM nN bB vV cC xX yY <> 37 - 30
DC.B      %000000000      ; 9N 8N 7N .N NL -_ .: ,; 3F - 38

```

KMHCapsable:

```

DC.B      %000000000      ; 47 - 40
DC.B      %000000000      ; 4F - 48
DC.B      %000000000      ; 57 - 50
DC.B      %000000000      ; 5F - 58
DC.B      %000000000      ; 67 - 60
DC.B      %000000000      ; 6F - 68
DC.B      %000000000      ; 77 - 70

```

KMLRepeatable:

```

DC.B      %111111111      ; 7 6& 5% 4$ 3s 2" 1! [] 07 - 00
DC.B      %101111111      ; 0N NL ' ' B? 0= 9) 8( 0F - 08
DC.B      %111111111      ; iI uU zZ tT rR eE wW qQ 17 - 10
DC.B      %111011111      ; 3N 2N 1N NL +* UU pP oO 1F - 18
DC.B      %111111111      ; kK jJ hH gG fF dD sS aA 27 - 20
DC.B      %111011111      ; 6N 5N 4N NL #' AA OO 1L 2F - 28
DC.B      %111111111      ; mM nN bB vV cC xX yY <> 37 - 30
DC.B      %111101111      ; 9N 8N 7N .N NL -_ .: ,; 3F - 38

```

KMHRepeatable:

```

DC.B      %010001111      ; 47 - 40
DC.B      %111101000      ; 4F - 48
DC.B      %111111111      ; 57 - 50
DC.B      %011111111      ; 5F - 58
DC.B      %000000000      ; 67 - 60
DC.B      %000000000      ; 6F - 68
DC.B      %000000000      ; 77 - 70

```

KMLowMapType:

```

DC.B      KC_VANILLA      ; []          $00
DC.B      KCF_SHIFT+KCF_ALT ; 1!
DC.B      KCF_SHIFT+KCF_ALT ; 2"
DC.B      KCF_SHIFT+KCF_ALT ; 3s

```

|      |                            |           |      |
|------|----------------------------|-----------|------|
| DC.B | KCF_SHIFT+KCF_ALT          | ; 4\$     |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; 5%      |      |
| DC.B | KC_VANILLA                 | ; 6&      |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; 7/      |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; 8(      | \$08 |
| DC.B | KCF_SHIFT+KCF_ALT          | ; 9)      |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; 0=      |      |
| DC.B | KC_VANILLA                 | ; B?      |      |
| DC.B | KCF_DEAD+KCF_SHIFT+KCF_ALT | ; ''      |      |
| DC.B | KC_VANILLA                 | ;         |      |
| DC.B | KCF_NOP                    | ; NL      |      |
| DC.B | KC_NOQUAL                  | ; ON      |      |
| DC.B | KC_VANILLA                 | ; qQ      | \$10 |
| DC.B | KC_VANILLA                 | ; wW      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; eE      |      |
| DC.B | KC_VANILLA                 | ; rR      |      |
| DC.B | KC_VANILLA                 | ; tT      |      |
| DC.B | KC_VANILLA                 | ; zZ      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; uU      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; iI      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; oO      | \$18 |
| DC.B | KC_VANILLA                 | ; pP      |      |
| DC.B | KC_VANILLA                 | ; omlatuU |      |
| DC.B | KC_VANILLA                 | ; +*      |      |
| DC.B | KCF_NOP                    | ; NL      |      |
| DC.B | KC_NOQUAL                  | ; 1N      |      |
| DC.B | KC_NOQUAL                  | ; 2N      |      |
| DC.B | KC_NOQUAL                  | ; 3N      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; aA      | \$20 |
| DC.B | KC_VANILLA                 | ; sS      |      |
| DC.B | KC_VANILLA                 | ; dD      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; fF      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; gG      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; hH      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; jJ      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; kK      |      |
| DC.B | KC_VANILLA                 | ; lL      | \$28 |
| DC.B | KCF_SHIFT+KCF_ALT          | ; umlotoO |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; umlotaA |      |
| DC.B | KC_VANILLA                 | ; #^      |      |
| DC.B | KCF_NOP                    | ; NL      |      |
| DC.B | KC_NOQUAL                  | ; 4N      |      |
| DC.B | KC_NOQUAL                  | ; 5N      |      |
| DC.B | KC_NOQUAL                  | ; 6N      |      |
| DC.B | KCF_SHIFT                  | ; <>      | \$30 |
| DC.B | KCF_DEAD+KC_VANILLA        | ; yY      |      |
| DC.B | KC_VANILLA                 | ; xX      |      |
| DC.B | KC_VANILLA                 | ; cC      |      |
| DC.B | KC_VANILLA                 | ; vV      |      |
| DC.B | KC_VANILLA                 | ; bB      |      |
| DC.B | KCF_DEAD+KC_VANILLA        | ; nN      |      |
| DC.B | KC_VANILLA                 | ; mM      |      |
| DC.B | KCF_SHIFT+KCF_ALT          | ; ,;      | \$38 |
| DC.B | KCF_SHIFT+KCF_ALT          | ; .:      |      |
| DC.B | KC_VANILLA                 | ; -_      |      |
| DC.B | KCF_NOP                    | ; NL      |      |
| DC.B | KC_NOQUAL                  | ; .N      |      |
| DC.B | KC_NOQUAL                  | ; 7N      |      |
| DC.B | KC_NOQUAL                  | ; 8N      |      |
| DC.B | KC_NOQUAL                  | ; 9N      |      |

KMHighMapType:

|      |                  |             |      |
|------|------------------|-------------|------|
| DC.B | KCF_DEAD+KCF_ALT | ; SPACE     | \$40 |
| DC.B | KC_NOQUAL        | ; BACKSPACE |      |

```

DC.B    KCF_STRING+KCF_SHIFT    ; TAB
DC.B    KC_NOQUAL                ; ENTER
DC.B    KCF_CONTROL              ; RETURN
DC.B    KCF_ALT                  ; ESCAPE
DC.B    KC_NOQUAL                ; DELETE
DC.B    KCF_NOP

DC.B    KCF_NOP                  ;
DC.B    KCF_NOP                  ;
DC.B    KC_NOQUAL                ; -N
DC.B    KCF_NOP
DC.B    KCF_STRING+KCF_SHIFT    ; CURSOR UP
DC.B    KCF_STRING+KCF_SHIFT    ; CURSOR DOWN
DC.B    KCF_STRING+KCF_SHIFT    ; CURSOR FORWARD
DC.B    KCF_STRING+KCF_SHIFT    ; CURSOR BACKWARD

DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 1      $50
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 2
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 3
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 4
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 5
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 6
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 7
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 8

DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 9      $58
DC.B    KCF_STRING+KCF_SHIFT    ; FKEY 10
DC.B    KCF_SHIFT+KCF_CONTROL    ; Numeric Pad [{
DC.B    KCF_SHIFT+KCF_CONTROL    ; Numeric Pad ]}
DC.B    KC_NOQUAL                ; Numeric Pad /
DC.B    KC_NOQUAL                ; Numeric Pad *
DC.B    KC_NOQUAL                ; Numeric Pad +
DC.B    KCF_STRING              ; Help

DC.B    KCF_NOP                  ; Left Shift  $60
DC.B    KCF_NOP                  ; Right Shift
DC.B    KCF_NOP                  ; Caps Lock
DC.B    KCF_NOP                  ; Control
DC.B    KCF_NOP                  ; Left Alt
DC.B    KCF_NOP                  ; Right Alt
DC.B    KCF_NOP                  ; Left Amiga
DC.B    KCF_NOP                  ; Right Amiga

DC.B    KCF_NOP                  ; Left Mouse Button $68
DC.B    KCF_NOP                  ; Right Mouse Button
DC.B    KCF_NOP                  ; Middle Mouse Button
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP

DC.B    KCF_NOP                  ; $70
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP
DC.B    KCF_NOP

```

#### KMLowMap:

```

DC.B    '!', '$B9, '!', '1' ; 1, !, super 1, !
DC.B    '$B2, '@', '"', '2' ; 2, ", @, super 2
DC.B    '#', '$B3, '$A7, '3' ; 3, section, super 3, #
DC.B    '$A2, '$B0, '$', '4' ; 4, $, degree, cents
DC.B    '%', '$BC, '%', '5' ; 5, %, 1/4, %
DC.B    '^', '$BD, '&', '6' ; 6, &, 1/2, ^
DC.B    '&', '$BE, '/', '7' ; 7, /, 3/4, &

```

\$00

|      |                         |                              |      |
|------|-------------------------|------------------------------|------|
| DC.B | '*', \$B7, '(' , '8'    | ; 8, (, bullet, *            | \$08 |
| DC.B | '(' , \$AB, ')' , '9'   | ; 9, ), <<, (                |      |
| DC.B | ' ' , \$BB, '=' , '0'   | ; 0, =, >>, )                |      |
| DC.B | '_ ' , \$DF, '?' , \$DF | ; sharp s, ?, -, _           |      |
| DC.L | key0C                   | ; dead ' , dead ' , =, +     |      |
| DC.B | '  ' , ' ' , ' ' , ' '  | ;  , ,                       |      |
| DC.L | 0                       | ; NOP                        |      |
| DC.B | 0,0,0,'0'               | ; numeric pad 0 (0N)         |      |
| DC.B | \$C5,\$E5,'Q','q'       | ; q, Q, dot a, dot A         | \$10 |
| DC.B | \$B0,\$B0,'W','w'       | ; w, W, dot, dot             |      |
| DC.L | key12                   | ; e, E, (c), (c)             |      |
| DC.B | \$AE,\$AE,'R','r'       | ; r, R, (r), (r)             |      |
| DC.B | \$DE,\$FE,'T','t'       | ; t, T, thorn, THORN         |      |
| DC.B | \$A5,\$A4,'Z','z'       | ; z, Z, IMS, Yen             |      |
| DC.L | key16                   | ; u, U, micro, micro         |      |
| DC.L | key17                   | ; i, I, inverted !, broken   |      |
| DC.L | key18                   | ; o, O, slash o, slash O     | \$18 |
| DC.B | \$B6,\$B6,'P','p'       | ; p, P, paragraph, paragraph |      |
| DC.B | '{' , '[' , \$DC,\$FC   | ; umlaut u, umlaut U, [, {   |      |
| DC.B | ' ' , ' ' , '*' , '+'   | ; +, *, ], )                 |      |
| DC.L | 0                       | ; NOP                        |      |
| DC.B | 0,0,0,'1'               | ; numeric pad 1 (1N)         |      |
| DC.B | 0,0,0,'2'               | ; numeric pad 2 (2N)         |      |
| DC.B | 0,0,0,'3'               | ; numeric pad 3 (3N)         |      |
| DC.L | key20                   | ; a, A, ae, AE               | \$20 |
| DC.B | \$A7,\$DF,'S','s'       | ; s, S, sharp s, section     |      |
| DC.B | \$D0,\$F0,'D','d'       | ; d, D, bar d, bar D         |      |
| DC.L | key23                   | ; f, F, dead '               |      |
| DC.L | key24                   | ; g, G, dead '               |      |
| DC.L | key25                   | ; h, H, dead ^               |      |
| DC.L | key26                   | ; j, J, dead ^               |      |
| DC.L | key27                   | ; k, K, dead "               |      |
| DC.B | \$A3,\$A3,'L','l'       | ; l, L, pound, pound         | \$28 |
| DC.B | ' ' , ' ' , \$D6,\$F6   | ; umlaut o, umlaut O, ;, :   |      |
| DC.B | ' ' , \$27,\$C4,\$E4    | ; umlaut a, umlaut A, ' , "  |      |
| DC.B | ' ' , '#' , ' ' , '#'   | ; #, ^, #, ^                 |      |
| DC.L | 0                       | ; NOP                        |      |
| DC.B | 0,0,0,'4'               | ; numeric pad 4 (4N)         |      |
| DC.B | 0,0,0,'5'               | ; numeric pad 5 (5N)         |      |
| DC.B | 0,0,0,'6'               | ; numeric pad 6 (6N)         |      |
| DC.B | 0,0,'>','<'             | ; <, >                       | \$30 |
| DC.L | key31                   | ; y, Y, +/-, not             |      |
| DC.B | \$F7,\$D7,'X','x'       | ; x, X, times, divide        |      |
| DC.B | \$C7,\$E7,'C','c'       | ; c, C, c cedilla, C cedilla |      |
| DC.B | \$AA,\$AA,'V','v'       | ; v, V, female ordinal       |      |
| DC.B | \$BA,\$BA,'B','b'       | ; b, B, male ordinal         |      |
| DC.L | key36                   | ; n, N, SHY, overbar         |      |
| DC.B | \$BF,\$B8,'M','m'       | ; m, M, cedilla, inverted ?  |      |
| DC.B | '< ' , ' ' , ' ' , ' '  | ; <, ;, <, <                 | \$38 |
| DC.B | '> ' , ' ' , ' ' , ' '  | ; >, :, >, >                 |      |
| DC.B | '?' , ' ' , ' ' , ' '   | ; -, _ , / , ?               |      |
| DC.L | 0                       | ; NOP                        |      |
| DC.B | 0,0,0,'.'               | ; numeric pad . (N)          |      |
| DC.B | 0,0,0,'7'               | ; numeric pad 7 (7N)         |      |
| DC.B | 0,0,0,'8'               | ; numeric pad 8 (8N)         |      |
| DC.B | 0,0,0,'9'               | ; numeric pad 9 (9N)         |      |

# KMHighMap:

|      |               |      |
|------|---------------|------|
| DC.L | key40         | \$40 |
| DC.B | 0,0,0,\$08    |      |
| DC.L | key42         |      |
| DC.B | 0,0,0,\$0D    |      |
| DC.B | 0,0,\$0A,\$0D |      |
| DC.B | 0,0,\$9B,\$1B |      |
| DC.B | 0,0,0,\$7F    |      |

|      |                                       |      |
|------|---------------------------------------|------|
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     | \$48 |
| DC.B | 0,0,0,'--'                            |      |
| DC.L | 0                                     |      |
| DC.L | key4C                                 |      |
| DC.L | key4D                                 |      |
| DC.L | key4E                                 |      |
| DC.L | key4F                                 |      |
| DC.L | key50                                 |      |
| DC.L | key51                                 | \$50 |
| DC.L | key52                                 |      |
| DC.L | key53                                 |      |
| DC.L | key54                                 |      |
| DC.L | key55                                 |      |
| DC.L | key56                                 |      |
| DC.L | key57                                 |      |
| DC.L | key58                                 |      |
| DC.L | key59                                 | \$58 |
| DC.B | \$1B,\$1B,'{'','[' ; numeric pad [, { |      |
| DC.B | \$1D,\$1D,'{'','[' ; numeric pad [, { |      |
| DC.B | 0,0,0,'/' ; numeric pad /             |      |
| DC.B | 0,0,0,'*' ; numeric pad *             |      |
| DC.B | 0,0,0,'+' ; numeric pad +             |      |
| DC.L | key5F                                 |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     | \$60 |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     | \$68 |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     | \$70 |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |
| DC.L | 0                                     |      |

;------ possible dead keys

|        |      |                               |                       |
|--------|------|-------------------------------|-----------------------|
| key0C: | DC.B | DPF_DEAD,1+(6<<DP_2DFACSHIFT) | ; dead '              |
|        | DC.B | DPF_DEAD,2+(6<<DP_2DFACSHIFT) | ; dead \              |
|        | DC.B | 0,'=',0,'+'                   | ; =, +                |
| key23: | DC.B | 0,'f',0,'F'                   | ; f, F                |
|        | DC.B | DPF_DEAD,1+(6<<DP_2DFACSHIFT) | ; dead '              |
|        | DC.B | DPF_DEAD,1+(6<<DP_2DFACSHIFT) | ; dead \              |
|        | DC.B | 0,\$06,0,\$06,0,\$86,0,\$86   | ; control translation |
| key24: | DC.B | 0,'g',0,'G'                   | ; g, G                |
|        | DC.B | DPF_DEAD,2+(6<<DP_2DFACSHIFT) | ; dead \              |
|        | DC.B | DPF_DEAD,2+(6<<DP_2DFACSHIFT) | ; dead \              |

```

DC.B      0,$07,0,$07,0,$87,0,$87      ; control translation
key25:
DC.B      0,'h',0,'H'                    ; h, H
DC.B      DPF_DEAD,3,DPF_DEAD,3          ; dead ^, dead ^
DC.B      0,$08,0,$08,0,$88,0,$88      ; control translation
key26:
DC.B      0,'j',0,'J'                    ; j, J
DC.B      DPF_DEAD,4,DPF_DEAD,4          ; dead ~, dead ~
DC.B      0,$0A,0,$0A,0,$8A,0,$8A      ; control translation
key27:
DC.B      0,'k',0,'K'                    ; k, K
DC.B      DPF_DEAD,5,DPF_DEAD,5          ; dead ", dead "
DC.B      0,$0B,0,$0B,0,$8B,0,$8B      ; control translation

```

;----- deadable keys (modified by dead keys)

```

key12:
          ; e, E, (c), (c)
DC.B      DPF_MOD,key12u-key12,DPF_MOD,key12s-key12
DC.B      0,$A9,0,$A9
DC.B      0,$05,0,$05,0,$85,0,$85 ; control translation
key12u:
DC.B      'e',$E9,$E8,$EA,'e',$EB
DC.B      $E9,$E9,$EA,$E9,$E9,$E9
DC.B      $E8,$EA,$E8,$E8,$E8,$E8
key12s:
DC.B      'E',$C9,$C8,$CA,'E',$CB
DC.B      $C9,$C9,$CA,$C9,$C9,$C9
DC.B      $C8,$CA,$C8,$C8,$C8,$C8
key16:
          ; u, U, micro, micro
DC.B      DPF_MOD,key16u-key16,DPF_MOD,key16s-key16
DC.B      0,$B5,0,$B5
DC.B      0,$15,0,$15,0,$95,0,$95 ; control translation
key16u:
DC.B      'u',$FA,$F9,$FB,'u',$FC
DC.B      $FA,$FA,$FB,$FA,$FA,$FA
DC.B      $F9,$FB,$F9,$F9,$F9,$F9
key16s:
DC.B      'U',$DA,$D9,$DB,'U',$DC
DC.B      $DA,$DA,$DB,$DA,$DA,$DA
DC.B      $D9,$DB,$D9,$D9,$D9,$D9
key17:
          ; i, I, inverted !, broken !
DC.B      DPF_MOD,key17u-key17,DPF_MOD,key17s-key17
DC.B      0,$A1,0,$A6
DC.B      0,$09,0,$09,0,$89,0,$89 ; control translation
key17u:
DC.B      'i',$ED,$EC,$EE,'i',$EF
DC.B      $ED,$ED,$EE,$ED,$ED,$ED
DC.B      $EC,$EE,$EC,$EC,$EC,$EC
key17s:
DC.B      'I',$CD,$CC,$CE,'I',$CF
DC.B      $CD,$CD,$CE,$CD,$CD,$CD
DC.B      $CC,$CE,$CC,$CC,$CC,$CC
key18:
          ; o, O, bar o, bar O
DC.B      DPF_MOD,key18u-key18,DPF_MOD,key18s-key18
DC.B      0,$F8,0,$D8
DC.B      0,$0F,0,$0F,0,$8F,0,$8F ; control translation
key18u:
DC.B      'o',$F3,$F2,$F4,$F5,$F6
DC.B      $F3,$F3,$F4,$F3,$F3,$F3
DC.B      $F2,$F4,$F2,$F2,$F2,$F2
key18s:
DC.B      'O',$D3,$D2,$D4,$D5,$D6
DC.B      $D3,$D3,$D4,$D3,$D3,$D3
DC.B      $D2,$D4,$D2,$D2,$D2,$D2
key20:
          ; a, A, ae, AE
DC.B      DPF_MOD,key20u-key20,DPF_MOD,key20s-key20
DC.B      0,$E6,0,$C6

```

```

DC.B      0,$01,0,$01,0,$81,0,$81 ; control translation
key20u:
DC.B      'a',$E1,$E0,$E2,$E3,$E4
DC.B      $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
DC.B      $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is '
key20s:
DC.B      'A',$C1,$C0,$C2,$C3,$C4
DC.B      $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
DC.B      $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is '
key36:
          ; n, N, SHY, overbar
DC.B      DPF_MOD,key36u-key36,DPF_MOD,key36s-key36
DC.B      0,$AD,0,$AF
DC.B      0,$0E,0,$0E,0,$8E,0,$8E ; control translation
key36u:
DC.B      'n','n','n','n','$F1,'n'
DC.B      'n','n','n','n','n','n'
DC.B      'n','n','n','n','n','n'
key36s:
DC.B      'N','N','N','N','$D1,'N'
DC.B      'N','N','N','N','N','N'
DC.B      'N','N','N','N','N','N'
key31:
          ; y, Y, +/-, not
DC.B      DPF_MOD,key31u-key31,DPF_MOD,key31s-key31
DC.B      0,$B1,0,$AC
DC.B      0,$19,0,$19,0,$99,0,$99 ; control translation
key31u:
DC.B      'y',$FD,'y','y','y','$FF
DC.B      $FD,$FD,$FD,$FD,$FD,$FD
DC.B      'y','y','y','y','y','y'
key31s:
DC.B      'Y',$DD,'Y','Y','Y','Y'
DC.B      $DD,$DD,$DD,$DD,$DD,$DD
DC.B      'Y','Y','Y','Y','Y','Y'
key40:
          ; space,,NBSP, and accents
DC.B      DPF_MOD,key40d-key40,0,$A0
key40d:
DC.B      ' ',,$B4,' ',' ',' ',' '$A8
DC.B      $B4,' ',' '$B4,$B4,$B4,$B4
DC.B      ' ',' ',' ',' ',' ',' '

```

;----- string keys

```

key42:
DC.B      key42ue-key42us,key42us-key42
DC.B      key42se-key42ss,key42ss-key42
key42us:
DC.B      $09
key42ue:
key42ss:
DC.B      $9B,'Z'
key42se:
key4C:
DC.B      key4Cue-key4Cus,key4Cus-key4C
DC.B      key4Cse-key4Css,key4Css-key4C
key4Cus:
DC.B      $9B,'A'
key4Cue:
key4Css:
DC.B      $9B,'T'
key4Cse:
key4D:
DC.B      key4Due-key4Dus,key4Dus-key4D
DC.B      key4Dse-key4Dss,key4Dss-key4D
key4Dus:
DC.B      $9B,'B'
key4Due:

```



key4Dss: DC.B \$9B, 'S'  
 key4Dse:  
 key4E: DC.B key4Eue-key4Eus, key4Eus-key4E  
 DC.B key4Ese-key4Ess, key4Ess-key4E  
 key4Eus: DC.B \$9B, 'C'  
 key4Eue:  
 key4Ess: DC.B \$9B, ' ', 'e'  
 key4Ese:  
 key4F: DC.B key4Fue-key4Fus, key4Fus-key4F  
 DC.B key4Fse-key4Fss, key4Fss-key4F  
 key4Fus: DC.B \$9B, 'D'  
 key4Fue:  
 key4Fss: DC.B \$9B, ' ', 'A'  
 key4Fse:  
 key50: DC.B key50ue-key50us, key50us-key50  
 DC.B key50se-key50ss, key50ss-key50  
 key50us: DC.B \$9B, '0~'  
 key50ue:  
 key50ss: DC.B \$9B, '10~'  
 key50se:  
 key51: DC.B key51ue-key51us, key51us-key51  
 DC.B key51se-key51ss, key51ss-key51  
 key51us: DC.B \$9B, '1~'  
 key51ue:  
 key51ss: DC.B \$9B, '11~'  
 key51se:  
 key52: DC.B key52ue-key52us, key52us-key52  
 DC.B key52se-key52ss, key52ss-key52  
 key52us: DC.B \$9B, '2~'  
 key52ue:  
 key52ss: DC.B \$9B, '12~'  
 key52se:  
 key53: DC.B key53ue-key53us, key53us-key53  
 DC.B key53se-key53ss, key53ss-key53  
 key53us: DC.B \$9B, '3~'  
 key53ue:  
 key53ss: DC.B \$9B, '13~'  
 key53se:  
 key54: DC.B key54ue-key54us, key54us-key54  
 DC.B key54se-key54ss, key54ss-key54  
 key54us: DC.B \$9B, '4~'  
 key54ue:  
 key54ss: DC.B \$9B, '14~'

```

key54se:

key55:
    DC.B    key55ue-key55us, key55us-key55
    DC.B    key55se-key55ss, key55ss-key55
key55us:
    DC.B    $9B, '5~'
key55ue:
key55ss:
    DC.B    $9B, '15~'
key55se:

key56:
    DC.B    key56ue-key56us, key56us-key56
    DC.B    key56se-key56ss, key56ss-key56
key56us:
    DC.B    $9B, '6~'
key56ue:
key56ss:
    DC.B    $9B, '16~'
key56se:

key57:
    DC.B    key57ue-key57us, key57us-key57
    DC.B    key57se-key57ss, key57ss-key57
key57us:
    DC.B    $9B, '7~'
key57ue:
key57ss:
    DC.B    $9B, '17~'
key57se:

key58:
    DC.B    key58ue-key58us, key58us-key58
    DC.B    key58se-key58ss, key58ss-key58
key58us:
    DC.B    $9B, '8~'
key58ue:
key58ss:
    DC.B    $9B, '18~'
key58se:

key59:
    DC.B    key59ue-key59us, key59us-key59
    DC.B    key59se-key59ss, key59ss-key59
key59us:
    DC.B    $9B, '9~'
key59ue:
key59ss:
    DC.B    $9B, '19~'
key59se:

key5F:
    DC.B    key5Fe-key5Fs, key5Fs-key5F
key5Fs:
    DC.B    $9B, '?~'
key5Fe:

KMName:
    DC.B    'd', 0
kmEnd

END

```

# **Chapter 34**

## **Gameport Device**

### **Introduction**

The gameport device manages access to the Amiga gameport connectors for the operating system. There are two units in the gameport device, unit 0 and unit 1. On the Amiga 2000, unit 0 controls the left gameport connector (connector 1), and unit 1 controls the right gameport connector (connector 2). Unit 0 is labelled “joy 1” on an A500 and “1” on an A1000. Unit 1 is labelled “joy 2” on an A500 and “2” on an A1000.

To use the gameport device, you must define the type of device connected to the gameport and define how the device is to respond. The gameport device can be set up to return the controller status immediately or only when certain conditions have been met.

When the input device is operating, unit 0 is usually dedicated to gathering mouse events. Therefore, the example listed at the end of this chapter employs unit 1, which is usually not used by the system. (For applications that take over the machine without starting up the input device, unit 0 can perform the same functions as unit 1.)

When a gameport device unit responds to a request for input, it creates an input event. The contents of the input event will vary based on the type of device and the trigger conditions you have declared.

# Gameport Device Commands

The gameport device supports the following system functions:

| Command              | Operation  |
|----------------------|--|
| <b>OpenDevice()</b>  | Obtain shared use of one unit of the gameport device.                |
| <b>CloseDevice()</b> | Relinquish use of the gameport device.                               |
| <b>DoIO()</b>        | Initiate a command and wait for it to complete (synchronous access). |
| <b>SendIO()</b>      | Initiate a command and return immediately (asynchronous access).     |
| <b>AbortIO()</b>     | Abort a command already in the queue.                                |

The gameport device also responds to the following commands:

| I/O Command           | Operation  |
|-----------------------|--|
| <b>CMD_CLEAR</b>      | Clear gameport device's input buffer.  |
| <b>GPD_ASKCTYPE</b>   | Inquire the type of the controller being monitored.  |
| <b>GPD_SETCTYPE</b>   | Set the type of the controller to be monitored.  |
| <b>GPD_SETTRIGGER</b> | Preset the conditions that will trigger a gameport event.  |
| <b>GPD_ASKTRIGGER</b> | Inquire the conditions that have been preset for triggering.   |
| <b>GPD_READEVENT</b>  | Read one or more gameport events from an initialized unit.   |
| <b>GPD_ASKCTYPE</b>   | You use this command to find out what kind of controller has been specified for a particular unit. As of this writing, there are five different legal controller types defined in the include file <i>devices/gameport.h</i> . |

| Controller Type          | Description   |
|--------------------------|---|
| <b>GPCT_MOUSE</b>        | Mouse controller.                                     |
| <b>GPCT_ABSJOYSTICK</b>  | Absolute (digital) joystick.                          |
| <b>GPCT_RELJOYSTICK</b>  | Relative (digital) joystick.                          |
| <b>GPCT_ALLOCATED</b>    | Custom controller.                                    |
| <b>GPCT_NOCONTROLLER</b> | "No controller" flag to indicate unit is free to use. |

- A mouse controller can report input events for one, two, or three buttons and for positive or negative (x,y) movements. A trackball controller or car-driving controller is generally of the same type and can be declared as a mouse controller.
- An absolute joystick reports one single event for each change of its current location. If, for example, the joystick is centered and a user pushes the stick forward and holds it in that position, only one single forward-switch event will be generated.
- A relative joystick, on the other hand, is comparable to an absolute joystick with "autorepeat" installed. As long as the user holds the stick in a position other than centered, the gameport device continues to generate position reports.

- There is currently no system software support for proportional joysticks or proportional controllers (e.g., paddles). If you write custom code to read proportional controllers or other controllers (e.g., light pen) please make sure that you issue **GPD\_SETCTYPE** (explained below) with controller type **GPCT\_ALLOCATED** to insure that other applications know the connector is being used.

**GPD\_ASKCTYPE** puts the controller type into the data area that you specify with the command. Here is a sample call:

```
struct IOStdReq *game_io_msg = NULL;

BYTE GetControllerType()
{
    BYTE controller_type = 0;

    game_io_msg->io_Command = GPD_ASKCTYPE;          /* get type of controller */
    game_io_msg->io_Data = (APTR)&controller_type; /* place data here */
    game_io_msg->io_Length = 1;
    DoIO(game_io_msg);
    return (controller_type);
}
```

The BYTE value returned corresponds to one of the five controller types noted above.

## **GPD\_SETCTYPE**

This command establishes the type of controller that is to be connected to the specific unit of the gameport device. You must have already successfully opened that unit before issuing the command.

The gameport device is a shared device; many tasks may have it open at any given time. Hence, a high level protocol has been established to prevent multiple tasks from reading the same unit at the same time.

### **Step 1:**

Call **GPD\_ASKCTYPE** and check for a **GPCT\_NOCONTROLLER** return. *Never* issue **GPD\_SETCTYPE** without checking whether the desired gameport unit is in use.

### **Step 2:**

If **GPCT\_NOCONTROLLER** is returned, you have access to the gameport. Set the allocation flag to **GPCT\_MOUSE**, **GPCT\_ABSJOYSTICK** or **GPCT\_RELJOYSTICK** if you use a system supported controller, or **GPCT\_ALLOCATED** if you read directly from the hardware registers.

The following function demonstrates how to do this correctly:

```
struct IOStdReq *game_io_msg = NULL;

BOOL set_controller_type(type)
BYTE type;
{
    BOOL success = FALSE;
    BYTE controller_type = 0;

    Forbid(); /*critical section start */
    game_io_msg->io_Command = GPD_ASKCTYPE; /* inquire current status */
    game_io_msg->io_Length = 1;
    game_io_msg->io_Flags = IOF_QUICK;
    game_io_msg->io_Data = (APTR)&controller_type; /* put answer in here */
    DoIO(game_io_msg);

    /* No one is using this device unit, let's claim it */
    if (controller_type == GPCT_NOCONTROLLER)
```

```

    {
        game_io_msg->io_Command = GPD_SETCTYPE;
        game_io_msg->io_Flags = IOF_QUICK;
        game_io_msg->io_Length = 1;
        game_io_msg->io_Data = (APTR)&type;
        DoIO( game_io_msg);
        success = TRUE;
        UnitOpened = TRUE;
    }
    Permit(); /* critical section end */

    /* success can be TRUE or FALSE, see above */
    return(success);
}

```

### Step 3:

The program must set the controller type back to GPCT\_NOCONTROLLER upon exiting your program:

```

struct IOStdReq *game_io_msg = NULL;

void free_gp_unit()
{
    BYTE type = GPCT_NOCONTROLLER;
    game_io_msg->io_Command = GPD_SETCTYPE;
    game_io_msg->io_Flags = IOF_QUICK;
    game_io_msg->io_Length = 1;
    game_io_msg->io_Data = (APTR)&type;
    DoIO( game_io_msg);
}

```

This three step protocol allows applications to share the gameport device in a system compatible way.

## GPD\_SETTRIGGER

Once you have correctly allocated a unit of the gameport as explained above, use this command to specify the conditions that can trigger a gameport event. The device won't reply to your read request until the trigger conditions have been satisfied.

The information needed for gameport trigger setting is placed into a GamePortTrigger data structure which is defined in the include file *devices/gameport.h*:

```

struct GamePortTrigger
{
    UWORD   gpt_Keys;           /* key transition triggers */
    UWORD   gpt_Timeout;        /* time trigger (vertical blank units) */
    UWORD   gpt_XDelta;         /* X distance trigger */
    UWORD   gpt_YDelta;         /* Y distance trigger */
};

```

- Setting GPTF\_UPKEYS enables the reporting of upward transitions. Setting GPTF\_DOWNKEYS enables the reporting of downward transitions. These flags may both be specified.
- The field gpt\_Timeout specifies the time interval (in vertical blank units) between reports in the absence of another trigger condition. In other words, an event is generated every gpt\_Timeout ticks. Vertical blank units may differ from country to country (e.g 60 Hz NTSC, 50 Hz PAL.)

To find out the exact frequency use this code fragment:

```
#include <exec/execbase.h>
extern struct ExecBase *SysBase;

UBYTE get_frequency(void)
{
    UBYTE hertz;
    hertz = SysBase->VBlankFrequency;
    return(hertz)
}
```

- The **gpt\_XDelta** and **gpt\_YDelta** fields specify the x and y distances which, if exceeded, trigger a report.

For a mouse controller, you can trigger on a certain minimum-sized move in either the x or y direction, on up or down transitions of the mouse buttons, on a timed basis, or any combination of these conditions.

For example, suppose you normally signal mouse events if the mouse moves at least 10 counts in either the x or y directions. If you are moving the cursor to keep up with mouse movements and the user moves the mouse less than 10 counts, after a period of time you will want to update the position of the cursor to exactly match the mouse position. Thus the timed report of current mouse counts would be preferred. The following structure would be used:

```
#define XMOVE 10
#define YMOVE 10

struct GamePortTrigger gpt =
{
    GPTF_UPKEYS + GPTF_DOWNKEYS, /* trigger on all key transitions */
    1800, /* and every 36(PAL) or 30(NTSC) seconds */
    XMOVE, /* for any 10 in an x or y direction */
    YMOVE
};
```

For a joystick controller, you can select timed reports as well as button-up and button-down report trigger conditions. For an absolute joystick specify a value of one (1) for the **gpt\_XDelta** and **gpt\_YDelta** fields or you will not get any direction events. You set the trigger conditions by using the following code or its equivalent:

```
struct IOStdReq *game_io_msg = NULL;

void set_trigger_conditions(struct GamePortTrigger *gpt)
{
    game_io_msg->io_Command = GPD_SETTRIGGER; /* set trigger conditions */
    game_io_msg->io_Data = (APTR)gpt; /* put trigger condition info here */
    game_io_msg->io_Length = sizeof(struct GamePortTrigger);
    DoIO(game_io_msg);
}
```

#### NOTE

If a task sets trigger conditions and does not ask for the position reports the gameport device will queue them up anyway. If the trigger conditions occur again and the gameport device buffer is filled, the additional triggers will be ignored until the buffer is read by a device read request (**GPD\_READEVENT**) or a system **CMD\_CLEAR** command flushes the buffer.

## **GPD\_ASKTRIGGER**

This command retrieves the conditions that must be met by a gameport unit before a pending read request will be satisfied. These conditions are set by the command **GPD\_SETTRIGGER** discussed above.

## **GPD\_READEVENT**

This command reads the internal buffer of the unit of the gameport device opened with **OpenDevice()**. It then puts the event information into the buffer pointed to by the **io\_Data** field of the **IOStdRequest** structure.

```
struct InputEvent game_event; /* defined in <devices/inputevent.h> */
struct IOStdRequest *game_io_msg = NULL;

void send_read_request()
{
    game_io_msg->io_Command = GPD_READEVENT;
    game_io_msg->io_Length = sizeof (struct InputEvent);
    game_io_msg->io_Data = (APTR)&game_event;
    SendIO(game_io_msg); /* Asynchronous */
}
```

The **game\_event.ie\_code** field will contain the event recorded by the gameport device. Check the include file *devices/inputevent.h* for legal event types.

The following example accurately demonstrates all previously discussed information. It is highly recommended that you incorporate the **set\_controller\_type()** function into your program, allowing multiple applications to access the gameport.



## Joystick Example Program

```
/* joy.c - gameport.device joystick example

    compiled with LATTICE 5.02:  LC -bl -cfist -v -y joy.c
    linked with Blink 5.04:  Blink FROM LIB:c.o,joy.o TO joy
                           LIBRARY LIB:LC.lib,LIB:Amiga.lib
*/

#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/devices.h>
#include <devices/gameport.h>
#include <devices/inputevent.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL/C handling */
#endif

#define XMOVE 1
#define YMOVE 1

/* our functions */
BOOL set_controller_type(BYTE);
void set_trigger_conditions(struct GamePortTrigger *);
void send_read_request(void);
void check_move(void);
void flush_buffer(void);
void free_gp_unit(void);
void cleanup(void);
void cleanexit(UBYTE *,LONG);

extern struct ExecBase *SysBase;

struct InputEvent game_event; /* where input event will be stored */
struct IOStdReq *game_io_msg = NULL;
struct MsgPort *game_msg_port = NULL;

/* trigger on all joystick key transitions */
struct GamePortTrigger joytrigger =
{
    GPTF_UPKEYS+GPTF_DOWNKEYS,
    0,
    XMOVE,
    YMOVE
};

UBYTE    hertz; /* vertical blank frequency */
SHORT    codeval,error;
SHORT    button_count, timeouts = 0;
BOOL     UnitOpened, DeviceOpened = FALSE;

void main(int argc,char **argv)
{
    /* Create port for gameport device communications */
    if(!(game_msg_port = CreatePort("RKM_game_port",0)))
        cleanexit(" Error: Can't create port\n",RETURN_FAIL);

    /* Create message block for device IO */
    if(!(game_io_msg = CreateStdIO(game_msg_port)))
        cleanexit(" Error: Can't create IO request\n",RETURN_FAIL);

    /* Open the right/back (unit 1, number 2) gameport.device unit */
```

```

if(error=OpenDevice("gameport.device",1,game_io_msg,0))
    cleanexit(" Error: Can't open gameport.device\n",RETURN_FAIL);
else DeviceOpened=TRUE;

/* Set controller type to joystick */
if (!(set_controller_type(GPCT_ABSJOYSTICK)))
    cleanexit(" Error: Gameport unit in use\n",RETURN_FAIL);

/* Specify the trigger conditions */
set_trigger_conditions(&joytrigger);

printf("\n >>> gameport.device joystick Demo <<<\n\n");
if (hertz == 60) printf(" We are running on a US system (%u Hz).\n",hertz);
if (hertz == 50) printf(" We are running on a PAL system (%u Hz).\n",hertz);
printf(" Attach joystick to right port (A2000) or rear port (A1000).\n");
printf(" Then move joystick and click its button(s).\n\n");
printf(" To exit program press and release fire button 3 times. \n");
printf(" The program also exists if no activity occurs for 1 minute.\n\n");

/* Clear device buffer. There might still be events left */
flush_buffer(); /* To start from a known state */

/* From now on, just read input events into the event buffer,
 * one at a time. READEVENT waits for the preset conditions. */

send_read_request(); /* Send the initial gameport read request */

while(timeouts < 3)
{
    Wait(1L << game_msg_port->mp_SigBit); /* Wait for joystick action */
    GetMsg(game_msg_port); /* Remove message from message port */

    codeval = game_event.ie_Code;
    switch(codeval)
    {
        case IECODE_LBUTTON:
            printf(" FIRE BUTTON PRESSED \n");
            break;

        case (IECODE_LBUTTON | IECODE_UP_PREFIX):
            printf(" FIRE BUTTON RELEASED \n");
            button_count++;
            if (button_count == 3)cleanexit(" Terminated \n",RETURN_OK);
            break;

        case IECODE_NOBUTTON:
            timeouts++; /* Program will timeout after 1 minute */
            button_count = 0;
            break;

        default:
            break;
    }

    check_move(); /* Check for change in position */

    /* We can now re-use our game_event... Send the next read request */
    send_read_request();
}

printf("\n Terminating program (one minute with no activity sensed). \n");
cleanexit(" Timeout \n",RETURN_OK);
}

/* end of main */

/* function definitions */

BOOL set_controller_type(type)
BYTE type;
{
    BOOL success = FALSE;
    BYTE controller_type = 0;

```

```

Forbid(); /*critical section start */
game_io_msg->io_Command = GPD_ASKCTYPE; /* inquire current status */
game_io_msg->io_Length = 1;
game_io_msg->io_Flags = IOF_QUICK;
game_io_msg->io_Data = (APTR)&controller_type; /* put answer in here */
DoIO(game_io_msg);

/* No one is using this device unit, let's claim it */
if (controller_type == GPCT_NOCONTROLLER)
{
    game_io_msg->io_Command = GPD_SETCTYPE;
    game_io_msg->io_Flags = IOF_QUICK;
    game_io_msg->io_Length = 1;
    game_io_msg->io_Data = (APTR)&type;
    DoIO( game_io_msg);
    success = TRUE;
    UnitOpened = TRUE;
}
Permit(); /* critical section end */
return(success);
}

void set_trigger_conditions(struct GamePortTrigger *gpt)
{
    /* get vertical blank frequency
    * US = 60 Hz, PAL = 50 Hz */
    hertz = SysBase->VBlankFrequency;

    /* trigger every 20 seconds */
    joytrigger.gpt_Timeout = (UWORD)hertz * 20;

    game_io_msg->io_Command = GPD_SETTRIGGER;
    game_io_msg->io_Length = (LONG)sizeof(struct GamePortTrigger);
    game_io_msg->io_Data = (APTR)gpt;
    DoIO(game_io_msg);
}

void check_move()
{
    WORD xmove, ymove;
    xmove = game_event.ie_X;
    ymove = game_event.ie_Y;

    if(xmove != 0 || ymove != 0)
    {
        printf(" x = %2ld , y = %2ld -->",xmove, ymove);

        if (xmove == 1 && ymove == 0) printf(" RIGHT \n");
        if (xmove == -1 && ymove == 0) printf(" LEFT \n");
        if (xmove == 0 && ymove == 1) printf(" DOWN \n");
        if (xmove == 0 && ymove == -1) printf(" UP \n");

        if (xmove == 1 && ymove == 1) printf(" RIGHT DOWN \n");
        if (xmove == -1 && ymove == 1) printf(" LEFT DOWN \n");
        if (xmove == 1 && ymove == -1) printf(" RIGHT UP \n");
        if (xmove == -1 && ymove == -1) printf(" LEFT UP \n");

        timeouts = 0;
    }
}

void flush_buffer()
{
    game_io_msg->io_Command = CMD_CLEAR;
    game_io_msg->io_Flags = IOF_QUICK;
    DoIO(game_io_msg);
}

void send_read_request()
{
    game_io_msg->io_Command = GPD_READEVENT;
    game_io_msg->io_Length = sizeof(struct InputEvent);

```

```

game_io_msg->io_Data = (APTR)&game_event;
SendIO(game_io_msg); /* Asynchronous - message will return later */
}

void free_gp_unit()
{
    BYTE type = GPCT_NOCONTROLLER;
    game_io_msg->io_Command = GPD_SETCTYPE;
    game_io_msg->io_Flags = IOF_QUICK;
    game_io_msg->io_Length = 1;
    game_io_msg->io_Data = (APTR)&type;
    DoIO( game_io_msg);
}

void cleanexit(UBYTE *s, LONG n)
{
    if(*s) printf(s);
    /* Free gameport unit so other applications can use it ! */
    if (UnitOpened) free_gp_unit();
    cleanup();
    exit(n);
}

void cleanup()
{
    if(DeviceOpened) CloseDevice( game_io_msg);
    if(game_io_msg) DeleteStdIO(game_io_msg);
    if(game_msg_port) DeletePort(game_msg_port);
}
/* eof joy.c */

```

# Chapter 35

## Input Device

This chapter describes the Amiga input device which is the central collection point for input events disseminated throughout the system. The best way to describe the input device is a manager of a stream with feeders. The input device itself and other modules such as the file system add events to the stream; so do input device “users”—programs or other devices that use parts of the stream or change it in some way. Feeders of the input device include the keyboard device, timer device, gameport device. There may be other feeders depending on the system and software configuration. The keyboard, gameport, and timer devices are special cases in that the input device opens them and asks them for input. Other feeders are “active”—they send the input to the input device. Users of the input device include Intuition and the console device.

### Introduction

The input device is automatically opened by the console device when the system boots. When the input device is opened, a task named “input.device” is started. The input device task communicates directly with the keyboard device to obtain raw key events. It also communicates with the gameport device to obtain mouse button and mouse movement events and with the timer device to obtain time events. In addition to these events, you can add your own input events to the input device, to be fed to the handler chain (see below).

The keyboard device is accessible directly (see the “Keyboard Device” chapter). However, once the input.device task has started, you should not read events from the keyboard device directly, since doing so will deprive the input device of the events and confuse key repeating.

The gameport device has two units. As you view the Amiga, looking at the gameport connectors, connector “1” is assigned as the primary mouse input for Intuition and contributes gameport input events to the input event stream. Connector “2” is handled by the other gameport unit and is currently unassigned. While the input device task is running, that task expects to read the input from connector 1. Direct use of the gameport device is covered in a separate chapter of this manual.

The timer device is used to generate time events for the input device. It is also used to control key repeat rate and key repeat threshold. The timer device is a shared-access device and is described in its own separate chapter.

## Input Device Commands

The input device allows the following system functions:

| Command              | Operation  |
|----------------------|--|
| <b>OpenDevice()</b>  | Obtain shared use of the input device.           |
| <b>CloseDevice()</b> | Relinquish use of the input device.              |
| <b>DoIO()</b>        | Initiate a command, and wait for it to complete. |
| <b>SendIO()</b>      | Initiate a command, and return immediately.      |
| <b>AbortIO()</b>     | Abort a command already in the queue.            |

Only the Start, Stop, Invalid, and Flush commands have been implemented for this device. All other standard commands are no-operations.

The input device also supports the device-specific commands shown in the table below.

Table 35-1: Input Device Commands

| I/O Command           | Operation   |
|-----------------------|---|
| <b>IND_WRITEEVENT</b> | Propagate an input event stream to all devices.   |
| <b>IND_ADDHANDLER</b> | Add an input-stream handler into the handler chain.   |
| <b>IND_REMHANDLER</b> | Remove an input-stream handler from the handler chain.                                      |
| <b>IND_SETTHRESH</b>  | Set the repeating key hold-down time before repeat starts.                                  |
| <b>IND_SETPERIOD</b>  | Set the period at which a repeating key repeats.  |
| <b>IND_SETMPORT</b>   | Set the gameport port to which the mouse is connected.                                      |
| <b>IND_SETMTRIG</b>   | Set conditions that must be met by a mouse before a pending read request will be satisfied. |
| <b>IND_SETMTYPE</b>   | Set the type of device at the mouse port.   |

The device-specific commands are described below. First though, it may be helpful to consider the types of input events that the input device deals with. An input event is a data structure that describes the following:

- The class of the event—often describes the device that generated the event.

- The subclass of the event—space for more information if needed.
- The code—keycode if keyboard, button information if mouse, others.
- A qualifier such as “Alt key also down,” “key repeat active”.
- A position field that contains a data address or a mouse position count.
- A time stamp, to determine the sequence in which the events occurred.
- A link-field by which input events are linked together.

The various types of input events are listed in the include file *devices/inputevent.h*. That information is not repeated here. You can find more information about input events in the chapters titled “Gameport Device” and “Console Device.”

There is a difference between simply receiving an input event from a device and actually becoming a handler of an input event stream. A handler is a routine that is passed an input event list. It is up to the handler to decide if it can process the input events. If the handler does not recognize an event, it leaves it undisturbed in the event list.

#### NOTE

Handlers can themselves generate new linked lists of events which can be passed down to lower priority handlers.

### IND\_ADDHANDLER COMMAND

You add a handler to the chain using the command `IND_ADDHANDLER`. Here is a typical C-language call to the `IND_ADDHANDLER` function. This assumes that you have a properly initialized `IOStdReq` and have already called `OpenDevice()`.

```
struct Interrupt *inputHandler;
struct IOStdReq *inputReqBlk;

inputHandler->is_Code=ButtonSwap;          /* Address of code */
inputHandler->is_Data=NULL;                  /* User Value passed in A1 */
inputHandler->is_Node.ln_Pri=100;            /* Priority in food chain */
inputHandler->is_Node.ln_Name=NameString;    /* Name of handler */

inputReqBlk->io_Data=(APTR)inputHandler;    /* Point to the structure */
inputReqBlk->io_Command=IND_ADDHANDLER;     /* Set command ... */
DoIO((struct IORequest *)inputReqBlk);      /* DoIO( ) the command */
```

Intuition is one of the input device handlers and normally distributes most of the input events. Intuition inserts itself at priority position 50. The console.device sits at priority position 0. You can choose the position in the chain at which your handler will be inserted by setting the priority field in the list-node part of the interrupt data structure you pass to this routine.

## NOTE

Any processing time expended by a handler subtracts from the time available before the next event happens. Therefore, handlers for the input stream *must* be fast. For this reason it is recommended that the handlers be written in assembly.

### Rules for Input Device Handlers

The following rules should be followed when you are designing an input handler:

- If an input handler is capable of processing a specific kind of an input event and that event has no links (ie `_NextEvent = 0`), the handler can end the handler chain by returning a NULL (0) value.
- If there are multiple events linked together, the handler is free to delink an event from the input event chain, thereby passing a shorter list of events to subsequent handlers. The starting address of the modified list is the return value.
- If a handler wishes to add new events to the chain that it passes to a lower-priority handler, it may initialize memory to contain the new event or event chain. The handler, when it again gets control on the next round of event handling, should assume nothing about the current contents of the memory blocks attached to the event chain. Lower priority handlers may have modified the memory as they handled their part of the event. The handler that allocates the memory for this purpose should keep track of the starting address and the size of this memory chunk so that the memory can be returned to the free memory list when it is no longer needed.

Your assembly language handler routine should be structured similar to the following pseudo-language statement:

```
newEventChain = yourHandlerCode(oldEventChain, yourHandlerData);  
d0             =             a0             a1
```

where

- `yourHandlerCode` is the entry point to your routine
- `oldEventChain` is the starting address for the current chain of input events
- `yourHandlerData` is a user-definable value, usually a pointer to some data structure your handler requires.
- `newEventChain` is the starting address of an event chain which you are passing to the next handler, if any

When your handler code is called, the event chain is passed in A0 and the handler data is passed in A1. (You may choose not to use A1) When your code returns, it should return the pointer to the event chain in D0. If all of the events were removed by the routine, return NULL. A NULL (0) value terminates the handling thus freeing more CPU resources.

Memory that you use to describe a new input event that you have added to the event chain is available for reuse or deallocation when the handler is called again or after the `IND_REMHANDLER` command for the handler is complete. There is no guarantee that any field in the event is unchanged since a handler may change any field of an event that comes through the food chain.



## NOTE

Altering a repeat key report will confuse the input device when it tries to stop the repeating after the key is raised.

Because `IND_ADDHANDLER` installs a handler in any position in the handler chain, it can, for example, ignore specific types of input events as well as act upon and modify existing streams of input. It can even create new input events for Intuition or other programs to interpret.

## IND\_REMHANDLER COMMAND

You remove a handler from the handler chain with the command `IND_REMHANDLER`. Assuming that you have a properly initialized `IOStdReq` block as a result of a call to `OpenDevice( )` (for the input device) and you have already added the handler using `IND_ADDHANDLER`, (see example above) here is a typical C-language call to the `IND_REMHANDLER` function:

```
struct Interrupt *inputHandler;
struct IOStdReq *inputReqBlk;

inputReqBlk->io_Data=(APTR)inputHandler; /* Which handler to REM */
inputReqBlk->io_Command=IND_REMHANDLER; /* The REM command */
DoIO((struct IORequest *)inputReqBlk); /* Send the command */
```

## IND\_WRITEEVENT COMMAND

Typically, input events are internally generated by the timer device, keyboard device, and gameport device. A *user* can also generate an input event and send it to the input device. It will then be treated as any other event and passed through to the input handler chain. You can create your own stream of events and then send them to the input device using the `IND_WRITEEVENT` command.

This example sends in a few phony mouse-movement events.

```
/*
 * InputDevice example
 *
 * This example adds a few mouse movements to the input chain...
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/input.h>
#include <devices/inputevent.h>

#include <proto/all.h>

VOID main(VOID)
{
    struct IOStdReq *inputReqBlk;
    struct MsgPort *inputPort;
    struct InputEvent *FakeEvent;
    short loop;
    short num;
    short numloop;

    if (inputPort=CreatePort(NULL, NULL))
    {
        if (FakeEvent=AllocMem(sizeof(struct InputEvent), MEMF_PUBLIC))
        {
```

```

if (inputReqBlk=(struct IOStdReq *)CreateExtIO(inputPort,
                                              sizeof(struct IOStdReq)))
{
    if (!OpenDevice("input.device",NULL,
                  (struct IORequest *)inputReqBlk,NULL))
    {
        for (numloop=0;numloop<4;numloop++)
            for (loop=0;loop<8;loop++)
                for (num=0;num<20;num++)
                {
                    FakeEvent->ie_NextEvent=NULL;
                    FakeEvent->ie_Class=IECLASS_RAWMOUSE;
                    FakeEvent->ie_Code=IECODE_NOBUTTON;
                    FakeEvent->ie_Qualifier=IEQUALIFIER_RELATIVEMOUSE;
                    FakeEvent->ie_X=0;
                    FakeEvent->ie_Y=0;

                    switch (loop)
                    {
                        case 0: FakeEvent->ie_X=1;
                        case 1: FakeEvent->ie_Y=1; break;
                        case 2: FakeEvent->ie_Y=1;
                        case 3: FakeEvent->ie_X=-1; break;
                        case 4: FakeEvent->ie_X=-1;
                        case 5: FakeEvent->ie_Y=-1; break;
                        case 6: FakeEvent->ie_Y=-1;
                        case 7: FakeEvent->ie_X=1; break;
                    }

                    inputReqBlk->io_Data=(APTR)FakeEvent;
                    inputReqBlk->io_Command=IND_WRITEEVENT;
                    inputReqBlk->io_Flags=0;
                    inputReqBlk->io_Length=sizeof(struct InputEvent);
                    DoIO((struct IORequest *)inputReqBlk);
                }
            CloseDevice((struct IORequest *)inputReqBlk);
        DeleteExtIO((struct IORequest *)inputReqBlk);
    }
    FreeMem(FakeEvent,sizeof(struct InputEvent));
}
DeletePort(inputPort);
}

```

## NOTE

This command propagates the input event thru the handler chain. The handlers may link other events onto the end of this event, or modify the contents of the data structure you constructed in any way it wishes. This means do not rely on any of the data being the same from event to event.

## IECLASS\_POINTERPOS

One event of special note is **IECLASS\_POINTERPOS**. This event, when sent to the input device, is fed into Intuition to specify a position for the mouse pointer **RELATIVE TO THE INTUITION VIEW ORIGIN IN 640 by 400 COORDINATES**. (*This will cause strange results on screens with higher resolutions.*) The coordinates are provided in the pseudo-fields **ie\_X** and **ie\_Y**. Internally, Intuition will convert this event into the proper **RAWMOUSE** event, replacing **ie\_X/Y** with a suitable relative mouse motion. The presence of the qualifier **IEQUALIFIER\_RELATIVEMOUSE** has the same effect as using **IECLASS\_RAWMOUSE**.

## IND\_SETTHRESH COMMAND

This command sets the timing in seconds and microseconds for the input device to indicate how long a user must hold down a key before it begins to repeat. This command is normally performed by the Preferences tool or by Intuition when it notices that the Preferences have been changed. If you wish, you can call this function.

This command takes a **timerequest** IO request (just like the timer device).

## IND\_SETPERIOD COMMAND

This command sets the time period between key repeat events once the initial period threshold has elapsed.

This command also takes a **timerequest** IO request. Again, it is a command normally issued by Intuition and preset by the Preferences tool. A typical calling sequence is shown below; change the timing period values to suit your application.

```
/*
 * InputDevice example
 *
 * This example changes the threshold and period of the key repeat...
 */
#include <exec/types.h>
#include <exec/memory.h>
#include <devices/input.h>
#include <devices/timer.h>
#include <proto/all.h>

VOID main(VOID)
{
    struct timerequest *inputReqBlk;
    struct MsgPort      *inputPort;

    if (inputPort=CreatePort(NULL,NULL))
    {
        if (inputReqBlk=(struct timerequest *)CreateExtIO(inputPort,
                                                             sizeof(struct timerequest)))
        {
            if (!OpenDevice("input.device",NULL,
                           (struct IORequest *)inputReqBlk,NULL))
            {
                inputReqBlk->tr_node.io_Command=IND_SETTHRESH;
                inputReqBlk->tr_time.tv_secs=1;
                inputReqBlk->tr_time.tv_micro=500000; /* 1.5 seconds */

                DoIO((struct IORequest *)inputReqBlk);

                inputReqBlk->tr_node.io_Command=IND_SETPERIOD;
                inputReqBlk->tr_time.tv_secs=0;
                inputReqBlk->tr_time.tv_micro=12000; /* .012 seconds */

                DoIO((struct IORequest *)inputReqBlk);

                CloseDevice((struct IORequest *)inputReqBlk);
            }
            DeleteExtIO((struct IORequest *)inputReqBlk);
        }
        DeletePort(inputPort);
    }
}
```

## Input Device and Intuition

There are several ways to receive information from the various devices that are part of the input device. The first way is to communicate directly with the device. This method is not recommended while the input device task is running -- which is most of the time. The second way is to become a handler for the stream of events which the input device produces. That method is shown above.

The third method of getting input from the input device is to retrieve the data from the console device or from the IDCMP (Intuition Direct Communications Message Port). See the Intuition chapter for more information on IDCMP messages. See the console chapter for more information on console device I/O.

## Sample Program

```
----- InputSwap.a -----
/*
 * InputDevice example
 *
 * This example swaps the function of the left and right mouse buttons
 * The C code is just the wrapper that installs and removes the
 * input.device handler that does the work.
 *
 * The handler is written in assembly code since it is important that
 * handlers be as fast as possible while processing the input events.
 *
 * Compile and link as follows:
 *
 * LC -bl -cfist -v -w InputSwap.c
 *
 * Cape assemble:
 *
 * Casm -a InputHandler.a -i INCLUDE: -o InputHandler.o
 *
 * BLink:
 *
 * BLink from LIB:c.o+InputSwap.o+inputhandler.o LIB LIB:lcs.lib LIB:amiga.lib TO InputSwap
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/input.h>
#include <intuition/intuition.h>

#include <proto/all.h>

UBYTE NameString[]="Button Swap";

struct NewWindow mywin=
{
    0,0,124,10,0,1,CLOSEWINDOW,
    WINDOWDRAG|WINDOWCLOSE|SIMPLE_REFRESH|NOCAREREFRESH,
    NULL,NULL,NameString,NULL,NULL,0,0,0,0,WBENCHSCREEN
};

extern VOID ButtonSwap();

extern struct IntuitionBase *IntuitionBase;
```

```

/*
 * This routine opens a window and waits for the one event that
 * can happen (CLOSEWINDOW) This is just to let the user play with
 * the swapped buttons and then close the program...
 */
VOID WaitForUser(VOID)
{
    struct Window *win;

    if (IntuitionBase=(struct IntuitionBase *)
        OpenLibrary("intuition.library",0L))
    {
        if (win=OpenWindow(&mywin))
        {
            WaitPort(win->UserPort);
            ReplyMsg(GetMsg(win->UserPort));

            CloseWindow(win);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
}

VOID main(VOID)
{
    struct IOStdReq *inputReqBlk;
    struct MsgPort *inputPort;
    struct Interrupt *inputHandler;

    if (inputPort=CreatePort(NULL,NULL))
    {
        if (inputHandler=AllocMem(sizeof(struct Interrupt),
            MEMF_PUBLIC|MEMF_CLEAR))
        {
            if (inputReqBlk=(struct IOStdReq *)CreateExtIO(inputPort,
                sizeof(struct IOStdReq)))
            {
                if (!OpenDevice("input.device",NULL,
                    (struct IORequest *)inputReqBlk,NULL))
                {
                    inputHandler->is_Code=ButtonSwap;
                    inputHandler->is_Data=NULL;
                    inputHandler->is_Node.ln_Pri=100;
                    inputHandler->is_Node.ln_Name=NameString;
                    inputReqBlk->io_Data=(APTR)inputHandler;
                    inputReqBlk->io_Command=IND_ADDHANDLER;
                    DoIO((struct IORequest *)inputReqBlk);

                    WaitForUser();

                    inputReqBlk->io_Data=(APTR)inputHandler;
                    inputReqBlk->io_Command=IND_REMHANDLER;
                    DoIO((struct IORequest *)inputReqBlk);

                    CloseDevice((struct IORequest *)inputReqBlk);
                }
                DeleteExtIO((struct IORequest *)inputReqBlk);
            }
            FreeMem(inputHandler,sizeof(struct Interrupt));
        }
        DeletePort(inputPort);
    }
}

----- InputHandler.a -----
*
* InputHandler that does a Left/Right mouse button swap...
*
*****
*
* Required includes...
*

```

```

INCLUDE "exec/types.i"
INCLUDE "exec/io.i"
INCLUDE "devices/inpustevent.i"

*
*****
*
* Make the entry point external...
*
xdef _ButtonSwap
*
*****
*
* This is the input handler that will swap the
* mouse buttons for left handed use.
*
* The event list gets passed to you in a0.
* The is_Data field is passed to you in a1.
* This example does not use the is_Data field...
*
* On exit you must return the event list in d0. In this way
* you could add or remove items from the event list.
*
*****
*
* The handler gets called here...
*
_ButtonSwap: move.l a0,-(sp) ; Save the event list
*
* Since the event list could be a linked list, we start a loop
* here to handle all of the events passed to us.
*
CheckLoop: move.w ie_Qualifier(a0),d1 ; Get qualifiers...
           move.w d1,d0 ; Two places...
*
* Since we are changing left and right mouse buttons, we need to make
* sure that we change the qualifiers on all of the messages. The
* left and right mouse buttons are tracked in the message qualifiers
* for use in such things as dragging. To make sure that we continue
* to drag correctly, we change the qualifiers.
*
CheckRight: btst #IEQUALIFIERB_RBUTTON,d1 ; Check for right
           beq.s NoRight
           bset #IEQUALIFIERB_LEFTBUTTON,d0 ; Set the left...
           beq.s CheckLeft
NoRight: bclr #IEQUALIFIERB_LEFTBUTTON,d0 ; Clear the left...
*
CheckLeft: btst #IEQUALIFIERB_LEFTBUTTON,d1 ; Check for left
           beq.s NoLeft
           bset #IEQUALIFIERB_RBUTTON,d0 ; Set the right...
           beq.s CheckLeft
NoLeft: bclr #IEQUALIFIERB_RBUTTON,d0 ; Clear the right...
*
           move.w d0,ie_Qualifier(a0) ; Save back...
*
* The actual button up/down events are transmitted as the
* code field in RAWMOUSE events. The code field must be
* checked and modified when needed on RAWMOUSE events. If the
* event is not a RAWMOUSE, we are done with it.
*
           cmp.b #IECLASS_RAWMOUSE,ie_Class(a0) ; Check for mouse
           bne.s NextEvent ; If not, next...
*
           move.w ie_Code(a0),d0 ; Get code...
           move.w d0,d1 ; Save...
           and.w #$7F,d0 ; Mask UP_PREFIX
           cmp.w #IECODE_LBUTTON,d0 ; Check for Left...
           beq.s SwapThem ; If so, swap...
           cmp.w #IECODE_RBUTTON,d0 ; Check for Right...
           bne.s NextEvent ; If not, next...
*
SwapThem: eor.w #1,d1 ; Flip bottom bit
           move.w d1,ie_Code(a0) ; Save it...

```

```

*
* The event list is linked via a pointer to the next event
* in the first element of the structure. That is why it is not
* necessary to use: move.l ie_NextEvent(a0),d0
*
* The reason I move to d0 first is that this also checks for zero.
* The last event in the list will have a NULL ie_NextEvent field.
* This is NOT as standard EXEC list where the node after the last
* node is NULL. Input events are single-linked for performance.
*
NextEvent:    move.l  (a0),d0                ; Get next event
              move.l  d0,a0                ; into a0...
              bne.s   CheckLoop            ; Do some more.
*
* All done, just return the event list... (in d0)
*
              move.l  (sp)+,d0             ; Get event list back...
              rts                    ; return from handler...

```

# **Chapter 36**

## **Keyboard Device**

### **Introduction**

The keyboard device gives system access to the Amiga keyboard. When you send this device the command to read one or more keystrokes from the keyboard, for each keystroke (whether key-up or key-down) the keyboard device creates a data structure called an input event to describe what happened. A keyboard input event includes the key code (including up or down transition status), information about the current state of the left and right Shift/Alt/Amiga keys, the state of the Control and CapsLock key, and whether the key came from the numeric keypad area.

Thus, the keyboard device provides more information than simply the “raw” key input that might be obtained by directly reading the hardware registers. In addition, the keyboard device can buffer keystrokes for you. If your task takes more time to process prior keystrokes, the keyboard device senses additional keystrokes and saves several keystrokes as a type-ahead feature. If your task takes an exceptionally long time to read this information from the keyboard, any keystrokes queued up beyond the number the system can handle will be ignored. Normally, the input device task requests for and processes keyboard events, turning them into input device events so that no keystrokes are lost. You can find more information about keyboard event-queuing in the “Input Device” chapter and in the Intuition chapter “Input and Output Methods.”



## Keyboard Device Commands

The following system functions are used to send commands to the keyboard device.

| Command              | Operation                                       |
|----------------------|---|
| <b>OpenDevice()</b>  | Obtain shared use of the keyboard device        |
| <b>CloseDevice()</b> | Relinquish use of the keyboard device           |
| <b>DoIO()</b>        | Initiate a command, and wait for it to complete |
| <b>SendIO()</b>      | Initiate a command, and return immediately      |
| <b>AbortIO()</b>     | Abort a command already in the queue            |

The keyboard device also responds to the following commands:

| I/O Command                 | Operation  |
|-----------------------------|--|
| <b>KBD_ADDRESETHANDLER</b>  | Add a reset handler to the device  |
| <b>KBD_REMRESETHANDLER</b>  | Remove a reset handler from the device   |
| <b>KBD_RESETHANDLERDONE</b> | Indicate that a handler has completed its job and reset could possibly occur now |
| <b>KBD_READMATRIX</b>       | Read the state of every key in the keyboard                                      |
| <b>KBD_READEVENT</b>        | Read one (or more) key event from the keyboard device                            |

### **KBD\_READMATRIX**

This command lets you discover the current state of every key in the key matrix (UP = 0, DOWN = 1). You provide a data area that is at least large enough to hold one bit per key, approximately 16 bytes. The keyboard layout is shown in the figure below, indicating the raw numeric value that each key transmits when it is pressed. This value is the numeric position that the key occupies in the key matrix.

|             |            |          |          |          |          |          |          |          |          |           |         |         |              |       |           |            |    |    |    |         |         |         |         |             |
|-------------|------------|----------|----------|----------|----------|----------|----------|----------|----------|-----------|---------|---------|--------------|-------|-----------|------------|----|----|----|---------|---------|---------|---------|-------------|
| ESC<br>45   | F1<br>50   | F2<br>51 | F3<br>52 | F4<br>53 | F5<br>54 | F6<br>55 | F7<br>56 | F8<br>57 | F9<br>58 | F10<br>59 |         |         |              |       |           |            |    |    |    |         |         |         |         |             |
| 00          | 01         | 02       | 03       | 04       | 05       | 06       | 07       | 08       | 09       | 0A        | 0B      | 0C      | 0D           | 41    | DEL<br>46 | HELP<br>5F | 5A | 5B | 5C | 5D      |         |         |         |             |
| ← →<br>42   | Q<br>10    | W<br>11  | E<br>12  | R<br>13  | T<br>14  | Y<br>15  | U<br>16  | I<br>17  | O<br>18  | P<br>19   | 1A      | 1B      | RETURN<br>44 |       |           |            |    |    |    |         | 7<br>3D | 8<br>3E | 9<br>3F | 4A          |
| CTRL<br>63  | CAPS<br>62 | A<br>20  | S<br>21  | D<br>22  | F<br>23  | G<br>24  | H<br>25  | J<br>26  | K<br>27  | L<br>28   | :       | ;       | 2A           | 2B    |           |            |    |    |    |         | 4<br>2D | 5<br>2E | 6<br>2F | 5E          |
| SHIFT<br>60 | 30         | Z<br>31  | X<br>32  | C<br>33  | V<br>34  | B<br>35  | N<br>36  | M<br>37  | ^        | >         | ?<br>39 | 3A      | 61           | SHIFT |           |            |    |    |    |         | 1<br>1D | 2<br>1E | 3<br>1F | ENTER<br>43 |
| ALT<br>64   | A<br>66    | 40       |          |          |          |          |          |          |          |           |         | A<br>67 | ALT<br>65    |       |           |            |    |    |    | 0<br>0F |         |         | 3C      |             |

|           |            |
|-----------|------------|
| DEL<br>46 | HELP<br>5F |
|-----------|------------|

|         |         |         |
|---------|---------|---------|
| A<br>4C |         |         |
| ^<br>4F | V<br>4D | ><br>4E |

|         |         |         |             |
|---------|---------|---------|-------------|
| (<br>5A | )<br>5B | /<br>5C | *<br>5D     |
| 7<br>3D | 8<br>3E | 9<br>3F | -<br>4A     |
| 4<br>2D | 5<br>2E | 6<br>2F | +<br>5E     |
| 1<br>1D | 2<br>1E | 3<br>1F | ENTER<br>43 |
| 0<br>0F |         |         | .<br>3C     |

Figure 36-1: Raw Key Matrix

The following example will read the matrix and display the up-down state of all of the elements in the matrix in a table. If you read the column header and then the row number as a hex number, it would correspond to the raw key code.

```

/*
 * Keyboard device matrix example...
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <devices/keyboard.h>

#include <proto/exec.h>

#include <stdio.h>

int CXBRK(VOID) { return(0); }

/*
 * There are keycodes from 0x00 to 0x77, (with 0x78 to 0x7F reserved)
 * so the matrix needs to be of 0x80 bits in size, or 0x80/8 which
 * is 0x10 or 16 bytes...
 */
#define MATRIX_SIZE 16L

/*
 * This assembles the matrix for display that translates directly
 * to the RAW key value of the key that is up or down
 */
VOID Display_Matrix(UBYTE *keyMatrix)
{
    SHORT bitcount;
    SHORT bytecount;
    SHORT mask;
    USHORT twobyte;

    printf("\n    0 1 2 3 4 5 6 7");
    printf("\n +-----");
    for (bitcount=0;bitcount<16;bitcount++)
    {
        printf("\n%x |",bitcount);
    }

```

```

        mask=1 << bitcount;
        for (bytecount=0;bytecount<16;bytecount+=2)
        {
            twobyte=keyMatrix[bytecount] | (keyMatrix[bytecount+1] << 8);
            if (twobyte & mask) printf(" *");
            else printf(" -");
        }
    }
    printf("\n\n");
}

VOID main(int argc, char *argv[])
{
    struct IOStdReq *keyRequest;
    struct MsgPort *keyPort;
    UBYTE *keyMatrix;

    if (keyPort=CreatePort(NULL,NULL))
    {
        if (keyRequest=(struct IOStdReq *)CreateExtIO(keyPort,
            sizeof(struct IOStdReq)))
        {
            if (!OpenDevice("keyboard.device",NULL,
                (struct IORequest *)keyRequest,NULL))
            {
                if (keyMatrix=AllocMem(MATRIX_SIZE,MEMF_PUBLIC|MEMF_CLEAR))
                {
                    keyRequest->io_Command=KBD_READMATRIX;
                    keyRequest->io_Data=(APTR)keyMatrix;
                    keyRequest->io_Length=13; /* MUST for 1.2/1.3 */
                    DoIO((struct IORequest *)keyRequest);

                    /* Check for CLI startup... */
                    if (argc) Display_Matrix(keyMatrix);

                    FreeMem(keyMatrix,MATRIX_SIZE);
                }
                CloseDevice((struct IORequest *)keyRequest);
            }
            DeleteExtIO((struct IORequest *)keyRequest);
        }
        DeletePort(keyPort);
    }
}

```

### NOTE

Although the `io_Length` field is described as being the size of the matrix you will accept, you *must* set this field to 13 for V34 and earlier versions of Kickstart.

To find the status of a particular key — for example, to find out if the F2 key is down — you find the bit that specifies the current state by dividing the key matrix value by 8. Since hex 51 = 81, this indicates that the bit is in byte number 10 of the matrix. Then take the same number (decimal 81) and use modulo 8 to determine which bit position within that byte represents the state of the key. This yields a value of 1. So, by reading bit position 1 of byte number 10, you determine the status of the function key F2.

### KBD\_ADDRESETHANDLER

This command adds a routine to a chain of reset-handlers. When a user presses the key sequence Ctrl-left Amiga-right Amiga (the reset sequence), the keyboard device senses this and calls a prioritized chain of reset-handlers. These might be thought of as clean-up routines that “must” be performed before reset is allowed to occur. For example, if a disk write is in progress, the system should finish that before resetting the hardware so as not to corrupt the contents of the disk.

## NOTE

If you add your own handler to this chain, you *must* ensure that your handler allows the rest of reset processing to occur. Reset *must* continue to function. Also, if you don't execute your reset code fast enough, the system will still reboot (about 10 seconds).

It is also important to note that not all Amigas handle reset processing in the same way. On the A500, the reset key sequence sends a hardware reset signal and never goes through the reset handlers. Also some of the early A2000s (i.e., German keyboards with the function keys the same size as the ESC-key) do not handle the reset via the reset handlers. It is thus recommended that your application not rely on the reset handler abilities of the keyboard device.

Reset handlers are just like any other handler and are added to the handler list with an **Interrupt** structure. The priority field in the list node of the interrupt structure establishes the sequence in which reset handlers are processed by the system. Keyboard reset handlers are currently limited to the priority values of a software interrupt, that is, values of -32, -16, 0, 16, and 32)

Only the `is_Data` field is passed to a reset handler. Any return value from it is ignored. All keyboard reset handlers are activated if time permits. Normally, a reset handler will just signal the requisite task and return. The task would then do whatever it needed and signal the reset handler that it is done.

## KBD\_REMRESETHANDLER

This command is used to remove a keyboard reset handler from the system. You need to supply the same Interrupt structure to this command that you used with the KBD\_ADDRESETHANDLER command.

## KBD\_RESETHANDLERDONE

This command tells the system that your reset handling code has completed. If you are the last outstanding reset handler, the system will reset after this call.

## NOTE

After 10 seconds, the system will still reboot, regardless of outstanding reset handlers.

Here is an example program that installs a reset handler and either waits for the reboot or for the user to close the window. If there was a reboot, the window will close and, if executed from the CLI, it will display a few messages. If the user closes the window, the handler is removed and the program exits cleanly.

```
; Compile, assemble, and link as follows:
; Cape assembler...
;
LC -cfist -b1 -v -w keyreset.c
CAsm -a keyhandler.a -i INCLUDE: -o keyhandler.o
BLink FROM c.o+keyreset.o+keyhandler.o LIB LIB:lc.lib+LIB:amiga.lib TO keyreset

----- keyreset.c -----
/*
 * Keyboard device reset handler example...
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/ports.h>
#include <exec/memory.h>
```

```

#include <devices/keyboard.h>
#include <intuition/intuition.h>

#include <proto/all.h>

#include <stdio.h>

int CXBRK(VOID) { return(0); }

extern VOID ResetHandler();

UBYTE NameString[]="Reset Handler Test";

struct NewWindow mywin=
{
    0,0,178,10,0,1,CLOSEWINDOW,
    WINDOWDRAG|WINDOWCLOSE|SIMPLE_REFRESH|NOCAREREFRESH,
    NULL,NULL,NameString,NULL,NULL,0,0,0,0,WBENCHSCREEN
};

extern struct IntuitionBase *IntuitionBase;

struct MyData
{
    struct Task *MyTask;
    ULONG MySignal;
};

/*
 * This routine opens a window and waits for the one event that
 * can happen (CLOSEWINDOW) or the signal from the reset handler.
 */
short WaitForUser(ULONG MySignal)
{
    struct Window *win;
    short ret=0;

    if (IntuitionBase=(struct IntuitionBase *)
        OpenLibrary("intuition.library",0L))
    {
        if (win=OpenWindow(&mywin))
        {
            ret=(MySignal==Wait(MySignal | (1L << win->UserPort->mp_SigBit)));
            CloseWindow(win);
        }
        CloseLibrary((struct Library *)IntuitionBase);
    }
    return(ret);
}

VOID main(int argc, char *argv[])
{
    struct IOStdReq *keyReqBlk;
    struct MsgPort *keyPort;
    struct Interrupt *keyHandler;
    struct MyData MyDataStuff;
    ULONG MySignal;

    if ((MySignal=AllocSignal(-1L))!=-1)
    {
        MyDataStuff.MyTask=FindTask(NULL);
        MyDataStuff.MySignal=1L << MySignal;
        if (keyPort=CreatePort(NULL,NULL))
        {
            if (keyHandler=AllocMem(sizeof(struct Interrupt),
                MEMF_PUBLIC|MEMF_CLEAR))
            {
                if (keyReqBlk=(struct IOStdReq *)CreateExtIO(keyPort,
                    sizeof(struct IOStdReq)))
                {
                    if (!OpenDevice("keyboard.device",NULL,
                        (struct IORequest *)keyReqBlk,NULL))

```

```

    {
        keyHandler->is_Code=ResetHandler;
        keyHandler->is_Data=(APTR) &MyDataStuff;

        /*
         * Note that only software interrupt priorities
         * can be used for the .ln_Pri on the reset
         * handler...
         */
        keyHandler->is_Node.ln_Pri=16;

        keyHandler->is_Node.ln_Name=NameString;
        keyReqBlk->io_Data=(APTR) keyHandler;
        keyReqBlk->io_Command=KBD_ADDRESETHANDLER;
        DoIO((struct IORequest *)keyReqBlk);

        if (WaitForUser(MyDataStuff.MySignal))
        {
            if (argc) /* Check for CLI */
            {
                printf("System going down\n");
                printf("Cleaning up...\n");
                /* Show a delay, like cleanup... */
                Delay(20);
                printf("**Poof*\n");
            }

            /* We are done with our cleanup */
            keyReqBlk->io_Data=(APTR) keyHandler;
            keyReqBlk->io_Command=KBD_RESETHANDLERDONE;
            DoIO((struct IORequest *)keyReqBlk);
            /*
             * Note that since the above call
             * tells the system it is safe to reboot
             * and will cause the reboot if this
             * task was the last to say so, the call
             * never really returns... The system
             * just reboots...
             */
        }

        keyReqBlk->io_Data=(APTR) keyHandler;
        keyReqBlk->io_Command=KBD_REMRESETHANDLER;
        DoIO((struct IORequest *)keyReqBlk);

        CloseDevice((struct IORequest *)keyReqBlk);
    }
    DeleteExtIO((struct IORequest *)keyReqBlk);
}
FreeMem(keyHandler, sizeof(struct Interrupt));
}
DeletePort(keyPort);
}
FreeSignal(MySignal);
}
}

----- keyhandler.a -----
* Keyboard reset handler that signals the task in the structure...
*
*****
* Required includes...
*
    INCLUDE "exec/types.i"
    INCLUDE "exec/io.i"
    INCLUDE "devices/keyboard.i"
*
    xref    _AbsExecBase    ; We get this from outside...
    xref    _LVOSignal      ; We get this from outside...
*
*****
* Make the entry point external...

```

```

*
*      xdef      _ResetHandler
*
*****
* This is the structure that is passed in A1 in this example...
*
*      STRUCTURE      MyData,0
*      APTR            MyTask
*      ULONG           MySignal
*
*****
* This is the input handler
* The is_Data field is passed to you in a1.
*
* The handler gets called here...
*
_ResetHandler:  move.l  MySignal(a1),d0 ; Get signal to send
               move.l  MyTask(a1),a1   ; Get task
*
* Now signal the task...
*
               move.l  a6,-(sp)         ; Save the stack...
               move.l  _AbsExecBase,a6 ; Get ExecBase
               jsr     _LVOSignal(a6)   ; Send the signal
               move.l  (sp)+,a6         ; Restore A6
*
* Return to let other handlers execute.
*
               rts                      ; return from handler...
*
*****

```

## KBD\_READEVENT

Reading keyboard events is normally not done through direct access to the keyboard device. See the chapter “Input Device,” for the intimate linkage between that device and the keyboard device. This section is provided primarily to show you the component parts of a keyboard input event.

The keyboard matrix figure shown at the beginning of this chapter gives the code value that each key places into the `ie_Code` field of the input event for a key-down event. For a key-up event, a value of hexadecimal 80 is or’ed with the value shown above. Additionally, if either shift key is down, or if the key is one of those in the numeric keypad, the qualifier field of the keyboard input event will be filled in accordingly. In V34 and earlier versions of Kickstart, the keyboard device does not set the numeric qualifier for the keypad keys ‘(, ’), ‘/, ‘\*’ and ‘+’.

When you ask to read events from the keyboard, the call will not be satisfied until at least one keyboard event is available to be returned. The `io_Length` field must contain the number of bytes available in `io_Data` to insert events into. Thus, you should use a multiple of the number of bytes in an `InputEvent` (see example below).

### NOTE

The keyboard device can queue up several keystrokes without a task requesting a report of keyboard events. However, when the keyboard event buffer has been filled with no task interaction, additional keystrokes will be discarded.

# Example Keyboard Read-event Program

Shown below is an example keyboard.device read-event program:

```
/* The following example does not work very well in a system where
 * input.device is active since input.device also actively calls for
 * keyboard events via this call. For that reason, you will not get all of
 * the keyboard events. Neither will the input device; no one will be happy.
 *
 * Keyboard device read event example...
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <devices/inputevent.h>
#include <devices/keyboard.h>

#include <proto/exec.h>

#include <stdio.h>

int CXBRK(VOID) { return(0); }

VOID Display_Event(struct InputEvent *keyEvent)
{
    printf("Got key event: KeyCode: %2x  Qualifiers: %4x\n",
           keyEvent->ie_Code,
           keyEvent->ie_Qualifier);
}

VOID main(int argc, char *argv[])
{
    struct IOStdReq    *keyRequest;
    struct MsgPort     *keyPort;
    struct InputEvent  *keyEvent;
    SHORT              loop;

    if (keyPort=CreatePort(NULL,NULL))
    {
        if (keyRequest=(struct IOStdReq *)CreateExtIO(keyPort,
                                                       sizeof(struct IOStdReq)))
        {
            if (!OpenDevice("keyboard.device",NULL,
                           (struct IORequest *)keyRequest,NULL))
            {
                if (keyEvent=AllocMem(sizeof(struct InputEvent),MEMF_PUBLIC))
                {
                    for (loop=0;loop<4;loop++)
                    {
                        keyRequest->io_Command=KBD_READEVENT;
                        keyRequest->io_Data=(APTR)keyEvent;

                        /*
                         * We want 1 event, so we just set the
                         * length field to the size, in bytes
                         * of the event.  For multiple events,
                         * set this to a multiple of that size.
                         * The keyboard device NEVER fills partial
                         * events...
                         */
                        keyRequest->io_Length=sizeof(struct InputEvent);
                        DoIO((struct IORequest *)keyRequest);
                    }
                }
            }
        }
    }
}
```



```

        /* Check for CLI startup... */
        if (argc) Display_Event(keyEvent);
    }
    FreeMem(keyEvent, sizeof(struct InputEvent));
}
CloseDevice((struct IORequest *)keyRequest);
DeleteExtIO((struct IORequest *)keyRequest);
DeletePort(keyPort);
}
}

```

## Chapter 37

# Narrator Device

This chapter describes the narrator device which, together with the translator library, provides all the Amiga's text-to-speech functions. It provides an example that can be used to evaluate how parameters passed to the device can affect the output. This chapter also contains a non-technical explanation of how to effectively utilize the speech device. In addition, a more technical explanation is provided for those who are interested in how the speech is actually produced.

### Introduction

The speech system on the Amiga is divided into two subsystems:

- The *translator library*, which contains a function that translates English strings into phonetic strings.
- The *narrator device*, which, given a phonetic string, communicates with the audio device to actually produce human-like speech.

The subsystems may be used individually; you can pretranslate the English, and just use the narrator device to speak phonetic strings at run-time. Please see the Exec documentation for general information on how to use libraries and devices.

# The Translator Library

The translator library provides a single function, `Translate()`, that converts an English language string into a phonetic string. To use this function, you must first open the library. Setting a global variable, `TranslatorBase`, to the value returned from the call to `OpenLibrary()` enables the Amiga linker to correctly locate the translator library:

```
struct Library *TranslatorBase;

TranslatorBase = OpenLibrary("translator.library", REVISION);
if(TranslatorBase != NULL)
{
    /* use translator here -- library open */
}
```

## NOTE

Since translator is a disk-based library, the call to `OpenLibrary()` will work only if the `LIBS:` directory contains *translator.library*.

## USING THE TRANSLATE FUNCTION

Once the library is open, you can call the translate function:

```
#define BUFLen 500

APTR EnglStr;          /* pointer to sample input string */
LONG EnglLen;          /* input length */
UBYTE PhonBuffer[BUFLen]; /* place to put the translation */
LONG rtnCode;          /* return code from function */

EnglStr = "This is Amiga speaking."; /* a test string */
EnglLen = strlen(EnglStr);
rtnCode = Translate(EnglStr, EnglLen, (APTR)&PhonBuffer[0], BUFLen);
```

The input string will be translated into its phonetic equivalent and can be used to feed the narrator device. If you receive a non-zero return code, you haven't provided enough output buffer space to hold the entire translation. In this case, the `Translate()` function breaks the translation at the end of a word in the input stream and returns the position in the input stream at which the translation ended. You can use the output buffer, then call the `Translate()` function again, starting at this original ending position, to continue the translation where you left off. This method will sound smoothest if the ending position ends on sentence boundaries.

## NOTE

The value returned is *negative*. Therefore, you must use `-(rtnCode)` as the starting point for a new translation.

As with all other libraries of functions, if you have successfully opened the translator library for use, be sure to close it before your program exits by calling `CloseLibrary()`. If the system needs memory resources, it can then expunge the closed libraries to gain additional space. For more information on the translator, refer to the "Translator Library" chapter.

# The Narrator Device

The narrator device on the Amiga provides two basic functions:

- You can write to the device and ask it to speak a phonetically-encoded string in a specific manner—pitch, male/female, various speaking rates, and so on.
- You can read from the device. As it speaks, the device can generate mouth data for you and you can use this data to perform a graphics rendering of a face and mouth.

## OPENING THE NARRATOR DEVICE

To use the narrator device, you must first open the device. The narrator device is disk-resident. For the **OpenDevice()** call to succeed, the narrator device must be present in the directory currently assigned by AmigaDOS to the *DEVS:* directory.

To communicate with the narrator device, like any other device, you must pass an **IORequest** block to **OpenDevice()**. The request used by the narrator device for a write is a special format called a **narrator\_rb**. The request used for a read is also a special format, called a **mouth\_rb**. Both requests are described in the sections that follow. A sample **OpenDevice()** sequence for the narrator device follows. Notice that two request blocks are created, one for writing to the device and one for reading from it. For brevity, the error checking is left out of this code segment. It is, however, utilized in the sample program later on.

```
struct MsgPort write_port;
struct narrator_rb voice_io;

struct MsgPort read_port;
struct mouth_rb mouth_io;

OpenDevice("narrator.device", 0, &voice_io, 0);

write_port.mp_SigBit      = AllocSignal(-1);
write_port.mp_Node.ln_Name = "speech write";
write_port.mp_Node.ln_Type = NT_MSGPORT;
write_port.mp_Flags       = PA_SIGNAL;
write_port.mp_SigTask     = FindTask(NULL);

NewList(&write_port.mp_MsgList);
```

## CONTENTS OF THE WRITE REQUEST BLOCK

The standard I/O part of the request is set up much as with any other device, the beauty of Exec showing again.

```
voice_io.message.io_Command = CMD_WRITE;
voice_io.message.io_Offset  = 0;
voice_io.message.io_Data    = PhonBuffer;
voice_io.message.io_Length  = strlen(PhonBuffer);

voice_io.message.io_Message.mn_Node.ln_Type = NT_MESSAGE;
voice_io.message.io_Message.mn_Length      = sizeof(voice_io);
voice_io.message.io_Message.mn_ReplyPort   = &write_port;
```

You can control several characteristics of the speech, as indicated in the narrator request block structure shown below.

```
struct narrator_rb
{
    struct IOStdReq message; /* Standard IOReq */
    UWORD rate; /* Speaking rate (words/minute) */
    UWORD pitch; /* Baseline pitch in Hertz */
    UWORD mode; /* Pitch mode */
    UWORD sex; /* Sex of voice */
    UBYTE *ch_masks; /* Pointer to allocation mask */
    UWORD nm_masks; /* Number of masks */
    UWORD volume; /* Volume. 0 (off) thru 64 */
    UWORD sampfreq; /* Audio sampling freq */
    UBYTE mouths; /* If non-zero, generate mouths */
    UBYTE chanmask; /* Which ch mask used (internal) */
    UBYTE numchan; /* Num ch masks used (internal) */
    UBYTE pad; /* For alignment, system use only */
};
```

where

**message.io\_Data**

points to the phonetic input string, which must be terminated with a Q#U.

**message.io\_Length**

is the length of the string (including the Q#U and the terminating NULL). The narrator will parse the input string until a Q#U or NULL is encountered or **io\_Length** is reached, whichever comes first.

**rate** is the speed in words per minute that you wish it to speak (range: 40 - 400).

**pitch**

is the baseline pitch. If you are using an expressive voice rather than a monotone, the pitch will vary above and below this baseline pitch (range: 65 - 320).

**mode**

determines whether you have a monotonic (0) or expressive (1) voice.

**sex** determines if the voice is male (0) or female (1).

**volume and sampfreq**

are passed on to the audio device; **sampfreq** affects both pitch and rate proportionally. Raising it above the default also causes the narrator to use proportionally more CPU time while speaking.

**mouths**

is set to nonzero before starting a write if you want to read mouths using the read command while the system is speaking.

**chanmask, numchan, pad**

are used by narrator for audio allocation keys and structure alignment.

Here is a code fragment which fills in the speech control fields of the **narrator\_rb** structure:

```

struct narrator_rb voice_io;
BYTE audio_chan[] = {3, 5, 10, 12};

voice_io.ch_masks = audio_chan;
voice_io.nm_masks = sizeof(audio_chan);
voice_io.mouths = 1;

/* These are THE parameters affecting speech. */
voice_io.rate = DEFRATE;
voice_io.pitch = DEFPITCH;
voice_io.mode = DEFMODE;
voice_io.sex = DEFSEX;
voice_io.volume = DEFVOL;
voice_io.sampfreq = DEFFREQ;

```

The system default values are shown in the files *devices/narrator.h* and *devices/narrator.i*. When you call **OpenDevice()**, the system initializes the request block to the default values. If you want other than the defaults, you must change them *after* the device is open.

## CONTENTS OF THE READ REQUEST

Read requests are only needed if the programmer wants to get mouth widths and heights as the narrator is speaking.

The **mouth\_rb** data structure follows. Notice that it is an extended form of the **narrator\_rb** structure.

```

struct mouth_rb
{
    struct narrator_rb voice;    /* Speech IORB */
    UBYTE width;                /* Width (returned value) */
    UBYTE height;               /* Height (returned value) */
    UBYTE shape;                /* Internal use, do not modify */
    UBYTE pad;                  /* For alignment */
};

```

The fields **width** and **height** will, on completion of a read-request, contain an integer value proportional to the mouth width and height that are appropriate to the phoneme currently being spoken. When you send a read request, the system does not return a response until one of two things happens. Either a different mouth size is available (this prevents you from drawing and redrawing the same shape or having to check whether or not it is the same) or the speaking has completed. You must check the error return field when the read request block is returned to determine if the request block contains a new mouth shape or simply is returning status of **ND\_NoWrite** (no write in progress, all speech ended for this request).

## PERFORMING A WRITE AND A READ

You normally perform a write command by using the functions **BeginIO()** or **SendIO()** to transmit the request block to the narrator device. This allows the narrator's task to begin the I/O, while your task is free to do something else. The something else may be issuing a series of read commands to the device to determine mouth shapes and drawing them on-screen. The following sample set of function calls implements both the write and read commands in a single loop. Again, error checking is deferred to the sample program.

### NOTE

This code segment handles the write with a **SendIO()**, then uses a loop on **DoIO()** to handle the reads. The sample program uses **SendIO()** for both the write and read, allowing the use of only one **Wait()**, a useful technique when handling messages from multiple sources.

The processing loop is as follows:

```
struct narrator_rb *writeNarrator;
struct mouth_rb *readNarrator;

/* tell it how many characters the translate function returned */
writeNarrator->message.io_Length = strlen(outputstring);

/* tell it where to find the string to speak */
writeNarrator->message.io_Data = outputstring;

/* return immediately, run tasks concurrently */
SendIO(writeNarrator);

readNarrator->voice.message.io_Error = 0;
while(readNarrator->voice.message.io_Error != ND_NoWrite)
{
    DoIO(readNarrator);
    /* Put task to sleep waiting for return of the
     * message block. It will have either a different
     * mouth shape or the the error field will show
     * that no write is in progress.
     */
    DrawMouth(readNarrator->width, readNarrator->height);
    /* An example DrawMouth is part of the sample program, below. */
}

WaitIO(writeNarrator); /* remove the write message from the
                        * write_port so that it can be reused */
```

The loop continues to send read requests to the narrator device until the speech output has ended. `DoIO()` automatically removes the read request block from the `read_port` for reuse. `SendIO()` is used to transmit the write request. When it completes, the write request will be appended to the `write_port`, and must be removed before it can be reused.

As with all other devices, if you have successfully opened the narrator device for use, be sure to close it before your program exits.

## Example Program

The example program listed at the end of this chapter shows how to use the translator and narrator, with or without the reading of mouth sizes. Run the program from the CLI. It takes a quoted string as its argument, translates it into phonemes and says them using the narrator. Use the `-h` flag to get a list of options the program supports.

Try experimenting with the narrator device by using values other than the defaults in the narrator I/O request block, changing them before the write command is sent to the device.

## How to Write Phonetically for Narrator

This section describes in detail the procedure used to specify phonetic strings to the *narrator* speech synthesizer. No previous experience with phonetics is required. The only thing you may need is a good pronouncing dictionary for those times when you doubt your own ears. You do not have to learn a foreign language or computer language. You are just going to learn how to write down the English that comes out of your own mouth. In writing phonetically you do not have to know how a word is spelled, just how it is said.

Narrator works on utterances at the sentence level. Even if you want to say only one word, narrator will treat it as a complete sentence. Therefore, narrator wants one of two punctuation marks to appear at the end of every sentence—a period (.) or a question mark (?). The period is used for almost all utterances and will cause a final fall in pitch to occur at the end of a sentence. The question mark is used at the end of yes/no questions only, and results in a final rise in pitch. For example, the question, *Do you enjoy using your Amiga?* would take a question mark at the end because the answer to the question is either yes or no. The question, *What is your favorite color?* would not take a question mark and should be followed by a period. If no punctuation appears at the end of a string, narrator will append a dash to it which will cause a short pause. Narrator recognizes other punctuation marks as well, but these are left for later discussion.

## PHONETIC SPELLING

Utterances are usually written phonetically using an alphabet of symbols known as I.P.A. (for “International Phonetic Alphabet”). This alphabet is found at the front of most good dictionaries. The symbols can be hard to learn and are not available on computer keyboards, so the Advanced Research Projects Agency (ARPA) came up with *Arpabet*, a way of representing each symbol using one or two upper-case letters. Narrator uses an expanded version of *Arpabet* to specify phonetic sounds.

A phonetic sound, or *phoneme*, is a basic speech sound, almost a speech atom. Working backwards, sentences can be broken into words, words into syllables, and syllables into phonemes. The word *cat* has three letters and (coincidentally) three phonemes. Looking at the table of phonemes we find the three sounds that make up the word *cat*. They are the phonemes K, AE, and T, written as KAET. The word *cent* translates as S, EH, N and T, or SEHNT. Notice that both words begin with a letter *c* but because the *c* says *k* in *cat* we use the phoneme K. In *cent* the letter *c* says *s* so we use the phoneme S. You may also have noticed that there is no C phoneme. These examples show that a word rarely sounds like it looks in English and introduces you to a very important concept of phonetic spelling: spell it like it sounds, not like it looks.

## CHOOSING THE RIGHT VOWEL

Phonemes, like letters, are divided into the two categories of vowels and consonants. Loosely defined, a vowel is a continuous sound made with the vocal cords vibrating and air exiting the mouth (as opposed to the nose). All vowels use a two-letter code. A consonant is any other sound, such as those made by rushing air (like S or TH), or by interruptions in air flow by the lips or tongue (B or T). Consonants use a one or two-letter code.

In English we write with only five vowels: a, e, i, o and u. It would be easy if we only *said* five vowels. Unfortunately, we say more than 15 vowels. Narrator provides for most of them. You choose the proper vowel by listening. Say the word aloud, perhaps extending the vowel sound you want to hear. Compare the sound you are making to the sounds made by the vowels in the example of the phoneme list. For example, the *a* in *apple* sounds the same as the *a* in *cat*, not like the *a* in *Amiga*, *talk*, or *made*. Notice also that some of the example words in the list do not even use any of the same letters contained in the phoneme code; for example, AA as in *bottle*.

Vowels are divided into two groups: those that maintain the same sound throughout their durations and those that change their sound. The ones that change are called *diphthongs*. Some of us were taught the terms *long* and *short* to describe vowel sounds. Diphthongs fall into the long category, but these two terms are inadequate to fully differentiate between vowels and should be avoided. The diphthongs are the last six vowels listed in the table. Say the word *made* out loud very slowly. Notice how the *a* starts out like the *e* in *bet* but ends up like the *e* in *beet*. The *a* therefore is a diphthong in this word and we would use EY to represent it. Some speech synthesis systems require you to specify the changing sounds in diphthongs as separate elements, but narrator takes care of the assembly of diphthongal sounds for you.



## CHOOSING THE RIGHT CONSONANT

Consonants are divided into many categories by phoneticians, but we need not concern ourselves with most of them. Picking the correct consonant is very easy if you pay attention to just two categories: voiced and unvoiced. A voiced consonant is made with the vocal cords vibrating, and an unvoiced one is made when the vocal cords are silent. Sometimes English uses the same letter combinations to represent both. Compare the *th* in *thin* and in *then*. Notice that the first is made with air rushing between the tongue and upper teeth. In the second, the vocal cords are vibrating also. The voiced *th* phoneme is DH, the unvoiced is TH. Therefore, *thin* is spelled with the phonemes TH, IH, N or THIHN, and *then* is spelled DH, EH, N or DHEHN. A sound that is particularly subject to mistakes is voiced and unvoiced *s* spelled Z or S. To put it clearly, *bats* ends in S, *suds* ends in Z. What kind of *s* does *closet* have? How about *close*? Say all of these words aloud to find out. Actually *close* changes its meaning when the *s* is voiced or unvoiced: *I love to be close to you.* versus *What time do you close?*

Another sound that causes some confusion is the *r* sound. There are two different r-like phonemes in the narrator alphabet: R under the consonants and ER under the vowels. Which one do you use? Use ER if the *r* sound is the vowel sound in the syllable. Words that take ER are *absurd*, *computer* and *flirt*. Use R if the *r* sound precedes or follows another vowel sound in that syllable, such as in *car*, *write*, or *craft*. The word *rooster* uses both kinds of *r*. Can you tell which is which?

## CONTRACTIONS AND SPECIAL SYMBOLS

There are several phoneme combinations that appear very often in English words. Some of these are caused by our laziness in pronunciation. Take the word *connector* for example. The *o* in the first syllable is almost swallowed out of existence. You would not use the AA phoneme; you would use the AX instead. It is because of this *relaxation* of vowels that we find ourselves using AX and IX very often. Since this relaxation frequently occurs before *l*, *m* and *n*, narrator has a shortcut for typing these combinations. Instead of *personal* being spelled PERSIXNAXL, we can spell it PERSINUL, making it a little more readable. *Anomaly* goes from AXNAAMAXLIY to UNAAMULIY, and KAAMBIXNEYSHIXN becomes KAAMBINNEYSHIN for *combination*. It may be hard to decide whether to use the AX or IX brand of relaxed vowel. The only way to find out is to try both and see which sounds best.

Other special symbols are used internally by narrator. Sometimes they are inserted into or substituted for part of your input sentence. You can type them in directly if you wish. The most useful is probably the Q or glottal stop; an interruption of air flow in the glottis. The word *Atlantic* has one between the *t* and the *l*. Narrator knows there should be a glottal stop there and saves you the trouble of typing it. But narrator is only close to perfect, so sometimes a word or word pair might slip by that would have sounded better with a Q stuck in someplace.

## STRESS AND INTONATION

It is not enough to tell narrator what you want said. For the best results you must also tell narrator *how* you want it said. In this way you can alter a sentence's meaning, stress important words, and specify the proper accents in polysyllabic words. These things improve the naturalness and thus the intelligibility of narrator's spoken output.

Stress and intonation are specified by the single digits 1-9 following a vowel phoneme code. Stress and intonation are two different things but are specified by a single number. Stress is, among other things, the elongation of a syllable. A syllable is either stressed or not, so the *presence* of a number after the vowel in a syllable indicates stress on that syllable. The *value* of the number indicates the intonation. These numbers are referred to here as *stress marks* but keep in mind that they also affect intonation. Intonation here means the pitch pattern or contour of an utterance. The higher the stress mark, the higher the potential for an accent in pitch (a rise and fall). A sentence's basic contour is comprised of a quickly rising pitch gesture up to the first stressed syllable in the sentence, followed by a slowly declining tone throughout the sentence, and finally a quick fall to a low pitch on the last syllable. The presence of additional stressed syllables causes the pitch to break its slow, declining pattern with rises and falls around each stressed syllable. Narrator uses a very sophisticated procedure to generate natural pitch contours based on how you mark the stressed syllables.

## HOW AND WHERE TO PUT THE STRESS MARKS

The stress marks go immediately to the right of vowel phoneme codes. The word *cat* has its stress marked after the AE so we get KAE5T or KAE9T. You generally have no choice about the location of a number; there is definitely a right and wrong location. Either a number should go after a vowel or it should not. Narrator will not flag an error if you forget to put a stress mark in or if you place one on the wrong vowel. It will only tell you if a stress mark is in the wrong place, such as after a consonant.

The rules for placing stress marks are as follows:

- Always place a stress mark in a *content* word. A content word is one that contains some meaning. Nouns, verbs, and adjectives are all content words. *Boat*, *huge*, *tonsils* and *hypertensive* are all content words; they tell the listener what you are talking about. Words like *but*, *the*, *if* and *is* are not content words. They do not convey any real-world meaning at all but are required to make the sentence function. Thus, they are given the name *function words*.
- Always place a stress mark on the accented syllable(s) of polysyllabic words, whether they are content or function words. A polysyllabic word is any word of more than one syllable. *Commodore* has its stress (or accent as it is often called) on the first syllable and would be spelled KAA5MAXDOHR. *Computer* is stressed on the second syllable, producing KUMPYUW5TER.

If you are in doubt about which syllable gets the stress, look the word up in a dictionary and you will find an accent mark over the stressed syllable. If more than one syllable in a word receives stress, they usually are not of equal value. These are referred to as primary and secondary stresses. The word *understand* has its first and last syllables stressed, with *stand* getting primary stress and *un* secondary, which produces AH1NDERSTAE4ND. Syllables with secondary stress should be marked with a value of only 1 or 2.

Compound words (words with more than one root) such as *baseball*, *software*, *lunchwagon*, and *houseboat* can be written as one word but should be thought of as separate words when marking stress. Thus, *lunchwagon* would be spelled LAH5NCHWAE2GIN. Notice that *lunch* got a higher stress mark than *wagon*. This is common in compound words; the first word usually receives the primary stress.

## WHAT STRESS VALUE DO I USE?

If you get the spelling and stress mark positions correct, you are 95 percent of the way to a good sounding sentence. The next thing to do is decide on the stress mark values. They can be roughly related to parts of speech, and you can use table shown below as a guide to assigning values.

Table 37-1: Recommended Stress Values

| Part of Speech   | Stress Value |               |
|------------------|--------------|---------------|
| Exclamations     | 9            |               |
| Adverbs          | 7            |               |
| Quantifiers      | 7            |               |
| Nouns            | 5            |               |
| Adjectives       | 5            |               |
| Verbs            | 4            |               |
| Pronouns         | 3            |               |
| Secondary stress | 1            | (sometimes 2) |
| Articles         | 0            | (no stress)   |
| Prepositions     | 0            |               |
| Conjunctions     | 0            |               |

The above values merely suggest a range. If you want attention directed to a certain word, raise its value. If you want to downplay a word, lower it. Sometimes even a function word can be the focus of a sentence. It is quite conceivable that the word *to* in the sentence *Please deliver this to Mr. Smith.* could receive a stress mark of 9. This would add focus to the word *to* indicating that the item should be delivered to Mr. Smith in person.

## PUNCTUATION

In addition to the period or question mark that is required at the end of a sentence, narrator recognizes several other punctuation marks: dashes, commas, and parentheses. The comma goes where you would normally put a comma in an English sentence. It causes narrator to pause with a slightly rising pitch, indicating that there is more to come. The use of additional commas—that is, more than would be required for written English—is often helpful. They serve to set clauses off from one another. There is a tendency for a listener to lose track of the meaning of a sentence if the words run together. Read your sentence aloud while pretending to be a newscaster. The locations for additional commas should leap out at you.

The dash serves almost the same purpose as the comma, except that the dash does not cause the pitch to rise so severely. A rule of thumb is: Use dashes to divide phrases, commas to divide clauses. For a definition of these terms, consult a high school English textbook.

Parentheses provide additional information to narrator's intonation function. They should be put around noun phrases of two or more content words. This means that the noun phrase, *a giant yacht* should be surrounded with parentheses because it contains two content words, *giant* and *yacht*. The phrase *my friend* should not have parentheses around it because it contains only one content word. Noun phrases can get pretty big, like *the silliest guy I ever saw* or *a big basket of fruit and nuts*. The parentheses really are most effective around these

large phrases; the smaller ones can sometimes go without. The effect of parentheses is subtle, and in some sentences you might not even notice their presence. In sentences of great length, however, they help provide for a very natural contour.

## HINTS FOR INTELLIGIBILITY

There are a few tricks you can use to improve the intelligibility of a sentence. Often, a polysyllabic word is more recognizable than a monosyllabic word. For instance, instead of saying *huge*, say *enormous*. The longer version contains information in every syllable, thus giving the listener three times the chance to hear it correctly. This can be taken to extremes, so try not to say things like "This program has a plethora of insects in it."

Another good practice is to keep sentences to an optimal length. Writing for reading and writing for speaking are two different things. Try not to write a sentence that cannot be easily spoken in one breath. Such a sentence tends to give the impression that the speaker has an infinite lung capacity. Try to keep sentences confined to one main idea. A run-on sentence tends to lose its meaning after a while.

New terms should be highly stressed the first time they are heard. If you are doing a tutorial or something similar, stress a new term at its first occurrence. All subsequent occurrences of that term need not be stressed as highly because it is now "old news."

The above techniques are but a few ways to enhance the performance of narrator. You will probably find some of your own. Have fun.

## EXAMPLE OF ENGLISH AND PHONETIC TEXTS

Cardiomyopathy. I had never heard of it before, but there it was listed as the form of heart disease that felled not one or two but all three of the artificial heart recipients. A little research produced some interesting results. According to an article in the Nov. 8, 1984, *New England Journal of Medicine*, cigarette smoking causes this lethal disease that weakens the heart's pumping power. While the exact mechanism is not clear, Dr. Arthur J. Hartz speculated that nicotine or carbon monoxide in the smoke somehow poisons the heart and leads to heart failure.

KAAIRDYOWMAYAA5PAXTHIY. AY /HAED NEHIVER HER4D AXV IHT BIXFOH5R, BAHT DHEH5R IHT WAHZ - LIH4STIXD AEZ (DHAX FOH5RM AXV /HAA5RT DIHZIY5Z) DHAET FEH4LD (NAAT WAH5N OHR TUW5) - BAHT (AO7L THRIY5 AXV DHAX AA5RTAXFIHSHUL /HAA5RT RIXSIH5PIYINTS). (AH LIH5TUL RIXSER5CH) PROHDUW5ST (SAHM IH5NTRIHSIHNX RIXZAH5LTS). AHKOH5RDIHNX TUW (AEN AA5RTIHKUL IHN DHAX NOWVEH5MBER EY2TH NAY5NTIYNEYTIYFOH1R NUW IY5NXGLIND JER5NUL AXV MEH5DIXSIN), (SIH5GEREHT SMOW5KIH5NX) KAO4ZIHZ (DHIHS LIY5THUL DIHZIY5Z) DHAET WIY4KINZ (DHAX /HAA5RTS PAH4MPIHNX PAW2ER). WAYL (DHIY IHGZAE5KT MEH5KINIXZUM) IHZ NAAT KLIY5R, DAA5KTER AA5RTER JEY2 /HAA5RTS SPEH5KYULEYTIHD DHAET NIH5KAXTIYN, OHR KAA5RBIN MUNAA5KSAYD IHN DHAX SMOW5K - SAH5M/HAW1 POY4ZINZ DHAX /HAA5RT, AEND LIY4DZ TUW (/HAA5RT FEY5LYER).

## CONCLUDING REMARKS

This guide should get you off to a good start in phonetic writing for narrator. The only way to get really proficient is to practice. Many people become good at it in as little as one day. Others make continual mistakes because they find it hard to let go of the rules of English spelling, so trust your ears.

## The More Technical Explanation

The narrator speech synthesis system is a computer model of the human speech production process. It attempts to produce accurately spoken utterances of any English sentence, given only a phonetic representation as input. Another program in the system, translator, derives the required phonetic spelling from English text. Timing and pitch contour are produced automatically by the synthesizer software.

In humans, the physical act of producing speech sounds begins in the lungs. To create a voiced sound, the lungs force air through the vocal folds (sometimes called the vocal cords), which are held under tension and which periodically interrupt the flow of air, thus creating a buzz-like sound. This buzz, which has a spectrum rich in harmonics, then passes through the vocal tract and out the lips, which alters its spectrum drastically. This is because the vocal tract acts as a frequency filter, selectively reinforcing some harmonics and suppressing others.

It is this filtering that gives a speech sound its identity. The amplitude versus frequency graph of the filtering action is called the *vocal tract transfer function*. Changing the shape of the throat, tongue, and mouth retunes the filter system to accentuate different frequencies.

The sound travels as a pressure wave through the air, and it causes the listener's eardrum to vibrate. The ear and brain of the listener decode the incoming frequency pattern. From this the listener can subconsciously make a judgment about what physical actions were performed by the speaker to make the sound. Thus the speech chain is completed, the speaker having encoded his physical actions on a buzz via selective filtering and the listener having turned the sound into guesses about physical actions by frequency decoding.

Now that we know how humans do it, how does the Amiga do it? It turns out that the vocal tract transfer function is not random, but tends to accentuate energy in narrow regions called *formants*. The formant positions move fairly smoothly as we speak, and it is the formant frequencies to which our ears are sensitive. So, luckily, we do not have to model throat, tongue, teeth and lips with our computer, we can imitate formant action.

A good representation of speech requires up to five formants, but only the lowest three are required for intelligibility. We begin with an oscillator that produces a waveform similar to that which is produced by the vocal folds, and we pass it through a series of resonators, each tuned to a different formant frequency. By controlling the volume and pitch of the oscillator and the frequencies of the resonators, we can produce highly intelligible and natural-sounding speech. Of course the better the model, the better the speech; but more importantly, experience has shown that the better the *control* of the model's parameters, the better the speech.

Oscillators, volume controls and resonators can all be simulated mathematically in software, and it is by this method that the narrator system operates. The input phonetic string is converted into a series of target values for the various parameters illustrated. A system of rules then operates on the string to determine things such as the duration of each phoneme and the pitch contour. Transitions between target values are created and smoothed to produce natural continuous changes from one sound to the next.

New values are computed for each parameter for every 8 milliseconds of speech, which produces about 120 acoustic changes per second. These values drive a mathematical model of the speech synthesizer. The accuracy of this simulation is quite good. Human speech has more formants than the narrator model, but they are low in energy content.

The human speech production mechanism is a complex and wonderful thing. The more we learn about it, the better we can make our computer simulations. Meanwhile, we can use synthetic speech as yet another computer output device to enhance the man/machine dialogue.

## Table of Phonemes

Table 12-2 lists all the available phonemes.

Table 37-2: Phonemes

| Vowels  |                  |         |                 |
|---------|------------------|---------|-----------------|
| Phoneme | Example          | Phoneme | Example         |
| IY      | beet, eat        | IH      | bit, in         |
| EH      | bet, end         | AE      | bat, ad         |
| AA      | bottle, on       | AH      | but, up         |
| AO      | ball, awl        | UH      | book, soot      |
| ER      | bird, early      | OH      | border          |
| AX*     | about, calibrate | IX*     | solid, infinite |

\*AX and IX should never be used in stressed syllables.

| Diphthongs |           |         |               |
|------------|-----------|---------|---------------|
| Phoneme    | Example   | Phoneme | Example       |
| EY         | bay, aid  | AY      | bide, I       |
| OY         | boy, oil  | AW      | bound, owl    |
| OW         | boat, own | UW      | brew, boolean |

### Consonants

| Phoneme | Example   | Phoneme | Example        |
|---------|-----------|---------|----------------|
| R       | red       | L       | long           |
| W       | wag       | Y       | yellow, beauty |
| M       | men       | N       | no             |
| NX      | sing      | SH      | shy            |
| S       | soon      | TH      | thin           |
| F       | fed       | ZH      | pleasure       |
| Z       | has, zoo  | DH      | then           |
| V       | very      | WH      | when           |
| CH      | check     | J       | judge          |
| /H      | hole      | /C      | loch           |
| B       | but       | P       | put            |
| D       | dog       | T       | toy            |
| K       | Commodore | G       | guest          |

### Special Symbols

| Phoneme | Example                |
|---------|------------------------|
| DX      | pity (tongue flap)     |
| Q       | kitt-en (glottal stop) |
| QX      | pause (silent vowel)   |

### Contractions

(see text)

|    |   |     |
|----|---|-----|
| UL | = | AXL |
| IL | = | IXL |
| UM | = | AXM |
| IM | = | IXM |
| UN | = | AXN |
| IN | = | IXN |

### Digits and Punctuation

|            |  |
|------------|--|
| Digits 1-9 | Syllabic stress, ranging from secondary through emphatic |
| .          | Period—sentence final character                          |
| ?          | Question mark—sentence final character                   |
| -          | Dash—phrase delimiter                                    |
| ,          | Comma—clause delimiter                                   |
| ()         | Parentheses—noun phrase delimiters (see text)            |

```

/* A Simple program to show speech on the amiga.
 * If you do not define the flag FACE_ON,
 * ALL code involved with reading mouth shapes is excluded.
 * If you do not define the flag PARSE, ALL code involved
 * with parsing the command line is excluded, and defaults
 * are used. Code by Dave Lucas.
 *
 * Lattice use lc -bl -cfast -v -y. Link with lc.lib and amiga.lib.
 *
 *   FACE_ON      PARSE      `Lines'o'Source      `Executeable size
 *   0             0         185                   8.8K
 *   0             1         360                  10.7K
 *   1             0         355                  10.4K
 *   1             1         540                  12.3K
 */

#define FACE_ON
#define PARSE

#include <exec/types.h>
#include <exec/io.h>
#include <intuition/intuition.h>
#include <devices/narrator.h>
#include <libraries/translator.h>
#include <libraries/dos.h>

#include <proto/all.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* The length of the English and phonetic buffers. */
#define ENGLISH_TEXT_LEN 25 /* Just long enough for default */
/* Input line is restricted to nnn chars (AmigaDOS), but */
#define PHONEME_TEXT_LEN 512 /* Phonemes are longer than english. */

void DrawFace(void);
void DrawMouth(USHORT half_w, USHORT half_h);
extern LONG ParseArgs (int argc, char **argv, UBYTE *show_face, UWORD *sex,
                      UWORD *inflect, UWORD *samp, UWORD *pitch, UWORD *speed,
                      UWORD *vol, char **text);
extern void BadOpt(char *option);
void CleanUp(void);

/* Which audio channels to use. */
BYTE audio_chan[] = {3, 5, 10, 12};

/* Pointer to translator library vectors. */
struct Library *TranslatorBase = NULL;

struct MsgPort write_port;
struct narrator_rb voice_io;

/* Indicative of the Open() return. */
UBYTE NarratorOpenError = -1;
/* Indicative of a Translations success. */
UBYTE TranslatorError = 0;

UBYTE EnglBuffer[ENGLISH_TEXT_LEN] = "This is amiga speaking.";
UBYTE PhonBuffer[PHONEME_TEXT_LEN] = "DHIHS IHZ AHMIY3GAH SPIY4KIHNX.";

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

#ifdef FACE_ON
/* Pen numbers to draw gadget borders/images/text with. */
#define REDP 3 /* Color in register 3 once was red. */
#define BLKP 2 /* Color in register 2 was black. */
#define WHTP 1 /* Color in register 1 was white. */
#define BLUP 0 /* Color in register 0 was blue. */

```



```

/* These are used for face rendering. */
#define FACE_ET      12 /* top of eyes */
#define FACE_EB      22 /* bottom of eyes */
#define FACE_LIP_T    23 /* top of mouth clear area */
#define FACE_MID_H    42 /* the middle of the mouth, heightwise */
#define FACE_H        80 /* overall window height */
#define FACE_ELL      24 /* left eye left side */
#define FACE_ELR      40 /* left eye right side */
#define FACE_INIT_W   32 /* left point for initial mouth line */
#define FACE_MID_W    56 /* the middle of the mouth, widthwise */
#define FACE_ERL      72 /* right eye left side */
#define FACE_ERR      88 /* right eye right side */
#define FACE_W        120 /* overall window width */

struct NewWindow NewFaceWindow =
{
    0, 11,                /* Start LeftEdge, TopEdge. */
    FACE_W, FACE_H,       /* Width, Height. */
    -1, -1,              /* DetailPen, BlockPen. */
    0,                   /* IDCMP FLAGS. */
    WINDOWDRAG | WINDOWDEPTH | GIMMEZEROZERO, /* Flags. */
    NULL,                /* No pointer to FirstGadget. */
    NULL,                /* No pointer to first CheckMark. */
    NULL,                /* No Title. */
    NULL,                /* No pointer to Screen. */
    NULL,                /* No pointer to BitMap. */
    FACE_W, FACE_H,       /* Minimum sizeable to (NA-not sizeable). */
    FACE_W, FACE_H,       /* Maximum sizeable to (NA-not sizeable). */
    WBENCHSCREEN          /* Type of screen window appears in. */
};

struct Window *FaceWindow = NULL;
struct IntuiMessage *MyIntuiMessage;

struct MsgPort read_port;
struct mouth_rb mouth_io;
#endif /* FACE_ON defined */

/** Start of code *****/
VOID main(int argc, char **argv)
{
    ULONG Signals;        /* Wait() tells me which to look at. */
    UBYTE *pp_string;
    UWORD rate, pitch, mode, sex, volume, sampfreq;
    UBYTE show_face = 0;
    /* Let Cleanup() know these signals not allocated yet. */
    write_port.mp_SigBit = -1;
#ifdef FACE_ON
    read_port.mp_SigBit = -1;
#endif /* FACE_ON defined */

    pp_string = &EnglBuffer[0];

    rate = DEFRATE;
    pitch = DEFPITCH;
    mode = DEFMODE;
    sex = DEFSEX;
    volume = DEFVOL;
    sampfreq = DEFFREQ;

#ifdef PARSE
    ParseArgs(argc, argv, &show_face, &sex, &mode, &sampfreq, &pitch, &rate,
        &volume, (APTR)&pp_string);
#endif /* PARSE defined */

    /* Open those libraries that the program uses directly. */
    if ((IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 33)) == NULL)
    {
        fprintf(stderr, "Can't open the intuition library0);
        Cleanup();
        exit(RETURN_WARN);
    }

```

```

}

if ((GfxBase = (struct GfxBase *)
OpenLibrary("graphics.library", 33)) == NULL)
{
    fprintf(stderr, "Can't open the graphics library0);
    CleanUp();
    exit(RETURN_WARN);
}

if ((TranslatorBase = (struct Library *)
OpenLibrary("translator.library", 0L)) == NULL)
{
    fprintf(stderr, "Can't open the translator library0);
    CleanUp();
    exit(RETURN_WARN);
}

if ((TranslatorError = Translate((UBYTE *)pp_string,
strlen(pp_string), (UBYTE *)PhonBuffer, PHONEME_TEXT_LEN)) != 0)
{
    fprintf(stderr, "Translator won't. (%lx)0,TranslatorError);
}

if ((NarratorOpenError =
OpenDevice("narrator.device",0, (struct IORequest *) &voice_io, 0))!=0)
{
    fprintf(stderr, "Can't open the narrator device0);
    CleanUp();
    exit(RETURN_WARN);
}

/* Set up the write port, allocate the signal */
/* and the message. */
if ((write_port.mp_SigBit = AllocSignal(-1)) == -1)
{
    fprintf(stderr, "Couldn't Allocate write Signal bit0);
    CleanUp();
    exit(RETURN_WARN);
}

write_port.mp_Node.ln_Name = "speech_write";
write_port.mp_Node.ln_Type = NT_MSGPORT;
write_port.mp_Flags = PA_SIGNAL;
write_port.mp_SigTask = (struct Task *)FindTask(NULL);

NewList(&write_port.mp_MsgList);

/* Set up the write channel information. */
voice_io.message.io_Command = CMD_WRITE;
voice_io.message.io_Offset = 0;
voice_io.message.io_Data = (APTR)PhonBuffer;
voice_io.message.io_Length = strlen(PhonBuffer);

voice_io.message.io_Message.mn_Node.ln_Type = NT_MSGPORT;
voice_io.message.io_Message.mn_Length = sizeof(voice_io);
voice_io.message.io_Message.mn_ReplyPort = &write_port;
voice_io.message.io_Unit = 0;

voice_io.ch_masks = (UBYTE *)audio_chan;
voice_io.nm_masks = sizeof(audio_chan);
voice_io.mouths = show_face;
voice_io.rate = rate;
voice_io.pitch = pitch;
voice_io.mode = mode;
voice_io.sex = sex;
voice_io.volume = volume;
voice_io.sampfreq = sampfreq;

#ifdef FACE_ON
if (show_face)
{
    /* Set up the read port, allocate the signal */

```

```

/* and the message. */
read_port.mp_Node.In_Type = NT_MSGPORT;
read_port.mp_Flags = PA_SIGNAL;
if ((read_port.mp_SigBit = AllocSignal(-1)) == -1)
{
    fprintf(stderr, "Couldn't Allocate read Signal bit0);
    CleanUp();
    exit(RETURN_WARN);
}
read_port.mp_SigTask = (struct Task *)FindTask(NULL);
NewList(&read_port.mp_MsgList);

/* Set up the read channel information. */
mouth_io.voice = voice_io;
mouth_io.width = 0;
mouth_io.height = 0;
mouth_io.voice.message.io_Message.mn_ReplyPort = &read_port;
mouth_io.voice.message.io_Command = CMD_READ;
mouth_io.voice.message.io_Error = 0;

if ((FaceWindow = OpenWindow(&NewFaceWindow))
    == NULL)
{
    fprintf(stderr, "Couldn't open the face window.0);
    CleanUp();
    exit(RETURN_WARN);
}
DrawFace();
}
#endif /* FACE_ON defined */

SendIO((struct IORequest *) &voice_io );

#ifdef FACE_ON
    if (show_face)
    {
        SendIO((struct IORequest *) &mouth_io );
    }
#endif /* FACE_ON defined */

/* Wait() lets the rest of the system run while */
/* this program sleeps. */
for (;;)
{
#ifdef FACE_ON
    if (show_face)
        Signals = Wait(
            (1L << voice_io.message.io_Message.mn_ReplyPort->mp_SigBit)
            | (1L << mouth_io.voice.message.io_Message.mn_ReplyPort->mp_SigBit));
    else
#endif /* FACE_ON defined */
        Signals = Wait(
            (1L << voice_io.message.io_Message.mn_ReplyPort->mp_SigBit));

/* A voice SendIO() (Write) has completed. */
if (Signals & (1L <<
    voice_io.message.io_Message.mn_ReplyPort->mp_SigBit))
{
    /* Was it Successful? filter out the abort error. */
    if (voice_io.message.io_Error == -2)
        voice_io.message.io_Error = 0;
    if (voice_io.message.io_Error != 0)
    {
        fprintf(stderr, "Narrator won't. (%ld)0,
            voice_io.message.io_Error);
        voice_io.message.io_Error = 0;
    }
    CleanUp();
    exit(RETURN_OK);
}

#endif /* FACE_ON

```

```

/* A mouth SendIO() (Read) has completed. */
if (Signals & (1L <<
    mouth_io.voice.message.io_Message.mn_ReplyPort->mp_SigBit))
{
    USHORT LipWidth, LipHeight;

    LipWidth = mouth_io.width << 2;
    LipHeight = mouth_io.height;

    DrawMouth(LipWidth, LipHeight);

    /* On occasion, the first request for a mouth shape is
     * handled before the narrator realizes it's speaking.
     * That's why this error is ignored.
     */
    if (mouth_io.voice.message.io_Error == ND_NoWrite)
        mouth_io.voice.message.io_Error = 0;

    if (mouth_io.voice.message.io_Error == 0)
        SendIO((struct IORequest *) &mouth_io);
    else
    {
        fprintf(stderr, "Narrator won't mouth. (%ld)0,
            mouth_io.voice.message.io_Error);
        Cleanup();
        exit(RETURN_WARN);
    }
}
#endif /* FACE_ON defined */
} /* The for will never exit through here, no Cleanup() needed. */
} /* main */

/* handle abort like this !!!! */
/* AbortIO(&voice_io);
voice_io.message.io_Error = 0;
mouth_io.voice.message.io_Error = 0;
*/

#ifdef FACE_ON
void DrawFace(void)
{
    /* Set pen to White, fill whole window. */
    SetAPen(FaceWindow->RPort, WHTP);
    RectFill(FaceWindow->RPort, 0, 0, FACE_W, FACE_H);

    /* Set pen to Blue, do eyes. */
    SetAPen(FaceWindow->RPort, BLUP);
    RectFill(FaceWindow->RPort, FACE_ELL, FACE_ET, FACE_ELR, FACE_EB);
    RectFill(FaceWindow->RPort, FACE_ERL, FACE_ET, FACE_ERR, FACE_EB);

    /* Do gob. */
    DrawMouth(FACE_INIT_W, 1);
}

void DrawMouth(USHORT half_w, USHORT half_h)
{
    WaitBOVP(&FaceWindow->WScreen->ViewPort);

    /* Clear the entire mouth area. */
    SetAPen(FaceWindow->RPort, WHTP);
    RectFill(FaceWindow->RPort, 0, FACE_LIP_T, FACE_W, FACE_H);

    /* Draw a new mouth. */
    SetAPen(FaceWindow->RPort, REDP);

    Move(FaceWindow->RPort, FACE_MID_W - half_w, FACE_MID_H);
    Draw(FaceWindow->RPort, FACE_MID_W, FACE_MID_H - half_h);
    Draw(FaceWindow->RPort, FACE_MID_W + half_w, FACE_MID_H);
    Draw(FaceWindow->RPort, FACE_MID_W, FACE_MID_H + half_h);
    Draw(FaceWindow->RPort, FACE_MID_W - half_w, FACE_MID_H);
}
#endif /* FACE_ON defined */

```

```

#ifdef PARSE
/** *****/
LONG
ParseArgs (int argc,
           char **argv,
           UBYTE *show_face,
           UWORD *sex,
           UWORD *inflect,
           UWORD *samp,
           UWORD *pitch,
           UWORD *speed,
           UWORD *vol,
           char **text)
{
    int len;

    argc--;
    argv++;

    while (argc > 0 && argv[0][0] == '-')
    {
        len = strlen(*argv);
        switch (argv[0][1])
        {
            case 'm':          /* -Male */
                if ((!strcmp(*argv, "-m", len)) ||
                    (!strcmp(*argv, "-male", len)))
                {
                    *sex = FEMALE;
                }
                else
                {
                    BadOpt(*argv);
                    return(1);
                }
                break;

            case 'f':          /* -face -Female */
                if ((!strcmp(*argv, "-f", len)) ||
                    (!strcmp(*argv, "-female", len)))
                {
                    *sex = FEMALE;
                }
#ifdef FACE_ON
                else if (!strcmp(*argv, "-face", len))
                {
                    *show_face = 1;
                }
#endif
                else /* FACE_ON defined */
                {
                    BadOpt(*argv);
                    return(1);
                }
                break;

            case 'r':          /* -Robotic */
                if ((!strcmp(*argv, "-r", len)) ||
                    (!strcmp(*argv, "-robotic", len)))
                {
                    *inflect = ROBOTICF0;
                }
                else
                {
                    BadOpt(*argv);
                    return(1);
                }
                break;

            case 'n':          /* -Natural */
                if ((!strcmp(*argv, "-n", len)) ||
                    (!strcmp(*argv, "-natural", len)))

```

```

        {
            *inflect = NATURALF0;
        }
        else
        {
            BadOpt(*argv);
            return(1);
        }
        break;

    case 's':          /* -Speed <frequency> */
        if ((!strcmp(*argv, "-s", len)) ||
            (!strcmp(*argv, "-speed", len)))
        {
            argc--; argv++;
            *speed = atoi(*argv);
        }
        else
        {
            BadOpt(*argv);
            return(1);
        }
        break;

    case 'p':          /* -pitch <frequency> */
        if ((!strcmp(*argv, "-p", len)) ||
            (!strcmp(*argv, "-pitch", len)))
        {
            argc--; argv++;
            *pitch = atoi(*argv);
        }
        else
        {
            BadOpt(*argv);
            return(1);
        }
        break;

    case 'v':          /* -vol <level> */
        if ((!strcmp(*argv, "-v", len)) ||
            (!strcmp(*argv, "-vol", len)) ||
            (!strcmp(*argv, "-volume", len)))
        {
            argc--; argv++;
            *vol = atoi(*argv);
        }
        else
        {
            BadOpt(*argv);
            return(1);
        }
        break;

    default:
        BadOpt(*argv);
        return(1);
        break;
}
argc--; argv++;

/* The last arg is the english string to speak. */
if (argc > 0)
{
    *text = *argv;
}

if (argc > 1)
{
    fprintf(stderr, "Arguments after text ignored.0);
}

```

```

/* Narrator has it's limits, be sure to abide by them. */
if (*speed > MAXRATE)
    *speed = MAXRATE;
if (*speed < MINRATE)
    *speed = MINRATE;

if (*pitch > MAXPITCH)
    *pitch = MAXPITCH;
if (*pitch < MINPITCH)
    *pitch = MINPITCH;

if (*samp > MAXFREQ)
    *samp = MAXFREQ;
if (*samp < MINFREQ)
    *samp = MINFREQ;

if (*vol > MAXVOL)
    *vol = MAXVOL;
if (*vol < MINVOL)
    *vol = MINVOL;
}

void BadOpt(char *option)
{
    char *whoami;
    whoami = "sayit";

    fprintf(stderr, "%s: option
    fprintf(stderr, "Usage: %s0, whoami);
    fprintf(stderr, " -m or -male -f or -female0);
    fprintf(stderr, " -r or -robot -n or -natural0);
    fprintf(stderr, " -s or -speed <WPM 40-400>0);
    fprintf(stderr, " -p or -pitch <65-320>0);
    fprintf(stderr, " -v or -volume <0-64>0);
#ifdef FACE_ON
    fprintf(stderr, " -face0);
#endif /* FACE_ON defined */
    fprintf(stderr, " [

    Cleanup();
}
#endif /* PARSE defined */

/* Deallocate any memory, and close all of the
 * windows/screens/devices/libraries in reverse order to
 * make things work smoothly. And be sure to check
 * that the open/allocation was successful before
 * closing/deallocating.
 */
void Cleanup(void)
{
    if (write_port.mp_SigBit != -1)
        FreeSignal(write_port.mp_SigBit);
#ifdef FACE_ON
    if (read_port.mp_SigBit != -1)
        FreeSignal(read_port.mp_SigBit);
    if (FaceWindow != NULL)
        CloseWindow(FaceWindow);
#endif /* FACE_ON defined */
    if (NarratorOpenError == 0)
        CloseDevice((struct IOREquest *) &voice_io);
    if (TranslatorBase != NULL)
        CloseLibrary(TranslatorBase);
    if (GfxBase != NULL)
        CloseLibrary(GfxBase);
    if (IntuitionBase != NULL)
        CloseLibrary(IntuitionBase);
    return;
}

```

# Chapter 38

## Parallel Device

This chapter describes software access to the Centronics-compatible parallel port. The parallel port is primarily intended for output to printers. The Amiga parallel port includes extensions for bi-directional IO. The `parallel.device` is based on the foundation of Exec device IO, with extensions for parameter setting and control.

### Introduction

The parallel device may be opened in either the exclusive or shared access modes. Device-specific parameters may be specified using the `PDCMD_SETPARAMS` command.

The `parallel.device` is very similar to the `serial.device`. See the “Serial Device” chapter to learn more about advanced use of the device IO system. Also see the include file `devices/parallel.h` and the AutoDocs for for the `parallel.device` (AutoDocs may be found in the Addison-Wesley *ROM Kernel Manual: Includes & Autodocs*).



## Opening & Closing the Parallel Device

The following is a complete example of writing to the parallel.device. During the open the parallel device pays attention to just one flag; **PARF\_SHARED**. For consistency, the other flag bits should also be properly set. Full descriptions of all flags will be given later. When the parallel device is opened, it fills the latest default parameter settings into the **IOExtPar** block.

```
/* parallel.c - Simple, abortable example of parallel.device usage
 * Compile with Lattice 5.04: LC -L -catsfq
 */
#include <exec/types.h>
#include <devices/parallel.h>
#include <libraries/dos.h>
#ifdef LATTICE
#include <proto/exec.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
void main(void);
#endif

#define DEVICE_NAME "parallel.device"
#define UNIT_NUMBER 0

void main()
{
    struct MsgPort *ParallelMP;          /* Define storage for one pointer */
    struct IOExtPar *ParallelIO;         /* Define storage for one pointer */
    ULONG          WaitMask;             /* Collect all signals here */
    ULONG          Temp;                 /* Hey, we all need pockets :- ) */

    if( ParallelMP=CreatePort(0,0) )
    {
        if( ParallelIO=(struct IOExtPar *)
            CreateExtIO(ParallelMP,sizeof(struct IOExtPar)) )
        {
            ParallelIO->io_ParFlags=0;    /* Example of setting flags */

            if ( OpenDevice(DEVICE_NAME,UNIT_NUMBER,ParallelIO,0) )
                printf("Parallel.device did not open\n");
            else
            {
                /* Precalculate a wait mask for the CTRL-C, CTRL-F and message
                 * port signals. When one or more signals are received,
                 * Wait() will return. Press CTRL-C to exit the example.
                 * Press CTRL-F to wake up the example without doing anything.
                 * NOTE: A signal may show up without an associated message!
                 */
                WaitMask = SIGBREAKF_CTRL_C |
                           SIGBREAKF_CTRL_F |
                           1L << ParallelMP->mp_SigBit;

                ParallelIO->IOPar.io_Command = CMD_WRITE;
                ParallelIO->IOPar.io_Length  = -1;
                ParallelIO->IOPar.io_Data    = (APTR)"Hey, Jude! \015\012";
                SendIO(ParallelIO);          /* execute write */

                printf("Sleeping until CTRL-C, CTRL-F, or write finish\n");
                while(1)
                {
                    Temp = Wait(WaitMask);
                    printf("Just woke up (YAWN!)\n");

                    if( SIGBREAKF_CTRL_C & Temp)
                        break;
                }
            }
        }
    }
}
```

```

        if( CheckIO(ParallelIO) ) /* If request is complete... */
        {
            WaitIO(ParallelIO); /* clean up and remove reply */
            printf("%ld bytes sent\n",ParallelIO->IOPar.io_Actual);
            break;
        }
    }

    AbortIO(ParallelIO); /* Ask device to abort request, if pending */
    WaitIO(ParallelIO); /* Wait for abort, then clean up */

    CloseDevice(ParallelIO);
}
DeleteExtIO(ParallelIO);
}
DeletePort(ParallelMP);
}
}

```

The functions **CreatePort()** and **CreateExtIO()** are part of **amiga.lib**. See the “Linker Libraries” appendix and the “Serial Device” chapter for more information.

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**. When the last close is performed, the device will deallocate all resources and buffers. The latest parameter settings will be saved for the next open. All IO Requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**:

```

AbortIO(ParallelIO); /* Ask device to abort request, if pending */
WaitIO(ParallelIO); /* Wait for abort, then clean up */

CloseDevice(ParallelIO);

```

### About The First Example

The above example code may seem needlessly complex. The **DoIO()** function is considerably easier to use. However, **DoIO()** is not always appropriate for executing **CMD\_READ** or **CMD\_WRITE** commands. **DoIO()** will not return until the IO request has finished. If a printer is disconnected or off-line, the write request may *never* finish. Read requests will not finish until characters arrive at the port. See the “Serial Device” chapter for more solutions and information about the **SendIO()** and **AbortIO()** functions.

## Termination of Reads

Reading from the parallel device can terminate early if an error occurs, or if end of file is sensed. You can specify a set of possible end-of-file characters that the parallel device is to look for in the input stream. These are contained in an `io_TermArray` that you provide with the `PDCMD_SETPARAMS` command. *Note:* `io_TermArray` is used only when EOF mode is selected.

If EOF mode is selected, each input data character that is read into the user's data block is compared against those in `io_TermArray`. If a match is found, the `IORequest` is terminated as complete, and the count of characters read (including the `TermChar`) is stored in `io_Actual`. To keep this search overhead as efficient as possible, the parallel device requires that the array of characters be in descending order (an example is shown in the `PDCMD_SETPARAMS` summary in the parallel.device AutoDocs. The array has eight bytes and all must be valid (that is, do not pad with zeros unless zero is a valid EOF character). Fill to the end of the array with the least-value `TermChar`. When making an arbitrary choice of EOF character(s), it is advisable to use the lowest value(s) available.

## Setting Parallel Parameters

You can control the parallel parameters shown in the following table. The parameter name within the parallel `IOExtPar` data structure is shown below. All of the fields described in this section are filled with defaults when you call `OpenDevice()`. Thus, you need not worry about any parameter that you do not need to change.

Table 38-1: Parallel Parameters (IOExtPar)

| Parameter Name             | Characteristic It Controls   |
|----------------------------|--|
| <code>io_PExtFlags</code>  | Reserved for future use.   |
| <code>io_PTermArray</code> | A byte-array of eight termination characters, must be in descending order. If <code>EOFMODE</code> is set in the parallel flags, this array specifies eight possible choices of character to use as an end-of-file mark. See the <code>PDCMD_SETPARAMS</code> summary page in the "Device Summaries" appendix and the section above titled "Termination of the Read" for more information. |
| <code>io_Status</code>     | Printer status bits returned by the <code>PDCMD_QUERY</code> command. See the <i>devices/parallel.h</i> file for bit definitions. Bit 0 reflects the state of the printer busy line. Bit 1 is for paper out. Bit 2 is the printer selected line. Other bits are reserved. Note that the printer selected line is also connected to Ring Indicator on the A500 and A2000 machines.          |
| <code>io_ParFlags</code>   | Explained below; see "PARALLEL FLAGS."   |

## PARALLEL FLAGS (io\_ParFlags)

The flags shown in the following table can be set to affect the operation of the parallel device. Note that the default state of all of these flags is zero.

Table 38-2: Parallel Flags (io\_ParFlags)

| Flag Name           | Effect on Device Operation   |
|---------------------|--|
| <b>PARF_EOFMODE</b> | Set this bit if you want the parallel device to check I/O characters against <b>io_TermArray</b> and terminate the <b>IORequest</b> immediately if an end-of-file character has been encountered. <i>Note:</i> This bit can be set and reset directly in the user's <b>IOExtPar</b> block without a call to <b>PDCMD_SETPARAMS</b> . |
| <b>PARF_SHARED</b>  | Set this bit if you want to allow other tasks to simultaneously access the parallel port. The default is exclusive access. If someone already has the port, whether for exclusive or shared access, and you ask for exclusive access, your <b>OpenDevice()</b> call will fail (must be modified before <b>OpenDevice()</b> ).        |

## SETTING THE PARAMETERS

You change parallel parameters by setting the flags and parameters as you desire, and then transmitting a **PDCMD\_SETPARAMS** command to the device. Here is an example:

```
ParallelIO->IOPar.io_Command = PDCMD_SETPARAMS;  
ParallelIO->io_ParFlags      &= ~PARF_EOFMODE;  
DoIO(ParallelIO);  
printf("Error setting parameters!\n");
```

The above code fragment modifies one bit in **io\_ParFlags**, then sends the command.

### NOTE

A parameter change should not be performed while an IO request is actually being processed, because it might invalidate already active request handling. Therefore you should use **PDCMD\_SETPARAMS** only when you have no parallel IO requests pending.

## Errors from the Parallel Device

The possible error returns from the parallel device are listed in table 14-3.

Table 38-3: Parallel Device Errors

|                                       |   |                    |
|---------------------------------------|---|--------------------|
| <code>#define ParErr_DevBusy</code>   | 1 | -Device busy       |
| <code>#define ParErr_BufToBig</code>  | 2 | -Out of memory     |
| <code>#define ParErr_InvParam</code>  | 3 | -Invalid parameter |
| <code>#define ParErr_LineErr</code>   | 4 |                    |
| <code>#define ParErr_NotOpen</code>   | 5 | -Unused            |
| <code>#define ParErr_PortReset</code> | 6 |                    |
| <code>#define ParErr_InitErr</code>   | 7 |                    |

# Chapter 39

## Printer Device

### Introduction

The printer device offers a way of doing configuration-independent output to a printer attached to the Amiga. It can be thought of as a filter: it takes standard commands as input and translates them into commands understood by the printer. The commands sent to the printer are defined in a specific printer driver program. For each type of printer in use, a driver (or the driver of a compatible printer) should be present in the *devs:printers* directory.

The printer device is totally transparent to an application. It uses information set up by the Workbench Preferences tool to identify the type of printer, whether it is serial or parallel, etc. It also offers the flexibility to send raw information to the printer for special non-standard or unsupported features. Raw data transfer is not recommended for conventional text and graphics since it will result in applications that will only work with certain printers. By using the standard *printer.device* interface, an application can perform device independent output to a printer.

There are two ways of doing output to the *printer.device*:

- **PRT:**—the AmigaDOS printer device

PRT: may be opened just like any other AmigaDOS file. You may send standard escape sequences to PRT: to specify the options you want as shown in the command table below. The escape sequences are interpreted by the printer driver, translated into printer-specific escape sequences and forwarded to the printer. When

using PRT: the escape sequences and data must be sent as a character stream. Using PRT: is by far the easiest way of doing text output to a printer.

- **printer.device**—to directly access the printer device itself

By opening the printer device directly, you have full control over the printer. You can either send standard escape sequences as shown in the command table below or send raw characters directly to the printer with no processing at all. Doing this would be similar to sending raw characters to SER: or PAR: from AmigaDOS. (Since this interferes with device-independence it is strongly discouraged). Direct access to the printer device also allows you to transmit device I/O commands, such as reset, flush or a raster dump on a graphics-capable printer.

#### NOTE

All “raw escape sequences” to the printer transmitted through the printer device must take the form of a character stream.

## Using the Printer Device as an AmigaDOS File

### OPENING THE AMIGADOS PRINTER DEVICE

You can open the printer device just as though it were a normal AmigaDOS output file. Here is an example program segment that accomplishes this:

```
struct FileHandle *file;

file = Open( "PRT:", MODE_NEWFILE );
if (file == 0) exit(PRINTER_WONT_OPEN);
```

Then, to print use code like this:

```
actual_length = Write(file, dataLocation, length);
```

where

**file** is a file handle.

#### **dataLocation**

is a pointer to the first character in the output stream you wish to write. This stream can contain the standard escape sequences as shown in the command table below. The printer command aRAW can be used in the stream if character translation is not desired.

#### **length**

is the length of the output stream.

#### **actual\_length**

is the actual length of the write. For the printer device, if there are no errors, this will be the same as the length of write requested. The only exception is if you specify a value of -1 for length. In this case, -1 for length means that a null (0) terminated stream is being written to the printer device. The device returns the count of characters written prior to encountering the null. If it returns a value of -1 in **actual\_length**, there has been an error.

## CLOSING THE AMIGADOS PRINTER DEVICE

When the printer I/O is complete, you should close the device. Don't keep the device open when you are not using it. The user may have changed the printer settings by using the Workbench Preferences tool. There's also the possibility the printer has been turned off and on again causing the printer to switch to its own default settings. Every time the printer device is opened, it reads the current Preferences settings. Hence, by always opening the printer device just before printing and always closing it afterwards, you ensure that your application is using the current Preferences settings. Here is a sample function call that you could use:

```
Close(file);
```

### NOTE

Printer I/O through the DOS versions of the printer device must be done by a process, *not* by a task. DOS utilizes information in the process control block and would become confused if a simple task attempted to perform these activities. Printer I/O using the printer device directly, however, *can* be performed by a task.

## Using the Printer Device Directly

In order to send commands and data to the printer device directly, data structures need to be set up. These structures enable us to use the standard functions provided by Exec, like `DoIO()` and `SendIO()`, to send I/O requests to the device. (See the Exec chapter "Input/Output" in this manual for detailed information about I/O requests.)

### DATA STRUCTURES USED DURING PRINTER I/O

There are three distinct kinds of data structures required by the printer I/O routines. Some of the printer device I/O commands, such as `CMD_START` and `CMD_WRITE` require only an `IStdReq` data structure. Others, such as `PRD_PRTCOMMAND` or `PRD_DUMP_RPORT`, require a larger data structure called `IOPrtCmdReq` (for "Printer Command Request") or `IODRPReq` (for "Dump a RastPort Request") structure. These two structures are defined in the include file `devices/printer.h`. For convenience, the printer device has defined a single data structure, called `printerIO`, that can be used to represent any of the three different kinds of printer communications request blocks.

The data structure type `printerIO` used in the following examples is a C-language union defined as:

```
union printerIO
{
    struct IStdReq    ios;
    struct IODRPReq   iodrp;
    struct IOPrtCmdReq iopc;
};
```

This means that one memory area can be used to represent three distinct forms of memory layout for the three different types of data structures that must be used to pass commands to the printer device. If you use the function `CreateExtIO()`, you can automatically allocate enough memory to hold the largest structure in the union statement.



## CREATING AN I/O REQUEST

Printer I/O, like the I/O of other devices, requires that you create an I/O request message that you pass to the printer device for processing. The message contains the command as well as a data area. For a write, there will be a pointer in the data area to the stream of information you wish to write to the printer.

The following example functions can be used to create and delete the message block that you use for printer communications.

```
/* Create printer device I/O message block.
 * message port is in request->ios.io_Message.mn_ReplyPort
 */

union printerIO *CreatePrtReq()
{
    struct MsgPort *prtport;
    union printerIO *request;

    if (!(prtport = CreatePort(NULL, 0)))
        return(0);
    if (!(request = (union printerIO *)CreateExtIO(prtport,
        sizeof(union printerIO))))
    {
        DeletePort(prtport);
        return(0);
    }
    return(request);
}

/* Delete printer device I/O message block. */

void DeletePrtReq(request)
union printerIO *request;
{
    struct MsgPort *prtport;

    prtport = request->ios.io_Message.mn_ReplyPort;
    DeleteExtIO((struct IORequest *)request);
    DeletePort(prtport);
}
```

Error handling is not shown here. It is deferred to the full example later in this chapter.

The routines `CreatePort()` and `CreateExtIO()`, which are part of *amiga.lib*, are described in the Exec chapters “Messages and Ports” and “Input/Output”.

## OPENING THE PRINTER DEVICE

Once you’ve created the I/O request message, you can open the printer device itself, using code like the following:

```
/* Open the printer device */

int OpenPrinter(request)
union printerIO *request;
{
    return(OpenDevice("printer.device", 0, request, 0));
}
```

This routine returns a value of zero if the printer device was opened successfully and a value other than zero if it did not open.

## SENDING I/O COMMANDS TO THE PRINTER DEVICE

Once the printer device is opened successfully, device I/O commands can be sent to it by setting up the I/O request and calling the appropriate I/O routine. The following commands are recognized by the printer device and can be used in an I/O request:

### **CMD\_FLUSH**

Abort all stopped I/O at the unit.

### **CMD\_INVALID**

Is an invalid command and sets the device error appropriately.

### **CMD\_RESET**

Reset the printer device without destroying handles.

### **CMD\_START**

Restart the unit after a stop command.

### **CMD\_STOP**

Pause current and queued I/O requests immediately.

### **CMD\_WRITE**

Send output to printer.

### **PRD\_DUMPRTPORT**

Dump a raster port to a printer with graphic capabilities.

### **PRD\_PRTCOMMAND**

Send a command to the printer.

### **PRD\_QUERY**

Query printer port/line status.

### **PRD\_RAWWRITE**

Send output to printer without processing it.

To use these device I/O commands, you set up the appropriate I/O request block with the I/O command and other specific information, then call **DoIO()** or **SendIO()** to issue the commands to the printer device. To transmit the standard device I/O commands (flush, invalid, reset, start, stop) to the printer device all that is needed is the command itself and a pointer to the I/O request block (which contains the device and unit information). A small function like the following could be used to do this.

```
/* send 'standard' I/O device Command to printer device */

int PrintDevCommand(request, devcommand)
union printerIO *request;
UWORD devcommand;
{
    request->ios.io_Command = devcommand;
    return(DoIO((struct IORequest *)request));
}
```

The other device I/O commands need more information to function properly and are discussed separately below.

## WRITING TEXT TO THE PRINTER

To output text to the printer the device I/O commands `CMD_WRITE` and `PRD_RAWWRITE` can be used. `CMD_WRITE` lets the printer device process the character stream to the printer. Possibly embedded escape sequences will be "translated" into appropriate escape sequences for the printer. `PRD_RAWWRITE`, on the other hand, passes the data directly along to the printer without *any* processing. Hence to use `PRD_RAWWRITE` the caller has to "know" what the printer will accept as a command. Moreover, `PRD_RAWWRITE` will, unlike `CMD_WRITE`, ignore the printer Preferences settings (like Pitch, Spacing, etc.) Unless the printer has already been initialized by another command, the printer's own default settings will be used when printing raw, not the user's Preferences settings.

The following two routines might be used to send text and raw text to the printer.

### NOTE

For the device I/O commands `CMD_WRITE` and `PRD_RAWWRITE` the `IOStdReq` structure of the union `printerIO` must be used.

```
/* Send a NULL-terminated string to the printer */

/* Assumes printer device is open and printerMsg is correctly initialized.
 * Watches for embedded "escape-sequences" and handles them as defined.
 */

int PrintString(request, string)
union printerIO *request;
UBYTE *string;
{
    request->ios.io_Command = CMD_WRITE;
    request->ios.io_Data = (APTR) string;
    request->ios.io_Length = -1L;
    /* if -1, the printer assumes it has been given
     * a null-terminated string.
     */
    return(DoIO((struct IORequest *)request));
}

/* Send RAW character stream to the printer directly,
 * avoid "escape-sequence" parsing by the device.
 */

int
PrintRawString(request, buffer, count)
union printerIO *request;      /* a properly initialized request block */
UBYTE *buffer;                /* where is the output stream of characters */
ULONG count;                  /* how many characters to output */
{
    /* queue a printer raw write */
    request->ios.io_Command = PRD_RAWWRITE;
    request->ios.io_Data = (APTR) buffer;
    request->ios.io_Length = count;
    return(DoIO((struct IORequest *)request));
}
```

## SENDING PRINTER COMMANDS TO THE PRINTER

As mentioned before, it is possible to embed printer commands (escape sequences) into the character stream and send them to the printer using the CMD\_WRITE device I/O command. It is also possible to use the *printer command names* using the device I/O command PRD\_PRTCOMMAND with the IOPrtCmdReq data structure. This gives you a mnemonic way of setting the printer to your program needs. The following routine is an example of how to do this:

```
int PrintPrtCommand(request, command, p0, p1, p2, p3)
union printerIO *request;
UWORD command;           /* the printer command */
UBYTE p0, p1, p2, p3;     /* and its parameters */
{
    /* queue a printer command */
    request->iopc.io_Command = PRD_PRTCOMMAND;
    request->iopc.io_PrtCommand = command;
    request->iopc.io_Parm0 = p0;
    request->iopc.io_Parm1 = p1;
    request->iopc.io_Parm2 = p2;
    request->iopc.io_Parm3 = p3;
    return(DoIO((struct IORequest *)request));
}
```

As an example, suppose you wanted to set the left and right margins on your printer to columns 1 and 79, respectively. Here is a sample call to the above PrintPrtCommand() function for this purpose:

```
PrintPrtCommand(aSLRM, 1, 79, 0, 0);
```

Consult the command function table listed below for other printer commands.

## PRINTER COMMAND DEFINITIONS

The following table describes the supported printer functions.

### NOTE

Not all printers support every command. Unsupported commands will either be ignored or simulated using available functions.

To transmit a command to the printer device, you can either formulate a character stream containing the material shown in the "Escape Sequence" column of the table below or send an PRD\_PRTCOMMAND device I/O command to the printer device with the "Name" of the function you wish to perform.

Again, recall that SER: and PAR: and the PRD\_RAWWRITE device I/O command will ignore all of these and pass them directly on to the attached device.

Table 39-1: Printer Device Command Functions

| Name    | Cmd No. | Escape Sequence | Function   | Defined by:   |
|---------|---------|-----------------|--|---------------|
| aRIS    | 0       | ESCc            | Reset  | ISO           |
| aRIN    | 1       | ESC#1           | Initialize   | +++           |
| aIND    | 2       | ESCD            | Lf   | ISO           |
| aNEL    | 3       | ESCE            | Return,lf  | ISO           |
| aRI     | 4       | ESCM            | Reverse lf   | ISO           |
| aSGR0   | 5       | ESC[0m          | Normal char set  | ISO           |
| aSGR3   | 6       | ESC[3m          | Italics on   | ISO           |
| aSGR23  | 7       | ESC[23m         | Italics off  | ISO           |
| aSGR4   | 8       | ESC[4m          | Underline on   | ISO           |
| aSGR24  | 9       | ESC[24m         | Underline off  | ISO           |
| aSGR1   | 10      | ESC[1m          | Boldface on  | ISO           |
| aSGR22  | 11      | ESC[22m         | Boldface off   | ISO           |
| aSFC    | 12      | ESC[nm          | Set foreground color<br>where n stands for a pair<br>of ASCII digits, 3 followed<br>by any number 0-9 (See ISO<br>Color Table) | ISO           |
| aSBC    | 13      | ESC[nm          | Set background color<br>Where n stands for<br>a pair of ASCII digits, 4<br>followed by any number 0-9<br>(See ISO Color Table) | ISO           |
| aSHORP0 | 14      | ESC[0w          | Normal pitch   | DEC           |
| aSHORP2 | 15      | ESC[2w          | Elite on   | DEC           |
| aSHORP1 | 16      | ESC[1w          | Elite off  | DEC           |
| aSHORP4 | 17      | ESC[4w          | Condensed fine on  | DEC           |
| aSHORP3 | 18      | ESC[3w          | Condensed off  | DEC           |
| aSHORP6 | 19      | ESC[6w          | Enlarged on  | DEC           |
| aSHORP5 | 20      | ESC[5w          | Enlarged off   | DEC           |
| aDEN6   | 21      | ESC[6"z         | Shadow print on  | DEC (sort of) |
| aDEN5   | 22      | ESC[5"z         | Shadow print off   | DEC           |
| aDEN4   | 23      | ESC[4"z         | Doublestrike on  | DEC           |
| aDEN3   | 24      | ESC[3"z         | Doublestrike off   | DEC           |
| aDEN2   | 25      | ESC[2"z         | NLQ on   | DEC           |
| aDEN1   | 26      | ESC[1"z         | NLQ off  | DEC           |
| aSUS2   | 27      | ESC[2v          | Superscript on   | +++           |
| aSUS1   | 28      | ESC[1v          | Superscript off  | +++           |
| aSUS4   | 29      | ESC[4v          | Subscript on   | +++           |
| aSUS3   | 30      | ESC[3v          | Subscript off  | +++           |
| aSUS0   | 31      | ESC[0v          | Normalize the line   | +++           |
| aPLU    | 32      | ESCL            | Partial line up  | ISO           |
| aPLD    | 33      | ESCK            | Partial line down  | ISO           |
| aFNT0   | 34      | ESC(B           | US char set  | DEC           |
| aFNT1   | 35      | ESC(R           | French char set  | DEC           |

|         |    |          |                         |               |
|---------|----|----------|-------------------------|---------------|
| aFNT2   | 36 | ESC(K    | German char set         | DEC           |
| aFNT3   | 37 | ESC(A    | UK char set             | DEC           |
| aFNT4   | 38 | ESC(E    | Danish I char set       | DEC           |
| aFNT5   | 39 | ESC(H    | Swedish char set        | DEC           |
| aFNT6   | 40 | ESC(Y    | Italian char set        | DEC           |
| aFNT7   | 41 | ESC(Z    | Spanish char set        | DEC           |
| aFNT8   | 42 | ESC(J    | Japanese char set       | +++           |
| aFNT9   | 43 | ESC(6    | Norwegian char set      | DEC           |
| aFNT10  | 44 | ESC(C    | Danish II char set      | +++           |
| aPROP2  | 45 | ESC[2p   | Proportional on         | +++           |
| aPROP1  | 46 | ESC[1p   | Proportional off        | +++           |
| aPROP0  | 47 | ESC[0p   | Proportional clear      | +++           |
| aTSS    | 48 | ESC[n E  | Set proportional offset | ISO           |
| aJFY5   | 49 | ESC[5 F  | Auto left justify       | ISO           |
| aJFY7   | 50 | ESC[7 F  | Auto right justify      | ISO           |
| aJFY6   | 51 | ESC[6 F  | Auto full justify       | ISO           |
| aJFY0   | 52 | ESC[0 F  | Auto justify off        | ISO           |
| aJFY3   | 53 | ESC[3 F  | Letter space (justify)  | ISO (special) |
| aJFY1   | 54 | ESC[1 F  | Word fill(auto center)  | ISO (special) |
| aVERP0  | 55 | ESC[0z   | 1/8" line spacing       | +++           |
| aVERP1  | 56 | ESC[1z   | 1/6" line spacing       | +++           |
| aSLPP   | 57 | ESC[nt   | Set form length n       | DEC           |
| aPERF   | 58 | ESC[nq   | Perf skip n (n>0)       | +++           |
| aPERF0  | 59 | ESC[0q   | Perf skip off           | +++           |
| aLMS    | 60 | ESC#9    | Left margin set         | +++           |
| aRMS    | 61 | ESC#0    | Right margin set        | +++           |
| aTMS    | 62 | ESC#8    | Top margin set          | +++           |
| aBMS    | 63 | ESC#2    | Bottom margin set       | +++           |
| aSTBM   | 64 | ESC[n;nr | T&B margins             | DEC           |
| aSLRM   | 65 | ESC[n;ns | L&R margin              | DEC           |
| aCAM    | 66 | ESC#3    | Clear margins           | +++           |
| aHTS    | 67 | ESCH     | Set horiz tab           | ISO           |
| aVTS    | 68 | ESCJ     | Set vertical tabs       | ISO           |
| aTBC0   | 69 | ESC[0g   | Clr horiz tab           | ISO           |
| aTBC3   | 70 | ESC[3g   | Clear all h tab         | ISO           |
| aTBC1   | 71 | ESC[1g   | Clr vertical tabs       | ISO           |
| aTBC4   | 72 | ESC[4g   | Clr all v tabs          | ISO           |
| aTBCALL | 73 | ESC#4    | Clr all h & v tabs      | +++           |
| aTBSALL | 74 | ESC#5    | Set default tabs        | +++           |
| aEXTEND | 75 | ESC[n"x  | Extended commands       | +++           |
| aRAW    | 76 | ESC[n"r  | Next n chars are raw    | +++           |

**Legend:**

ISO indicates that the sequence has been defined by the International Standards Organization. This is also very similar to ANSI x3.64.

DEC indicates a control sequence defined by Digital Equipment Corporation.

+++ indicates a sequence unique to Amiga.

n stands for a decimal number expressed as a set of ASCII digits. For example, in the aRAW string ESC[5"rHELLO, n is substituted by 5, the number of RAW characters you send to the printer.

### ISO Color Table

- 0 Black
- 1 Red
- 2 Green
- 3 Yellow
- 4 Blue
- 5 Magenta
- 6 Cyan
- 7 White
- 8 NC
- 9 Default

### DUMPING A RASTPORT TO THE PRINTER

You can dump a RastPort (drawing area) to a graphics capable printer by sending the device I/O command PRD\_DUMPRPORT the printer, along with several parameters that define how the dump is to be accomplished.

For using PRD\_DUMPRPORT, the data structure IODRPREq is needed. Here is an overview of the possible arguments for the RastPort dump.

|              |   |
|--------------|---|
| io_Command   | PRD_DUMPRPORT   |
| io_RastPort  | A pointer to a RastPort. The RastPort's BitMap could be in FASTMEM. |
| io_ColorMap  | A pointer to a ColorMap. This could be a custom one.                |
| io_Modes     | The view modes from a ViewPort structure.                           |
| io_SrcX      | X offset in the RastPort to start printing from.                    |
| io_SrcY      | Y offset in the RastPort to start printing from.                    |
| io_SrcWidth  | Width of the RastPort to print from io_SrcX.                        |
| io_SrcHeight | Height of the RastPort to print from io_SrcY.                       |
| io_DestCols  | Width of the dump in printer pixels.                                |
| io_DestRows  | Height of the dump in printer pixels.                               |
| io_Special   | Flag bits (described below).  |

Looking at these arguments you can see the enormous flexibility the printer device offers for dumping a RastPort. The RastPort pointed to could be totally custom defined. This flexibility means it is possible to build a BitMap with the resolution of the printer. This would result in having one pixel of our BitMap correspond to one pixel of the printer. In other words only the resolution of the output device would limit our final result. With 12 bit planes and a custom ColorMap, you could dump 4096 colors—without the HAM limitation—to a suitable printer. The offset,

width and height parameters allow dumps of any desired part of the picture. Finally the **ViewPort** mode, **io\_DestCols**, **io\_DestRows** parameters, together with the **io\_Special** flags define how the dump will appear on paper and aid in getting the correct aspect ratio.

## PRINTER SPECIAL FLAGS

Following a description of the valid flag bits for **io\_Special** which can be used for a **RastPort** dump.

|                           |   |
|---------------------------|---|
| <b>SPECIAL_ASPECT</b>     | Allows one of the dimensions to be reduced/expanded to preserve the correct aspect ratio of the printout.   |
| <b>SPECIAL_CENTER</b>     | Centers the image between the left and right edge of the paper.   |
| <b>SPECIAL_NOFORMFEED</b> | Prevents the page from being ejected after a graphics dump. Usually used to mix graphics and text or multiple graphics dump on a page oriented printer (normally a laser printer).  |
| <b>SPECIAL_NOPRINT</b>    | The print size will be computed, and set in <b>io_DestCols</b> and <b>io_DestRows</b> , but won't print. This way the application can see what the actual printsize in printerpixels would be.  |
| <b>SPECIAL_TRUSTME</b>    | Instructs the printer not to send a reset before and after the dump. This flag is obsolete for V1.3 (and higher) drivers.   |
| <b>SPECIAL_DENSITY1-7</b> | This flag bit is set by the user in Preferences. Refer to 'CHANGING THE PRINTER PREFERENCES SETTINGS' if you want to change to density of the printout. (Or any other setting for that matter.)   |
| <b>SPECIAL_FULLCOLS</b>   | The width is set to the maximum possible, as determined by the printer or the configuration limits.   |
| <b>SPECIAL_FULLROWS</b>   | The height is set to the maximum possible, as determined by the printer or the configuration limits.  |
| <b>SPECIAL_FRACCOLS</b>   | Informs the printer device that the value in <b>io_DestCols</b> is to be taken as a longword binary fraction of the maximum for the dimension. For example if <b>io_DestCols</b> would be 0x8000, the width would be 1/2 (0x8000 / 0xffff) of the width of the paper. |
| <b>SPECIAL_FRACROWS</b>   | Informs the printer device that the value in <b>io_DestRows</b> is to be taken as a longword binary fraction for the dimension.   |
| <b>SPECIAL_MILCOLS</b>    | Informs the printer device that the value in <b>io_DestCols</b> is specified in thousands of an inch. For example if <b>io_DestCols</b> would be 8000, the width of the printout would be 8.000 inch.   |
| <b>SPECIAL_MILROWS</b>    | Informs the printer device that the value in <b>io_DestRows</b> is specified in thousands of an inch.   |

## PRINTING WITH CORRECTED ASPECT RATIO

Using the special flags it is fairly easy to ensure a graphic dump will have the correct aspect ratio on paper. There are some considerations though when printing a non-displayed **RastPort**. One way to get a corrected aspect ratio dump is to calculate the printer's ratio from **XDotsInch** and **YDotsInch** (taking into account that the printer may not have square pixels) and then adjust the width and height parameters accordingly. You then ask for a non-aspect-ratio-corrected dump since you already corrected it yourself.

Another possibility is having the printer device do it for you. To get a correct calculation you could build your **RastPort** dimensions in two ways:



- 1) Using an integer multiple of one of the standard (NTSC) display resolutions and setting the **io\_Modes** argument accordingly. For example if your **RastPort** dimensions were 1280 x 800 (an even multiple of 640 x 400) you would set **io\_Modes** to **LACE | HIRES**. Setting the **SPECIAL\_ASPECT** flag would enable the printer device to calculate the aspect ratio of the image properly.
- 2) Using an arbitrary sized **RastPort** and setting **GfxBase->NormalDPMX** and **GfxBase->NormalDPMY** to its dimensions. Again, setting the **SPECIAL\_ASPECT** flag would enable the printer device to calculate the aspect ratio of the image, based on the information you provided. This method could also be used for PAL-sized **RastPorts**.

#### NOTE

After the graphics dump you should restore **GfxBase->NormalDPMX** and **GfxBase->NormalDPMY** to their original values.

The actual code to dump the **RastPort** could be like the following example routine:

```
void DumpRPort(request, rastPort, colorMap, modes, sx, sy, sw, sh, dc, dr, s)
union printerIO *request;
struct RastPort *rastPort;
struct ColorMap *colorMap;
ULONG modes;
UWORD sx, sy, sw, sh;
LONG dc, dr;
UWORD s;
{
    request->iodrp.io_Command = PRD_DUMPRPORT;
    request->iodrp.io_RastPort = rastPort;
    request->iodrp.io_ColorMap = colorMap;
    request->iodrp.io_Modes = modes;
    request->iodrp.io_SrcX = sx;
    request->iodrp.io_SrcY = sy;
    request->iodrp.io_SrcWidth = sw;
    request->iodrp.io_SrcHeight = sh;
    request->iodrp.io_DestCols = dc;
    request->iodrp.io_DestRows = dr;
    request->iodrp.io_Special = s;
    SendIO((struct IORequest *)request);
}
```

The asynchronous **SendIO()** routine is used in this example instead of the synchronous **DoIO()**. A call to **DoIO()** does not return until the I/O request is finished. A call to **SendIO()** returns immediately. This allows your task to do other processing such as checking if the user wants to abort the I/O request. When writing a lot of text, or raw data to the printer, this should be seriously considered for **CMD\_WRITE** and **PRD\_RAWWRITE** too. The following code is an simplified example of using **SendIO()** to issue an I/O request while giving the user a way to abort it:

```
union printerIO      *prtrequest;
struct Window        *wn;
struct MsgPort       *prtport;
struct IntuiMessage  *msg;

ULONG signal, usersig, printersig;
BOOL EINDE = FALSE;

/* Assuming the I/O request message has been created and the printer
 * device is opened.
 */

usersig = 1 << wn->UserPort->mp_SigBit;
printersig = 1 << prtport->mp_SigBit;

SendIO(prtrequest);
signal = Wait(usersig | printersig) /* wait until either user or printer
                                signals */
```

```

if (signal & usersig)
{
    /* User expressed need to abort */
    while (msg = (struct IntuiMessage *)GetMsg(wn->UserPort))
        ReplyMsg(msg);
    EINDE = TRUE;
}
if (signal & printersig)
{
    /* Printer is either ready or has an error to report */
    /* Error is in printerIO->iodrp.io_Error, show user if problem */
    while (msg = GetMsg(prtport));
}

if (EINDE)
{
    AbortIO(prtreq);
    WaitIO(prtreq);          /* wait for reply */
}
/* 'prtreq' can be used again */

```

### NOTE

It is possible that the printer has been instructed to receive a certain amount of data and is still in an "expecting" state if an I/O request has been aborted by the user. This means the printer would try to finish the job with the data the next I/O request might send. Currently the best way to overcome this problem is to reset the printer.

## HANDLING PRINTER ERROR CODES

It is good practice to inform the user about any errors that may have come back from an I/O request instead of simply not printing. The currently defined errors are:

|                         |  |
|-------------------------|--|
| PDERR_NOERR             | Clean exit. No errors.   |
| PDERR_CANCEL            | User canceled job.   |
| PDERR_NOTGRAPHICS       | Printer cannot output graphics.  |
| PDERR_INVERTHAM         | OBSOLETE   |
| PDERR_BADDIMENSION      | Print dimensions are illegal.  |
| PDERR_DIMENSIONOVERFLOW | OBSOLETE   |
| PDERR_INTERNALMEMORY    | No memory available for internal variables.  |
| PDERR_BUFFERMEMORY      | No memory available for print buffer.  |
| PDERR_TOOKCONTROL       | The printer driver does the graphic dump entirely on it's own.   |
|                         | The printer device can assume the dump has been done. The calling application will not be informed of this internal error. |

The following example will dump the **RastPort** of a window to the printer, wait for either the printer to finish or the user to cancel the dump and act accordingly.

```

/* WindowDump.c 10/89
 * Compiled with Lattice C 5.04: LC -bl -cfist -v -y
 * Linkage: c.o+WindowDump.o Library lib:amiga.lib+lib:lc.lib
 */

#include <exec/types.h>
#include <devices/printer.h>
#include <exec/io.h>
#include <graphics/display.h>
#include <graphics/gfxbase.h>
#include <graphics/rastport.h>
#include <graphics/view.h>
#include <intuition/intuition.h>

```

```

#include <libraries/dos.h>
#include <libraries/dosextens.h>

#ifdef LATTICE
#include <proto/all.h>
#include <stdio.h>
#include <stdlib.h>
#endif

void main(void);
union printerIO *CreatePrtReq(void);
void DeletePrtReq(union printerIO *);
int OpenPrinter (union printerIO *);
void ClosePrinter(union printerIO *);
void DumpRPort(union printerIO *, struct RastPort *, struct ColorMap *,
               ULONG, UWORD, UWORD, UWORD, UWORD, LONG, LONG, UWORD);
void openLibraries(void);
struct Window *doWindow(SHORT, SHORT);
void cleanexit(UBYTE *);

union printerIO
{
    struct IOStdReq ios;
    struct IODRPreq iodr;
    struct IOPrtCmdReq iopc;
};

static UBYTE *prtErrorText[] =
{
    " EVERYTHING'S FINE",
    " USER CANCELED DUMP",
    " NOT A GRAPHICS PRINTER",
    " SHOULDN'T GET THIS ONE",
    " ILLEGAL DIMENSIONS",
    " SHOULDN'T GET THIS ONE",
    " NO MEMORY FOR VARIABLES",
    " NO MEMORY FOR BUFFER"
};

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Window *window;
union printerIO *printerReq;
BOOL PDOPEN = FALSE;

void main()
{
    struct IntuiMessage *msg;
    struct MsgPort *port;
    struct RastPort *rp;
    struct ViewPort *vp;

    ULONG usersig, printersig, signal;
    BOOL P_ABORT = FALSE;
    BOOL U_ABORT = FALSE;

    openLibraries();
    window = doWindow(320, 100);

    if (!(printerReq = CreatePrtReq()))
        cleanexit("Can't create printer request");

    if (OpenPrinter(printerReq))
        cleanexit("Can't open printer device\n");

    PDOPEN = TRUE;

    port = printerReq->ios.io_Message.mn_ReplyPort;

    usersig = 1 << window->UserPort->mp_SigBit;
    printersig = 1 << port->mp_SigBit;

```

```

rp = window->RPort;
vp = &window->WScreen->ViewPort;

DumpRPort(printerReq, rp, vp->ColorMap, vp->Modes,
  (UWORD)window->BorderLeft, (UWORD)window->BorderTop,
  (UWORD)(window->Width - window->BorderRight - window->BorderLeft),
  (UWORD)(window->Height - window->BorderBottom - window->BorderTop),
  0L, 0L,
  SPECIAL_ASPECT|SPECIAL_FULLROWS);
FOREVER
{
  signal = Wait(usersig | printersig);
  if (signal & usersig)
  {
    while(msg = (struct IntuiMessage *)GetMsg(window->UserPort))
    {
      if(msg->Class == CLOSEWINDOW) U_ABORT = TRUE;
      ReplyMsg((struct Message *)msg);
    }
  }
  if (signal & printersig)
  {
    if (printerReq->iodrp.io_Error != 0)
    {
      SetWindowTitles(window,
        prtErrorText[printerReq->iodrp.io_Error], NULL);
      Delay(150);
    }
    P_ABORT = TRUE;
    while((struct MsgPort *)GetMsg(port));
  }

  if (U_ABORT)
  {
    AbortIO((struct IORequest *)printerReq);
    WaitIO((struct IORequest *)printerReq);
  }
  if (P_ABORT | U_ABORT)
    cleanexit("");
}
}

/* open and fill a window on the Workbench screen */
struct Window *doWindow(width, height)
SHORT width, height;
{
  struct NewWindow nw =
  {
    0,0,0,0,0,1,CLOSEWINDOW,ACTIVATE|WINDOWCLOSE|WINDOWDEPTH|SMART_REFRESH|
    WINDOWDRAG|RMBTRAP,NULL,NULL,"<- Close To Abort Dump",
    NULL,NULL,-1,-1,-1,-1,WBENCHSCREEN
  };

  struct RastPort *rp;
  struct Window *wn;
  register COUNT n;
  register SHORT mc, mr;

  nw.Height = height;
  nw.Width = width;

  if(!(wn = (struct Window *)OpenWindow(&nw)))
    cleanexit("Can't open window\n");

  rp = wn->RPort;
  mc = wn->Width - (wn->BorderRight+1);
  mr = wn->Height - (wn->BorderBottom+1);

  SetDrMd(rp,0);
  SetAPen(rp, 1);
}

```

```

RectFill(rp, wn->BorderLeft, wn->BorderTop, mc, mr);

for (n = wn->BorderLeft; n < mc; n+=4)
{
    Move(rp, (mc / 2) + wn->BorderLeft + 1, (mr / 2) + wn->BorderBottom + 1);
    SetAPen(rp, 0);
    Draw(rp, n, wn->BorderTop);
    Move(rp, (mc / 2) + wn->BorderLeft + 1, (mr / 2) + wn->BorderBottom + 1);
    SetAPen(rp, 3);
    Draw(rp, n, mr);
}
for (n = wn->BorderTop; n < mr; n+=4)
{
    Move(rp, (mc / 2) + wn->BorderLeft + 1, (mr / 2) + wn->BorderBottom + 1);
    SetAPen(rp, 2);
    Draw(rp, wn->BorderLeft, n);
    Move(rp, (mc / 2) + wn->BorderLeft + 1, (mr / 2) + wn->BorderBottom + 1);
    Draw(rp, mc, n);
}
return(wn);
}

void openLibraries()
{
    if(!(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",33)))
        cleanexit("Can't open graphics.lib\n");
    if(!(IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 33)))
        cleanexit("Can't open intuition.lib\n");
}

void cleanexit(s)
UBYTE *s;
{
    struct IntuiMessage *msg;
    SHORT c;

    if (PDOPEN)
        CloseDevice((struct IORequest *)printerReq);

    if (printerReq)
        DeletePrtReq(printerReq);

    if(window)
    {
        while(msg = (struct IntuiMessage *)GetMsg(window->UserPort))
            ReplyMsg((struct Message *)msg);
        CloseWindow(window);
    }

    if(GfxBase) CloseLibrary((struct Library *)GfxBase);
    if(IntuitionBase) CloseLibrary((struct Library *)IntuitionBase);

    if (*s)
    {
        printf("%s\n", s);
        printf("<<< Hit Return to continue >>>\n"); /* just in case we're */
        c = getchar(); /* run from Workbench */
        exit(RETURN_FAIL);
    }
    exit(RETURN_OK);
}

/* the printer i/o routines we use */

union printerIO *CreatePrtReq()
{
    struct MsgPort *prtport;
    union printerIO *request;

    if (!(prtport = CreatePort(NULL,0)))

```

```

        return(0);
    if (!(request = (union printerIO *)CreateExtIO(prtport,
        sizeof(union printerIO))))
    {
        DeletePort(prtport);
        return(0);
    }
    return(request);
}

void DeletePrtReq(request)
union printerIO *request;
{
    struct MsgPort *prtport;

    prtport = request->ios.io_Message.mn_ReplyPort;
    DeleteExtIO((struct IORequest *)request);
    DeletePort(prtport);
}

int
OpenPrinter(request)
union printerIO *request;
{
    return(OpenDevice("printer.device",0,(struct IORequest *)request,0));
}

void
DumpRPort(request,rastPort, colorMap, modes, sx,sy, sw,sh, dc,dr, s)
union printerIO *request;
struct RastPort *rastPort;
struct ColorMap *colorMap;
ULONG modes;
UWORD sx, sy, sw, sh;
LONG dc, dr;
UWORD s;
{
    request->iodr.io_Command = PRD_DUMPRT;
    request->iodr.io_RastPort = rastPort;
    request->iodr.io_ColorMap = colorMap;
    request->iodr.io_Modes = modes;
    request->iodr.io_SrcX = sx;
    request->iodr.io_SrcY = sy;
    request->iodr.io_SrcWidth = sw;
    request->iodr.io_SrcHeight = sh;
    request->iodr.io_DestCols = dc;
    request->iodr.io_DestRows = dr;
    request->iodr.io_Special = s;
    SendIO((struct IORequest *)request);
}

```

## STRIP PRINTING

Strip printing is a method which allows you to print a picture that normally requires a large print buffer when there is not much memory available. This would allow, for example, a **RastPort** to be printed at a higher resolution than it was drawn in. Strip printing is done by creating a temporary **RastPort** as wide as the source **RastPort**, but not as high. The source **RastPort** is then rendered, a strip at a time, into the temporary **RastPort** which is dumped to the printer.

The height of the strip to dump must be an integer multiple of the printer's **NumRows** if a non-aspect-ratio-corrected image is to be printed.

For an aspect-ratio-corrected image, the **SPECIAL\_NOPRINT** flag will have to be used to find an **io\_DestRows** that is an integer multiple of **NumRows**. This can be done by varying the source height and asking for a **SPECIAL\_NOPRINT** dump until **io\_DestRows** holds a number that is an integer multiple of the printer's

## **NumRows.**

If SMOOTHING is to work with strip printing, a raster line above and below the actual area should be added. The line above should be the last line from the previous strip, the line below should be the first line of the next strip. Of course, the first strip should not have a line added above and the last strip should not have a line added below.

The following is a strip printing scenario for a **RastPort** which is 200 lines high.

### **First strip**

- copy source line 0 through 50 (51 lines) to strip **RastPort** lines 0 through 50 (51 lines).
- **io\_SrcY = 0, io\_Height = 50.**
- the printer device can see there is no line above the first line to dump (since **SrcY = 0**) and that there is a line below the last line to dump (since there is a 51 line **RastPort** and only 50 lines are dumped).

### **Second strip**

- copy source line 49 through 100 (52 lines) to strip **RastPort** lines 0 through 51 (52 lines).
- **io\_SrcY = 1, io\_Height = 50.**
- the printer device can see there is a line above the first line to dump (since **SrcY = 1**) and that there is a line below the last line to dump (since there is a 52 line **RastPort** and only 50 lines are dumped).

### **Third strip**

- copy source line 99 through 150 (52 lines) to strip **RastPort** lines 0 through 51 (52 lines).
- **io\_SrcY = 1, io\_Height = 50.**
- the printer device can see there is a line above the first line to dump (since **SrcY = 1**) and that there is a line below the last line to dump (since there is a 52 line **RastPort** and only 50 lines are dumped).

### **Fourth strip**

- copy source line 149 through 199 (51 lines) to strip **RastPort** lines 0 through 50 (51 lines).
- **io\_SrcY = 1, io\_Height = 50.**
- the printer device can see there is a line above the first line to dump (since **SrcY = 1**) and that there is no line below the last line to dump (since there is a 51 line **RastPort** and only 50 lines are dumped).

## **GETTING INFORMATION ABOUT THE PRINTER**

The device I/O command **PRD\_QUERY** can be used to get the status of the printer port and registers. The result is returned in two **UBYTES**. The status is returned in the **io\_Data** field. The printer type, either serial or parallel, is returned in the **io\_Actual** field.

| io_Data | BIT   | ACTIVE | FUNCTION (SERIAL DEVICE)         |
|---------|-------|--------|----------------------------------|
| LSB     | 0     | low    | reserved                         |
|         | 1     | low    | reserved                         |
|         | 2     | low    | reserved                         |
|         | 3     | low    | Data Set Ready                   |
|         | 4     | low    | Clear To Send                    |
|         | 5     | low    | Carrier Detect                   |
|         | 6     | low    | Ready To Send                    |
| MSB     | 7     | low    | Data Terminal Ready              |
|         | 8     | high   | read buffer overflow             |
|         | 9     | high   | break sent (most recent output)  |
|         | 10    | high   | break received (as latest input) |
|         | 11    | high   | transmit x-OFFed                 |
|         | 12    | high   | receive x-OFFed                  |
|         | 13-15 | high   | reserved                         |

| io_Data | BIT | ACTIVE | FUNCTION (PARALLEL DEVICE) |
|---------|-----|--------|----------------------------|
|         | 0   | high   | printer busy (offline)     |
|         | 1   | high   | paper out                  |
|         | 2   | high   | printer selected           |
|         | 3-7 |        | reserved                   |

**io\_Actual**                      1-parallel, 2-serial

A function like the following could be used to query the port status:

```

struct PStat
{
    UBYTE LSB;
    UBYTE MSB;
};

struct PStat *GetPrintStat(request)
union printerIO *request;
{
    struct PStat *status = NULL;

    request->ios.io_Command = PRD_QUERY;
    request->ios.io_Data = (APTR)status;
    DoIO((struct IORequest *)request);

    /* request->ios.io_Actual contains type of connection */
    return(status);
}

```

For more information about the printer in use, the **PrinterData** and **PrinterExtendedData** data structures can be read. These data structures are defined in *devices/prtbase.h*. The following example shows how to read them:

```

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#ifdef LATTICE
#include <stdio.h>
#include <proto/all.h>
#endif

union printerIO
{
    struct IOStdReq ios;
    struct IODRPReq iodrp;
    struct IOPrtCmdReq iopc;
};

union printerIO *printerReq;
struct PrinterData *PD;
struct PrinterExtendedData *PED;

```



```

void main(void);
extern int OpenPrinter(union printerIO *);

void main()
{
    /* open the printer device if it opened... */
    if(! (OpenPrinter(printerReq)))
    {
        /* get pointer to printer data */
        PD = (struct PrinterData *)printerReq->iodrp.io_Device;

        /* get pointer to printer extended data */
        PED = &PD->pd_SegmentData->ps_PED;

        /* let's see what's there */
        printf("PrinterName = '%s', Version=%u, Revision=%u\n",
            PED->ped_PrinterName, PD->pd_SegmentData->ps_Version,
            PD->pd_SegmentData->ps_Revision);

        printf("PrinterClass=%u, ColorClass=%u\n",
            PED->ped_PrinterClass, PED->ped_ColorClass);

        printf("MaxColumns=%u, NumCharSets=%u, NumRows=%u\n",
            PED->ped_MaxColumns, PED->ped_NumCharSets,
            PED->ped_NumRows);

        printf("MaxXDots=%lu, MaxYDots=%lu, XDotsInch=%u, YDotsInch=%u\n",
            PED->ped_MaxXDots, PED->ped_MaxYDots,
            PED->ped_XDotsInch, PED->ped_YDotsInch);

        CloseDevice((struct IORequest *)printerReq); /* close the printer device */
    }
    else
        printf("Can't open printer.device\n");
}

```

## CHANGING THE PRINTER PREFERENCES SETTINGS

The user preferences can be changed without running the Workbench Preferences tool. This can be done by referring to `PD->pd_Preferences`, which contains the latest preferences. See *intuition/preferences.h* for a description of this structure.

The application program is responsible for range checking if the user is able to change the preferences from within the application. Listed below are the printer preferences items and their valid ranges.

### Text Preferences

|                  |   |                        |
|------------------|---|------------------------|
| PrintPitch       | - | PICA, ELITE, FINE      |
| PrintQuality     | - | DRAFT, LETTER          |
| PrintSpacing     | - | SIX_LPI, EIGHT_LPI     |
| PrintLeftMargin  | - | 1 to PrintRightMargin  |
| PrintRightMargin | - | PrintLeftMargin to 999 |
| PaperLength      | - | 1 to 999               |

### Graphic Preferences

|                |   |   |
|----------------|---|---|
| PrintImage     | - | IMAGE_POSITIVE, IMAGE_NEGATIVE  |
| PrintAspect    | - | ASPECT_HORIZ, ASPECT_VERT   |
| PrintShade     | - | SHADE_BW, SHADE_GREYSCALE, SHADE_COLOR  |
| PrintThreshold | - | 1 to 15   |
| PrintFlags     | - | CORRECT_RED, CORRECT_GREEN, CORRECT_BLUE,<br>CENTER_IMAGE, IGNORE_DIMENSIONS,<br>BOUNDED_DIMENSIONS, ABSOLUTE_DIMENSIONS,<br>PIXEL_DIMENSIONS, MULTIPLY_DIMENSIONS,<br>INTEGER_SCALING, ORDERED_DITHERING,<br>HALFTONE_DITHERING, FLOYD_DITHERING,<br>ANTI_ALIAS, GREY_SCALE2 |
| PrintMaxWidth  | - | 0 to 65535  |
| PrintMaxHeight | - | 0 to 65535  |
| PrintDensity   | - | 1 to 7  |
| PrintXOffset   | - | 0 to 255  |

## ADDITIONAL NOTES ABOUT GRAPHIC DUMPS

1. When dumping a 1 bitplane image select the B&W mode in preferences. This is much faster than a grey-scale or color dump.
2. Horizontal dumps are much faster than vertical dumps because they use the blitter to read the pixel data.
3. Smoothing doubles the print time. Use it for final copy only.
4. F-S dithering doubles the print time. Ordered and half-tone dithering incur no extra overhead.
5. The lower the density, the faster the printout.
6. Friction-fed paper tends to be much more accurate than tractor-fed paper in terms of vertical dot placement (i.e., less horizontal strips or white lines).
7. Densities which use more than one pass tend to produce muddy grey-scale or color printouts. It is recommended not to choose these densities when doing a grey-scale or color dump.

## Creating a Printer Driver

Creating the printer-dependent modules for the printer device involves writing the data structures and code, compiling and assembling them, and linking to produce an Amiga binary object file. Each driver contains at least six modules: *printertag.asm*, *init.asm*, *transfer.c*, *data.c*, *dospecial.c* and *render.c*. Depending on the special capabilities of the printer driver, additional modules may be needed. For example, some standard Workbench drivers use the modules *prtready.c* and *density.c*. This section will describe the function of all the printer driver modules. At the end of this chapter source code is listed for four of the Workbench printer drivers for you to use as an example.

The first piece of the printer driver is the **PrinterSegment** structure described in *devices/prtbase.h* (this is pointed to by the BPTR returned by the LoadSeg() of the object file). The **PrinterSegment** contains the **PrinterExtendedData** (PED) structures (also described in *devices/prtbase.h*) at the beginning of the object. The PED structure contains data describing the capabilities of the printer, as well as pointers to code and other data. Here is the assembly code for a sample **PrinterSegment**, which would be linked to the beginning of the sequence of files as *printertag.asm*.

```

*****
*
*   printer device dependent code tag
*
*****
; named sections are easier to exactly place in the linked file
SECTION printer

*----- Included Files -----

INCLUDE "exec/types.i"
INCLUDE "exec/nodes.i"

INCLUDE "revision.i" ; contains VERSION & REVISION

INCLUDE "devices/prtbase.i"

*----- Imported Names -----

XREF    _Init
XREF    _Expunge
XREF    _Open
XREF    _Close
XREF    _CommandTable
XREF    _DoSpecial
XREF    _Render
XREF    _ExtendedCharTable

*----- Exported Names -----

XDEF    _PEDData

*****

; in case anyone tries to execute this
MOVEQ   #0,D0
RTS

DC.W    VERSION      ; must be 35 (for V1.3)
DC.W    REVISION     ; your own revision number

_PEDData:
DC.L    printerName  ; pointer to the printer name
DC.L    _Init        ; pointer to the initialization code
DC.L    _Expunge     ; pointer to the expunge code
DC.L    _Open        ; pointer to the open code
DC.L    _Close       ; pointer to the close code
DC.B    PPC_COLORGFX ; PrinterClass
DC.B    PCC_YMCB     ; ColorClass
DC.B    136          ; MaxColumns
DC.B    10           ; NumCharSets
DC.W    8            ; NumRows
DC.L    1632         ; MaxXDots
DC.L    0            ; MaxYDots
DC.W    120          ; XDotsInch
DC.W    72           ; YDotsInch
DC.L    _CommandTable ; pointer to Command Strings
DC.L    _DoSpecial   ; pointer to Command Code function
DC.L    _Render      ; pointer to Graphics Render function
DC.L    30           ; Timeout

DC.L    _ExtendedCharTable ; pointer to 8BitChar table

DS.L    1            ; Flag for PrintMode (reserve space)
DC.L    0            ; pointer to ConvFunc (char conversion function),
; usually null

printerName:
DC.B    'EpsonX'
DC.B    0
END

```

The printer name should be the brand name of the printer that is available for use by programs wishing to be specific about the printer name in any diagnostic or instruction messages. The four functions at the top of the structure are used to initialize this printer-dependent code:

**(\*(PED->ped\_Init))(PD);**

This is called when the printer-dependent code is loaded and provides a pointer to the printer device for use by the printer-dependent code. It can also be used to open up any libraries or devices needed by the printer-dependent code.

**(\*(PED->ped\_Expunge))();**

This is called immediately before the printer-dependent code is unloaded, to allow it to close any resources obtained at initialization time.

**(\*(PED->ped\_Open))(ior);**

This is called in the process of an **OpenDevice()** call, after the Preferences are read and the correct primitive I/O device (parallel or serial) is opened. It must return zero if the open is successful, or non-zero to terminate the open and return an error to the user.

**(\*(PED->ped\_Close))(ior);**

This is called in the process of a **CloseDevice()** call to allow the printer-dependent code to close any resources obtained at open time.

The **pd\_** variable provided as a parameter to the initialization call is a pointer to the **PrinterData** structure described in *devices/prtbase.h* and *devices/prtbase.i*. This is also the same as the **io\_Device** entry in printer I/O requests.

#### **pd\_SegmentData**

This points back to the **PrinterSegment**, which contains the PED.

#### **pd\_PrintBuf**

This is available for use by the printer-dependent code—it is not otherwise used by the printer device.

**(\*pd\_PWrite)(data, length);**

This is the interface routine to the primitive I/O device. This routine uses two I/O requests to the primitive device, so writes are double-buffered. The data parameter points to the byte data to send, and the length is the number of bytes.

**(\*pd\_PBothReady)();**

This waits for both primitive I/O requests to complete. This is useful if your code does not want to use double buffering. If you want to use the same data buffer for successive **pd\_PWrites**, you must separate them with a call to this routine.

#### **pd\_Preferences**

This is the copy of Preferences in use by the printer device, obtained when the printer was opened.

The timeout field is the number of seconds that an I/O request from the printer device will remain posted and unsatisfied to the primitive I/O device (parallel or serial) before the timeout requester is presented to the user. This value should be large enough to avoid the requester during normal printing.

The **PrintMode** field is a flag which indicates whether text has been printed or not (1 means printed, 0 means not printed). This flag is used in drivers for page oriented printers to indicate that there is no alphanumeric data waiting for a formfeed.

## WRITING A GRAPHICS PRINTER DRIVER

Designing the graphics portion of a custom printer driver consists of two steps: writing the printer-specific **Render()**, **Transfer()** and **SetDensity()** functions, and replacing the printer-specific values in *printertag.asm*. **Render()**, **Transfer()** and **SetDensity()** comprise *render.c*, *transfer.c*, and *density.c* modules, respectively.

.IX **Render()**

A printer that does *not* support graphics has a very simple form of **Render()**; it returns an error. Here is sample code for **Render()** for a non-graphics printer (such as an Alphacom or Diablo 630):

```
#include "exec/types.h"
#include "devices/printer.h"
int Render()
{
    return (PDERR_NOTGRAPHICS);
}
```

The following section describes the contents of a typical driver for a printer that does support graphics.

### RENDER()

This function is the main printer-specific code module and consists of seven parts referred to here as cases:

- Pre-Master initialization
- Master initialization
- Putting the pixels in a buffer
- Dumping a pixel buffer to the printer
- Clearing and initializing the pixel buffer
- Closing down
- Switching to the next color

For each case, **Render()** receives four long variables as parameters: *ct*, *x*, *y* and *status*. These parameters are described below for each of the seven cases that **Render()** must handle.

#### Pre-Master initialization (Case 5)

Parameters:

*ct* - 0 or pointer to the **IODRPreq** structure passed to **PCDumpRPort**  
*x* - **io\_Special** flag from the **IODRPreq** structure  
*y* - 0

When the printer device is first opened, **Render()** is called with *ct* set to 0, to give the driver a chance to set up the density values before the actual graphic dump is called.

The parameter passed in *x* will be the **io\_Special** flag which contains the density and other SPECIAL flags. The only flags used at this point are the DENSITY flags, all others should be ignored. Never call PWrite during this case. When you are finished handling this case, return PDERR\_NOERR.

#### Master initialization (Case 0).

##### Parameters:

*ct* - pointer to a **IODRPreq** structure  
*x* - width (in pixels) of printed picture  
*y* - height (in pixels) of printed picture

#### NOTE

At this point the printer device has already checked that the values are within range for the printer. This is done by checking values listed in *printertag.asm*.

The *x* and *y* value should be used to allocate enough memory for a command and data buffer for the printer. If the allocation fails, PDERR\_BUFFERMEMORY should be returned. In general, the buffer needs to be large enough for the commands and data required for one pass of the print head. These typically take the following form:

<start gfx cmd> <data> <end gfx cmd>

The <start gfx cmd> should contain any special, one-time initializations that the printer might require such as:

- a) Carriage Return - some printers start printing graphics without returning the printhead. Sending a CR assures that printing will start from the left edge.
- b) Uni-Directional - some printer which have a bi-directional mode produce non-matching vertical lines during a graphics dump, giving a wavy result. To fix this, your driver could set the printer to uni-directional mode.
- c) Clear margins - some printers force graphic dumps to be done within the text margins, thus they should be cleared.
- d) Other commands - enter the graphics mode, set density, etc.

In addition to the memory for commands and data, a multi-pass color printer must allocate enough buffer space for each of the different color passes.

The printer should never be reset during the master initialization case. This will cause problems during multiple dumps. Also, the pointer to the **IODRPreq** structure in *ct* should not be used except for those rare printers which require it to do the dump themselves. Return the PDERR\_TOOKCONTROL error in that case so that the printer device can exit gracefully.

The example *render.c* functions listed at the end of this chapter use double buffering to reduce the dump time which is why the AllocMem() calls are for BUFSIZE *times two*, where BUFSIZE represents the amount of memory for one entire print cycle. However, contrary to the example source code, allocating the two buffers independently of each other is recommended. A request for one large block of contiguous memory might be refused. Two smaller requests are more likely to be granted.

### Putting the pixels in a buffer (Case 1).

#### Parameters:

ct - pointer to a `PrtInfo` structure.  
x - PCM color code (if the printer is `PCC_MULTI_PASS`).  
y - printer row # (the range is 0 to pixel height - 1).

In this case, you are passed an entire row of YMCB intensity values (Yellow, Magenta, Cyan, Black). To handle this case, you call the `Transfer()` function in the *transfer.c* module. You should return `PDERR_NOERR` after handling this case. The PCM-defines for the x parameter from the file *devices/prtgfx.h* are `PCMYELLOW`, `PCMMAGENTA`, `PCMCYAN` and `PCMBLACK`.

### Dumping a pixel buffer to the printer (Case 2).

#### Parameters:

ct - 0  
x - 0  
y - # of rows sent (the range is 1 to `NumRows`).

At this point the data can be Run Length Encoded (RLE) if your printer supports it. If the printer doesn't support RLE, the data should be white-space stripped. This involves scanning the buffer from back to front for the first occurrence of a non-zero value. Only the data up to the first non-zero value should be sent to the printer. This will significantly reduce print times.

The value of y can be used to advance the paper the appropriate number of pixel lines, if your printer supports that feature. This helps prevent white lines from appearing between graphic dumps.

You can also do post-processing on the buffer at this point. For example, if your printer uses the hexadecimal number `$03` as a command and requires the sequence `$03 $03` to send `$03` as data, you would probably want to scan the buffer and expand any `$03`'s to `$03 $03` during this case. Of course, you'll need to allocate space somewhere in order to expand the buffer.

The error from `PWrite` should be returned after this call.

### Clearing and initializing the pixel buffer (Case 3)

#### Parameters:

ct - 0  
x - 0  
y - 0

The printer driver does not send blank pixels so you must initialize the buffer to the value your printer uses for blank pixels (usually 0). Clearing the buffer should be the same for all printers. Initializing the buffer is printer specific, and it includes placing the printer-specific control codes in the buffer ahead of and behind where the data will go.

This call is made before each case 2 call. Clear your active print buffer—remember you are double buffering—and initialize it if necessary. After this call `PDERR_NOERR` should be returned.

## Closing Down (Case 4).

### Parameters:

ct - error code  
x - io\_Special flag from the IODRPreReq structure  
y - 0

This call is made at the end of the graphic dump, or if the graphic dump was cancelled for some reason. At this point you should free the printer's buffer memory. Determine if memory was allocated by checking if **PD->pd\_PrintBuf** is not NULL. If memory was allocated then you must wait for the print buffers to clear (by calling **PBothReady**) and then de-allocate the memory. If the printer—usually a page oriented printer—requires a page eject command, it can be given here. Before you do, though, you should check the **SPECIAL\_NOFORMFEED** bit in x. Don't issue the command if it is set.

If the error condition in ct is **PDERR\_CANCEL** then you should not **PWrite**. This error indicates that the user is trying to cancel the dump for whatever reason. Each additional **PWrite** will generate another printer trouble requester. Obviously, this is not desirable.

During this render case **PWrite** could be used to:

- a) reset the line spacing. If the printer doesn't have an advance 'n' dots command, then you'll probably advance the paper by changing the line spacing. If you do, then you should set it back to either 6 or 8 lpi (depending on Preferences).
- b) set bi-directional mode if you selected uni-directional mode in render case 0.
- c) set black text; some printers print the text in the last color used, even if it was in graphics mode.
- d) restore the margins if you cancelled the margins in render case 0.
- e) any other command needed to exit the graphics mode, eject the page, etc.

Either **PDERR\_NOERR** or the error from **PWrite** should be returned after this call.

## Switching to the next color (Case 6)

This call provides support for printers which require that colors be sent in separate passes. When this call is made, you should instruct the printer to advance its color panel. This case is only needed for printers of the type **PCC\_MULTI\_PASS**, such as the CalComp ColorMaster.

## TRANSFER()

**Transfer()** dithers and renders an entire row of pixels passed to it by the **Render()** function. When **Transfer()** gets called, it is passed 5 parameters; a pointer to a **PrtInfo** structure in **PInfo**, the row number in y, a pointer to the buffer in **ptr**, a pointer to the color buffers in **colors** and the buffer offset in **BufOffset** for interleaved printing.

The dithering process of **Transfer()** might entail thresholding, grey-scale dithering, or color-dithering each destination pixel. If **PInfo->pi\_threshold** is non-zero, then the dither value is **PInfo->pi\_threshold^15**. If **PInfo->pi\_threshold** is zero, then the dither value is computed by:



```
*(PInfo->pi_dmatrix + ((y & 3) * 2) + (x & 3))
```

where *x* is initialized to **PInfo->pi\_xpos** and is incremented for each of the destination pixels. Since the printer device uses a 4x4 dither matrix, you must calculate the dither value exactly as given above. Otherwise, your driver will be non-standard and the results will be unpredictable.

The **Transfer()** function renders by putting a pixel in the print buffer based on the dither value. If the intensity value for the pixel is greater than the dither value as computed above, then the pixel should be put in the print buffer. If it is less than, or equal to the dither value, it should be skipped to process the next pixel.

| Printer ColorClass | Type of Dithering | Rendering logic   |
|--------------------|-------------------|---|
| PCC_BW             | Thresholding      | Test the black value against the threshold value to see if you should render a black pixel.   |
|                    | Grey Scale        | Test the black value against the dither value to see if you should render a black pixel.  |
|                    | Color             | NA (See HP_LaserJet transfer.c file)  |
| PCC_YMC            | Thresholding      | Test the black value against the threshold value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.   |
|                    | Grey Scale        | Test the black value against the dither value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.  |
|                    | Color             | Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.  |
| PCC_YMCB           | Thresholding      | Test the black value against the threshold value to see if you should render a black pixel.   |
|                    | Grey Scale        | Test the black value against the dither value to see if you should render a black pixel.  |
|                    | Color             | Test the black value against the dither value to see if you should render a black pixel. If black is not rendered, then test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan. (See EpsonQ, EpsonX or Xerox_4020 transfer.c file) |
| PCC_YMC_BW         | Thresholding      | Test the black value against the threshold value to see if you should render a black pixel.   |
|                    | Grey Scale        | Test the black value against the dither value to see if you should render a black pixel.  |
|                    | Color             | Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.  |

In general, if black is rendered for a specific printer dot, then the YMC values should be ignored, since the combination of YMC is black. It is recommended that the printer buffer be constructed so that the order of colors printed is yellow, magenta, cyan and black, to prevent smudging and minimize color contamination on ribbon color printers.

The example *transfer.c* files are provided in C for demonstration only. Writing this module in assembler can cut the time needed for a graphic dump in half. The EpsonX *transfer.asm* file is an example of this.

## SETDENSITY()

**SetDensity()** is a short function which implements multiple densities. **SetDensity()** is called in the Pre-master initialization case of the **Render()** function. It is passed the density code in **density\_code**. This is used to select the desired density or, if the user asked for a higher density than is supported, the maximum density available. **SetDensity()** should also handle narrow and wide tractor paper sizes.

Densities below 80 dpi should not be supported in **SetDensity()**, so a minimum of 640 dots across for a standard 8.5x11-inch paper is guaranteed. This results in a 1-1 correspondence of dots on the printer to dots on the screen in standard screen sizes. The *density.c* example for the HP LaserJet is an exception to this rule. Its minimum density is 75 dpi, since the original HP LaserJet has too little memory to output a full page at a higher density.

## PRINTERTAG.ASM

For a graphic printer the printer-specific values that need to be filled in in *printertag.asm* are as follows:

### MaxXDots

the maximum number of dots the printer can print across the page.

### MaxYDots

the maximum number of dots the printer can print down the page. Generally, if the printer supports roll or form feed paper, this value should be 0 indicating that there is no limit. If the printer has a definite y dots maximum (as a laser printer does), this number should be entered here.

### XDotsInch

the dot density in x (supplied by **SetDensity()**, if it exists).

### YDotsInch

the dot density in y (supplied by **SetDensity()**, if it exists).

### PrinterClass

the printer class of the printer. Current choices are:

|                       |   |
|-----------------------|---|
| <b>PPC_BWALPHA</b>    | - black&white alphanumeric, no graphics.  |
| <b>PPC_BWGFX</b>      | - black&white (only) graphics.            |
| <b>PPC_COLORALPHA</b> | - color alphanumeric, no graphics.        |
| <b>PPC_COLORGFX</b>   | - color (and maybe black&white) graphics. |

### ColorClass

the color class the printer falls into. Current choices are:

|                   |  |
|-------------------|--|
| <b>PCC_BW</b>     | - Black&White only                                 |
| <b>PCC_YMC</b>    | - Yellow Magenta Cyan only.                        |
| <b>PCC_YMC_BW</b> | - Yellow Magenta Cyan or Black&White, but not both |
| <b>PCC_YMCB</b>   | - Yellow Magenta Cyan Black                        |
| <b>PCC_WB</b>     | - White&Black only, 0 is BLACK                     |
| <b>PCC_BGR</b>    | - Blue Green Red                                   |
| <b>PCC_BGR_WB</b> | - Blue Green Red or Black&White                    |
| <b>PCC_BGRW</b>   | - Blue Green Red White                             |

#### **NumRows**

the number of pixel rows printed by one pass of the print head. This number is used by the non-printer-specific code to determine when to make a render case 2 call to you. You have to keep this number in mind when determining how big a buffer you'll need to store one print cycle's worth of data.

### **WRITING AN ALPHANUMERIC PRINTER DRIVER**

The alphanumeric portion of the printer driver is designed to convert ANSI x3.64 style commands into the specific escape codes required by each individual printer. For example, the ANSI code for underline-on is ESC[4m. The Commodore MPS-1250 printer would like a ESC[-1 to set underline-on. The HP LaserJet accepts ESC[&dD as a start underline command. By using the printer driver all printers may be handled in a similar manner.

There are two parts to the alphanumeric portion of the printer driver: the **CommandTable** data table and the **DoSpecial()** routine.

#### **Command Table**

The **CommandTable** is used to convert all escape codes that can be handled by simple substitution. It has one entry per ANSI command supported by the printer driver. When you are creating a custom **CommandTable**, you must maintain the order of the commands in the same sequence as that shown in *printer.h* and *printer.i*. By placing the specific codes for your printer in the proper position, the conversion takes place automatically.

#### **NOTE**

If the code for your printer requires a decimal 0 (an ASCII NULL character), you enter this NULL into the **CommandTable** as octal 376 (decimal 254).

Placing an octal value of 377 (255 decimal) in a position in the command table indicates to the printer device that no simple conversion is available on this printer for this ANSI command. For example, if a daisy-wheel printer does not have a foreign character set, 377 octal (255 decimal) is placed in that position in the command table. However, 377 in a position can also mean that the ANSI command is to be handled by code located in the **DoSpecial()** function. For future compatibility *all* V1.3 printer commands should be present in the command table, and those not supported by the printer filled with the dummy entry 377.

#### **DoSpecial()**

The **DoSpecial()** function is meant to implement all the ANSI functions that cannot be done by simple substitution, but can be handled by a more complex sequence of control characters sent to the printer. These are functions that need parameter conversion, read values from Preferences, and so on. Complete routines can also be placed in

*dospecial.c*. For instance, in a driver for a page oriented-printer such as the HP LaserJet, the dummy Close() routine from the *init.asm* file would be replaced by a real Close() routine in *dospecial.c*. This close routine would handle ejecting the paper after text has been sent to the printer and the printer has been closed.

The DoSpecial() function is set up as follows:

```
#include "exec/types.h"
#include "devices/printer.h"
#include "devices/prtbase.h"

extern struct PrinterData *PD;

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Parms)
UBYTE outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Parms[];
{
    /* code begins here... */
}
```

where

**command**

points to the command number. The *devices/printer.h* file contains the definitions for the routines to use (aRIN is initialize, and so on).

**vline**

points to the value for the current line position.

**currentVMI**

points to the value for the current line spacing.

**crlfFlag**

points to the setting of the "add line feed after carriage return" flag.

**Parms**

contain whatever parameters were given with the ANSI command.

**outputBuffer**

points to the memory buffer into which the converted command is returned.

Almost every printer will require an aRIN (initialize) command in DoSpecial(). This command reads the printer settings from Preferences and creates the proper control sequence for the specific printer. It also returns the character set to normal (not italicized, not bold, and so on). Other functions depend on the printer.

Certain functions are implemented both in the CommandTable and in the DoSpecial() routine. These are functions such as superscript, subscript, PLU (partial line up), and PLD (partial line down), which can often be handled by a simple conversion. However, certain of these functions must also adjust the printer device's line-position variable.

**NOTE**

Some printers lose data when sent their own reset command. For this reason it is recommended that if the printer's own reset command is used, PD->pd\_PWaitEnabled should be defined to be a character that the printer will not print. This character should be put in the reset string before and after the reset character(s) in the command table.

In the EpsonX[CBM\_MPS-1250] **DoSpecial()** function you'll see

```
if (*command == aRIS)
{
    /* reset command */
    PD->pd_PWaitEnabled = 253; /* identical to \375 */
}
```

while in the command table the string for reset is defined as "\375\033@\375". This means that when the printer device outputs the reset string "\033@", it will first see the "\375", wait a second and output the reset string. While the printer is resetting, the printer device gets the second "\375" and waits another second. This ensures that no data will be lost if a reset command is embedded in a string.

## Printertag.asm

For an alphanumeric printer the printer-specific values that need to be filled in in *printertag.asm* are as follows:

### MaxColumns

the maximum number of columns the printer can print across the page.

### NumCharSets

the number of character sets which can be selected.

### 8BitChars

a pointer to an extended character table. If the field is null, the default table will be used.

### ConvFunc

a pointer to a character conversion routine. If the field is null, no conversion routine will be used.

## Extended Character table

The **8BitChars** field could contain a pointer to a table of characters for the ASCII codes \$a0 to \$ff. The symbols for these codes are shown in the "IFF" chapter of the *ROM Kernel Reference Manual: Includes & Autodocs*. If this field contains a NULL, it means no specific table is provided for the driver, and the default table is to be used instead.

Care should be taking when generating this table, because of the way the table is parsed by the printer device. Valid expressions in the table include \011 where 011 is an octal number, \000 for null and \n where n is a 1-3 digit decimal number. To enter an actual backslash in the table requires the somewhat awkward \\\\. As an example, here is a list of the first entries of the EpsonxX[CBM\_MPS-1250] table:

```
char *ExtendedCharTable[] =
{
    " ", /* NBSPP*/
    "\033R\007[\033R\0", /* i */
    "c\010|", /* c| */
    "\033R\003#\033R\0", /* L- */
    "\033R\005$\033R\0", /* o */
    "\033R\010\\\033R\0", /* Y- */
    "|", /* | */
    "\033R\002@\033R\0", /* SS */
    "\033R\001~\033R\0", /* " */
    "c", /* copyright */
    "\033S\00a\010_\033T", /* a_ */
    "<", /* << */
    "-", /* - */
}
```

```

    "-",          /* SHY */
    "r",          /* registered trademark */
    "-",          /* - */
    /* more entries go here */
};

```

## Character Conversion Routine

The **ConvFunc** field contains a pointer to a character conversion function that allows you to selectively translate any character to a combination of other characters. If no translation conversion is necessary (for most printers it isn't) the field should contain a null.

**ConvFunc()** arguments are a pointer to a buffer, the character currently processed, and a CR/LF flag. The **ConvFunc()** function should return a -1 if no conversion has been done. If the character is not to be added to the buffer, a 0 can be returned. If any translation is done, the number of characters added to the buffer must be returned.

Besides simple character translation, the **ConvFunc()** function can be used to add features like underlining to a printer which doesn't support them automatically. A global flag could be introduced that could be set or cleared by the **DoSpecial()** function. Depending on the status of the flag the **ConvFunc()** routine could, for example, put the character, a backspace and an underline character in the buffer and return 3, the number of characters added to the buffer.

The **ConvFunc()** function for this could look like the following example:

```

#define DO_UNDERLINE    0x01
#define DO_BOLD         0x02
/* etc */

external short myflags;

int
ConvFunc(buffer, c, crlf_flag)
char *buffer, c;
int crlf_flag
{
    int nr_of_chars_added = 0;

    /* for this example we only do this for chars in the 0x20-0x7e range */
    /* Conversion of ESC (0x1b) and CSI (0x9b) is NOT recommended */

    if (c > 0x1f && c < 0x7f)
    {
        /* within space - ~ range ? */
        if (myflags & DO_UNDERLINE)
        {
            *buffer++ = c;          /* the character itself */
            *buffer++ = 0x08;       /* a backspace */
            *buffer++ = '_';        /* an underline char */
            nr_of_chars_added = 3;  /* added three chars to buffer */
        }
        if (myflags & DO_BOLD)
        {
            if (nr_of_chars_added)
            {
                /* already have added something */
                *buffer++ = 0x08;    /* so we start with backspace */
                ++nr_of_chars_added; /* and increment the counter */
            }
            *buffer++ = c;
            *buffer++ = 0x08;
            *buffer++ = c;
            ++nr_of_chars_added;
            if (myflags & DO_UNDERLINE)
            {
                /* did we do underline too? */
                *buffer++ = 0x08;    /* then backspace again*/
            }
        }
    }
}

```

```

        *buffer++ = ' ';          /* (printer goes crazy by now) */
        nr_of_chars_added += 2;   /* two more chars */
    }
}
if (nr_of_chars_added)
    return(nr_of_chars_added);    /* total nr of chars we added */
else
    return(-1);                  /* we didn't do anything */
}

```

In **DoSpecial()** the flagbits could be set or cleared, with code like the following:

```

if (*command == aRIS)          /* reset command */
    myflags = 0;               /* clear all flags */

if (*command == aRIN)          /* initialize command */
    myflags = 0;

if (*command == aSGR0)         /* 'PLAIN' command */
    myflags = 0;

if (*command == aSGR4)         /* underline on */
    myflags |= DO_UNDERLINE;   /* set underline bit */

if (*command == aSGR24)        /* underline off */
    myflags &= ~DO_UNDERLINE;  /* clear underline bit */

if (*command == aSGR1)         /* bold on */
    myflags |= DO_BOLD;        /* set bold bit */

if (*command == aSGR22)        /* bold off */
    myflags &= ~DO_BOLD;       /* clear bold bit */

```

Try to keep the expansions to a minimum, so the throughput will not be slowed down too much, and to reduce the possibility of data overrunning the printer device buffer.

## TESTING THE PRINTER DRIVER

Before releasing a printer driver it should be thoroughly tested. Though labor intensive, the alphanumeric part of a driver can be easily tested. The graphics part is more difficult. Following are some recommendations on how to test this part.

Start with a black and white (threshold 8), grey scale and color dump of the same picture. The color dump should be in color of course. The grey scale dump should be like the color dump, except it will consist of patterns of black dots. The black and white dump will have solid black and solid white areas.

Next do a dump with the **DestX** and **DestY** dots set to an even multiple of the **XDotsInch** and **YDotsInch** for the printer. For example, if the printer has a resolution of 120 x 144 dpi, a 480 x 432 dump could be done. This should produce a printed picture which covers 4 x 3 inches on paper. If the width of the picture is off, then the wrong value for **XDotsInch** has been put in *printertag.asm*. If the height of the picture is off, the wrong value for **YDotsInch** is in *printertag.asm*.

Do a color dump as wide as the printer can handle with the density set to 7.

Make sure that the printer doesn't force graphic dumps to be done within the text margins. This can easily be tested by setting the text margins to 30 and 50, the pitch to 10 cpi and then doing a graphic dump wider than 2 inches. The dump should be left justified and as wide as you instructed. If the dump starts at character position 30 and is cut off at position 50, the driver will have to be changed. These changes involve clearing the margins before the dump (Case 0) and restoring the margins after the dump (Case 4). An example of this is present in every **Render()** source

example listed at the end of this chapter.

#### NOTE

Before the graphic dump, some text must be sent to the printer to have the printer device initialize the printer. The "text" sent does not have to contain any printable characters (i.e., you can send a carriage return or other control characters).

As a final test, construct an image with a white background that has objects in it surrounded by white space. Dump this as black&white, grey scale and color printout. This will test the white-space stripping or RLE and the ability of the driver to handle null lines. The white data areas should be separated by at least as many lines of white space as the NumRows value in the *printertag.asm* file.

## Example Printer Driver Source Code

As an aid in writing printer drivers, source code for four different classes of printers is supplied. All drivers have been successfully generated with Lattice C 5.04 and Lattice Assembler 5.04. The example drivers are:

|             |  |
|-------------|--|
| EpsonX      | A YMCB, 8 pin, multi-density interleaved printer.    |
| EpsonQ      | A YMCB, 24 pin, multi-density printer.               |
| HP_Laserjet | A black&white, multi-density, page-oriented printer. |
| Xerox_4020  | A YMCB, color inkjet printer with RLE.               |

## macros.i

All printer drivers use the following include file *macros.i* for *init.asm*.

```
*****
*
* Copyright 1985, Commodore-Amiga Inc. All rights reserved.
* No part of this program may be reproduced, transmitted,
* transcribed, stored in retrieval system, or translated into
* any language or computer language, in any form or by any
* means, electronic, mechanical, magnetic, optical, chemical,
* manual or otherwise, without the prior written permission of
* Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
* Pennsylvania, 19380
*
*****
*
* printer device macro definitions
*
*****
*----- external definition macros -----
XREF_EXE      MACRO
XREF          _LVO\1
ENDM

XREF_DOS      MACRO
XREF          _LVO\1
ENDM
```



```
XREF_GFX      MACRO
XREF          _LVO\1
ENDM
```

```
XREF_ITU      MACRO
XREF          _LVO\1
ENDM
```

```
*----- library dispatch macros -----
```

```
CALLEXE      MACRO
CALLLIB _LVO\1
ENDM
```

```
LINKEXE      MACRO
LINKLIB _LVO\1,_SysBase
ENDM
```

```
LINKDOS      MACRO
LINKLIB _LVO\1,_DOSBase
ENDM
```

```
LINKGFX      MACRO
LINKLIB _LVO\1,_GfxBase
ENDM
```

```
LINKITU      MACRO
LINKLIB _LVO\1,_IntuitionBase
ENDM
```

# EPSONX

For the EpsonX driver, both the assembly and C version of Transfer() are supplied. In the Makefile the (faster) assembly version is used to generate the driver.

The EpsonX driver can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
CFLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
LINK = lc:blink
LIB = lib:amiga.lib+lib:lc.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o+density.o
TARGET = EpsonX

    @$(LC) $(CFLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o density.o transfer.o
    @$(LINK) <WITH <
    FROM $(OBJ)
    TO $(TARGET)
    LIBRARY $(LIB)
    NODEBUG SC SD VERBOSE MAP $(TARGET).map H
    <

init.o: init.asm
    @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm epsonx_rev.i
    @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.asm
    @$(ASM) $(ASMFLAGS) transfer.asm

dospecial.o: dospecial.c

data.o: data.c

density.o: density.c

render.o: render.c

install:
    @copy $(TARGET) to devs:printers
```

## EPSONX: PRINTERTAG.ASM

```
TTL      '$Header: printer.2,v 1.3 89/11/07 14:36:56 carolyn Exp $'
*****
*
*   Copyright 1985, Commodore-Amiga Inc.   All rights reserved.
*   No part of this program may be reproduced, transmitted,
*   transcribed, stored in retrieval system, or translated into
*   any language or computer language, in any form or by any
*   means, electronic, mechanical, magnetic, optical, chemical,
*   manual or otherwise, without the prior written permission of
*   Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
*   Pennsylvania, 19380
*
*****
```

```

*
* printer device dependent code tag
*
* Source Control
* -----
* $Header: printer.2,v 1.3 89/11/07 14:36:56 carolyn Exp $
*
* $Locker: mks $
*
* $Log: printer.2,v $
* Revision 1.3 89/11/07 14:36:56 carolyn
* changed Loas Gatos address to WC
*
* Revision 1.2 89/11/03 05:28:05 dan
* Added corrections from QA
* Corrected grammar, etc.
* Dan B.
*
* Revision 1.1 89/10/31 16:05:17 ken
* Initial revision
*
* Revision 1.2 88/04/14 12:03:35 daveb
* V1.3 Gamma 11 release
*
* Revision 1.1 87/10/27 15:30:30 daveb
* V1.3 gamma 1 check-in
*
* Revision 1.0 87/08/20 14:10:02 daveb
* added to rcs
*
* Revision 1.0 87/08/20 13:27:35 daveb
* added to rcs
*
* Revision 1.3 87/08/03 11:05:54 daveb
* added null ptr to char conversion function at end of table
*
* Revision 1.2 87/07/30 10:35:12 daveb
* added 'DS.L 1' at end to reserve space for PrintMode
*
* Revision 1.1 87/07/21 11:37:42 daveb
* added 'PPC_VERSION_2' to PrinterClass
*
* Revision 1.0 87/07/21 11:37:10 daveb
* added to rcs
*
* Revision 32.6 86/06/30 21:07:52 andy
* *** empty log message ***
*
* Revision 32.5 86/06/30 20:54:48 andy
* enabled 8 bit characters
*
* Revision 32.4 86/06/11 16:16:44 andy
* *** empty log message ***
*
* Revision 32.3 86/06/10 12:57:11 andy
* Corrected printer name
*
* Revision 32.2 86/02/12 18:16:13 kodiak
* YDotsInch -> 72
*
* Revision 32.1 86/02/10 14:32:51 kodiak
* add null 8BitChars field
*
* Revision 32.0 86/02/10 14:22:56 kodiak
* added to rcs for updating
*
* Revision 1.1 85/10/09 23:57:45 kodiak
* replace reference to pdata w/ prtbased
*
* Revision 1.0 85/10/09 23:57:39 kodiak
* added to rcs for updating in version 1
*

```

```

* Revision 25.1 85/06/16 01:02:15 kodiak
* *** empty log message ***
*
* Revision 25.0 85/06/15 06:40:00 kodiak
* added to rcs
*
* Revision 25.0 85/06/13 18:53:36 kodiak
* added to rcs
*
*

```

```

*****

```

```

SECTION          printer

```

```

*----- Included Files -----

```

```

INCLUDE          "exec/types.i"
INCLUDE          "exec/nodes.i"
INCLUDE          "exec/strings.i"

INCLUDE          "epsonX_rev.i"

INCLUDE          "devices/prtbase.i"

```

```

*----- Imported Names -----

```

```

XREF             _Init
XREF             _Expunge
XREF             _Open
XREF             _Close
XREF             _CommandTable
XREF             _PrinterSegmentData
XREF             _DoSpecial
XREF             _Render
XREF             _ExtendedCharTable

```

```

*----- Exported Names -----

```

```

XDEF             _PEDData

```

```

*****

```

```

MOVEQ    #0,D0          ; show error for OpenLibrary()
RTS
DC.W     VERSION
DC.W     REVISION

_PEDData:
DC.L     printerName
DC.L     _Init
DC.L     _Expunge
DC.L     _Open
DC.L     _Close
DC.B     PPC_COLORGFX    ;PrinterClass
DC.B     PCC_YMCB        ; ColorClass
DC.B     136             ; MaxColumns
DC.B     10              ; NumCharSets
DC.W     8               ; NumRows
DC.L     1632            ; MaxXDots
DC.L     0               ; MaxYDots
DC.W     120             ; XDotsInch
DC.W     72              ; YDotsInch
DC.L     _CommandTable   ; Commands
DC.L     _DoSpecial
DC.L     _Render
DC.L     30              ; Timeout
DC.L     _ExtendedCharTable ; 8BitChars
DS.L     1               ; PrintMode (reserve space)
DC.L     0               ; ptr to char conversion function

printerName:
DC.B     "EwoutTest"
DC.B     0

```

END

## EPSONX\_REV.I

VERSION EQU 35  
REVISION EQU 0

## EPSONX: INIT.ASM

```
TTL '$Header: printer.2,v 1.3 89/11/07 14:36:56 carolyn Exp $'
*****
*
* Copyright 1985, Commodore-Amiga Inc. All rights reserved.
* No part of this program may be reproduced, transmitted,
* transcribed, stored in retrieval system, or translated into
* any language or computer language, in any form or by any
* means, electronic, mechanical, magnetic, optical, chemical,
* manual or otherwise, without the prior written permission of
* Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
* Pennsylvania, 19380
*
*****
*
* printer device functions
*
* Source Control
* -----
* $Header: printer.2,v 1.3 89/11/07 14:36:56 carolyn Exp $
*
* $Locker: mks $
*
* $Log: printer.2,v $
* Revision 1.3 89/11/07 14:36:56 carolyn
* changed Loas Gatos address to WC
*
* Revision 1.2 89/11/03 05:28:05 dan
* Added corrections from QA
* Corrected grammar, etc.
* Dan B.
*
* Revision 1.1 89/10/31 16:05:17 ken
* Initial revision
*
* Revision 1.1 88/04/14 12:03:14 daveb
* V1.3 Gamma 11 release
*
* Revision 1.0 87/08/20 14:10:17 daveb
* added to rcs
*
* Revision 1.1 85/10/09 19:27:20 kodiak
* remove _stdout variable
*
* Revision 1.0 85/10/09 19:23:23 kodiak
* added to rcs for updating in version 1
*
* Revision 25.0 85/06/16 01:01:22 kodiak
* added to rcs
*
*****
*
SECTION printer
*----- Included Files -----
```

```

INCLUDE      "exec/types.i"
INCLUDE      "exec/nodes.i"
INCLUDE      "exec/lists.i"
INCLUDE      "exec/memory.i"
INCLUDE      "exec/ports.i"
INCLUDE      "exec/libraries.i"

INCLUDE      "macros.i"

```

\*----- Imported Functions -----\*

```

XREF_EXE      CloseLibrary
XREF_EXE      OpenLibrary
XREF          _AbsExecBase

```

```

XREF          _PEDData

```

\*----- Exported Globals -----\*

```

XDEF          _Init
XDEF          _Expunge
XDEF          _Open
XDEF          _Close
XDEF          _PD
XDEF          _PED
XDEF          _SysBase
XDEF          _DOSBase
XDEF          _GfxBase
XDEF          _IntuitionBase

```

\*\*\*\*\*

```

SECTION      printer, DATA
_PD          DC.L      0
_PED         DC.L      0
_SysBase     DC.L      0
_DOSBase     DC.L      0
_GfxBase     DC.L      0
_IntuitionBase DC.L      0

```

\*\*\*\*\*

```

SECTION      printer, CODE
_Init:
    MOVE.L    4(A7), _PD
    LEA       _PEDData(PC), A0
    MOVE.L    A0, _PED
    MOVE.L    A6, -(A7)
    MOVE.L    _AbsExecBase, A6
    MOVE.L    A6, _SysBase

```

```

*          ;----- open the dos library
    LEA       DLName(PC), A1
    MOVEQ     #0, D0
    CALLEXE   OpenLibrary
    MOVE.L    D0, _DOSBase
    BEQ       initDLErr

```

```

*          ;----- open the graphics library
    LEA       GLName(PC), A1
    MOVEQ     #0, D0
    CALLEXE   OpenLibrary
    MOVE.L    D0, _GfxBase
    BEQ       initGLErr

```

```

*          ;----- open the intuition library
    LEA       ILName(PC), A1
    MOVEQ     #0, D0
    CALLEXE   OpenLibrary

```

```

        MOVE.L D0,_IntuitionBase
        BEQ    initILerr

pdiRts:    MOVEQ    #0,D0

        MOVE.L (A7)+,A6
        RTS

initPAerr:    MOVE.L _IntuitionBase,A1
        LINKEXE CloseLibrary

initILerr:    MOVE.L _GfxBase,A1
        LINKEXE CloseLibrary

initGLerr:    MOVE.L _DOSBase,A1
        LINKEXE CloseLibrary

initDLerr:    MOVEQ    #-1,D0
        BRA.S    pdiRts

ILName:      DC.B    'intuition.library'
              DC.B    0

DLName:      DC.B    'dos.library'
              DC.B    0

GLName:      DC.B    'graphics.library'
              DC.B    0
              DS.W    0

```

```

*-----
_Expunge:    MOVE.L _IntuitionBase,A1
              LINKEXE CloseLibrary

              MOVE.L _GfxBase,A1
              LINKEXE CloseLibrary

              MOVE.L _DOSBase,A1
              LINKEXE CloseLibrary

```

```

*-----
_Open:       MOVEQ    #0,D0
              RTS

```

```

*-----
_Close:      MOVEQ    #0,D0
              RTS

              END

```

**EPSONX: DATA.C**

```

/*
    Data.c table for EpsonX driver.
    David Berezowski - March/88.
*/

char *CommandTable[] =
{
    "\375\033@\375", /* 00 aRIS reset */
    "\377", /* 01 aRIN initialize */
    "\012", /* 02 aIND linefeed */
    "\015\012", /* 03 aNEL CRLF */
    "\377", /* 04 aRI reverse LF */

    /* 05 aSGR0 normal char set */
    "\0335\033-\376\033F",
    "\0334", /* 06 aSGR3 italics on */
    "\0335", /* 07 aSGR23 italics off */
    "\033-\001", /* 08 aSGR4 underline on */
    "\033-\376", /* 09 aSGR24 underline off */
    "\033E", /* 10 aSGR1 boldface on */
    "\033F", /* 11 aSGR22 boldface off */
    "\377", /* 12 aSFC set foreground color */
    "\377", /* 13 aSBC set background color */

    /* 14 aSHORP0 normal pitch */
    "\033P\022\033W\376",
    /* 15 aSHORP2 elite on */
    "\033M\022\033W\376",
    "\033P", /* 16 aSHORP1 elite off */
    /* 17 aSHORP4 condensed fine on */
    "\017\033P\033W\376",
    "\022", /* 18 aSHORP3 condensed fine off */
    "\033W\001", /* 19 aSHORP6 enlarge on */
    "\033W\376", /* 20 aSHORP5 enlarge off */

    "\377", /* 21 aDEN6 shadow print on */
    "\377", /* 22 aDEN5 shadow print off */
    "\033G", /* 23 aDEN4 double strike on */
    "\033H", /* 24 aDEN3 double strike off */
    "\033x\001", /* 25 aDEN2 NLQ on */
    "\033x\376", /* 26 aDEN1 NLQ off */

    "\033S\376", /* 27 aSUS2 superscript on */
    "\033T", /* 28 aSUS1 superscript off */
    "\033S\001", /* 29 aSUS4 subscript on */
    "\033T", /* 30 aSUS3 subscript off */
    "\033T", /* 31 aSUS0 normalize the line */
    "\377", /* 32 aPLU partial line up */
    "\377", /* 33 aPLD partial line down */

    "\033R\376", /* 34 aFNT0 Typeface 0 */
    "\033R\001", /* 35 aFNT1 Typeface 1 */
    "\033R\002", /* 36 aFNT2 Typeface 2 */
    "\033R\003", /* 37 aFNT3 Typeface 3 */
    "\033R\004", /* 38 aFNT4 Typeface 4 */
    "\033R\005", /* 39 aFNT5 Typeface 5 */
    "\033R\006", /* 40 aFNT6 Typeface 6 */
    "\033R\007", /* 41 aFNT7 Typeface 7 */
    "\033R\010", /* 42 aFNT8 Typeface 8 */
    "\033R\011", /* 43 aFNT9 Typeface 9 */
    "\033R\012", /* 44 aFNT10 Typeface 10 */

    "\033p1", /* 45 aPROP2 proportional on */
    "\033p0", /* 46 aPROP1 proportional off */
    "\377", /* 47 aPROP0 proportional clear */
    "\377", /* 48 aTSS set proportional offset */
    "\377", /* 49 aJFY5 auto left justify */
    "\377", /* 50 aJFY7 auto right justify */
    "\377", /* 51 aJFY6 auto full justify */
    "\377", /* 52 aJFY0 auto justify off */
    "\377", /* 53 aJFY3 letter space */
    "\377", /* 54 aJFY1 word fill */
}

```



```

"\0330",      /* 55 aVERP0 1/8" line spacing */
"\0332",      /* 56 aVERP1 1/6" line spacing */
"\033C",      /* 57 aSLPP set form length */
"\033N",      /* 58 aPERF perf skip n (n > 0) */
"\033O",      /* 59 aPERF0 perf skip off */

"\377",      /* 60 aLMS set left margin */
"\377",      /* 61 aRMS set right margin */
"\377",      /* 62 aTMS set top margin */
"\377",      /* 63 aBMS set bottom margin */
"\377",      /* 64 aSTBM set T&B margins */
"\377",      /* 65 aSLRM set L&R margins */
"\377",      /* 66 aCAM clear margins */

"\377",      /* 67 aHTS set horiz tab */
"\377",      /* 68 aVTS set vert tab */
"\377",      /* 69 aTBC0 clear horiz tab */
"\033D\376", /* 70 aTBC3 clear all horiz tabs */
"\377",      /* 71 aTBC1 clear vert tab */
"\033B\376", /* 72 aTBC4 clear all vert tabs */
              /* 73 aTBCALL clear all h & v tabs */
"\033D\376\033B\376",
              /* 74 aTBSALL set default tabs */
"\033D\010\020\030\040\050\060\070\100\110\120\130\376",

"\377",      /* 75 aEXTEND extended commands */
"\377"       /* 76 aRAW next 'n' chars are raw */
};

/*
For each character from character 160 to character 255, there is
an entry in this table, which is used to print (or simulate printing of)
the full Amiga character set. (see AmigaDos Developer's Manual, pp A-3)
This table is used only if there is a valid pointer to this table
in the PEDData table in the printertag.asm file, and the VERSION is
33 or greater. Otherwise, a default table is used instead.
To place non-printable characters in this table, you can either enter
them as in C strings (ie \011, where 011 is an octal number, or as
\\000 where 000 is any decimal number, from 1 to 3 digits. This is
usually used to enter a NUL into the array (C has problems with it
otherwise.), or if you forgot your octal calculator. On retrospect,
was a poor choice for this function, as you must say \\ to enter a
backslash as a backslash. Live and learn...
*/
char *ExtendedCharTable[] =
{
    " ",      /* NBSPP */
    "\033R\007[\033R\0", /* i */
    "c\010|", /* c| */
    "\033R\003#\033R\0", /* L- */
    "\033R\005$\033R\0", /* o */
    "\033R\010\\\033R\0", /* Y- */
    "|",      /* | */
    "\033R\002@\033R\0", /* SS */

    "\033R\001~\033R\0", /* " */
    "c",      /* copyright */
    "\033S\0a\010_\033T", /* a_ */
    "<",      /* << */
    "~",      /* - */
    "-",      /* SHY */
    "r",      /* registered trademark */
    "-",      /* - */

    "\033R\001[\033R\0", /* degrees */
    "+\010 ", /* +_ */
    "\033S\0002\033T", /* 2 */
    "\033S\0003\033T", /* 3 */
    "'",      /* ' */
    "u",      /* u */
    "P",      /* reverse P */
    "\033S\000.\033T", /* . */

```

```

",", /* , */
"\033S\0001\033T", /* 1 */
"\033R\001[\033R\0\010-", /* o */
">", /* > */
"\033S\0001\033T\010-\010\033S\0014\033T", /* 1/4 */
"\033S\0001\033T\010-\010\033S\0012\033T", /* 1/2 */
"\033S\0003\033T\010-\010\033S\0014\033T", /* 3/4 */
"\033R\007]\033R\0", /* upside down ? */

"A\010'", /* 'A */
"A\010'", /* 'A */
"A\010^", /* ^A */
"A\010~", /* ~A */
"\033R\002[\033R\0", /* "A */
"\033R\004]\033R\0", /* oA */
"\033R\004[\033R\0", /* AE */
"C\010,", /* C, */

"E\010'", /* 'E */
"\033R\011@\033R\0", /* 'E */
"E\010^", /* ^E */
"E\010\033R\001~\033R\0", /* "E */
"I\010'", /* 'I */
"I\010'", /* 'I */
"I\010^", /* ^I */
"I\010\033R\001~\033R\0", /* "I */

"D\010-", /* -D */
"\033R\007\\\\\033R\0", /* ~N */
"O\010'", /* 'O */
"O\010'", /* 'O */
"O\010^", /* ^O */
"O\010~", /* ~O */
"\033R\002\\\\\033R\0", /* "O */
"x", /* x */

"\033R\004\\\\\033R\0", /* 0 */
"U\010'", /* 'U */
"U\010'", /* 'U */
"U\010^", /* ^U */
"\033R\002]\033R\0", /* "U */
"Y\010'", /* 'Y */
"T", /* Thorn */
"\033R\002~\033R\0", /* B */

"\033R\001e\033R\0", /* 'a */
"a\010'", /* 'a */
"a\010^", /* ^a */
"a\010~", /* ~a */
"\033R\002[\033R\0", /* "a */
"\033R\004]\033R\0", /* oa */
"\033R\004[\033R\0", /* ae */
"\033R\001\\\\\033R\0", /* c, */

"\033R\001]\033R\0", /* 'e */
"\033R\001[\033R\0", /* 'e */
"e\010^", /* ^e */
"e\010\033R\001~\033R\0", /* "e */
"\033R\006~\033R\0", /* 'i */
"i\010'", /* 'i */
"i\010^", /* ^i */
"i\010\033R\001~\033R\0", /* "i */

"d", /* d */
"\033R\007]\033R\0", /* ~n */
"\033R\006]\033R\0", /* 'o */
"o\010'", /* 'o */
"o\010^", /* ^o */
"o\010~", /* ~o */
"\033R\002]\033R\0", /* "o */
":\010-", /* :- */

```

```

        "\033R\004|\033R\0",          /* o/ */
        "\033R\001|\033R\0",          /* `u */
        "u\010'",                      /* 'u */
        "u\010^",                      /* ^u */
        "\033R\002|\033R\0",          /* "u */
        "y\010'",                      /* 'y */
        "t",                          /* thorn */
        "y\010\033R\001~\033R\0"     /* "y */
};

```

## EPSONX: DOSPECIAL.C

```

/*
    DoSpecial for EpsonX driver.
    David Berezowski - March/88.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

#define LMARG      3
#define RMARG      6
#define MARGLEN    8

#define CONDENSED   7
#define PITCH       9
#define QUALITY     17
#define LPI         24
#define INITLEN     26

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Params)
char outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Params[];
{
    extern struct PrinterData *PD;

    int x = 0, y = 0;
    /*
        00-00  \375    wait
        01-03  \033L    set left margin
        04-06  \033Qq   set right margin
        07-07  \375    wait
    */
    static char initMarg[MARGLEN] = {0xfd,0x1b,'l','L',0x1b,'Q','q',0xfd};
    /*
        00-01  \0335    italics off
        02-04  \033-\000 underline off
        05-06  \033F    boldface off
        07-07  \022     cancel condensed mode
        08-09  \033P    select pica (10 cpi)
        10-12  \033W\000 enlarge off
        13-14  \033H    doublestrike off
        15-17  \033x\000 draft
        18-19  \033T    super/sub script off
        20-22  \033p0   proportional off
        23-24  \0332    6 lpi
        25-25  \015     carriage return
    */
    static char initThisPrinter[INITLEN] =
{0x1b,0x05,0x1b,'-',0x0,0x1b,'F',0x12,0x1b,'P',0x1b,'W',0x0,0x1b,'H',
 0x1b,'x',0x0,0x1b,'T',0x1b,'p',0x0,0x1b,0x02,0x0d};
    static BYTE ISOcolorTable[10] = {0, 5, 6, 4, 3, 1, 2, 0};

```

```

if (*command == aRIN)
{
    while (x < INITLEN)
    {
        outputBuffer[x] = initThisPrinter[x];
        x++;
    }

    if (PD->pd_Preferences.PrintQuality == LETTER)
    {
        outputBuffer[QUALITY] = 1;
    }

    *currentVMI = 36; /* assume 1/6 line spacing (36/216 => 1/6) */
    if (PD->pd_Preferences.PrintSpacing == EIGHT_LPI)
    {
        outputBuffer[LPI] = '0';
        *currentVMI = 27; /* 27/216 => 1/8 */
    }

    if (PD->pd_Preferences.PrintPitch == ELITE)
    {
        outputBuffer[PITCH] = 'M';
    }
    else if (PD->pd_Preferences.PrintPitch == FINE)
    {
        outputBuffer[CONDENSED] = '\017'; /* condensed */
        outputBuffer[PITCH] = 'P'; /* pica condensed */
    }

    Params[0] = PD->pd_Preferences.PrintLeftMargin;
    Params[1] = PD->pd_Preferences.PrintRightMargin;
    *command = aSLRM;
}

if (*command == aCAM)
{ /* cancel margins */
    y = PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
    if (PD->pd_Preferences.PrintPitch == PICA)
    {
        Params[1] = (10 * y) / 10;
    }
    else if (PD->pd_Preferences.PrintPitch == ELITE)
    {
        Params[1] = (12 * y) / 10;
    }
    else
    { /* fine */
        Params[1] = (17 * y) / 10;
    }
    Params[0] = 1;
    y = 0;
    *command = aSLRM;
}

if (*command == aSLRM)
{ /* set left and right margins */
    PD->pd_PWaitEnabled = 253;
    if (Params[0] == 0)
    {
        initMarg[LMARG] = 0;
    }
    else
    {
        initMarg[LMARG] = Params[0] - 1;
    }
    initMarg[RMARG] = Params[1];
    while (y < MARGLEN)
    {
        outputBuffer[x++] = initMarg[y++];
    }
    return(x);
}

```

```

}

if (*command == aPLU)
{
    if (*vline == 0)
    {
        *vline = 1;
        *command = aSUS2;
        return(0);
    }
    if (*vline < 0)
    {
        *vline = 0;
        *command = aSUS3;
        return(0);
    }
    return(-1);
}

if (*command == aPLD)
{
    if (*vline == 0)
    {
        *vline = -1;
        *command = aSUS4;
        return(0);
    }
    if (*vline > 0)
    {
        *vline = 0;
        *command = aSUS1;
        return(0);
    }
    return(-1);
}

if (*command == aSUS0)
{
    *vline = 0;
}
if (*command == aSUS1)
{
    *vline = 0;
}
if (*command == aSUS2)
{
    *vline = 1;
}
if (*command == aSUS3)
{
    *vline = 0;
}
if (*command == aSUS4)
{
    *vline = -1;
}

if (*command == aVERP0)
{
    *currentVMI = 27;
}

if (*command == aVERP1)
{
    *currentVMI = 36;
}

if (*command == aIND)
{ /* lf */
    outputBuffer[x++] = '\033';
    outputBuffer[x++] = 'J';
    outputBuffer[x++] = *currentVMI;
}

```

```

        return(x);
    }

    if (*command == aRI)
    { /* reverse lf */
        outputBuffer[x++] = '\033';
        outputBuffer[x++] = 'j';
        outputBuffer[x++] = *currentVMI;
        return(x);
    }

    if (*command == aSFC)
    {
        if (Parms[0] == 39)
        {
            Parms[0] = 30; /* set defaults */
        }
        if (Parms[0] > 37)
        {
            return(0); /* ni or background color change */
        }
        outputBuffer[x++] = '\033';
        outputBuffer[x++] = 'r';
        outputBuffer[x++] = ISOcolorTable[Parms[0] - 30];
        /*
        Kludge to get this to work on a CBM MPS-1250 which interprets
        'ESCr' as go into reverse print mode. The 'ESCT' tells it to
        get out of reverse print mode. The 'NULL' is ignored by the
        CBM MPS-1250 and required by all Epson printers as the
        terminator for the 'ESCtNULL' command which means select
        normal char set (which has no effect).
        */
        outputBuffer[x++] = '\033';
        outputBuffer[x++] = 't';
        outputBuffer[x++] = 0;
        return(x);
    }

    if (*command == aRIS)
    {
        PD->pd_PWaitEnabled = 253;
    }

    return(0);
}

```

## EPSONX: RENDER.C

```

/*
    EpsonX (EX/FX/JX/LX/MX/RX) driver.
    David Berezowski - October/87.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>

#include <devices/printer.h>
#include <devices/prtbase.h>

#define NUMSTARTCMD    7      /* # of cmd bytes before binary data */
#define NUMENDCMD      1      /* # of cmd bytes after binary data */
#define NUMTOTALCMD    (NUMSTARTCMD + NUMENDCMD) /* total of above */
#define NUMLFCMD       4      /* # of cmd bytes for linefeed */
#define MAXCOLORBUFS   4      /* max # of color buffers */

#define STARTLEN       19

```

```

#define PITCH          1
#define CONDENSED      2
#define LMARG          8
#define RMARG         11
#define DIREC          15

static ULONG TwoBufSize;
static UWORD RowSize, ColorSize, NumColorBufs, dpi_code, spacing;
static UWORD colorcodes[MAXCOLORBUFS];

void ClearAndInit(UBYTE *);
UBYTE *CompactBuf(UBYTE *, UBYTE *, LONG, int);

int
Render(ct, x, y, status)
long ct, x, y, status;
{
    extern void *AllocMem(), FreeMem();

    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    UBYTE *CompactBuf();
    static ULONG BufSize, TotalBufSize, dataoffset;
    static UWORD spacing, colors[MAXCOLORBUFS];
    /*
        00-01  \003P          set pitch (10 or 12 cpi)
        02-02  \022          set condensed fine (on or off)
        03-05  \033W\000     enlarge off
        06-08  \033ln       set left margin to n
        09-11  \033Qn       set right margin to n
        12-12  \015         carriage return
        13-15  \033U1       set uni-directional mode
        16-18  \033t\000     see kludge note below
        Kludge to get this to work on a CBM MPS-1250 which interprets
        'ESC'r as go into reverse print mode. The 'ESCt' tells it to
        get out of reverse print mode. The 'NULL' is ignored by the
        CBM MPS-1250 and required by all Epson printers as the
        terminator for the 'ESCtNULL' command which means select
        normal char set (which has no effect).
    */
    static UBYTE StartBuf[STARTLEN] =
    {0x1b, 'P', 0x12, 0x1b, 'W', 0x0, 0x1b, 'l', 'n', 0x1b, 'Q', 'n', 0x0d, 0x1b, 'U', 0x01,
     0x1b, 't', 0x0};
    /*
        "\033P\022\033W\000\033ln\033Qn\015\033U1\033t\000"; */

    UBYTE *ptr, *ptrstart;
    int err;

    switch(status)
    {
        case 0 : /* Master Initialization */
            /*
                ct      - pointer to IODRPreq structure.
                x      - width of printed picture in pixels.
                y      - height of printed picture in pixels.
            */
            RowSize = x;
            ColorSize = RowSize + NUMTOTALCMD;
            if (PD->pd_Preferences.PrintShade == SHADE_COLOR)
            {
                NumColorBufs = MAXCOLORBUFS;
                colors[0] = ColorSize * 3; /* Black */
                colors[1] = ColorSize * 0; /* Yellow */
                colors[2] = ColorSize * 1; /* Magenta */
                colors[3] = ColorSize * 2; /* Cyan */
                colorcodes[0] = 4; /* Yellow */
                colorcodes[1] = 1; /* Magenta */
                colorcodes[2] = 2; /* Cyan */
                colorcodes[3] = 0; /* Black */
            }
            else

```

```

    { /* grey-scale or black&white */
        NumColorBufs = 1;
        colors[0] = ColorSize * 0; /* Black */
        colorcodes[0] = 0; /* Black */
    }
    BufSize = ColorSize * NumColorBufs + NUMLFCMD;
    if (PED->ped_YDotsInch == 216)
    {
        TwoBufSize = BufSize * 3;
        TotalBufSize = BufSize * 6;
    }
    else if (PED->ped_YDotsInch == 144)
    {
        TwoBufSize = BufSize * 2;
        TotalBufSize = BufSize * 4;
    }
    else
    {
        TwoBufSize = BufSize * 1;
        TotalBufSize = BufSize * 2;
    }
    PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
    if (PD->pd_PrintBuf == NULL)
    {
        err = PDERR_BUFFERMEMORY;
    }
    else
    {
        dataoffset = NUMSTARTCMD;
        /*
            This printer prints graphics within its
            text margins. This code makes sure the
            printer is in 10 cpi and then sets the
            left and right margins to their minimum
            and maximum values (respectively). A
            carriage return is sent so that the
            print head is at the leftmost position
            as this printer starts printing from
            the print head's position. The printer
            is put into unidirectional mode to
            reduce wavy vertical lines.
        */
        StartBuf[PITCH] = 'P'; /* 10 cpi */
        StartBuf[CONDENSED] = '\022'; /* off */
        /* left margin of 1 */
        StartBuf[LMARG] = 0;
        /* right margin of 80 or 136 */
        StartBuf[RMARG] = PD->pd_Preferences.
            PaperSize == W_TRACTOR ? 136 : 80;
        /* uni-directional mode */
        StartBuf[DIREC] = '1';
        err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
    }
    break;

case 1 : /* Scale, Dither and Render */
    /*
        ct      - pointer to PrtInfo structure.
        x      - 0.
        y      - row # (0 to Height - 1).
    */
    Transfer(ct, y, &PD->pd_PrintBuf[dataoffset], colors,
        BufSize);
    err = PDERR_NOERR; /* all ok */
    break;

case 2 : /* Dump Buffer to Printer */
    /*
        ct      - 0.
        x      - 0.
        y      - # of rows sent (1 to NumRows).
    */

```



```

*/
/* white-space strip */
ptrstart = &PD->pd_PrintBuf[dataoffset - NUMSTARTCMD];
if (PED->ped_YDotsInch == 72)
{
    /* y range : 1 to 8 */
    y = y * 3 - spacing;
    ptr = CompactBuf(ptrstart + NUMSTARTCMD,
        ptrstart, y, 1);
}
else if (PED->ped_YDotsInch == 144)
{
    /* y range : 1 to 16 */
    ptr = CompactBuf(ptrstart + NUMSTARTCMD,
        ptrstart, 2, 1);
    if (y > 1)
    {
        ptr = CompactBuf(&PD->pd_PrintBuf[
            dataoffset + BufSize],
            ptr, y * 3 / 2 - 2, 0);
    }
}
else if (PED->ped_YDotsInch == 216)
{
    /* y range : 1 to 24 */
    ptr = CompactBuf(ptrstart + NUMSTARTCMD,
        ptrstart, 1, 1);
    if (y > 1)
    {
        ptr = CompactBuf(&PD->pd_PrintBuf[
            dataoffset + BufSize],
            ptr, 1, 0);
    }
    if (y > 2)
    {
        ptr = CompactBuf(&PD->pd_PrintBuf[
            dataoffset + BufSize * 2],
            ptr, y - 2, 0);
    }
}
err = (*PD->pd_PWrite)(ptrstart, ptr - ptrstart);
if (err == PDERR_NOERR)
{
    dataoffset = (dataoffset == NUMSTARTCMD ?
        TwoBufSize : 0) + NUMSTARTCMD;
}
break;

case 3 : /* Clear and Init Buffer */
/*
    ct      - 0.
    x      - 0.
    y      - 0.
*/
ClearAndInit(&PD->pd_PrintBuf[dataoffset]);
err = PDERR_NOERR;
break;

case 4 : /* Close Down */
/*
    ct      - error code.
    x      - io_Special flag from IODRPreq.
    y      - 0.
*/
err = PDERR_NOERR; /* assume all ok */
/* if user did not cancel print */
if (ct != PDERR_CANCEL)
{
    /* restore preferences pitch and margins */
    if (PD->pd_Preferences.PrintPitch == ELITE)
    {
        StartBuf[PITCH] = 'M'; /* 12 cpi */
    }
}

```

```

    }
    else if (PD->pd_Preferences.PrintPitch == FINE)
    {
        StartBuf[CONDENSED] = '\017'; /* on */
    }
    StartBuf[LMARG] =
        PD->pd_Preferences.PrintLeftMargin - 1;
    StartBuf[RMARG] =
        PD->pd_Preferences.PrintRightMargin;
    StartBuf[DIREC] = '0'; /* bi-directional */
    err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
}
(*(PD->pd_PBothReady))();
if (PD->pd_PrintBuf != NULL)
{
    FreeMem(PD->pd_PrintBuf, TotalBufSize);
}
break;

case 5 : /* Pre-Master Initialization */
/*
    ct      - 0 or pointer to IODRPreReq structure.
    x      - io_Special flag from IODRPreReq.
    y      - 0.
*/
/* kludge for sloppy tractor mechanism */
spacing = PD->pd_Preferences.PaperType == SINGLE ?
    1 : 0;
dpi_code = SetDensity(x & SPECIAL_DENSITYMASK);
err = PDERR_NOERR;
break;
}
return(err);
}

```

```

UBYTE *CompactBuf(ptrstart, ptr2start, y, flag)
UBYTE *ptrstart, *ptr2start;
long y;
int flag; /* 0 - not first pass, !0 - first pass */
{
    static int x;
    UBYTE *ptr, *ptr2;
    UBYTE **dummy;

    long ct;
    int i;

    ptr2 = ptr2start; /* where to put the compacted data */
    if (flag)
    {
        x = 0; /* flag no transfer required yet */
    }
    for (ct=0; ct<NumColorBufs; ct++, ptrstart += ColorSize)
    {
        i = RowSize;
        ptr = ptrstart + i - 1;
        while (i > 0 && *ptr == 0)
        {
            i--;
            ptr--;
        }
        if (i != 0)
        { /* if data */
            *(++ptr) = 13; /* <cr> */
            ptr = ptrstart - NUMSTARTCMD;
            *ptr++ = 27;
            *ptr++ = 'r';
            *ptr++ = colorcodes[ct]; /* color */
            *ptr++ = 27;
            *ptr++ = dpi_code; /* density */
            *ptr++ = i & 0xff;
        }
    }
}

```

```

        *ptr++ = i >> 8;                /* size */
        i += NUMTOTALCMD;
        if (x != 0)
        { /* if must transfer data */
            /* get src start */
            ptr = ptrstart - NUMSTARTCMD;
            dummy = &ptr; /* otherwise Lattice loses
                           track of the pointer..... */
            do
            { /* transfer and update dest ptr */
                *ptr2++ = *ptr++;
            } while (--i);
        }
        else
        { /* no transfer required */
            ptr2 += i; /* update dest ptr */
        }
    }
    if (i != RowSize + NUMTOTALCMD)
    { /* if compacted or 0 */
        x = 1; /* flag that we need to transfer next time */
    }
}
*ptr2++ = 13; /* cr */
*ptr2++ = 27;
*ptr2++ = 'J';
*ptr2++ = y; /* y/216 lf */
return(ptr2);
}

void ClearAndInit(ptr)
UBYTE *ptr;
{
    ULONG *lptr, i, j;

    /*
       Note : Since 'NUMTOTALCMD + NUMLFCMD' is > 3 bytes it is safe
       to do the following to speed things up.
    */
    i = TwoBufSize - NUMTOTALCMD - NUMLFCMD;
    j = (ULONG)ptr;
    if (!(j & 1))
    { /* if on a word boundary, clear by longs */
        i = (i + 3) / 4;
        lptr = (ULONG *)ptr;
        do
        {
            *lptr++ = 0;
        } while (--i);
    }
    else
    { /* clear by bytes */
        do
        {
            *ptr++ = 0;
        } while (--i);
    }
}

```

## EPSONX: DENSITY.C

```

/*
    Density module for EpsonX driver.
    David Berezowski - October/87.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

char
SetDensity(density_code)
ULONG density_code;
{
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    /* SPECIAL_DENSITY    0    1    2    3    4    5    6    7 */
    static int XDPI[8] = {120, 120, 120, 240, 120, 240, 240, 240};
    static int YDPI[8] = {72, 72, 144, 72, 216, 144, 216, 216};
    static char codes[8] = {'L', 'L', 'L', 'Z', 'L', 'Z', 'Z', 'Z'};

    PED->ped_MaxColumns =
        PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
    density_code /= SPECIAL_DENSITY1;
    /* default is 80 chars (8.0 in.), W_TRACTOR is 136 chars (13.6 in.) */
    PED->ped_MaxXDots =
        (XDPI[density_code] * PED->ped_MaxColumns) / 10;
    PED->ped_XDotsInch = XDPI[density_code];
    PED->ped_YDotsInch = YDPI[density_code];
    if ((PED->ped_YDotsInch = YDPI[density_code]) == 216)
    {
        PED->ped_NumRows = 24;
    }
    else if (PED->ped_YDotsInch == 144)
    {
        PED->ped_NumRows = 16;
    }
    else
    {
        PED->ped_NumRows = 8;
    }
    return(codes[density_code]);
}

```

## EPSONX: TRANSFER.ASM

```

*****
*
* Transfer routine for EpsonX
*
* David Berezowski - April/88
*
*****

INCLUDE "exec/types.i"

INCLUDE "intuition/intuition.i"
INCLUDE "devices/printer.i"
INCLUDE "devices/prtbase.i"
INCLUDE "devices/prtgfx.i"

XREF    _PD
XREF    _PED
XREF    _LVODebug
XREF    _AbsExecBase

XDEF    _Transfer

```

# SECTION CODE

```

_Transfer:
; Transfer(PInfo, y, ptr, colors, BufOffset)
; struct PrtInfo *PInfo          4-7
; UWORD y;                       8-11
; UBYTE *ptr;                    12-15
; UWORD *colors;                 16-19
; ULONG BufOffset                20-23

    movem.l d2-d6/a2-a4,-(sp)      ;save regs used

    movea.l 36(sp),a0              ;a0 = PInfo
    move.l 40(sp),d0               ;d0 = y
    movea.l 44(sp),a1              ;a1 = ptr
    movea.l 48(sp),a2              ;a2 = colors
    move.l 52(sp),d1               ;d1 = BufOffset

    move.l d0,d3                   ;save y
    moveq.l #3,d2                  ;d2 = y & 3
    and.w d0,d2                   ;d2 = (y & 3) << 2
    lsl.w #2,d2                    ;d2 = (y & 3) << 2
    movea.l pi_dmatrix(a0),a3      ;a3 = dmatrix
    adda.l d2,a3                   ;a3 = dmatrix + ((y & 3) << 2)

    movea.l _PED,a4                ;a4 = ptr to PED
    cmpi.w #216,ped_YDotsInch(a4) ;triple interleaving?
    bne.s 10$                     ;no
    divu.w #3,d0                   ;y /= 3
    swap.w d0                      ;d0 = y % 3
    mulu.w d0,d1                   ;BufOffset *= y % 3
    swap.w d0                      ;d0 = y / 3
    bra.s 30$

10$:    cmpi.w #144,ped_YDotsInch(a4) ;double interleaving?
    bne.s 20$                     ;no, clear BufOffset
    asr.w #1,d0                    ;y /= 2
    btst.b #0,d3                   ;odd pass? (Lattice doesn't like
    ; the '.b'
    btst #0,d3                     ;odd pass?
    bne.s 30$                     ;no, dont clear BufOffset

20$:    moveq.l #0,d1               ;BufOffset = 0

30$:    move.w d0,d6                ;d6 = bit to set
    not.b d6                       ;ptr += BufOffset
    adda.l d1,a1

    movea.l _PD,a4                 ;a4 = ptr to PD
    cmpi.w #SHADE_COLOR,pd_Preferences+pf_PrintShade(a4) ;color dump?
    bne not_color                  ;no

color:
; a0 - PInfo
; a1 - ptr (ptr + BufOffset)
; a2 - colors
; a3 - dmatrix ptr
; d0 - y
; d1 - BufOffset
; d6 - bit to set

    movem.l d7/a5-a6,-(sp)      ;save regs used

    movea.l a1,a4
    movea.l a1,a5
    movea.l a1,a6
    adda.w (a2)+,a1               ;a1 = ptr + colors[0] (bptr)
    adda.w (a2)+,a4               ;a4 = ptr + colors[1] (yptr)
    adda.w (a2)+,a5               ;a5 = ptr + colors[2] (mptr)
    adda.w (a2)+,a6               ;a6 = ptr + colors[3] (cptr)

    movea.l pi_ColorInt(a0),a2    ;a2 = ColorInt ptr

```

```

        move.w pi_width(a0),width      ;# of pixels to do
        move.w pi_xpos(a0),d2          ;d2 = x
        movea.l pi_ScaleX(a0),a0       ;a0 = ScaleX (sxptra)
        move.b d6,d7                   ;d7 = bit to set

; a0 - sxptr
; a1 - bptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr
; a4 - yptr
; a5 - mptr
; a6 - cptr
; d1 - Black
; d2 - x
; d3 - dvalue (dmatrix[x & 3])
; d4 - Yellow
; d5 - Magenta
; d6 - Cyan
; d7 - bit to set

cwidth_loop:
        move.b PCBLACK(a2),d1          ;d1 = Black
        move.b PCMYELLOW(a2),d4        ;d4 = Yellow
        move.b PCMMAGENTA(a2),d5       ;d5 = Magenta
        move.b PCMCYAN(a2),d6          ;d6 = Cyan
        addq.l #ce_SIZEOF,a2           ;advance to next entry

        move.w (a0)+,sx                ;# of times to use this pixel

csx_loop:
        moveq.l #3,d3
        and.w d2,d3                    ;d3 = x & 3
        move.b 0(a3,d3.w),d3           ;d3 = dmatrix[x & 3]

black:
        cmp.b d3,d1                    ;render black?
        ble.s yellow                   ;no, try ymc
        bset.b d7,0(a1,d2.w)           ;set black pixel
        bra.s csx_end

yellow:
        cmp.b d3,d4                    ;render yellow pixel?
        ble.s magenta                  ;no.
        bset.b d7,0(a4,d2.w)           ;set yellow pixel

magenta:
        cmp.b d3,d5                    ;render magenta pixel?
        ble.s cyan                     ;no.
        bset.b d7,0(a5,d2.w)           ;set magenta pixel

cyan:
        cmp.b d3,d6                    ;render cyan pixel?
        ble.s csx_end                  ;no, skip to next pixel.
        bset.b d7,0(a6,d2.w)           ;clear cyan pixel

csx_end
        addq.w #1,d2                    ;x++
        subq.w #1,sx                   ;sx--
        bne.s csx_loop
        subq.w #1,width                ;width--
        bne.s cwidth_loop

        movem.l (sp)+,d7/a5-a6         ;restore regs used
        bra exit

not_color:
; a0 - PInfo
; a1 - ptr
; a2 - colors
; a3 - dmatrix ptr
; d0 - y
; d6 - bit to set

```

```

        adda.w  (a2),a1          ;a1 = ptr + colors[0]
        move.w  pi_width(a0),d1  ;d1 = width
        subq.w  #1,d1           ;adjust for dbra

        move.w  pi_threshold(a0),d3 ;d3 = threshold, thresholding?
        beq.s   grey_scale       ;no, grey-scaling

threshold:
; a0 - PInfo
; a1 - ptr
; a3 - dmatrix ptr
; d1 - width-1
; d3 - threshold
; d6 - bit to set

        eori.b  #15,d3          ;d3 = dvalue
        movea.l pi_ColorInt(a0),a2 ;a2 = ColorInt ptr
        move.w  pi_xpos(a0),d2   ;d2 = x
        movea.l pi_ScaleX(a0),a0 ;a0 = ScaleX (sxptr)
        adda.w  d2,a1           ;ptr += x

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr (NOT USED)
; d1 - width
; d3 - dvalue
; d4 - Black
; d5 - sx
; d6 - bit to set

twidth_loop:
        move.b  PCMBLACK(a2),d4   ;d4 = Black
        addq.l  #ce_SIZEOF,a2     ;advance to next entry

        move.w  (a0)+,d5          ;d5 = # of times to use this pixel

        cmp.b   d3,d4             ;render this pixel?
        ble.s   tsx_end           ;no, skip to next pixel.
        subq.w  #1,d5             ;adjust for dbra

tsx_render:
        bset.b  d6,(a1)           ;yes, render this pixel sx times
                                      ;*(ptr) |= bit;

        adda.w  #1,a1             ;ptr++
        dbra    d5,tsx_render     ;sx--
        dbra    d1,twidth_loop    ;width--
        bra.s   exit              ;all done

tsx_end:
        adda.w  d5,a1             ;ptr += sx
        dbra    d1,twidth_loop    ;width--
        bra.s   exit

grey_scale:
; a0 - PInfo
; a1 - ptr
; a3 - dmatrix ptr
; d0 - y
; d1 - width-1
; d6 - bit to set

        movea.l pi_ColorInt(a0),a2 ;a2 = ColorInt ptr
        move.w  pi_xpos(a0),d2     ;d2 = x
        movea.l pi_ScaleX(a0),a0   ;a0 = ScaleX (sxptr)

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr
; d1 - width

```

```

; d2 - x
; d3 - dvalue (dmatrix[x & 3])
; d4 - Black
; d5 - sx
; d6 - bit to set

gwidth_loop:
    move.b  PCMBLACK(a2),d4        ;d4 = Black
    addq.l  #ce_SIZEOF,a2        ;advance to next entry

    move.w  (a0)+,d5              ;d5 = # of times to use this pixel
    subq.w  #1,d5                ;adjust for dbra

gsx_loop:
    moveq.l #3,d3
    and.w   d2,d3                ;d3 = x & 3
    move.b  0(a3,d3.w),d3        ;d3 = dmatrix[x & 3]

    cmp.b   d3,d4                ;render this pixel?
    ble.s   gsx_end              ;no, skip to next pixel.

    bset.b  d6,0(a1,d2.w)        ;*(ptr + x) |= bit

gsx_end:
    addq.w  #1,d2                ;x++
    dbra    d5,gsx_loop          ;sx--
    dbra    d1,gwidth_loop       ;width--

exit:
    movem.l (sp)+,d2-d6/a2-a4    ;restore regs used
    moveq.l #0,d0                ;flag all ok
    rts                          ;goodbye

sx      dc.w   0
width   dc.w   0

END

```

## EPSONX: TRANSFER.C

```

/*
    Transfer routine for EpsonX driver.
    David Berezowski - October/87.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <devices/prtgfx.h>

Transfer(PInfo, y, ptr, colors, BufOffset)
struct PrtInfo *PInfo;
UWORD y;                /* row # */
UBYTE *ptr;              /* ptr to buffer */
UWORD *colors;           /* indexes to color buffers */
ULONG BufOffset;         /* used for interleaved printing */
{
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    static UWORD bit_table[8] = {128, 64, 32, 16, 8, 4, 2, 1};
    union colorEntry *ColorInt;
    UBYTE *bptr, *yptr, *mptr, *cptr, Black, Yellow, Magenta, Cyan;
    UBYTE *dmatrix, dvalue, threshold;
    UWORD x, width, sx, *sxptr, color, bit, x3;

    /* printer non-specific, MUST DO FOR EVERY PRINTER */
    x = PInfo->pi_xpos;

```



```

ColorInt = PInfo->pi_ColorInt;
sxptr = PInfo->pi_ScaleX;
width = PInfo->pi_width;

/* printer specific */
if (PED->ped_YDotsInch == 216)
{
    BufOffset *= y % 3;
    y /= 3;
}
else if (PED->ped_YDotsInch == 144)
{
    BufOffset *= y % 1;
    y /= 2;
}
else
{
    BufOffset = 0;
}
bit = bit_table[y % 7];
bptr = ptr + colors[0] + BufOffset;
yptr = ptr + colors[1] + BufOffset;
mptr = ptr + colors[2] + BufOffset;
cptr = ptr + colors[3] + BufOffset;

/* pre-compute threshold; are we thresholding? */
if (threshold = PInfo->pi_threshold)
{ /* thresholding */
    dvalue = threshold ^ 15;
    bptr += x;
    do { /* for all source pixels */
        /* pre-compute intensity values for Black component */
        Black = ColorInt->colorByte[PCMBLACK];
        ColorInt++;

        sx = *sxptr++;

        do { /* use this pixel 'sx' times */
            if (Black > dvalue)
            {
                *bptr |= bit;
            }
            bptr++; /* done 1 more printer pixel */
        } while (--sx);
    } while (--width);
}
else
{ /* not thresholding, pre-compute ptr to dither matrix */
    dmatrix = PInfo->pi_dmatrix + ((y % 3) << 2);
    if (PD->pd_Preferences.PrintShade == SHADE_GREYSCALE)
    {
        do { /* for all source pixels */
            /* pre-compute intensity values for Black */
            Black = ColorInt->colorByte[PCMBLACK];
            ColorInt++;

            sx = *sxptr++;

            do { /* use this pixel 'sx' times */
                if (Black > dmatrix[x % 3])
                {
                    *(bptr + x) |= bit;
                }
                x++; /* done 1 more printer pixel */
            } while (--sx);
        } while (--width);
    }
    else
    { /* color */
        do { /* for all source pixels */
            /* compute intensity values for each color */
            Black = ColorInt->colorByte[PCMBLACK];

```

```

Yellow = ColorInt->colorByte[PCMYELLOW];
Magenta = ColorInt->colorByte[PCMMAGENTA];
Cyan = ColorInt->colorByte[PCMCYAN];
ColorInt++;

```

```

sx = *sxptr++;

```

```

do { /* use this pixel 'sx' times */
    x3 = x >> 3;
    dvalue = dmatrix[x & 3];
    if (Black > dvalue)
    {
        *(bptr + x) |= bit;
    }
    else
    { /* black not rendered */
        if (Yellow > dvalue)
        {
            *(yptr + x) |= bit;
        }
        if (Magenta > dvalue)
        {
            *(mptr + x) |= bit;
        }
        if (Cyan > dvalue)
        {
            *(cptr + x) |= bit;
        }
    }
    ++x; /* done 1 more printer pixel */
} while (--sx);
} while (--width);

```

```

}

```

```

}

```

```

}

```

# EPSONQ

The EpsonQ driver can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
CFLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
LINK = lc:blink
LIB = lib:amiga.lib+lib:lc.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o+density.o
TARGET = EpsonQ

    @$(LC) $(CFLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o density.o transfer.o
    @$(LINK) <WITH <
    FROM $(OBJ)
    TO $(TARGET)
    LIBRARY $(LIB)
    NODEBUG SC SD VERBOSE MAP $(TARGET).map H
    <

init.o: init.asm
    @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm EpsonQ_rev.i
    @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.c

dospecial.o: dospecial.c

data.o: data.c

density.o: density.c

render.o: render.c

install:
    @copy $(TARGET) to devs:printers
```

## EPSONQ: PRINTERTAG.ASM

```
TTL      '$Header: printer.3,v 1.1 89/10/31 16:05:49 ken Exp $'
*****
*
*   Copyright 1985, Commodore-Amiga Inc.  All rights reserved.
*   No part of this program may be reproduced, transmitted,
*   transcribed, stored in retrieval system, or translated into
*   any language or computer language, in any form or by any
*   means, electronic, mechanical, magnetic, optical, chemical,
*   manual or otherwise, without the prior written permission of
*   Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
*   Pennsylvania, 19380
*
*****
*
*   printer device dependent code tag
*
*   Source Control
*   -----
```

```

* $Header: printer.3,v 1.1 89/10/31 16:05:49 ken Exp $
*
* $Locker: $
*
* $Log: printer.3,v $
* Revision 1.1 89/10/31 16:05:49 ken
* Initial revision
*
* Revision 1.5 88/04/19 17:12:34 daveb
* V1.3 Gamma 13
*
* Revision 1.4 88/04/15 16:42:12 daveb
* fixed docs for devcon
* V1.3 Gamma 13
*
* Revision 1.3 87/10/27 15:26:31 daveb
* V1.3 gamma 1 check-in.
*
* Revision 1.2 87/10/01 09:17:56 daveb
* changed NumRows to 16 to support 'weak' 24-pin printers
* V1.3 beta 4 check-in
*
* Revision 1.1 87/08/21 10:01:13 daveb
* set XDotsInch and MaxXDots to default (60 dpi) values
*
* Revision 1.0 87/08/20 14:08:42 daveb
* added to rcs
*
* Revision 1.0 87/08/20 13:27:02 daveb
* added to rcs
*
* Revision 1.3 87/08/03 11:03:52 daveb
* added null ptr to char conversion function at end of table
*
* Revision 1.2 87/07/30 10:34:12 daveb
* added 'DS.L 1' at end to reserve space for PrintMode
*
* Revision 1.1 87/07/21 11:36:30 daveb
* added 'PPC_VERSION_2' to PrinterClass
*
* Revision 1.0 87/07/21 11:35:43 daveb
* added to rcs
*
*
*
*****

```

## SECTION printer

### \*----- Included Files -----

```

INCLUDE      "exec/types.i"
INCLUDE      "exec/nodes.i"
INCLUDE      "exec/strings.i"

INCLUDE      "epsonQ_rev.i"

INCLUDE      "devices/prtbase.i"

```

### \*----- Imported Names -----

```

XREF      _Init
XREF      _Expunge
XREF      _Open
XREF      _Close
XREF      _CommandTable
XREF      _PrinterSegmentData
XREF      _DoSpecial
XREF      _Render
XREF      _ExtendedCharTable

```

### \*----- Exported Names -----

XDEF

\_PEDData

\*\*\*\*\*

```

MOVEQ    #0,D0          ; show error for OpenLibrary()
RTS
DC.W     VERSION
DC.W     REVISION

_PEDData:
DC.L     printerName
DC.L     _Init
DC.L     _Expunge
DC.L     _Open
DC.L     _Close
DC.B     PPC_COLORGFX    ; PrinterClass
DC.B     PCC_YMCB        ; ColorClass
DC.B     136             ; MaxColumns
DC.B     10              ; NumCharSets
DC.W     24              ; NumRows
DC.L     1088            ; MaxXDots
DC.L     0               ; MaxYDots
DC.W     80              ; XDotsInch
DC.W     180             ; YDotsInch
DC.L     _CommandTable   ; Commands
DC.L     _DoSpecial
DC.L     _Render
DC.L     30              ; Timeout
DC.L     _ExtendedCharTable ; 8BitChars
DS.L     1               ; PrintMode (reserve space)
DC.L     0               ; ptr to char conversion function

printerName:
DC.B     "EpsonQ"
DC.B     0
END

```

**EPSONQ: EPSONQ\_REV.I**

```

VERSION    EQU 35
REVISION   EQU 0

```

**EPSONQ: INIT.ASM**

```

TTL      '$Header: printer.3,v 1.1 89/10/31 16:05:49 ken Exp $'
*****
*
*   Copyright 1985, Commodore-Amiga Inc.  All rights reserved.
*   No part of this program may be reproduced, transmitted,
*   transcribed, stored in retrieval system, or translated into
*   any language or computer language, in any form or by any
*   means, electronic, mechanical, magnetic, optical, chemical,
*   manual or otherwise, without the prior written permission of
*   Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
*   Pennsylvania, 19380
*
*****
*
*   printer device functions
*
*   Source Control
*   -----
*   $Header: printer.3,v 1.1 89/10/31 16:05:49 ken Exp $
*
*   $Locker: carolyn $

```

```

*
* $Log: printer.3,v $
* Revision 1.1  89/10/31  16:05:49  ken
* Initial revision
*
* Revision 1.1  88/04/14  12:03:14  daveb
* V1.3 Gamma 11 release
*
* Revision 1.0  87/08/20  14:10:17  daveb
* added to rcs
*
* Revision 1.1  85/10/09  19:27:20  kodiak
* remove _stdout variable
*
* Revision 1.0  85/10/09  19:23:23  kodiak
* added to rcs for updating in version 1
*
* Revision 25.0  85/06/16  01:01:22  kodiak
* added to rcs
*
*
*****

```

# ``` SECTION printer ```

## ``` *----- Included Files ----- ```

```

INCLUDE          "exec/types.i"
INCLUDE          "exec/nodes.i"
INCLUDE          "exec/lists.i"
INCLUDE          "exec/memory.i"
INCLUDE          "exec/ports.i"
INCLUDE          "exec/libraries.i"

INCLUDE          "macros.i"

```

## ``` *----- Imported Functions ----- ```

```

XREF_EXE         CloseLibrary
XREF_EXE         OpenLibrary
XREF             _AbsExecBase

XREF             _PEDData

```

## ``` *----- Exported Globals ----- ```

```

XDEF             _Init
XDEF             _Expunge
XDEF             _Open
XDEF             _Close
XDEF             _PD
XDEF             _PED
XDEF             _SysBase
XDEF             _DOSBase
XDEF             _GfxBase
XDEF             _IntuitionBase

```

```
*****
```

```

SECTION          printer,DATA
_PD              DC.L    0
_PED             DC.L    0
_SysBase         DC.L    0
_DOSBase         DC.L    0
_GfxBase         DC.L    0
_IntuitionBase   DC.L    0

```

```
*****
```

```
SECTION          printer,CODE
```

```

_Init:
    MOVE.L 4(A7),_PD
    LEA    PEDData(PC),A0
    MOVE.L A0,_PED
    MOVE.L A6,-(A7)
    MOVE.L _AbsExecBase,A6
    MOVE.L A6,_SysBase

*      ;----- open the dos library
    LEA    DLName(PC),A1
    MOVEQ  #0,D0
    CALLEXE OpenLibrary
    MOVE.L D0,_DOSBase
    BEQ    initDLerr

*      ;----- open the graphics library
    LEA    GLName(PC),A1
    MOVEQ  #0,D0
    CALLEXE OpenLibrary
    MOVE.L D0,_GfxBase
    BEQ    initGLErr

*      ;----- open the intuition library
    LEA    ILName(PC),A1
    MOVEQ  #0,D0
    CALLEXE OpenLibrary
    MOVE.L D0,_IntuitionBase
    BEQ    initILerr

    MOVEQ  #0,D0

pdIRts:
    MOVE.L (A7)+,A6
    RTS

initPAerr:
    MOVE.L _IntuitionBase,A1
    LINKEXE CloseLibrary

initILerr:
    MOVE.L _GfxBase,A1
    LINKEXE CloseLibrary

initGLErr:
    MOVE.L _DOSBase,A1
    LINKEXE CloseLibrary

initDLerr:
    MOVEQ  #-1,D0
    BRA.S  pdIRts

ILName:
    DC.B   'intuition.library'
    DC.B   0

DLName:
    DC.B   'dos.library'
    DC.B   0

GLName:
    DC.B   'graphics.library'
    DC.B   0
    DS.W   0

```

```

*-----
_Expunge:
    MOVE.L _IntuitionBase,A1
    LINKEXE CloseLibrary

    MOVE.L _GfxBase,A1
    LINKEXE CloseLibrary

    MOVE.L _DOSBase,A1

```

# LINKEXE CloseLibrary

```

-----
_Open:
        MOVEQ    #0,D0
        RTS

-----
_Close:
        MOVEQ    #0,D0
        RTS

        END

```

## EPSONQ: DATA.C

```

/*
    Data.c table for EpsonQ driver.
    David Berezowski - March/88.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

char *CommandTable[] = {
    "\375\033@\375", /* 00 aRIS reset */
    "\377",          /* 01 aRIN initialize */
    "\377",          /* 02 aIND linefeed */
    "\012\015",      /* 03 aNEL CRLF */
    "\377",          /* 04 aRI reverse LF */

    /* 05 aSGR0 normal char set */
    "\0335\033-\376\033F",
    "\0334",          /* 06 aSGR3 italics on */
    "\0335",          /* 07 aSGR23 italics off */
    "\033-\001",      /* 08 aSGR4 underline on */
    "\033-\376",      /* 09 aSGR24 underline off */
    "\033E",          /* 10 aSGR1 boldface on */
    "\033F",          /* 11 aSGR22 boldface off */
    "\377",          /* 12 aSFC set foreground color */
    "\377",          /* 13 aSBC set background color */

    /* 14 aSHORP0 normal pitch */
    "\033P\022\033W\376",
    /* 15 aSHORP2 elite on */
    "\033M\022\033W\376",
    "\033P",          /* 16 aSHORP1 elite off */
    /* 17 aSHORP4 condensed fine on */
    "\017\033P\033W\376",
    "\022",          /* 18 aSHORP3 condensed fine off */
    "\033W\001",      /* 19 aSHORP6 enlarge on */
    "\033W\376",      /* 20 aSHORP5 enlarge off */

    "\377",          /* 21 aDEN6 shadow print on */
    "\377",          /* 22 aDEN5 shadow print off */
    "\033G",          /* 23 aDEN4 double strike on */
    "\033H",          /* 24 aDEN3 double strike off */
    "\033x\001",      /* 25 aDEN2 NLQ on */
    "\033x\376",      /* 26 aDEN1 NLQ off */
}

```



```

"\033S\376", /* 27 aSUS2 superscript on */
"\033T", /* 28 aSUS1 superscript off */
"\033S\001", /* 29 aSUS4 subscript on */
"\033T", /* 30 aSUS3 subscript off */
"\033T", /* 31 aSUS0 normalize the line */
"\377", /* 32 aPLU partial line up */
"\377", /* 33 aPLD partial line down */

"\033R\376", /* 34 aFNT0 Typeface 0 */
"\033R\001", /* 35 aFNT1 Typeface 1 */
"\033R\002", /* 36 aFNT2 Typeface 2 */
"\033R\003", /* 37 aFNT3 Typeface 3 */
"\033R\004", /* 38 aFNT4 Typeface 4 */
"\033R\005", /* 39 aFNT5 Typeface 5 */
"\033R\006", /* 40 aFNT6 Typeface 6 */
"\033R\007", /* 41 aFNT7 Typeface 7 */
"\033R\010", /* 42 aFNT8 Typeface 8 */
"\033R\011", /* 43 aFNT9 Typeface 9 */
"\033R\012", /* 44 aFNT10 Typeface 10 */

"\033p1", /* 45 aPROP2 proportional on */
"\033p0", /* 46 aPROP1 proportional off */
"\377", /* 47 aPROP0 proportional clear */
"\377", /* 48 aTSS set proportional offset */
/* 49 aJFY5 auto left justify */
"\033x\001\033a\376", /* 50 aJFY7 auto right justify */
"\033x\001\033a\002", /* 51 aJFY6 auto full justify */
"\033x\001\033a\003", /* 52 aJFY0 auto justify off */
"\033a\376", /* 53 aJFY3 letter space */
"\377", /* 54 aJFY1 word fill */
"\033x\001\033a\001",

"\0330", /* 55 aVERP0 1/8" line spacing */
"\0332", /* 56 aVERP1 1/6" line spacing */
"\377", /* 57 aSLPP set form length */
"\377", /* 58 aPERF perf skip n (n > 0) */
"\0330", /* 59 aPERF0 perf skip off */

"\377", /* 60 aLMS set left margin */
"\377", /* 61 aRMS set right margin */
"\377", /* 62 aTMS set top margin */
"\377", /* 63 aBMS set bottom margin */
"\377", /* 64 aSTBM set T&B margins */
"\377", /* 65 aSLRM set L&R margins */
"\377", /* 66 aCAM clear margins */

"\377", /* 67 aHTS set horiz tab */
"\377", /* 68 aVTS set vert tab */
"\377", /* 69 aTBC0 clear horiz tab */
"\033D\376", /* 70 aTBC3 clear all horiz tabs */
"\377", /* 71 aTBC1 clear vert tab */
"\377", /* 72 aTBC4 clear all vert tabs */
"\033D\376", /* 73 aTBCALL clear all h & v tabs */
/* 74 aTBSALL set default tabs */

"\033D\010\020\030\040\050\060\070\100\110\120\130\140\150\160\170\200\376",

"\377", /* 75 aEXTEND extended commands */
"\377", /* 76 aRAW next 'n' chars are raw */
};

char *ExtendedCharTable[] = {
    " ", /* NBS */
    "\033R\007[\033R\0", /* i */
    "c\010|", /* cl */
    "\033R\003#\033R\0", /* L- */
    "\033R\005$\033R\0", /* o */
    "\033R\010\\\033R\0", /* Y- */
    "|", /* | */
    "\033R\002@\033R\0", /* SS */

```

```

"\033R\001~\033R\0",      /* " */
"c",                          /* copyright */
"\033S\0a\010_\033T",      /* a */
"<",                          /* < */
"-",                          /* - */
"_",                          /* SHY */
"r",                          /* registered trademark */
"-",                          /* - */

"\033R\001{\033R\0",        /* degrees */
"+\010 ",                     /* + */
"\033S\0002\033T",          /* 2 */
"\033S\0003\033T",          /* 3 */
"/",                           /* ' */
"u",                          /* u */
"p",                          /* reverse P */
"\033S\000.\033T",           /* . */

",",                           /* , */
"\033S\0001\033T",           /* 1 */
"\033R\001{\033R\0\010-",     /* o */
">",                           /* > */
"\033S\0001\033T\010-\010\033S\0014\033T", /* 1/4 */
"\033S\0001\033T\010-\010\033S\0012\033T", /* 1/2 */
"\033S\0003\033T\010-\010\033S\0014\033T", /* 3/4 */
"\033R\007}\033R\0",         /* upside down ? */

"A\010'",                     /* 'A */
"A\010'",                     /* 'A */
"A\010^",                     /* ^A */
"A\010~",                     /* ~A */
"\033R\002{\033R\0",         /* {A */
"\033R\004}\033R\0",         /* oA */
"\033R\004{\033R\0",         /* AE */
"C\010,",                     /* C, */

"E\010'",                     /* 'E */
"\033R\011e\033R\0",         /* 'E */
"E\010^",                     /* ^E */
"E\010\033R\001~\033R\0",   /* ~E */
"I\010'",                     /* 'I */
"I\010'",                     /* 'I */
"I\010^",                     /* ^I */
"I\010\033R\001~\033R\0",   /* ~I */

"D\010-",                     /* -D */
"\033R\007\\\\\033R\0",      /* \N */
"O\010'",                     /* 'O */
"O\010'",                     /* 'O */
"O\010^",                     /* ^O */
"O\010~",                     /* ~O */
"\033R\002\\\\\033R\0",      /* "O */
"x",                           /* x */

"\033R\004\\\\\033R\0",      /* O */
"U\010'",                     /* 'U */
"U\010'",                     /* 'U */
"U\010^",                     /* ^U */
"\033R\002}\033R\0",         /* }U */
"Y\010'",                     /* 'Y */
"T",                           /* Thorn */
"\033R\002~\033R\0",         /* B */

"\033R\001e\033R\0",         /* 'a */
"a\010'",                     /* 'a */
"a\010^",                     /* ^a */
"a\010~",                     /* ~a */
"\033R\002{\033R\0",         /* {a */
"\033R\004}\033R\0",         /* oa */
"\033R\004{\033R\0",         /* ae */
"\033R\001\\\\\033R\0",      /* c, */

```

```

"\033R\001\033R\0",      /* 'e */
"\033R\001\033R\0",      /* 'e */
"e\010",                  /* 'e */
"e\010\033R\001\033R\0", /* 'e */
"\033R\006\033R\0",      /* 'i */
"i\010",                  /* 'i */
"i\010",                  /* 'i */
"i\010\033R\001\033R\0", /* 'i */

"d",                       /* d */
"\033R\007\033R\0",      /* n */
"\033R\006\033R\0",      /* o */
"o\010",                  /* o */
"o\010",                  /* o */
"o\010",                  /* o */
"\033R\002\033R\0",      /* o */
":\010",                  /* :- */

"\033R\004\033R\0",      /* o/ */
"\033R\001\033R\0",      /* 'u */
"u\010",                  /* 'u */
"u\010",                  /* ^u */
"\033R\002\033R\0",      /* "u */
"y\010",                  /* 'y */
"t",                      /* thorn */
"y\010\033R\001\033R\0"  /* "y */
};

```

## EPSONQ: DOSPECIAL.C

```

/*
    DoSpecial for EpsonQ driver.
    David Berezowski - March/88.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Params)
char outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Params[];
{
    extern struct PrinterData *PD;

    int x = 0, y = 0;
    /*
        00-01  \0335      italics off
        02-04  \033-\000  underline off
        05-06  \033F      boldface off
        07-08  \033P      elite off
        09-09  \022       condensed fine off
        10-12  \033W\000  enlarge off
        13-14  \033H      double strike off
    */

```

```

15-17  \033x\000      NLQ off
18-19  \033T          normalize the line
20-22  \033R\000      US char set
23-25  \033p0         proportional off
26-27  \0332          6 lpi
28-28  \015           carriage-return

*/
static char initThisPrinter[29] =
{0x1b,'5',0x1b,'-',0xfe,0x1b,'F',0x1b,'P',0x12,0x1b,'W',0xfe,0x1b,'H',
 0x1b,'x',0xfe,0x1b,'T',0x1b,'R',0xfe,0x1b,'p','0',0x1b,'2',0xd};

static char initMarg[] = {0xfd,0x1b,'l','L',0x1b,'Q','R',0xfd};

static BYTE ISOcolorTable[] = {0, 5, 6, 4, 3, 1, 2, 0};

if (*command == aRIN) { /* initialize */
    while(x < 29) {
        if ((outputBuffer[x] = initThisPrinter[x]) == -2) {
            outputBuffer[x] = 0;
        }
        x++;
    }

    if (PD->pd_Preferences.PrintQuality == LETTER) {
        outputBuffer[17] = 1;
    }

    *currentVMI = 30; /* assume 1/6 line spacing */
    if (PD->pd_Preferences.PrintSpacing == EIGHT_LPI) {
        outputBuffer[27] = '0';
        *currentVMI = 22;
    }

    if (PD->pd_Preferences.PrintPitch == ELITE) {
        outputBuffer[8] = 'M';
    }
    else if (PD->pd_Preferences.PrintPitch == FINE) {
        outputBuffer[9] = 15;
    }

    Params[0] = PD->pd_Preferences.PrintLeftMargin;
    Params[1] = PD->pd_Preferences.PrintRightMargin;
    *command = aSLRM;
}

if (*command == aCAM) { /* cancel margins */
    y = PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
    if (PD->pd_Preferences.PrintPitch == PICA) {
        Params[1] = (10 * y) / 10;
    }
    else if (PD->pd_Preferences.PrintPitch == ELITE) {
        Params[1] = (12 * y) / 10;
    }
    else { /* fine */
        Params[1] = (17 * y) / 10;
    }
    Params[0] = 1;
    y = 0;
    *command = aSLRM;
}

if (*command == aSLRM) { /* set left&right margins */
    PD->pd_PWaitEnabled = 253; /* wait after this character */
    if (Params[0] == 0) {
        initMarg[3] = 0;
    }
    else {
        initMarg[3] = Params[0] - 1;
    }
    initMarg[6] = Params[1];
    while (y < 8) {
        outputBuffer[x++] = initMarg[y++];
    }
}

```

```

        }
        return(x);
    }

    if (*command == aPLU) { /* partial line up */
        if (*vline == 0) {
            *vline = 1;
            *command = aSUS2;
            return(0);
        }
        if (*vline < 0) {
            *vline = 0;
            *command = aSUS3;
            return(0);
        }
        return(-1);
    }

    if (*command == aPLD) { /* partial line down */
        if (*vline == 0) {
            *vline = -1;
            *command = aSUS4;
            return(0);
        }
        if (*vline > 0) {
            *vline = 0;
            *command = aSUS1;
            return(0);
        }
        return(-1);
    }

    if (*command == aSUS0) { /* normalize the line */
        *vline = 0;
    }

    if (*command == aSUS1) { /* superscript off */
        *vline = 0;
    }

    if (*command == aSUS2) { /* superscript on */
        *vline = 1;
    }

    if (*command == aSUS3) { /* subscript off */
        *vline = 0;
    }

    if (*command == aSUS4) { /* subscript on */
        *vline = -1;
    }

    if (*command == aVERP0) { /* 8 LPI */
        *currentVMI = 22;
    }

    if (*command == aVERP1) { /* 6 LPI */
        *currentVMI = 30;
    }

    if (*command == aSFC) { /* set foreground/background color */
        if (Parms[0] == 39) {
            Parms[0] = 30; /* set default (black) */
        }
        if (Parms[0] > 37) {
            return(0); /* no or background color change */
        }
        outputBuffer[x++] = 27;
        outputBuffer[x++] = 'r';
        outputBuffer[x++] = ISOcolorTable[Parms[0] - 30];
        return(x);
    }
}

```

```

    if (*command == aSLPP) { /* set form length */
        outputBuffer[x++] = 27;
        outputBuffer[x++] = 'C';
        outputBuffer[x++] = Params[0];
        return(x);
    }

    if (*command == aPERF) { /* perf skip n */
        outputBuffer[x++] = 27;
        outputBuffer[x++] = 'N';
        outputBuffer[x++] = Params[0];
        return(x);
    }

    if (*command == aRIS) { /* reset */
        PD->pd_PWaitEnabled = 253;
    }

    return(0);
}

```

## EPSONQ: RENDER.C

```

/*
    EpsonQ (LQ-800/LQ-850/LQ-1000/LQ-1050/LQ-1500/LQ-2500) driver.
    (tested on a Star NB24-15 (bw) and an Epson LQ-2500 (color) printer).
    David Berezowski - October/87.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

    Executables based on this information may be used in software for
    Commodore Amiga computers. All other rights reserved.

    This information is provided "as is"; no warranties are made. All use
    is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <devices/prtgfx.h>

#define NUMSTARTCMD      8      /* # of cmd bytes before binary data */
#define NUMENDCMD        1      /* # of cmd bytes after binary data */
#define NUMTOTALCMD (NUMSTARTCMD + NUMENDCMD) /* total of above */
#define NUMLFCMD         4      /* # of cmd bytes for linefeed */
#define MAXCOLORBUFS     4      /* max # of color buffers */

#define STARTLEN         16
#define PITCH             1
#define CONDENSED         2
#define LMARG             8
#define RMARG            11
#define DIREC            15

Render(ct, x, y, status)
long ct, x, y, status;
{
    extern void *AllocMem(), FreeMem();

    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    static UWORD RowSize, ColorSize, BufSize, TotalBufSize;

```

```

static UWORD dataoffset, dpi_code;
static UWORD colors[MAXCOLORBUFS]; /* color ptrs */
static UWORD colorcodes[MAXCOLORBUFS]; /* printer color codes */
static UWORD NumColorBufs; /* actually number of color buffers req. */
/*
    00-01  \003P          set pitch (10 or 12 cpi)
    02-02  \022          set condensed fine (on or off)
    03-05  \033W\000     enlarge off
    06-08  \033ln        set left margin to n
    09-11  \033Qn        set right margin to n
    12-12  \015          carriage return
    13-15  \033U1        set uni-directional mode
*/
static UBYTE StartBuf[STARTLEN] =
{0x1b,'P',0x12,0x1b,'W',0x0,0x1b,'1','n',0x1b,'Q','n',0x0d,0x1b,'U','1'};

UBYTE *ptr, *ptrstart, *ptr2, *ptr2start, **dummy;

int i, err;

switch(status) {
    case 0: /* Master Initialization */
        /*
            ct      - pointer to IODRPreq structure.
            x      - width of printed picture in pixels.
            y      - height of printed picture in pixels.
        */
        RowSize = x * 3;
        ColorSize = RowSize + NUMTOTALCMD;
        if (PD->pd_Preferences.PrintShade == SHADE_COLOR) {
            NumColorBufs = MAXCOLORBUFS;
            colors[0] = ColorSize * 3; /* Black */
            colors[1] = ColorSize * 0; /* Yellow */
            colors[2] = ColorSize * 1; /* Magenta */
            colors[3] = ColorSize * 2; /* Cyan */
            colorcodes[0] = 4; /* Yellow */
            colorcodes[1] = 1; /* Magenta */
            colorcodes[2] = 2; /* Cyan */
            colorcodes[3] = 0; /* Black */
        }
        else { /* grey-scale or black&white */
            NumColorBufs = 1;
            colors[0] = ColorSize * 0; /* Black */
            colorcodes[0] = 0; /* Black */
        }
        BufSize = ColorSize * NumColorBufs + NUMLFCMD;
        TotalBufSize = BufSize * 2;
        PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
        if (PD->pd_PrintBuf == NULL) {
            err = PDERR_BUFFERMEMORY; /* no mem */
        }
        else {
            dataoffset = NUMSTARTCMD;
            /*
                This printer prints graphics within its
                text margins. This code makes sure the
                printer is in 10 cpi and then sets the
                left and right margins to their minimum
                and maximum values (respectively). A
                carriage return is sent so that the
                print head is at the leftmost position
                as this printer starts printing from
                the print head's position. The printer
                is put into unidirectional mode to
                reduce wavy vertical lines.
            */
            StartBuf[PITCH] = 'P'; /* 10 cpi */
            StartBuf[CONDENSED] = '\022'; /* off */
            /* left margin of 1 */
            StartBuf[LMARG] = 0;
            /* right margin of 80 or 136 */
            StartBuf[RMARG] = PD->pd_Preferences.

```

```

        PaperSize == W_TRACTOR ? 136 : 80;
        /* uni-directional mode */
        StartBuf[DIREC] = '1';
        err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
    }
    break;

case 1: /* Scale, Dither and Render */
    /*
        ct      - pointer to PrtInfo structure.
        x      - 0.
        y      - row # (0 to Height - 1).
    */
    Transfer(ct, y, &PD->pd_PrintBuf[dataoffset], colors);
    err = PDERR_NOERR; /* all ok */
    break;

case 2: /* Dump Buffer to Printer */
    /*
        ct      - 0.
        x      - 0.
        y      - # of rows sent (1 to NumRows).
    */
    /* white-space strip */
    ptrstart = &PD->pd_PrintBuf[dataoffset];
    ptr2start = ptr2 = ptrstart - NUMSTARTCMD;
    x = 0; /* flag no transfer required yet */
    for (ct=0; ct<NumColorBufs;
        ct++, ptrstart += ColorSize) {
        i = RowSize;
        ptr = ptrstart + i - 1;
        while (i > 0 && *ptr == 0) {
            i--;
            ptr--;
        }
        if (i != 0) { /* if data */
            /* convert to # of pixels */
            i = (i + 2) / 3;
            ptr = ptrstart - NUMSTARTCMD;
            *ptr++ = 27;
            *ptr++ = 'r';
            *ptr++ = colorcodes[ct]; /* color */
            *ptr++ = 27;
            *ptr++ = '*';
            *ptr++ = dpi_code; /* density */
            *ptr++ = i & 0xff;
            *ptr++ = i >> 8; /* size */
            i *= 3; /* back to # of bytes used */
            *(ptrstart + i) = 13; /* cr */
            i += NUMTOTALCMD;
            /* if must transfer data */
            if (x != 0) {
                /* get src start */
                ptr = ptrstart - NUMSTARTCMD;
                /* otherwise Lattice loses
                   track of the pointer.... */
                dummy = &ptr;
                /* xfer and update dest ptr */
                do {
                    *ptr2++ = *ptr++;
                } while (--i);
            }
            else { /* no transfer required */
                /* update dest ptr */
                ptr2 += i;
            }
        }
        /* if compacted or 0 */
        if (i != RowSize + NUMTOTALCMD) {
            /* we need to transfer next time */
            x = 1;
        }
    }

```



```

    }
    *ptr2++ = 13; /* cr */
    *ptr2++ = 27;
    *ptr2++ = 'J';
    *ptr2++ = y; /* y/180 lf */
    err = (*(PD->pd_PWrite))(ptr2start, ptr2 - ptr2start);
    if (err == PDERR_NOERR) {
        dataoffset = (dataoffset == NUMSTARTCMD ?
            BufSize : 0) + NUMSTARTCMD;
    }
    break;

case 3: /* Clear and Init Buffer */
    /*
        ct      - 0.
        x      - 0.
        y      - 0.
    */
    ptr = &PD->pd_PrintBuf[dataoffset];
    i = BufSize - NUMTOTALCMD - NUMLFCMD;
    do {
        *ptr++ = 0;
    } while (--i);
    err = PDERR_NOERR; /* all ok */
    break;

case 4: /* Close Down */
    /*
        ct      - error code.
        x      - io_Special flag from IODRPreReq.
        y      - 0.
    */
    err = PDERR_NOERR; /* assume all ok */
    /* if user did not cancel print */
    if (ct != PDERR_CANCEL) {
        /* restore preferences pitch and margins */
        if (PD->pd_Preferences.PrintPitch == ELITE) {
            StartBuf[PITCH] = 'M'; /* 12 cpi */
        }
        else if (PD->pd_Preferences.PrintPitch == FINE) {
            StartBuf[CONDENSED] = '\017'; /* on */
        }
        StartBuf[LMARG] =
            PD->pd_Preferences.PrintLeftMargin - 1;
        StartBuf[RMARG] =
            PD->pd_Preferences.PrintRightMargin;
        StartBuf[DIREC] = '0'; /* bi-directional */
        err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
    }
    /* wait for both buffers to empty */
    (*(PD->pd_PBothReady))();
    if (PD->pd_PrintBuf != NULL) {
        FreeMem(PD->pd_PrintBuf, TotalBufSize);
    }
    break;

case 5: /* Pre-Master Initialization */
    /*
        ct      - 0 or pointer to IODRPreReq structure.
        x      - io_Special flag from IODRPreReq.
        y      - 0.
    */
    /*
        Kludge for weak power supplies.
        FANFOLD - use all 24 pins (default).
        SINGLE  - use only 16 pins.
    */
    PED->ped_NumRows = PD->pd_Preferences.PaperType ==
        SINGLE ? 16 : 24;
    dpi_code = SetDensity(x & SPECIAL_DENSITYMASK);
    err = PDERR_NOERR; /* all ok */
    break;

```

```

    }
    return(err);
}

```

## EPSONQ: DENSITY.C

```

/*
    Density module for EpsonQ driver.
    David Berezowski - October/87
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

char SetDensity(ULONG);

char SetDensity(density_code)
ULONG density_code;
{
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    /* SPECIAL_DENSITY    0    1    2    3    4    5    6    7 */
    static int XDPI[8] = {90, 90, 120, 180, 360, 360, 360, 360};
    static char codes[8] = {38, 38, 33, 39, 40, 40, 40, 40};

    PED->ped_MaxColumns =
        PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
    density_code /= SPECIAL_DENSITY1;
    /* default is 80 chars (8.0 in.), W_TRACTOR is 136 chars (13.6 in.) */
    PED->ped_MaxXDots = (XDPI[density_code] * PED->ped_MaxColumns) / 10;
    PED->ped_XDotsInch = XDPI[density_code];
    return(codes[density_code]);
}

```

## EPSONQ: TRANSFER.C

```

/*
    Transfer routine for EpsonQ driver.
    David Berezowski - October/87
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <devices/prtgfx.h>

```

```

Transfer(PInfo, y, ptr, colors)
struct PInfo *PInfo;
UWORD y;      /* row # */
UBYTE *ptr;    /* ptr to buffer */
UWORD *colors; /* indexes to color buffers */
{
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    static UWORD bit_table[8] = {128, 64, 32, 16, 8, 4, 2, 1};
    union colorEntry *ColorInt;
    UBYTE *bptr, *yptr, *mptr, *cptr, Black, Yellow, Magenta, Cyan;
    UBYTE *dmatrix, dvalue, threshold;
    UWORD x, width, sx, *sxptr, bit, x3, ymod;

    /* pre-compute */
    /* printer non-specific, MUST DO FOR EVERY PRINTER */
    x = PInfo->pi_xpos;
    ColorInt = PInfo->pi_ColorInt;
    sxptr = PInfo->pi_ScaleX;
    width = PInfo->pi_width;
    /* printer specific */
    x3 = x * 3;
    ymod = y % PED->ped_NumRows;
    bit = bit_table[ymod & 7];
    ptr += ymod >> 3;
    bptr = ptr + colors[0];
    yptr = ptr + colors[1];
    mptr = ptr + colors[2];
    cptr = ptr + colors[3];

    /* pre-compute threshold; are we thresholding? */
    if (threshold = PInfo->pi_threshold) { /* thresholding */
        dvalue = threshold ^ 15;
        bptr += x3;
        do { /* for all source pixels */
            /* pre-compute intensity values for each component */
            Black = ColorInt->colorByte[PCMBLACK];
            ColorInt++;

            sx = *sxptr++;

            do { /* use this pixel 'sx' times */
                if (Black > dvalue) {
                    *bptr |= bit;
                }
                bptr += 3;
            } while (--sx);
        } while (--width);
    }
    else { /* not thresholding, pre-compute ptr to dither matrix */
        dmatrix = PInfo->pi_dmatrix + ((y & 3) << 2);
        if (PD->pd_Preferences.PrintShade == SHADE_GREYSCALE) {
            bptr += x3;
            do { /* for all source pixels */
                /* compute intensity val for each component */
                Black = ColorInt->colorByte[PCMBLACK];
                ColorInt++;

                sx = *sxptr++;

                do { /* use this pixel 'sx' times */
                    if (Black > dmatrix[x & 3]) {
                        *bptr |= bit;
                    }
                    x++; /* done 1 more printer pixel */
                    bptr += 3;
                } while (--sx);
            } while (--width);
        }
        else { /* color */
            do { /* for all source pixels */

```

```

/* compute intensity val for each component */
Black = ColorInt->colorByte[PCMBLACK];
Yellow = ColorInt->colorByte[PCMYELLOW];
Magenta = ColorInt->colorByte[PCMMAGENTA];
Cyan = ColorInt->colorByte[PCMCYAN];
ColorInt++;

sx = *sxptra++;

do { /* use this pixel 'sx' times */
    dvalue = dmatrix[x & 3];
    if (Black > dvalue) {
        *(bptr + x3) |= bit;
    }
    else { /* black not rendered */
        if (Yellow > dvalue) {
            *(yptr + x3) |= bit;
        }
        if (Magenta > dvalue) {
            *(mptr + x3) |= bit;
        }
        if (Cyan > dvalue) {
            *(cptr + x3) |= bit;
        }
    }
    x++; /* done 1 more printer pixel */
    x3 += 3;
} while (--sx);
} while (--width);

```

# HP\_LASERJET

The driver for the HP\_LaserJet can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
CFLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
LINK = lc:blink
LIB = lib:amiga.lib+lib:lc.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o+density.o
TARGET = HP_LaserJet

    @$(LC) $(CFLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o density.o transfer.o
    @$(LINK) <WITH <
    FROM $(OBJ)
    TO $(TARGET)
    LIBRARY $(LIB)
    NODEBUG SC SD VERBOSE MAP $(TARGET).map H
    <

init.o: init.asm
    @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm hp_rev.i
    @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.c

dospecial.o: dospecial.c

data.o: data.c

density.o: density.c

render.o: render.c

install:
    @copy $(TARGET) to devs:printers
```

## HP\_LASERJET: PRINTERTAG.ASM

```
TTL      '$Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $'
*****
*
* Copyright 1985, Commodore-Amiga Inc. All rights reserved.
* No part of this program may be reproduced, transmitted,
* transcribed, stored in retrieval system, or translated into
* any language or computer language, in any form or by any
* means, electronic, mechanical, magnetic, optical, chemical,
* manual or otherwise, without the prior written permission of
* Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
* Pennsylvania, 19380
*
*****
*
* printer device dependent code tag
*
* Source Control
* -----
```

```

* $Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $
*
* $Locker: $
*
* $Log: printer.4,v $
* Revision 1.2 89/11/05 23:56:38 ken
* added blank page to the end
*
* Revision 1.1 89/10/31 16:06:08 ken
* Initial revision
*
* Revision 1.3 88/06/05 18:15:28 daveb
* V1.3 Gamma 15
*
* Revision 1.2 88/04/15 16:58:35 daveb
* fixed docs for devcon
* V1.3 Gamma 13
*
* Revision 1.1 87/10/27 15:33:12 daveb
* V1.3 gamma 1 check-in
*
* Revision 1.0 87/08/20 14:12:28 daveb
* added to rcs
*
* Revision 1.0 87/08/20 13:28:44 daveb
* added to rcs
*
* Revision 1.3 87/08/03 11:09:33 daveb
* added null ptr to char conversion function at end of table
*
* Revision 1.2 87/07/30 10:37:11 daveb
* added 'DS.L 1' at end to reserve space for PrintMode
*
* Revision 1.1 87/07/21 11:42:04 daveb
* added 'PPC_VERSION_2' to PrinterClass
*
* Revision 1.0 87/07/21 11:41:36 daveb
* added to rcs
*
* Revision 32.4 86/06/30 21:09:33 andy
* *** empty log message ***
*
* Revision 32.3 86/06/30 20:55:47 andy
* enabled 8 bit characters
*
* Revision 32.2 86/06/10 12:58:00 andy
* Corrected printer name
*
* Revision 32.1 86/02/10 14:33:17 kodiak
* add null 8BitChars field
*
* Revision 32.0 86/02/10 14:23:56 kodiak
* added to rcs for updating
*
* Revision 1.2 85/10/09 23:58:23 kodiak
* replace reference to pdata w/ prtbase
*
* Revision 1.1 85/10/09 16:11:31 kodiak
* daveb density changes
*
* Revision 25.1 85/06/16 01:02:15 kodiak
* *** empty log message ***
*
* Revision 25.0 85/06/15 06:40:00 kodiak
* added to rcs
*
* Revision 25.0 85/06/13 18:53:36 kodiak
* added to rcs
*
*****

```

```

SECTION          printer

*----- Included Files -----
INCLUDE          "exec/types.i"
INCLUDE          "exec/nodes.i"
INCLUDE          "exec/strings.i"

INCLUDE          "hp_rev.i"

INCLUDE          "devices/prtbase.i"

*----- Imported Names -----
XREF             _Init
XREF             _Expunge
XREF             _Open
XREF             _Close
XREF             _CommandTable
XREF             _PrinterSegmentData
XREF             _DoSpecial
XREF             _Render
XREF             _ExtendedCharTable
XREF             _ConvFunc

*----- Exported Names -----
XDEF             _PEDData

*****

MOVEQ    #0,D0          ; show error for OpenLibrary()
RTS
DC.W     VERSION
DC.W     REVISION

_PEDData:
DC.L     printerName
DC.L     _Init
DC.L     _Expunge
DC.L     _Open
DC.L     _Close
DC.B     PPC_BWGFx      ; PrinterClass
DC.B     PCC_BW         ; ColorClass
DC.B     0              ; MaxColumns
DC.B     0              ; NumCharSets
DC.W     1              ; NumRows
DC.L     600            ; MaxXDots
DC.L     795            ; MaxYDots
DC.W     75             ; XDotsInch
DC.W     75             ; YDotsInch
DC.L     _CommandTable  ; Commands
DC.L     _DoSpecial
DC.L     _Render
DC.L     30             ; Timeout
DC.L     _ExtendedCharTable ; 8BitChars
DS.L     1              ; PrintMode (reserve space)
DC.L     _ConvFunc      ; ptr to char conversion function

printerName:
DC.B     "HP_LaserJet"
DC.B     0
END

```

## HP\_LASERJET: HP\_REV.I

VERSION EQU 35  
REVISION EQU 0

## HP\_LASERJET: INIT.ASM

```
TTL      '$Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $'
*****
*
*   Copyright 1985, Commodore-Amiga Inc.   All rights reserved.
*   No part of this program may be reproduced, transmitted,
*   transcribed, stored in retrieval system, or translated into
*   any language or computer language, in any form or by any
*   means, electronic, mechanical, magnetic, optical, chemical,
*   manual or otherwise, without the prior written permission of
*   Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
*   Pennsylvania, 19380
*
*****
*
*   printer device functions
*
*   Source Control
*   -----
*   $Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $
*
*   $Locker: carolyn $
*
*   $Log: printer.4,v $
*   Revision 1.2  89/11/05  23:56:38  ken
*   added blank page to the end
*
*   Revision 1.1  89/10/31  16:06:08  ken
*   Initial revision
*
*   Revision 1.0  87/08/20  14:12:39  daveb
*   added to rcs
*
*   Revision 1.0  87/08/20  13:30:11  daveb
*   added to rcs
*
*   Revision 1.1  85/10/09  19:27:38  kodiak
*   remove _stdout variable
*
*   Revision 1.0  85/10/09  19:23:53  kodiak
*   added to rcs for updating in version 1
*
*   Revision 29.1 85/08/02  16:58:43  kodiak
*   remove dummy _Close routine -- it's used to finish print of last page.
*
*   Revision 29.0 85/08/02  16:58:17  kodiak
*   added to rcs for updating in version 29
*
*   Revision 25.0 85/06/16  01:01:22  kodiak
*   added to rcs
*
*****
SECTION      printer
*----- Included Files -----
```



```

INCLUDE      "exec/types.i"
INCLUDE      "exec/nodes.i"
INCLUDE      "exec/lists.i"
INCLUDE      "exec/memory.i"
INCLUDE      "exec/ports.i"
INCLUDE      "exec/libraries.i"

INCLUDE      "macros.i"

*----- Imported Functions -----

XREF_EXE      CloseLibrary
XREF_EXE      OpenLibrary
XREF          _AbsExecBase

XREF          _PEDData

*----- Exported Globals -----

XDEF          _Init
XDEF          _Expunge
XDEF          _Open
XDEF          _PD
XDEF          _PED
XDEF          _SysBase
XDEF          _DOSBase
XDEF          _GfxBase
XDEF          _IntuitionBase

*****
SECTION      printer, DATA
_PD          DC.L      0
_PED         DC.L      0
_SysBase     DC.L      0
_DOSBase     DC.L      0
_GfxBase     DC.L      0
_IntuitionBase DC.L      0

*****
SECTION      printer, CODE
_Init:
MOVE.L      4(A7), _PD
LEA        _PEDData(PC), A0
MOVE.L      A0, _PED
MOVE.L      A6, -(A7)
MOVE.L      _AbsExecBase, A6
MOVE.L      A6, _SysBase

*          ;----- open the dos library
LEA        DLName(PC), A1
MOVEQ      #0, D0
CALLEXE    OpenLibrary
MOVE.L      D0, _DOSBase
BEQ        initDLerr

*          ;----- open the graphics library
LEA        GLName(PC), A1
MOVEQ      #0, D0
CALLEXE    OpenLibrary
MOVE.L      D0, _GfxBase
BEQ        initGLErr

*          ;----- open the intuition library
LEA        ILName(PC), A1
MOVEQ      #0, D0
CALLEXE    OpenLibrary
MOVE.L      D0, _IntuitionBase

```

```

        BEQ      initILerr
pdiRts:  MOVEQ    #0,D0
        MOVE.L   (A7)+,A6
        RTS

initPAErr:
        MOVE.L   _IntuitionBase,A1
        LINKEXE  _CloseLibrary

initILerr:
        MOVE.L   _GfxBase,A1
        LINKEXE  _CloseLibrary

initGLErr:
        MOVE.L   _DOSBase,A1
        LINKEXE  _CloseLibrary

initDLerr:
        MOVEQ    #-1,D0
        BRA.S    pdiRts

ILName:  DC.B     'intuition.library'
        DC.B     0

DLName:  DC.B     'dos.library'
        DC.B     0

GLName:  DC.B     'graphics.library'
        DC.B     0
        DS.W     0

```

\*-----

```

_Expunge:
        MOVE.L   _IntuitionBase,A1
        LINKEXE  _CloseLibrary

        MOVE.L   _GfxBase,A1
        LINKEXE  _CloseLibrary

        MOVE.L   _DOSBase,A1
        LINKEXE  _CloseLibrary

```

\*-----

```

_Open:  MOVEQ    #0,D0
        RTS

        END

```

## HP\_LASERJET: DATA.C

```

/*
    Data.c table for HP_LaserJet (Plus and II compatible) driver.
    David Berezowski - March/88.
*/

```

```

/* Copyright (c) 1988 Commodore-Amiga, Inc.

```

Executables based on this information may be used in software for Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use is at your own risk, and no liability or responsibility is assumed.

\*/

```

char *CommandTable[] = {
    "\375\033E\375", /* 00 aRIS reset */
    "\377", /* 01 aRIN initialize */
    "\012", /* 02 aIND linefeed */
    "\015\012", /* 03 aNEL CRLF */
    "\033&a-1R", /* 04 aRI reverse LF */

    /* 05 aSGR0 normal char set */
    "\033&d@\033(sbs",
    "\033(s1S", /* 06 aSGR3 italics on */
    "\033(sS", /* 07 aSGR23 italics off */
    "\033&dD", /* 08 aSGR4 underline on */
    "\033&d@", /* 09 aSGR24 underline off */
    "\033(s5B", /* 10 aSGR1 boldface on */
    "\033(sB", /* 11 aSGR22 boldface off */
    "\377", /* 12 aSFC set foreground color */
    "\377", /* 13 aSBC set background color */

    "\033(s10h1T", /* 14 aSHORP0 normal pitch */
    "\033(s12h2T", /* 15 aSHORP2 elite on */
    "\033(s10h1T", /* 16 aSHORP1 elite off */
    "\033(s15H", /* 17 aSHORP4 condensed fine on */
    "\033(s10H", /* 18 aSHORP3 condensed fine off */
    "\377", /* 19 aSHORP6 enlarge on */
    "\377", /* 20 aSHORP5 enlarge off */

    "\033(s7B", /* 21 aDEN6 shadow print on */
    "\033(sB", /* 22 aDEN5 shadow print off */
    "\033(s3B", /* 23 aDEN4 double strike on */
    "\033(sB", /* 24 aDEN3 double strike off */
    "\377", /* 25 aDEN2 NLQ on */
    "\377", /* 26 aDEN1 NLQ off */

    "\377", /* 27 aSUS2 superscript on */
    "\377", /* 28 aSUS1 superscript off */
    "\377", /* 29 aSUS4 subscript on */
    "\377", /* 30 aSUS3 subscript off */
    "\377", /* 31 aSUS0 normalize the line */
    "\033&a-.5R", /* 32 aPLU partial line up */
    "\033=", /* 33 aPLD partial line down */

    "\033(s3T", /* 34 aFNT0 Typeface 0 */
    "\033(s0T", /* 35 aFNT1 Typeface 1 */
    "\033(s1T", /* 36 aFNT2 Typeface 2 */
    "\033(s2T", /* 37 aFNT3 Typeface 3 */
    "\033(s4T", /* 38 aFNT4 Typeface 4 */
    "\033(s5T", /* 39 aFNT5 Typeface 5 */
    "\033(s6T", /* 40 aFNT6 Typeface 6 */
    "\033(s7T", /* 41 aFNT7 Typeface 7 */
    "\033(s8T", /* 42 aFNT8 Typeface 8 */
    "\033(s9T", /* 43 aFNT9 Typeface 9 */
    "\033(s10T", /* 44 aFNT10 Typeface 10 */

    "\033(s1P", /* 45 aPROP2 proportional on */
    "\033(sP", /* 46 aPROP1 proportional off */
    "\033(sP", /* 47 aPROP0 proportional clear */
    "\377", /* 48 aTSS set proportional offset */
    "\377", /* 49 aJFY5 auto left justify */
    "\377", /* 50 aJFY7 auto right justify */
    "\377", /* 51 aJFY6 auto full justify */
    "\377", /* 52 aJFY0 auto justify off */
    "\377", /* 53 aJFY3 letter space */
    "\377", /* 54 aJFY1 word fill */

    "\033&l8D", /* 55 aVERP0 1/8" line spacing */
    "\033&l6D", /* 56 aVERP1 1/6" line spacing */
    "\377", /* 57 aSLPP set form length */
    "\033&l1L", /* 58 aPERF perf skip n (n > 0) */
    "\033&lL", /* 59 aPERF0 perf skip off */

```

```

"\377",      /* 60 aLMS set left margin      */
"\377",      /* 61 aRMS set right margin     */
"\377",      /* 62 aTMS set top margin       */
"\377",      /* 63 aBMS set bottom margin    */
"\377",      /* 64 aSTBM set T&B margins     */
"\377",      /* 65 aSLRM set L&R margins     */
"\0339\015", /* 66 aCAM clear margins       */

"\377",      /* 67 aHTS set horiz tab       */
"\377",      /* 68 aVTS set vert tab        */
"\377",      /* 69 aTBC0 clear horiz tab     */
"\377",      /* 70 aTBC3 clear all horiz tabs */
"\377",      /* 71 aTBC1 clear vert tab     */
"\377",      /* 72 aTBC4 clear all vert tabs */
"\377",      /* 73 aTBCALL clear all h & v tabs */
"\377",      /* 74 aTBSALL set default tabs */

"\377",      /* 75 aEXTEND extended commands */
"\377"       /* 76 aRAW next 'n' chars are raw */
};

```

```

char *ExtendedCharTable[] = {
/*
  " ", "!", "c", "L", "o", "Y", "|", "S",
  "\", "c", "a", "<", "~", "-", "r", "-",
  "*", "+", "2", "3", "'", "u", "p", ".",
  ",", "1", "o", ">", "/", "\\", "\\", "?",
  "A", "A", "A", "A", "A", "A", "A", "C",
  "E", "E", "E", "E", "I", "I", "I", "I",
  "D", "N", "O", "O", "O", "O", "O", "x",
  "O", "U", "U", "U", "U", "Y", "p", "B",
  "a", "a", "a", "a", "a", "a", "a", "c",
  "e", "e", "e", "e", "i", "i", "i", "i",
  "d", "n", "o", "o", "o", "o", "o", "/",
  "o", "u", "u", "u", "u", "y", "p", "y"
*/
  " ", "\270", "\277", "\273", "\272", "\274", "|", "\275",
  "\253", "c", "\371", "\373", "~", "\366", "r", "\260",
  "\263", "\376", "2", "3", "\250", "\363", "\364", "\362",
  ",", "1", "\372", "\375", "\367", "\370", "\365", "\271",
  "\241", "\340", "\242", "\341", "\330", "\320", "\323", "\264",
  "\243", "\334", "\244", "\245", "\346", "\345", "\246", "\247",
  "\343", "\266", "\350", "\347", "\337", "\351", "\332", "x",
  "\322", "\255", "\355", "\256", "\333", "\261", "\360", "\336",
  "\310", "\304", "\300", "\342", "\314", "\324", "\327", "\265",
  "\311", "\305", "\301", "\315", "\331", "\325", "\321", "\335",
  "\344", "\267", "\312", "\306", "\302", "\352", "\316", "-\010:",
  "\326", "\313", "\307", "\303", "\317", "\262", "\361", "\357"
};

```

**HP\_LASERJET: DOSPECIAL.C**

```

/*
    DoSpecial for HP_LaserJet driver.
    David Berezowski - March/88.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

#define LPI          7
#define CPI          15
#define QUALITY      17
#define INIT_LEN     30
#define LPP          7
#define FORM_LEN     11
#define LEFT_MARG    3
#define RIGHT_MARG   7
#define MARG_LEN     12

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Params)
char outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Params[];
{
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    UWORD textlength, topmargin;
    int x, y, j;
    static char initThisPrinter[INIT_LEN] =
        {0x1b, '&', 'd', '@', 0x1b, '&', 'l', '6', 'D', 0x1b, '(', 's', '0', 'b',
         '1', '0', 'h', '1', 'q', '0', 'p', '0', 's', '3', 't', '0', 'u', '1', '2', 'V'};

    static char initForm[FORM_LEN] =
        {0x1b, '&', 'l', '0', '0', '2', 'e', '0', '0', '0', 'F'};

    static char initMarg[MARG_LEN] =
        {0x1b, '&', 'a', '0', '0', '0', '1', '0', '0', '0', 'M', 0x0d};

    static char initTMarg[] =
        {0x1b, '&', 'l', '0', '0', '0', 'e', '0', '0', '0', 'F'};
    x = y = j = 0;

    if (*command == aRIN) {
        while(x < INIT_LEN) {
            outputBuffer[x] = initThisPrinter[x];
            x++;
        }
        outputBuffer[x++] = '\015';

        if (PD->pd_Preferences.PrintSpacing == EIGHT_LPI) {
            outputBuffer[LPI] = '8';
        }

        if (PD->pd_Preferences.PrintPitch == ELITE) {
            outputBuffer[CPI] = '2';
        }
        else if (PD->pd_Preferences.PrintPitch == FINE) {
            outputBuffer[CPI] = '5';
        }
    }
}

```

```

        if (PD->pd_Preferences.PrintQuality == LETTER) {
            outputBuffer[QUALITY] = '2';
        }

        j = x; /* set the formlength = textlength, top margin = 2 */
        textlength = PD->pd_Preferences.PaperLength;
        topmargin = 2;

        while (y < FORM_LEN) {
            outputBuffer[x++] = initForm[y++];
        }
        numberString(textlength, j + LPP, outputBuffer);

        Params[0] = PD->pd_Preferences.PrintLeftMargin;
        Params[1] = PD->pd_Preferences.PrintRightMargin;
        *command = aSLRM;
    }

    if (*command == aSLRM) {
        j = x;
        y = 0;
        while(y < MARG_LEN) {
            outputBuffer[x++] = initMarg[y++];
        }
        numberString(Params[0] - 1, j + LEFT_MARG, outputBuffer);
        numberString(Params[1] - 1, j + RIGHT_MARG, outputBuffer);
        return(x);
    }

    if ((*command == aSUS2) && (*vline == 0)) {
        *command = aPLU;
        *vline = 1;
        return(0);
    }

    if ((*command == aSUS2) && (*vline < 0)) {
        *command = aRI;
        *vline = 1;
        return(0);
    }

    if ((*command == aSUS1) && (*vline > 0)) {
        *command = aPLD;
        *vline = 0;
        return(0);
    }

    if ((*command == aSUS4) && (*vline == 0)) {
        *command = aPLD;
        *vline = -1;
        return(0);
    }

    if ((*command == aSUS4) && (*vline > 0)) {
        *command = aIND;
        *vline = -1;
        return(0);
    }

    if ((*command == aSUS3) && (*vline < 0)) {
        *command = aPLU;
        *vline = 0;
        return(0);
    }

    if(*command == aSUS0) {
        if (*vline > 0) {
            *command = aPLD;
        }
        if (*vline < 0) {
            *command = aPLU;
        }
    }

```

```

        *vline = 0;
        return(0);
    }

    if (*command == aPLU) {
        (*vline)++;
        return(0);
    }

    if (*command == aPLD){
        (*vline)--;
        return(0);
    }

    if (*command == aSTBM) {
        if (Parms[0] == 0) {
            Parms[0] = topmargin;
        }
        else {
            topmargin = --Parms[0];
        }

        if (Parms[1] == 0) {
            Parms[1] = textlength;
        }
        else {
            textlength=Parms[1];
        }
        while (x < 11) {
            outputBuffer[x] = initTMarg[x];
            x++;
        }
        numberString(Parms[0], 3, outputBuffer);
        numberString(Parms[1] - Parms[0], 7, outputBuffer);
        return(x);
    }

    if (*command == aSLPP) {
        while(x < 11) {
            outputBuffer[x] = initForm[x];
            x++;
        }
        /*restore textlength, margin*/
        numberString(topmargin, 3, outputBuffer);
        numberString(textlength, 7, outputBuffer);
        return(x);
    }

    if (*command == aRIS) {
        PD->pd_PWaitEnabled = 253;
    }

    return(0);
}

numberString(Param, x, outputBuffer)
UBYTE Param;
int x;
char outputBuffer[];
{
    if (Param > 199) {
        outputBuffer[x++] = '2';
        Param -= 200;
    }
    else if (Param > 99) {
        outputBuffer[x++] = '1';
        Param -= 100;
    }
    else {
        outputBuffer[x++] = '0'; /* always return 3 digits */
    }
}

```

```

        if (Param > 9) {
            outputBuffer[x++] = Param / 10 + '0';
        }
        else {
            outputBuffer[x++] = '0';
        }

        outputBuffer[x++] = Param % 10 + '0';
    }

ConvFunc(buf, c, flag)
char *buf, c;
int flag; /* expand lf into lf/cr flag (0=yes, else no) */
{
    if (c == '\014') { /* if formfeed (page eject) */
        PED->ped_PrintMode = 0; /* no data to print */
    }
    return(-1); /* pass all chars back to the printer device */
}

Close(ior)
struct printerIO *ior;
{
    if (PED->ped_PrintMode) { /* if data has been printed */
        (*(PD->pd_PWrite))("\014",1); /* eject page */
        (*(PD->pd_PBothReady))(); /* wait for it to finish */
        PED->ped_PrintMode = 0; /* no data to print */
    }
    return(0);
}

```

## HP\_LASERJET: RENDER.C

```

/*
    HP_LaserJet driver.
    David Berezowski - May/87.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

    Executables based on this information may be used in software for
    Commodore Amiga computers. All other rights reserved.

    This information is provided "as is"; no warranties are made. All use
    is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>
#include <devices/prtbase.h>
#include <devices/printer.h>

#define NUMSTARTCMD    7      /* # of cmd bytes before binary data */
#define NUMENDCMD      0      /* # of cmd bytes after binary data */
#define NUMTOTALCMD (NUMSTARTCMD + NUMENDCMD) /* total of above */

extern SetDensity();
/*
    00-04  \033&10L perf skip mode off
    05-11  \033*t075R set raster graphics resolution (dpi)
    12-16  \033*r0A start raster graphics
*/
char StartCmd[17] = {0x1b, '&', '1', '0', 'L', 0x1b, '*', 't', '0', '7', '5', 'R',
                    0x1b, '*', 'r', '0', 'A'};

Render(ct, x, y, status)

```



```

long ct, x, y, status;
{
    extern void *AllocMem(), FreeMem();

    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    static UWORD RowSize, BufSize, TotalBufSize, dataoffset;
    static UWORD huns, tens, ones; /* used to program buffer size */
    UBYTE *ptr, *ptrstart;
    int i, err;

    switch(status) {
        case 0 : /* Master Initialization */
            /*
                ct      - pointer to IODRPreq structure.
                x      - width of printed picture in pixels.
                y      - height of printed picture in pixels.
            */
            RowSize = (x + 7) / 8;
            BufSize = RowSize + NUMTOTALCMD;
            TotalBufSize = BufSize * 2;
            PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
            if (PD->pd_PrintBuf == NULL) {
                err = PDERR_BUFFERMEMORY; /* no mem */
            }
            else {
                ptr = PD->pd_PrintBuf;
                *ptr++ = 27;
                *ptr++ = '*';
                *ptr++ = 'b'; /* transfer raster graphics */
                *ptr++ = huns | '0';
                *ptr++ = tens | '0';
                *ptr++ = ones | '0'; /* printout width */
                *ptr = 'W'; /* terminator */
                ptr = &PD->pd_PrintBuf[BufSize];
                *ptr++ = 27;
                *ptr++ = '*';
                *ptr++ = 'b'; /* transfer raster graphics */
                *ptr++ = huns | '0';
                *ptr++ = tens | '0';
                *ptr++ = ones | '0'; /* printout width */
                *ptr = 'W'; /* terminator */
                dataoffset = NUMSTARTCMD;
                /* perf skip mode off, set dpi, start raster gfx */
                err = (*(PD->pd_PWrite))(StartCmd, 17);
            }
            break;

        case 1 : /* Scale, Dither and Render */
            /*
                ct      - pointer to PrtInfo structure.
                x      - 0.
                y      - row # (0 to Height - 1).
            */
            Transfer(ct, y, &PD->pd_PrintBuf[dataoffset]);
            err = PDERR_NOERR; /* all ok */
            break;

        case 2 : /* Dump Buffer to Printer */
            /*
                ct      - 0.
                x      - 0.
                y      - # of rows sent (1 to NumRows).
            */
            White-space strip.
            /*
                i = RowSize;
                ptrstart = &PD->pd_PrintBuf[dataoffset - NUMSTARTCMD];
                ptr = ptrstart + NUMSTARTCMD + i - 1;
                while (i > 0 && *ptr == 0) {
                    i--;

```

```

        ptr--;
    }
    ptr = ptrstart + 3; /* get ptr to density info */
    *ptr++ = (huns = i / 100) | '0';
    *ptr++ = (i - huns * 100) / 10 | '0';
    *ptr = i % 10 | '0'; /* set printout width */
    err = (*(PD->pd_PWrite))(ptrstart, i + NUMTOTALCMD);
    if (err == PDERR_NOERR) {
        dataoffset = (dataoffset == NUMSTARTCMD ?
            BufSize : 0) + NUMSTARTCMD;
    }
    break;

case 3 : /* Clear and Init Buffer */
    /*
        ct      - 0.
        x      - 0.
        y      - 0.
    */
    ptr = &PD->pd_PrintBuf[dataoffset];
    i = RowSize;
    do {
        *ptr++ = 0;
    } while (--i);
    break;

case 4 : /* Close Down */
    /*
        ct      - error code.
        x      - io_Special flag from IODRPreq struct
        y      - 0.
    */
    err = PDERR_NOERR; /* assume all ok */
    /* if user did not cancel the print */
    if (ct != PDERR_CANCEL) {
        /* end raster graphics, perf skip mode on */
        if ((err = (*(PD->pd_PWrite))
            ("\033*rB\033&l1L", 9)) == PDERR_NOERR) {
            /* if want to unload paper */
            if (!(x & SPECIAL_NOFORMFEED)) {
                /* eject paper */
                err = (*(PD->pd_PWrite))
                    ("\014", 1);
            }
        }
    }
    /*
        flag that there is no alpha data waiting that
        needs a formfeed (since we just did one)
    */
    PED->ped_PrintMode = 0;
    /* wait for both buffers to empty */
    (*(PD->pd_PBothReady))();
    if (PD->pd_PrintBuf != NULL) {
        FreeMem(PD->pd_PrintBuf, TotalBufSize);
    }
    break;

case 5 : /* Pre-Master Initialization */
    /*
        ct      - 0 or pointer to IODRPreq structure.
        x      - io_Special flag from IODRPreq struct
        y      - 0.
    */
    /* select density */
    SetDensity(x & SPECIAL_DENSITYMASK);
    break;
}
return(err);
}

```

## HP\_LASERJET: DENSITY.C

```
/*
    Density module for HP_LaserJet
    David Berezowski - May/87
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

SetDensity(density_code)
ULONG density_code;
{
    extern struct PrinterExtendedData *PED;
    extern char StartCmd[];

    /* SPECIAL_DENSITY      0   1   2   3   4   5   6   7 */
    static int XDPI[8] = {75, 75, 100, 150, 300, 300, 300, 300};
    static char codes[8][3] = {
        {'0','7','5'}, {'0','7','5'}, {'1','0','0'}, {'1','5','0'},
        {'3','0','0'}, {'3','0','0'}, {'3','0','0'}, {'3','0','0'};

    density_code /= SPECIAL_DENSITY1;
    PED->ped_MaxXDots = XDPI[density_code] * 8; /* 8 inches */
    PED->ped_MaxYDots = XDPI[density_code] * 10; /* 10 inches */
    PED->ped_XDotsInch = PED->ped_YDotsInch = XDPI[density_code];
    StartCmd[8] = codes[density_code][0];
    StartCmd[9] = codes[density_code][1];
    StartCmd[10] = codes[density_code][2];
}
```

## HP\_LASERJET TRANSFER.C

```
/*
    Transfer routine for HP_LaserJet driver.
    David Berezowski - October/87.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/prtgfx.h>

Transfer(PInfo, y, ptr)
struct PrtInfo *PInfo;
UWORD y;          /* row # */
UBYTE *ptr;       /* ptr to buffer */
```

```

static UBYTE bit_table[] = {128, 64, 32, 16, 8, 4, 2, 1};
UBYTE *dmatrix, Black, dvalue, threshold;
union colorEntry *ColorInt;
UWORD x, width, sx, *sxptra, bit;

/* pre-compute */
/* printer non-specific, MUST DO FOR EVERY PRINTER */
x = PInfo->pi_xpos; /* get starting x position */
ColorInt = PInfo->pi_ColorInt; /* get ptr to color intensities */
sxptra = PInfo->pi_ScaleX;
width = PInfo->pi_width; /* get # of source pixels */

/* pre-compute threshold; are we thresholding? */
if (threshold = PInfo->pi_threshold) { /* thresholding */
    dvalue = threshold ^ 15; /* yes, so pre-compute dither value */
    do { /* for all source pixels */
        /* pre-compute intensity value for Black */
        Black = ColorInt->colorByte[PCMBLACK];
        ColorInt++; /* bump ptr for next time */

        sx = *sxptra++;

        /* dither and render pixel */
        do { /* use this pixel 'sx' times */
            /* if we should render Black */
            if (Black > dvalue) {
                /* set bit */
                *(ptr + (x >> 3)) |= bit_table[x & 7];
            }
            ++x; /* done 1 more printer pixel */
        } while (--sx);
    } while (--width);
}
else { /* not thresholding, pre-compute ptr to dither matrix */
    dmatrix = PInfo->pi_dmatrix + ((y & 3) << 2);
    do { /* for all source pixels */
        /* pre-compute intensity value for Black */
        Black = ColorInt->colorByte[PCMBLACK];
        ColorInt++; /* bump ptr for next time */

        sx = *sxptra++;

        /* dither and render pixel */
        do { /* use this pixel 'sx' times */
            /* if we should render Black */
            if (Black > dmatrix[x & 3]) {
                /* set bit */
                *(ptr + (x >> 3)) |= bit_table[x & 7];
            }
            ++x; /* done 1 more printer pixel */
        } while (--sx);
    } while (--width);
}
}

```

## XEROX\_4020

The Xerox\_4020 driver can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
FLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
LINK = lc:blink
LIB = lib:amiga.lib+lib:lc.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o
TARGET = Xerox_4020

    @$(LC) $(FLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o transfer.o
    @$(LINK) <WITH <
    FROM $(OBJ)
    TO $(TARGET)
    LIBRARY $(LIB)
    NODEBUG SC SD VERBOSE MAP $(TARGET).map H
    <

init.o: init.asm
    @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm xerox_4020_rev.i
    @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.c

dospecial.o: dospecial.c

data.o: data.c

render.o: render.c

install:
    @copy $(TARGET) to devs:printers
```

## XEROX\_4020: PRINTERTAG.ASM

```
TTL      '$Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $'
*****
*
* Copyright 1985, Commodore-Amiga Inc. All rights reserved.
* No part of this program may be reproduced, transmitted,
* transcribed, stored in retrieval system, or translated into
* any language or computer language, in any form or by any
* means, electronic, mechanical, magnetic, optical, chemical,
* manual or otherwise, without the prior written permission of
* Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
* Pennsylvania, 19380
*
*****
*
* printer device dependent code tag
*
* Source Control
* -----
* $Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $
*
```

```

* $Locker: $
*
* $Log: printer.4,v $
* Revision 1.2 89/11/05 23:56:38 ken
* added blank page to the end
*
* Revision 1.1 89/10/31 16:06:08 ken
* Initial revision
*
* Revision 1.2 88/04/15 17:25:53 daveb
* fixed docs for devcon
* V1.3 Gamma 13
*
* Revision 1.1 88/01/15 15:52:05 daveb
* changed XDotsInch from 120 to 121.
* changed MaxXDots from 1088 to 1080
* V1.3 Gamma 6 release
*
* Revision 1.0 87/08/20 13:53:15 daveb
* added to rcs
*
* Revision 1.4 87/07/30 10:43:13 daveb
* added 'DS.L 1' at end to reserve space for PrintMode
*
* Revision 1.3 87/07/21 11:48:04 daveb
* added 'PPC_VERSION_2' to PrinterClass
*
* Revision 1.2 87/04/30 11:27:14 daveb
* changed MaxColumns from 80 to 90 (Programmer's Guide, pg 1-2)
* changed MaxXDots from 1024 to 1088 (same)
* changed YDotsInch from 240 to 120 (Programmer's Guide, pg 1-6)
*
* Revision 1.1 87/04/29 18:14:32 andy
* Initial revision
*
* Revision 32.4 86/06/30 21:05:59 andy
* *** empty log message ***
*
* Revision 32.3 86/06/30 20:51:12 andy
* enabled 8 bit char support
*
* Revision 32.2 86/06/10 12:55:43 andy
* Corrected printer name
*
* Revision 32.1 86/02/10 14:32:06 kodiak
* add null 8BitChars field
*
* Revision 32.0 86/02/10 14:21:40 kodiak
* added to rcs for updating
*
* Revision 1.1 85/10/09 23:56:42 kodiak
* replace reference to pdata w/ prtbase
*
* Revision 1.0 85/10/09 23:56:36 kodiak
* added to rcs for updating in version 1
*
* Revision 25.1 85/06/16 01:02:15 kodiak
* *** empty log message ***
*
* Revision 25.0 85/06/15 06:40:00 kodiak
* added to rcs
*
* Revision 25.0 85/06/13 18:53:36 kodiak
* added to rcs
*

```

```

*****

```

```

SECTION          printer

```

```

*----- Included Files -----

```

```

INCLUDE      "exec/types.i"
INCLUDE      "exec/nodes.i"
INCLUDE      "exec/strings.i"

INCLUDE      "xerox_4020_rev.i"

INCLUDE      "devices/prtbase.i"

*----- Imported Names -----
XREF         _Init
XREF         _Expunge
XREF         _Open
XREF         _Close
XREF         _CommandTable
XREF         _PrinterSegmentData
XREF         _DoSpecial
XREF         _Render
XREF         _ExtendedCharTable
XREF         _ConvFunc

*----- Exported Names -----
XDEF         _PEDData

*****

MOVEQ        #0,D0          ; show error for OpenLibrary()
RTS
DC.W          VERSION
DC.W          REVISION
_PEDData:
DC.L          printerName
DC.L          _Init
DC.L          _Expunge
DC.L          _Open
DC.L          _Close
DC.B          PPC_COLORGFX   ; PrinterClass
DC.B          PCC_YMCB       ; ColorClass
DC.B          90             ; MaxColumns
DC.B          1              ; NumCharSets
DC.W          4              ; NumRows
DC.L          1080           ; MaxXDots
DC.L          0              ; MaxYDots
DC.W          121            ; XDotsInch
DC.W          120            ; YDotsInch
DC.L          _CommandTable  ; Commands
DC.L          _DoSpecial
DC.L          _Render
DC.L          30             ; Timeout
DC.L          _ExtendedCharTable ; 8BitChars
DS.L          1              ; PrintMode (reserve space)
DC.L          0              ; ptr to char conversion function

printerName:
DC.B          "xerox_4020"
DC.B          0
END

```

**XEROX\_4020: XEROX\_4020\_REV.I**

VERSION EQU 35  
REVISION EQU 0

## XEROX\_4020: INIT.ASM

```
TTL '$Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $'
*****
*
* Copyright 1985, Commodore-Amiga Inc. All rights reserved.
* No part of this program may be reproduced, transmitted,
* transcribed, stored in retrieval system, or translated into
* any language or computer language, in any form or by any
* means, electronic, mechanical, magnetic, optical, chemical,
* manual or otherwise, without the prior written permission of
* Commodore-Amiga Incorporated, 1200 Wilson Drive, West Chester,
* Pennsylvania, 19380
*
*****
*
* printer device functions
*
* Source Control
* -----
* $Header: printer.4,v 1.2 89/11/05 23:56:38 ken Exp $
*
* $Locker: carolyn $
*
* $Log: printer.4,v $
* Revision 1.2 89/11/05 23:56:38 ken
* added blank page to the end
*
* Revision 1.1 89/10/31 16:06:08 ken
* Initial revision
*
* Revision 1.1 88/04/14 12:03:14 daveb
* V1.3 Gamma 11 release
*
* Revision 1.0 87/08/20 14:10:17 daveb
* added to rcs
*
* Revision 1.1 85/10/09 19:27:20 kodiak
* remove _stdout variable
*
* Revision 1.0 85/10/09 19:23:23 kodiak
* added to rcs for updating in version 1
*
* Revision 25.0 85/06/16 01:01:22 kodiak
* added to rcs
*
*****
SECTION printer
*----- Included Files -----
INCLUDE "exec/types.i"
INCLUDE "exec/nodes.i"
INCLUDE "exec/lists.i"
INCLUDE "exec/memory.i"
INCLUDE "exec/ports.i"
INCLUDE "exec/libraries.i"
INCLUDE "macros.i"
*----- Imported Functions -----
XREF_EXE CloseLibrary
```



```

XREF_EXE      OpenLibrary
XREF          _AbsExecBase

```

```

XREF          _PEDData

```

\*----- Exported Globals -----

```

XDEF          _Init
XDEF          _Expunge
XDEF          _Open
XDEF          _Close
XDEF          _PD
XDEF          _PED
XDEF          _SysBase
XDEF          _DOSBase
XDEF          _GfxBase
XDEF          _IntuitionBase

```

\*\*\*\*\*

```

SECTION      printer,DATA
_PD          DC.L      0
_PED         DC.L      0
_SysBase     DC.L      0
_DOSBase     DC.L      0
_GfxBase     DC.L      0
_IntuitionBase DC.L    0

```

\*\*\*\*\*

```

SECTION      printer,CODE
_Init:
    MOVE.L    4(A7),_PD
    LEA       _PEDData(PC),A0
    MOVE.L    A0,_PED
    MOVE.L    A6,-(A7)
    MOVE.L    _AbsExecBase,A6
    MOVE.L    A6,_SysBase

```

```

*          ;----- open the dos library
    LEA       DLName(PC),A1
    MOVEQ     #0,D0
    CALLEXE   OpenLibrary
    MOVE.L    D0,_DOSBase
    BEQ       initDLerr

```

```

*          ;----- open the graphics library
    LEA       GLName(PC),A1
    MOVEQ     #0,D0
    CALLEXE   OpenLibrary
    MOVE.L    D0,_GfxBase
    BEQ       initGLErr

```

```

*          ;----- open the intuition library
    LEA       ILName(PC),A1
    MOVEQ     #0,D0
    CALLEXE   OpenLibrary
    MOVE.L    D0,_IntuitionBase
    BEQ       initILErr

```

```

    MOVEQ     #0,D0
pdIRts:
    MOVE.L    (A7)+,A6
    RTS

```

```

initPAErr:
    MOVE.L    _IntuitionBase,A1
    LINKEXE   CloseLibrary

```

```

initILerr:
    MOVE.L  _GfxBase,A1
    LINKEXE CloseLibrary

initGLErr:
    MOVE.L  _DOSBase,A1
    LINKEXE CloseLibrary

initDLerr:
    MOVEQ   #-1,D0
    BRA.S   pdiRts

ILName:
    DC.B    'intuition.library'
    DC.B    0

DLName:
    DC.B    'dos.library'
    DC.B    0

GLName:
    DC.B    'graphics.library'
    DC.B    0
    DS.W    0

```

```

*-----
_Expunge:

```

```

    MOVE.L  _IntuitionBase,A1
    LINKEXE CloseLibrary

    MOVE.L  _GfxBase,A1
    LINKEXE CloseLibrary

    MOVE.L  _DOSBase,A1
    LINKEXE CloseLibrary

```

```

*-----
_Open:

```

```

    MOVEQ   #0,D0
    RTS

```

```

*-----
_Close:

```

```

    MOVEQ   #0,D0
    RTS

    END

```

## XEROX\_4020: DATA.C

```

/*
    Data.c table for Xerox 4020 driver.
    David Berezowski - March/88.
*/

```

```

/* Copyright (c) 1988 Commodore-Amiga, Inc.

```

```

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

```

```

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

```

```

char *CommandTable[]={
    /* 00 aRIS reset */
}

```

```

"\375\033\015P\375",
"\377",      /* 01 aRIN initialize */
"\012",      /* 02 aIND linefeed */
"\015\012",  /* 03 aNEL CRLF */
"\377",      /* 04 aRI reverse LF */

"\033R",     /* 05 aSGR0 normal char set */
"\377",      /* 06 aSGR3 italics on */
"\377",      /* 07 aSGR23 italics off */
"\033E",     /* 08 aSGR4 underline on */
"\033R",     /* 09 aSGR24 underline off */
"\377",      /* 10 aSGR1 boldface on */
"\377",      /* 11 aSGR22 boldface off */
"\377",      /* 12 aSFC set foreground color */
"\377",      /* 13 aSBC set background color */

"\033F0\033&", /* 14 aSHORP0 normal pitch */
"\033F2",     /* 15 aSHORP2 elite on */
"\033F0",     /* 16 aSHORP1 elite off */
"\033F4",     /* 17 aSHORP4 condensed fine on */
"\033F0",     /* 18 aSHORP3 condensed fine off */
"\033W20",    /* 19 aSHORP6 enlarge on */
"\033&",      /* 20 aSHORP5 enlarge off */

"\377",      /* 21 aDEN6 shadow print on */
"\377",      /* 22 aDEN5 shadow print off */
"\377",      /* 23 aDEN4 double strike on */
"\377",      /* 24 aDEN3 double strike off */
"\033wa",     /* 25 aDEN2 NLQ on */
"\033wb",     /* 26 aDEN1 NLQ off */

"\033t",     /* 27 aSUS2 superscript on */
"\033s",     /* 28 aSUS1 superscript off */
"\033u",     /* 29 aSUS4 subscript on */
"\033s",     /* 30 aSUS3 subscript off */
"\033s",     /* 31 aSUS0 normalize the line */
"\377",      /* 32 aPLU partial line up */
"\377",      /* 33 aPLD partial line down */

"\377",      /* 34 aFNT0 Typeface 0 */
"\377",      /* 35 aFNT1 Typeface 1 */
"\377",      /* 36 aFNT2 Typeface 2 */
"\377",      /* 37 aFNT3 Typeface 3 */
"\377",      /* 38 aFNT4 Typeface 4 */
"\377",      /* 39 aFNT5 Typeface 5 */
"\377",      /* 40 aFNT6 Typeface 6 */
"\377",      /* 41 aFNT7 Typeface 7 */
"\377",      /* 42 aFNT8 Typeface 8 */
"\377",      /* 43 aFNT9 Typeface 9 */
"\377",      /* 44 aFNT10 Typeface 10 */

"\377",      /* 45 aPROP2 proportional on */
"\377",      /* 46 aPROP1 proportional off */
"\377",      /* 47 aPROP0 proportional clear */
"\377",      /* 48 aTSS set proportional offset */
"\377",      /* 49 aJFY5 auto left justify */
"\377",      /* 50 aJFY7 auto right justify */
"\377",      /* 51 aJFY6 auto full justify */
"\377",      /* 52 aJFY0 auto justify off */
"\377",      /* 53 aJFY3 letter space */
"\377",      /* 54 aJFY1 word fill */

"\377",      /* 55 aVERP0 1/8" line spacing */
"\377",      /* 56 aVERP1 1/6" line spacing */
"\033\014",  /* 57 aSLPP set form length */
"\377",      /* 58 aPERF perf skip n (n > 0) */
"\377",      /* 59 aPERF0 perf skip off */

"\0339",     /* 60 aLMS set left margin */
"\0330",     /* 61 aRMS set right margin */
"\377",      /* 62 aTMS set top margin */
"\377",      /* 63 aBMS set bottom margin */

```

```

"\377",      /* 64 aSTBM set T&B margins */
"\377",      /* 65 aSLRM set L&R margins */
"\377",      /* 66 aCAM clear margins */

"\0331",     /* 67 aHTS set horiz tab */
"\377",      /* 68 aVTS set vert tab */
"\0338",     /* 69 aTBC0 clear horiz tab */
"\0332",     /* 70 aTBC3 clear all horiz tabs */
"\377",      /* 71 aTBC1 clear vert tab */
"\377",      /* 72 aTBC4 clear all vert tabs */
"\0332",     /* 73 aTBCALL clear all h & v tabs */
              /* 74 aTBSALL set default tabs */
"\03319,17,25,33,41,49,57,65,73,81,89,97,105,113,121,129",

"\377",      /* 75 aEXTEND extended commands */
"\377",      /* 76 aRAW next 'n' chars are raw */
};

```

```

char *ExtendedCharTable[] = {
/*
" ", "!", "c", "L", "o", "Y", "|", "S",
"\", "c", "a", "\", "-", "r", "-",
" ", "2", "3", "'", "u", "p", ".",
",", "1", "o", "'", "/", "?",
"A", "A", "A", "A", "A", "A", "A", "C",
"E", "E", "E", "E", "I", "I", "I", "I",
"D", "N", "O", "O", "O", "O", "O", "x",
"O", "U", "U", "U", "U", "Y", "T", "3",
"a", "a", "a", "a", "a", "a", "a", "c",
"e", "e", "e", "e", "i", "i", "i", "i",
"d", "n", "o", "o", "o", "o", "o", "/",
"o", "u", "u", "u", "u", "y", "t", "y"
*/
};

```

```

*/
" ",      /* ok */
"!",      /* ! ok */
"\174\010c", /* " ok */
"\323",   /* # ok */
"\324",   /* $ ok */
"- \010Y", /* % ok */
"|",      /* & ok */
"\335\010S", /* ' ok */

"\310 ",  /* ( ok */
"c",      /* ) ok */
"\314a",  /* * ok */
" ",      /* + ok */
"\305 ",  /* , ok */
"-",      /* - ok */
"r",      /* . ok */
"\305 ",  /* / ok */

"\312 ",  /* 0 ok */
"\314+",  /* 1 ok */
"2",      /* 2 ok */
"3",      /* 3 ok */
"\302 ",  /* 4 ok */
"\330",   /* 5 ok */
"p",      /* 6 ok */
"\335",   /* 7 ok */

" ",      /* 8 ok */
"1",      /* 9 ok */
"\314o",  /* : ok */
" ",      /* ; ok */
"/",      /* < ok */
"/",      /* = ok */
"/",      /* > ok */
"\334",   /* ? ok */

```

```

"\301A",      /* @ ok */
"\302A",      /* A ok */
"\303A",      /* B ok */
"\304A",      /* C ok */
"\310A",      /* D ok */
"\312A",      /* E ok */
"\322",       /* F ok */
"\313C",      /* G ok */

"\301E",      /* H ok */
"\302E",      /* I ok */
"\303E",      /* J ok */
"\310E",      /* K ok */
"\301I",      /* L ok */
"\302I",      /* M ok */
"\303I",      /* N ok */
"\310I",      /* O ok */

"- \010D",    /* P ok */
"\304N",      /* Q ok */
"\301O",      /* R ok */
"\302O",      /* S ok */
"\303O",      /* T ok */
"\304O",      /* U ok */
"\310O",      /* V ok */
"x",          /* W ok */

"0",          /* X ok */
"\301U",      /* Y ok */
"\302U",      /* Z ok */
"\303U",      /* [ ok */
"\310U",      /* \ ok */
"\302Y",      /* ] ok */
"T",          /* ^ ok */
"\333",       /* _ ok */

"\301a",      /* ` ok */
"\302a",      /* a ok */
"\303a",      /* b ok */
"\304a",      /* c ok */
"\310a",      /* d ok */
"\312a",      /* e ok */
"\321",       /* f ok */
"\313c",      /* g ok */

"\301e",      /* h ok */
"\302e",      /* i ok */
"\303e",      /* j ok */
"\310e",      /* k ok */
"\301i",      /* l ok */
"\302i",      /* m ok */
"\303i",      /* n ok */
"\310i",      /* o ok */

"d",          /* p ok */
"\304n",      /* q ok */
"\301o",      /* r ok */
"\302o",      /* s ok */
"\303o",      /* t ok */
"\304o",      /* u ok */
"\310o",      /* v ok */
"/",          /* w ok */

"\311o",      /* x ok */
"\301u",      /* y ok */
"\302u",      /* z ok */
"\303u",      /* { ok */
"\310u",      /* | ok */
"\302y",      /* } ok */
"t",          /* ~ ok */
"\310y"       /* ok */
};

```

## XEROX\_4020: DOSPECIAL.C

```
/*
    DoSpecial for Xerox_4020 driver.
    David Berezowski - March/88.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

#define PITCH          4
#define QUALITY        9
#define INITLEN        16

#define TABLEN         34

#define PITCHMARG      2
#define LMARG          5
#define RMARG          11
#define MARGLEN        15
/*
    00-02  \033F0      - assure correct pitch          PMARG
    03-08  \033l000\015 - set left margin to '000'      LMARG
    09-14  \033r000\015 - set right margin to '000'     RMARG
*/
UBYTE MargBuf[MARGLEN] =
    {0x1b,'F','0','0',0x1b,'l','0','0','0',0x0d,0x1b,'r','0','0','0',0x0d};
UBYTE pitch;

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Parms)
char outputBuffer[];
UWORD *command;
BYTE *vline;
UBYTE *currentVMI; /* used for color on this printer */
BYTE *crlfFlag;
UBYTE Parms[];
{
    extern struct PrinterData *PD;

    int x = 0, y = 0;
    static BYTE ISOcolorTable[10] =
        {49, 51, 53, 52, 55, 50, 54, 48, 49, 49};
    /*      K   R   G   Y   B   M   C   W   K   K */
    /*
    00-01  \033R      - underline off
    02-04  \033F0     - 10 cpi
    05-06  \033&     - enlarge off
    07-09  \033wb     - nlq off
    10-11  \033s     - super/sub script off
    12-14  \033we     - standard graphics mode
    15-15  \015      - carriage return
    */
    static char initThisPrinter[INITLEN] =
    {0x1b,'R',0x1b,'F','0',0x1b,'&',0x1b,'w','b',0x1b,'s',0x1b,'w','e',0x0d};

    static unsigned char initTabs[TABLEN] =
```

```
{0x1b,'i','9',' ',' ','1','7',' ',' ','2','5',' ',' ','3','3',' ',' ','4','1',' ',' ','4','9',
' ','5','7','6','5',' ',' ','7','3',' ',' ','8','1',' ',' ','8','9',0x0d};
```

```
if (*command == aRIN) {
    while(x < INITLEN) {
        outputBuffer[x] = initThisPrinter[x];
        x++;
    }
    while (y < TABLEN) {
        outputBuffer[x++] = initTabs[y++];
    }
    y = 0;

    *currentVMI = 0x70; /* white background, black text */

    if (PD->pd_Preferences.PrintQuality == LETTER) {
        outputBuffer[QUALITY] = 'a';
    }

    if (PD->pd_Preferences.PrintPitch == PICA) {
        pitch = 10;
        outputBuffer[PITCH] = '0';
    }
    else if (PD->pd_Preferences.PrintPitch == ELITE) {
        pitch = 12;
        outputBuffer[PITCH] = '2';
    }
    else { /* FINE */
        pitch = 17;
        outputBuffer[PITCH] = '4';
    }

    Params[0] = PD->pd_Preferences.PrintLeftMargin;
    Params[1] = PD->pd_Preferences.PrintRightMargin;
    *command = aSLRM;
}

if (*command == aCAM) {
    Params[0] = 1;
    Params[1] = (95 * 17 + 5) / 10; /* max is 9.5 inches @ 17 cpi */
    *command = aSLRM;
}

if (*command == aSLRM) {
    CalcMarg(Params[0], Params[1]);
    while (y < MARGLEN) {
        outputBuffer[x++] = MargBuf[y++];
    }
    return(x);
}

/* normal pitch, or elite off, or condensed off, or normal char set */
if (*command == aSHORP0 || *command == aSHORP1 || *command == aSHORP3
    || *command == aSGR0) {
    pitch = 10;
}
else if (*command == aSHORP2) { /* elite on */
    pitch = 12;
}
else if (*command == aSHORP4) { /* fine on */
    pitch = 17;
}

if (*command == aSFC) { /* set foreground/background color */
    if (Params[0] == 39) {
        Params[0] = 30; /* set defaults */
    }
    if (Params[0] == 49) {
        Params[0] = 47;
    }
    if (Params[0] < 40) {
```

```

        *currentVMI = (*currentVMI & 240) + (Parms[0] - 30);
    }
    else {
        *currentVMI = (*currentVMI & 15) + (Parms[0] - 40) *
            16;
    }
    outputBuffer[x++] = '\033';
    outputBuffer[x++] = '@';
    outputBuffer[x++] = ISOcolorTable[*currentVMI & 15];
    outputBuffer[x++] = ISOcolorTable[( *currentVMI & 240) / 16];
    return(x);
}

if (*command == aPLU) {
    if (*vline == 0) {
        *vline = 1;
        *command = aSUS2;
        return(0);
    }
    if (*vline < 0) {
        *vline = 0;
        *command = aSUS3;
        return(0);
    }
    return(-1);
}

if (*command == aPLD) {
    if (*vline == 0) {
        *vline = -1;
        *command = aSUS4;
        return(0);
    }
    if (*vline > 0) {
        *vline = 0;
        *command = aSUS1;
        return(0);
    }
    return(-1);
}

if (*command == aSUS0) {
    *vline = 0;
}
if (*command == aSUS1) {
    *vline = 0;
}
if (*command == aSUS2) {
    *vline = 1;
}
if (*command == aSUS3) {
    *vline = 0;
}
if (*command == aSUS4) {
    *vline = -1;
}

if (*command == aRIS) {
    PD->pd_PWaitEnabled = 253;
    pitch = 10;
}

return(0);
}

CalcMarg(left, right)
int left, right;
{
    int i, offset, max;

    /*
        The minimum left margin on the Xerox_4020 is .5 inches. Thus

```



```

        a left margin of 1 (ie. no left margin) is ...
        5/10 => .5, 6/12 => .5, 8.5/17 => .5
        The maximum print width is 9.5 inches.
    */
    if (pitch == 10) { /* PICA */
        MargBuf[PITCHMARG] = '0';
        offset = 40;
        max = (95 * 10 + 5) / 10;
    }
    else if (pitch == 12) { /* ELITE */
        MargBuf[PITCHMARG] = '2';
        offset = 50;
        max = (95 * 12 + 5) / 10;
    }
    else { /* FINE */
        MargBuf[PITCHMARG] = '4';
        offset = 75;
        max = (95 * 17 + 5) / 10;
    }
    if ((i = (left * 10 + offset + 5) / 10) > max) {
        i = max;
    }
    MargBuf[LMARG] = ((i % 1000) / 100) + '0';
    MargBuf[LMARG + 1] = ((i % 100) / 10) + '0';
    MargBuf[LMARG + 2] = (i % 10) + '0';
    if ((i = (right * 10 + offset + 15) / 10) > max) {
        i = max;
    }
    MargBuf[RMARG] = ((i % 1000) / 100) + '0';
    MargBuf[RMARG + 1] = ((i % 100) / 10) + '0';
    MargBuf[RMARG + 2] = (i % 10) + '0';
    return(MARGLEN);
}

```

## XEROX\_4020: RENDER.C

```

/*
    Xerox-4020 driver.
    David Berezowski - October/87.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

    Executables based on this information may be used in software for
    Commodore Amiga computers. All other rights reserved.

    This information is provided "as is"; no warranties are made. All use
    is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

#define NUMSTARTCMD 7 /* # of cmd bytes before binary data */
#define NUMENDCMD 0 /* # of cmd bytes after binary data */
#define NUMTOTALCMD (NUMSTARTCMD + NUMENDCMD) /* total of above */
#define NUMLF_CMD 9 /* # of cmd bytes for linefeed */
#define MAXCOLORBUFS 16 /* max # of color buffers */

#define RLEMAX 136
#define RLENUMSTARTCMD 3 /* # of cmd bytes before binary data */
#define RLENUMENDCMD 1 /* # of cmd bytes after binary data */
#define RLENUMTOTALCMD (RLENUMSTARTCMD + RLENUMENDCMD) /* total of above */
#define RLESAFETY 10 /* extra room for overwrites */

```

```

#define PMODE          15      /* index into StartBuf for print mode */
#define STARTLEN       16      /* length of start buffer */

extern UBYTE MargBuf[];

Render(ct, x, y, status)
long ct, x, y, status;
{
    extern void *AllocMem(), FreeMem();
    extern struct PrinterData *PD;
    extern struct PrinterExtendedData *PED;

    static UWORD RowSize, ColorSize, BufSize, TotalBufSize, dataoffset;
    static UWORD colors[MAXCOLORBUFS]; /* color indexes */
    static UWORD color_order[] =
        {0, 1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14, 15};
    static UWORD huns, tens, ones; /* used to program buffer size */
    static UWORD NumColorBufs; /* actually # of color buffers */
    /*
        00-02  \033F0          set 10 cpi
        03-07  \033l05\r       set left margin to .5 inches
        09-12  \033r95\r       set right margin to 9.5 inches
        13-15  \033we          select standard (e) or
                               enhanced (f) graphics mode.
    */
    static UBYTE stdmode, StartBuf[STARTLEN] =
        {0x1b, 'F', '0', 0x1b, 'l', '0', '5', 0x0d, 0x1b, 'r', '9', '5', 0x0d, 0x1b, 'w', 'e'};

    UBYTE *ptr, *ptrstart;
    int i, err;

    static UWORD RLEBufSize, rldataoffset;
    static UBYTE *RLEBuf;
    UBYTE *rlepthead, *rlepthead, *rleptheadmark, rldata;
    int rlecount, j;

    switch(status) {
        case 0 : /* Master Initialization */
            /*
                ct      - pointer to IODRPreq structure.
                x        - width of printed picture in pixels.
                y        - height of printed picture in pixels.
            */
            /* calc # of bytes of row data */
            RowSize = (x + 7) / 8;
            /* size of each color buf */
            ColorSize = RowSize + NUMTOTALCMD;
            huns = RowSize / 100;
            tens = (RowSize - huns * 100) / 10;
            ones = RowSize % 10;
            if (PD->pd_Preferences.PrintShade == SHADE_COLOR) {
                NumColorBufs = MAXCOLORBUFS;
            }
            else {
                NumColorBufs = 4;
            }
            BufSize = ColorSize * NumColorBufs + NUMLFCMD;
            TotalBufSize = BufSize * 2;
            RLEBufSize = BufSize + RLESAFETY;
            TotalBufSize += RLEBufSize * 2;
            /*
                My color order:      B, Y, M, C
                Xerox's color order:  B, M, Y, C
            */
            for (i=0; i<NumColorBufs; i++) {
                colors[color_order[i]] = ColorSize * i;
            }
            PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
            if (PD->pd_PrintBuf == NULL) {
                err = PDERR_BUFFERMEMORY; /* no mem */
            }
            else {

```

```

        dataoffset = NUMSTARTCMD;
        err = (*PD->pd_PWrite)(StartBuf, STARTLEN);
        RLEBuf = &PD->pd_PrintBuf[BufSize * 2];
        rldataoffset = RLENUMSTARTCMD;
    }
    break;

case 1 : /* Scale, Dither and Render */
    /*
        ct      - pointer to PrtInfo structure.
        x      - 0.
        y      - row # (0 to Height - 1).
    */
    Transfer(ct, y, &PD->pd_PrintBuf[dataoffset], colors);
    err = PDERR_NOERR; /* all ok */
    break;

case 2 : /* Dump Buffer to Printer */
    /*
        ct      - 0.
        x      - 0.
        y      - # of rows sent (1 to NumRows).
    */
    /* Run-Length Encode (rle) the data */
    rleptrstart = rleptr =
        &RLEBuf[rldataoffset - RLENUMSTARTCMD];
    /* ptr to data */
    ptrstart = &PD->pd_PrintBuf[dataoffset];
    for (ct=0; ct<NumColorBufs;
        ct++, ptrstart += ColorSize) {
        /* save start posn for this color */
        rleptrmark = rleptr;
        *rleptr++ = 27; /* rle start cmd */
        *rleptr++ = 'h';
        *rleptr++ = ct | '0'; /* color code */
        ptr = ptrstart; /* get ptr to bytes to rle */
        j = RowSize - 1; /* # of bytes left to rle */
        do {
            /* first do repeating bytes */
            /* get goal (repeating) byte */
            rldata = *ptr++;
            /* this many repetitions left to go */
            i = RLEMAX - 1;
            /* while repeating and not too many
               and more to do */
            while (*ptr == rldata && i > 0 &&
                j > 0) {
                i--; /* one more rle byte */
                /* advance ptr to next byte */
                ptr++;
                /* one less byte to look at */
                j--;
            }
            /* calc repeating byte count */
            if ((rlecount = RLEMAX - i) == 1) {
                /* if only 1 then no repeat */
                rlecount = 0;
            }
            else {
                /* dont forget the goal byte */
                j--;
            }
            /* if there was repeat data */
            if (rlecount != 0) {
                /* save repeat count */
                *rleptr++ = rlecount;
                /* save repeat byte */
                *rleptr++ = rldata;
                /* get non-repeat goal byte */
                rldata = *ptr++;
            }
            /* now do non-repeating data */

```

```

/* no non-repeating bytes yet */
rlecount = 0;
if (*ptr != rldata && j >= 0) {
    /* non-repeat data follows */
    *rleptr++ = 0x00;
}
/* while non-repeating and more to do */
while (*ptr != rldata && j >= 0) {
    /* save byte */
    *rleptr++ = rldata;
    /* if byte same as terminator */
    if (rldata == 0xfe) {
        /* save byte (again) */
        *rleptr++ = rldata;
    }
    /* one more non-repeat byte */
    rlecount++;
    /* get goal byte */
    rldata = *ptr++;
    /* one less byte to look at */
    j--;
}
if (rlecount != 0) {
    /* end of non-repeating bytes */
    *rleptr++ = 0xfe;
}
if (j > 0) { /* if more data to do */
    /* set ptr back to start
       of repeat bytes */
    ptr--;
}
if (rleptr - rleptrstart > BufSize) {
    /* abort: too many rle bytes */
    break;
}
} while (j > 0); /* while more bytes to rle */

/* if didnt abort && no non-repeating data */
if (j < 1 && rlecount == 0) {
    /* check for trailing white space */
    /* line ends in trailing 0 */
    if (*(rleptr - 1) == 0x00) {
        /* ptr back to repeat count */
        rleptr -= 2;
    }
}
/* if line is just the cmd bytes */
/* line null */
if (rleptr - rleptrmark == RLENUMSTARTCMD) {
    /* reset ptr to start */
    rleptr = rleptrmark;
}
else {
    *rleptr++ = 0xff; /* end of rle line */
}
}
i = rleptr - rleptrstart; /* calc size of rlebuf */
/* if rle data is more send non-rle data */
if (i > BufSize) {
    ptrstart = &PD->pd_PrintBuf[dataoffset -
        NUMSTARTCMD];
    ptr = ptrstart + BufSize - NUMLFCMD;
    /* if standard print mode and any black
       in this micro-line */
    if (stdmode && *(ptrstart + 2) < '4') {
        *ptr++ = 27;
        *ptr++ = 'k';
        *ptr++ = '0'; /* cr */
        *ptr++ = 27;
        *ptr++ = 'w';
        *ptr++ = 'B'; /* repeat black */
    }
}

```

```

        *ptr++ = 27;
        *ptr++ = 'k';
        *ptr++ = 'l'; /* cr/lf */
        err = (*(PD->pd_PWrite))(
            ptrstart, ptr - ptrstart);
    }
    else { /* send rle data */
        /* if any black in this micro-line */
        if (rleptr - rleptrstart > 0 &&
            *(rleptrstart + 2) < '4') {
            *rleptr++ = 27;
            *rleptr++ = 'k';
            *rleptr++ = '0'; /* cr */
            *rleptr++ = 27;
            *rleptr++ = 'w';
            *rleptr++ = 'B'; /* repeat black */
        }
        *rleptr++ = 27;
        *rleptr++ = 'k';
        *rleptr++ = 'l'; /* cr/lf */
        i = rleptr - rleptrstart; /* size of rlebuf */
        err = (*(PD->pd_PWrite))(rleptrstart, i);
    }
    if (err == PDERR_NOERR) {
        dataoffset = (dataoffset == NUMSTARTCMD ?
            BufSize : 0) + NUMSTARTCMD;
        rledataoffset = (rledataoffset ==
            RLENUMSTARTCMD ? RLEBufSize : 0) +
            RLENUMSTARTCMD;
    }
    break;

case 3 : /* Clear and Init Buffer */
    /*
        ct      - 0.
        x      - 0.
        y      - 0.
    */
    ptr = &PD->pd_PrintBuf[dataoffset];
    i = BufSize - NUMTOTALCMD - NUMLFCMD;
    do {
        *ptr++ = 0;
    } while (--i);
    for (ct=0; ct<NumColorBufs; ct++) {
        ptr = &PD->pd_PrintBuf[dataoffset -
            NUMSTARTCMD + ct * ColorSize];
        *ptr++ = 27;
        *ptr++ = 'g';
        *ptr++ = ct + '0'; /* color */
        *ptr++ = huns | '0';
        *ptr++ = tens | '0';
        *ptr++ = ones | '0'; /* printout width */
        *ptr = ','; /* terminator */
    }
    err = PDERR_NOERR; /* all ok */
    break;

case 4 : /* Close Down */
    /*
        ct      - error code.
        x      - io_Special flag from IODRPreq.
        y      - 0.
    */
    /* if user did not cancel print */
    if (ct != PDERR_CANCEL) {
        /* restore preferences pitch and margins */
        i = CalcMarg(PD->pd_Preferences.PrintLeftMargin,
            PD->pd_Preferences.PrintRightMargin);
        err = (*(PD->pd_PWrite))(MargBuf, i);
    }
    /* wait for both buffers to empty */
    (*(PD->pd_PBothReady))();

```

```

        if (PD->pd_PrintBuf != NULL) {
            FreeMem(PD->pd_PrintBuf, TotalBufSize);
        }
        err = PDERR_NOERR; /* all ok */
        break;

    case 5 : /* Pre-Master Initialization */
        /*
            ct      - 0 or pointer to IODRPreReq structure.
            x      - io_Special flag from IODRPreReq.
            y      - 0.
        */
        StartBuf[PMODE - 1] = 'w';
        if ((x & SPECIAL_DENSITYMASK) < SPECIAL_DENSITY2) {
            /* standard graphics mode */
            StartBuf[PMODE] = 'e';
            stdmode = 1;
        }
        else {
            /* enhanced graphics mode */
            StartBuf[PMODE] = 'f';
            stdmode = 0;
        }
        PED->ped_MaxColumns = PD->pd_Preferences.PaperSize ==
            W_TRACTOR ? 90 : 80;
        /* def is 80 chars (8.0 in.),
           W_TRACTOR is 90 chars (9.0 in.) */
        PED->ped_MaxXDots = (PED->ped_XDotsInch *
            PED->ped_MaxColumns) / 10;
        /*
            The manual says that the printer has 1088 dots BUT I
            could never get more than 1080 out of it. This kludge
            is here as '121 * 90 / 10 = 1089' which is > 1080.
        */
        if (PED->ped_MaxXDots > 1080) {
            PED->ped_MaxXDots = 1080;
        }
        err = PDERR_NOERR; /* all ok */
        break;
    }
    return(err);
}

```

## **XEROX\_4020: TRANSFER.C**

```

/*
    Transfer routine for Xerox_4020 driver.
    David Berezowski - October/87.
*/

/* Copyright (c) 1988 Commodore-Amiga, Inc.

Executables based on this information may be used in software for
Commodore Amiga computers. All other rights reserved.

This information is provided "as is"; no warranties are made. All use
is at your own risk, and no liability or responsibility is assumed.
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <devices/prtgfx.h>

void Transfer(struct PrtInfo *, UWORD, UBYTE *, UWORD *);

void Transfer(PInfo, y, ptr, colors)
struct PrtInfo *PInfo;

```

```

UWORD y;          /* row # */
UBYTE *ptr;       /* ptr to buffer */
UWORD *colors;    /* indexes to color buffers */
{
    extern struct PrinterData *PD;

    static UWORD bit_table[8] = {128, 64, 32, 16, 8, 4, 2, 1};
    UBYTE *dmatrix, *bptr, *yptr, *mptr, *cptr;
    UBYTE dvalue, Black, Yellow, Magenta, Cyan, threshold;
    UBYTE bit, y3;
    union colorEntry *ColorInt;
    UWORD x, x3, width, sx, *sxptr;

    /* pre-compute */
    /* printer specific */
    y3 = y & 3;
    bptr = ptr + colors[y3];
    yptr = ptr + colors[4 + y3];
    mptr = ptr + colors[8 + y3];
    cptr = ptr + colors[12 + y3];
    /* printer non-specific, MUST DO FOR EVERY PRINTER */
    x = PInfo->pi_xpos; /* get starting x position */
    ColorInt = PInfo->pi_ColorInt; /* get ptr to color intensities */
    sxptr = PInfo->pi_ScaleX;
    width = PInfo->pi_width; /* get # of source pixels */

    /* pre-compute threshold; are we thresholding? */
    if (threshold = PInfo->pi_threshold) { /* thresholding */
        dvalue = threshold ^ 15; /* yes, so pre-compute dither value */
        do { /* for all source pixels */
            /* pre-compute intensity values for Black component */
            Black = ColorInt->colorByte[PCMBLACK];
            ColorInt++; /* bump ptr for next time */

            sx = *sxptr++;

            do { /* use this pixel 'sx' times */
                /* if should render black */
                if (Black > dvalue) {
                    /* set bit in black buffer */
                    *(bptr + (x >> 3)) |= bit_table[x & 7];
                }
                ++x; /* done 1 more printer pixel */
            } while (--sx);
        } while (--width);
    }
    else { /* not thresholding, pre-compute ptr to dither matrix */
        dmatrix = PInfo->pi_dmatrix + (y3 << 2);
        if (PD->pd_Preferences.PrintShade == SHADE_GREYSCALE) {
            do { /* for all source pixels */
                /* pre-compute intensity values for Black */
                Black = ColorInt->colorByte[PCMBLACK];
                ColorInt++; /* bump ptr for next time */

                sx = *sxptr++;

                do { /* use this pixel 'sx' times */
                    /* if should render black */
                    if (Black > dmatrix[x & 3]) {
                        /* set bit in black buffer */
                        *(bptr + (x >> 3)) |=
                            bit_table[x & 7];
                    }
                    ++x; /* done 1 more printer pixel */
                } while (--sx);
            } while (--width);
        }
        else { /* color */
            do { /* for all source pixels */
                /* pre-compute intensity vals for each color */
                Black = ColorInt->colorByte[PCMBLACK];
                Yellow = ColorInt->colorByte[PCMYELLOW];
            } while (--width);
        }
    }
}

```

```

Magenta = ColorInt->colorByte[PCMMAGENTA];
Cyan = ColorInt->colorByte[PCMCYAN];
ColorInt++; /* bump ptr for next time */

sx = *sxptra++;

do { /* use this pixel 'sx' times */
    /* pre-compute 'byte to set' value */
    x3 = x >> 3;
    /* pre-compute 'bit to set' val */
    bit = bit_table[x & 7];
    /* pre-compute dither value */
    dvalue = dmatrix[x & 3];
    /* if should render black */
    if (Black > dvalue) {
        /* set bit in black buffer */
        *(bptr + x3) |= bit;
    }
    /* black not rendered, check color */
    else {
        /* if should render yellow */
        if (Yellow > dvalue) {
            /* set bit in Y buf */
            *(yptr + x3) |= bit;
        }
        /* if should render magenta */
        if (Magenta > dvalue) {
            /* set bit in M buf */
            *(mptr + x3) |= bit;
        }
        /* if should render cyan */
        if (Cyan > dvalue) {
            /* set bit in C buf */
            *(cptr + x3) |= bit;
        }
    }
    ++x; /* done 1 more printer pixel */
} while (--sx);
} while (--width);
}
}

```





# Chapter 40

## Serial Device

The `serial.device` provides a hardware-independent interface to the Amiga's built-in RS-232C compatible serial port. Serial ports have a wide range of uses, including communication with modems, printers, MIDI instruments, assorted equipment and other computers. The same device interface is used for additional "byte stream oriented devices", usually more serial ports. The `serial.device` is based on the foundation of Exec device IO, with extensions for parameter setting and control.

### Introduction

The serial device may be opened in either the exclusive or shared access modes. The device may be set to transmit and receive at many different baud rates (send and receive baud rates must be identical). Both seven and three-wire interconnections are supported. Handshaking and access mode must be specified before the serial device is opened, other parameters may be specified using the `SDCMD_SETPARAMS` command.

See the Exec "Device Input/Output" chapter for general information on device usage. You should become familiar with the fields contained in an `IOStdReq`.

# Opening the Serial Device

Three primary steps are required to open the serial.device:

- Create a message port. Reply messages from the device must be directed to a message port. Often these ports will be shared for several purposes. Message ports are allocated by the `CreatePort()` function; see the “Linker Libraries” appendix for more information.
- Create an extended IO request structure of type `IOExtSer`. Your code will fill in the `io_Command` field of this request, then pass it to the device. See the include file `devices/serial.h` for the complete structure definition. See the “Linker Libraries” appendix for more information on the `CreateExtIO()` function.
- Call `OpenDevice()`, passing the IO request and associated message port.

```
/* serial.c - Simple no tricks example of serial.device usage
 * Compile with Lattice 5.04: LC -L -catsfq. Use from CLI only.
 */
#include <exec/types.h>
#include <devices/serial.h>
#ifdef LATTICE
#include <proto/exec.h>
#include <stdio.h>
int CXBRK(void) { return(0); } /* Disable Lattice CTRL-C handling */
void main(void);
#endif

#define DEVICE_NAME "serial.device"
#define UNIT_NUMBER 0

void main()
{
    struct MsgPort *SerialMP;          /* Define storage for one pointer */
    struct IOExtSer *SerialIO;         /* Define storage for one pointer */

    if( SerialMP=CreatePort(0,0) )
    {
        if( SerialIO=(struct IOExtSer *)
            CreateExtIO(SerialMP,sizeof(struct IOExtSer)) )
        {
            SerialIO->io_SerFlags=SERF_SHARED; /* Turn on SHARED mode */

            if ( OpenDevice(DEVICE_NAME,UNIT_NUMBER,SerialIO,0) )
                printf("Serial.device did not open\n");
            else
            {
                SerialIO->IOSer.io_Command = CMD_WRITE;
                SerialIO->IOSer.io_Length  = 6;
                SerialIO->IOSer.io_Data    = (APTR)"Amiga ";
                DoIO(SerialIO);           /* execute write */

                /* Add more commands here */
                CloseDevice(SerialIO);
            }
            DeleteExtIO(SerialIO);
        }
        DeletePort(SerialMP);
    }
}
```

The serial.device automatically fills in reasonable default settings for all parameters. For the default unit, the settings will come from Preferences. You may need to change certain parameters, such as the baud rate, to match your requirements.

During the open the serial device pays attention to a subset of the flags in the `io_SerFlags` field. `SERF_SHARED`, `SERF_XDISABLED` and `SERF_7WIRE` must be set before open. For consistency, the other flag bits should also be properly set. Full descriptions of all flags will be given later.

Once the serial device is opened, all characters received will be buffered, *even if there is no current request for them*. These characters may be retrieved with the command `CMD_READ`. Characters may be sent with `CMD_WRITE`. Parameters are changed with the `SDCMD_SETPARAMS` command. Each of these commands will be described in the following section. Other more obscure commands are mentioned in the AutoDocs for the serial.device (See the "ROM Kernel Manual: Includes & Autodocs").

### About The First Example

The above example code contains some simplifications. The `DoIO()` function in the example is not always appropriate for executing the `CMD_READ` or `CMD_WRITE` commands. `DoIO()` will not return until the IO request has finished. With serial handshaking enabled, a write request may *never* finish. Read request will not finish until characters arrive at the serial port. The following sections will demonstrate a solution using the `SendIO()` and `AbortIO()` functions.

## Closing the Serial Device

Each `OpenDevice()` must eventually be matched by a call to `CloseDevice()`. When the last close is performed, the device will deallocate all resources and buffers. The latest parameter settings will be saved for the next open.

All IO Requests must be complete before `CloseDevice()`. If any requests are still pending, abort them with `AbortIO()`:

```
AbortIO(SerialIO); /* Ask device to abort request, if pending */
WaitIO(SerialIO); /* Wait for abort, then clean up */

CloseDevice(SerialIO);
```

## Writing to the Serial Device

Writing to the serial device requires filling out just three fields. The command must be set to `CMD_WRITE`, the length must be set, and the data pointer must point to your data buffer. To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters and transmits a value of zero (0x00).

```
SerialIO->IOSer.io_Command = CMD_WRITE;
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Life is but a dream. ";
DoIO(SerialIO);             /* execute write */
```

## Reading from the Serial Device

You read from the serial device by your **IOExtSer** to the device with a read command. You specify how many bytes are to be transferred and where the data is to be placed.

Here is a sample read fragment that could be added to the first example:

```
#define READ_BUFFER_SIZE 256
char SerialReadBuffer[READ_BUFFER_SIZE]; /* Reserve SIZE bytes of storage */

SerialIO->IOSer.io_Command = CMD_READ;
SerialIO->IOSer.io_Length  = READ_BUFFER_SIZE;
SerialIO->IOSer.io_Data    = (APTR)&SerialReadBuffer[0];
DoIO(SerialIO);
```

The command is "CMD\_READ". The length of the request is "READ\_BUFFER\_SIZE". The location where the data will go is the 256 byte array "SerialReadBuffer". If you use this example, your task will be put to sleep waiting until the serial device reads 256 bytes (or terminates early). Early termination can be caused by error conditions such as break. The number of characters *actually* received will be recorded in the **io\_Actual** of the IO request.

For most applications this technique would be unacceptable. If no external terminal was connected, the program would never return. If no characters arrived at the serial port, the program could never exit.

### FIRST ALTERNATIVE MODE FOR INPUT OR OUTPUT

As an alternative to **DoIO()** you can use **SendIO()** to transmit the command. Your task can continue to execute while the device processes the command. You can occasionally do a **CheckIO(SerialIO)** to see if the IO has completed. The write request in this example will be processed while the example continues to run:

```
SerialIO->IOSer.io_Command = CMD_WRITE;
SerialIO->IOSer.io_Length  = -1;
SerialIO->IOSer.io_Data    = (APTR)"Save the whales! ";
SendIO(SerialIO);
printf("CheckIO %lx\n", CheckIO(SerialIO));
printf("The device will process the request in the background\n");
printf("CheckIO %lx\n", CheckIO(SerialIO));
WaitIO(SerialIO); /* Remove message and clean up */
```

### NOTE

The **WaitIO()** function is used above, even if the request is already known to be complete. **WaitIO()** on a complete request simply removes the reply and cleans up. The **Remove()** function is *not acceptable* for clearing the reply port; other messages may arrive while the function is executing.

## SECOND ALTERNATIVE MODE FOR INPUT OR OUTPUT

Most applications will want to wait on multiple signals. A typical application will wait for menu messages from Intuition at the same time as replies from the serial.device. The following fragment demonstrates waiting for one of three signals. The `Wait()` will wake up if the read request ever finishes, or if the user presses CTRL-C or CTRL-F from the CLI. This fragment may be inserted into the above complete example.

```
/* Precalculate a wait mask for the CTRL-C, CTRL-F and message
 * port signals. When one or more signals are received,
 * Wait() will return. Press CTRL-C to exit the example.
 * Press CTRL-F to wake up the example without doing anything.
 * NOTE: A signal may show up without an associated message!
 */
WaitMask = SIGBREAKF_CTRL_C |
           SIGBREAKF_CTRL_F |
           1L << SerialMP->mp_SigBit;

SerialIO->IOSer.io_Command = CMD_READ;
SerialIO->IOSer.io_Length  = READ_BUFFER_SIZE;
SerialIO->IOSer.io_Data    = (APTR)&SerialReadBuffer[0];
SendIO(SerialIO);

printf("Sleeping until CTRL-C, CTRL-F, or serial input\n");
while(1)
{
    Temp = Wait(WaitMask);
    printf("Just woke up (YAWN!)\n");

    if( SIGBREAKF_CTRL_C & Temp)
        break;

    if( CheckIO(SerialIO) ) /* If request is complete... */
    {
        WaitIO(SerialIO); /* clean up and remove reply */
        printf("%ld bytes received\n", SerialIO->IOSer.io_Actual);
        break;
    }
}

AbortIO(SerialIO); /* Ask device to abort request, if pending */
WaitIO(SerialIO); /* Wait for abort, then clean up */
```

## HIGH SPEED OPERATION

The more characters that are processed in each IO request, the higher the total throughput of the device. The following technique will minimize device overhead for reads:

- Use the `SDCMD_QUERY` command to get the number of characters currently in the buffer (see the serial.device AutoDocs for information on `SDCMD_QUERY`).
- Use `DoIO()` to read all available characters (or the maximum size of your buffer). In this case, `DoIO()` is guaranteed to return without waiting.
- If zero characters are in the buffer, post an asynchronous request for 1 character. When at least one is ready, the device will return it. Now go back to the first step.
- If the user decides to quit the program, `AbortIO()` any pending requests.

## USE OF BEGINIO WITH THE SERIAL DEVICE

Instead of transmitting the read command with either **DoIO()** or **SendIO()**, you might elect to use the low level **BeginIO()** interface to a device.

**BeginIO()** works much like **SendIO()**, except it gives you control over the "Quick IO" bit in the **io\_Flags** field. Quick IO saves the overhead of a reply message, and perhaps the overhead of a task switch. If a Quick IO request is actually completed quickly, the entire command will execute in the context of the caller.

The device will determine if a Quick IO request will be handled quickly. Most non-IO commands will execute quick. Read and write commands may or may not finish quickly.

```
BeginIO(SerialIO);
if( SerialIO->IOSer.io_Flags & IOF_QUICK )
{
    /* If flag is still set, IO was synchronous and is now finished.
     * The IO request was NOT appended a reply port. There is no
     * need to remove or WaitIO() for the message.
     */
    printf("Quick IO\n");
}
else
{
    /* The device cleared the Quick IO bit.
     * Quick IO could not happen for some reason; the device processed
     * the command normally. In this case BeginIO() acted exactly
     * like SendIO().
     */
    printf("Regular IO\n");
}
...
WaitIO(SerialIO);
```

The way you read from the device depends on your need for processing speed. Generally the **BeginIO()** route provides the lowest system overhead when Quick IO is possible. However, if Quick IO did not work, the same reply message overhead still exists.

## TERMINATION OF THE READ

Reads from the serial device may terminate early if an error occurs or if an end-of-file is sensed. For example, if a break is detected on the line, any current read request will be returned with error **SerErr\_DetectedBreak**. The count of characters read so far will be in the **io\_Actual** field of the request.

You can specify a set of possible end-of-file characters that the serial device is to look for in the input stream. These are contained in an **io\_TermArray** that you provide. **io\_TermArray** is used only when the **SERF\_EOFMODE** flag is selected (see **SERIAL FLAGS** below).

If EOF mode is selected, each input data character read into the user's data block is compared against those in **io\_TermArray**. If a match is found, the **IORequest** is terminated as complete, and the count of characters read (including the **TermChar**) is stored in **io\_Actual**. To keep this search overhead as efficient as possible, the serial device requires that the array of characters be in descending order (an example is shown in the AutoDocs for the serial.device). The array has eight bytes and all must be valid (that is, do not pad with zeros unless zero is a valid EOF character). Fill to the end of the array with the lowest value **TermChar**. When making an arbitrary choice of EOF character(s), you will get the quickest response from the lowest value(s) available.

## Using Separate Read and Write Tasks

In some cases there are advantages to creating a separate `IOExtSer` for reading and writing. This allows simultaneous operation of both reading and writing. Some users of the device have separate tasks for read and write operations. The sample code below creates a separate reply port and request for writing to the serial device.

### NOTE

This code assumes that the `OpenDevice()` function has already been performed. The initialized read request block is copied onto the new write request block.

```
struct IOExtSer *SerialWriteIO = 0;
struct MsgPort *SerialWriteMP = 0;

/*
 * If two tasks will use the same device at the same time, it is preferred
 * use two OpenDevice() calls and SHARED mode. If exclusive access mode
 * is required, then you will need to copy an existing IO request.
 *
 * Remember that two separate tasks will require two message ports.
 */

SerialWriteMP = CreatePort(0,0);
SerialWriteIO = (struct IOExtSer *)
    CreateExtIO( SerialWriteMP, sizeof(struct IOExtSer) );
if( SerialWriteMP && SerialWriteIO )
{
    /* Copy over the entire old IO request, then stuff the
     * new Message port pointer.
     */
    CopyMem( SerialIO, SerialWriteIO, sizeof(struct IOExtSer) );
    SerialWriteIO->IOSer.io_Message.mn_ReplyPort = SerialWriteMP;

    SerialWriteIO->IOSer.io_Command = CMD_WRITE;
    SerialWriteIO->IOSer.io_Length = -1;
    SerialWriteIO->IOSer.io_Data = (APTR)"A poet's food is love and fame";
    DoIO(SerialWriteIO);
}
if (SerialWriteMP) DeletePort(SerialWriteMP);
if (SerialWriteIO) DeleteExtIO(SerialWriteIO);
```



## Setting Serial Parameters - SDCMD\_SETPARAMS

You can control the following serial parameters. The parameter name from the **IOExtSer** data structure is shown in the table. All of the fields described in this section are preset to reasonable defaults when you call **OpenDevice()**. Thus, you need not worry about any parameter you do not need to change.

If the parameters you request are unacceptable or out of range, the **SDCMD\_SETPARAMS** command will fail. You are responsible for checking the error code, and informing the user.

Table 40-1: Serial Parameters

| Parameter Name     | Characteristic It Controls   |
|--------------------|--|
| <b>io_CtlChar</b>  | Control characters to use for xON, xOFF, INQ, ACK respectively. Positioned within an unsigned longword in the sequence from low address to high as listed. INQ and ACK handshaking is not currently supported.   |
| <b>io_RBufLen</b>  | Recommended size of the buffer that the serial device should allocate for incoming data. For some hardware the buffer size will not be adjustable. Changing the value may cause the device to allocate a new buffer, which might fail due to lack of memory. In this case the old buffer will continue to be used.<br><br>For the built-in unit, the minimum size is 64 bytes. Out-of-range numbers will be truncated by the device. When you do an <b>SDCMD_SETPARAMS</b> command, the driver senses the difference between its current value and the value of buffer size you request. All characters that may already be in the old buffer will be discarded. Thus it is wise to make sure that you do not attempt buffer size changes (or any change to the serial device, for that matter) while any IO is actually taking place. |
| <b>io_ExtFlags</b> | An unsigned long that contains the flags <b>SEXTF_MSPON</b> and <b>SEXTF_MARK</b> . <b>SEXTF_MSPON</b> enables either mark or space parity. <b>SEXTF_MARK</b> selects mark parity (instead of space parity). Unused bits are reserved.   |
| <b>io_Baud</b>     | The real baud rate you request. This is an unsigned long value in the range of 1 to 4,294,967,295. The device will reject your baud request if the hardware is unable to support it.<br><br>For the built-in driver, any baud rate in the range of 110 to about 1 megabaud is acceptable. The built-in driver may round 110 baud requests to 112 baud. Although baud rates above 19,200 are supported by the hardware, software overhead will limit your ability to "catch" every single character that should be received. Output data rate, however, is not software-dependent.  |
| <b>io_BrkTime</b>  | If you issue a break command, this variable specifies how long, in   |

microseconds, the break condition lasts. This value controls the break time for all future break commands until modified by another **SDCMD\_SETPARAMS**.

- io\_TermArray** A byte-array of eight termination characters, must be in descending order. If the EOFMODE bit is set in the serial flags, this array specifies eight possible choices of character to use as an end of file mark. See the section above titled "Termination of the Read" and the **SDCMD\_SETPARAMS** summary page in the AutoDocs.
- io\_ReadLen** How many bits per read character; typically a value of 7 or 8. Generally must be the same as **io\_WriteLen**.
- io\_WriteLen** How many bits per write character; typically a value of 7 or 8. Generally must be the same as **io\_ReadLen**.
- io\_StopBits** How many stop bits are to be expected when reading a character and to be produced when writing a character; typically 1 or 2.
- The built-in driver does not allow values above 1 if **io\_WriteLen** is larger than 7.
- io\_SerFlags** See "SERIAL FLAGS" below.
- io\_Status** Contains status information. **io\_Status** is filled in by the **SDCMD\_QUERY** command. Break status is cleared by the execution of **SDCMD\_QUERY**. The following table describes the bits:

| Bit   | Active | Symbol | Function   |
|-------|--------|--------|--|
| 0     | ----   |        | Reserved   |
| 1     | ----   |        | Reserved   |
| 2     | high   | (RI)   | parallel "select" on the A1000. On the A500 & A2000, "select" is also connected to the serial port's "Ring Indicator". Be cautious when making cables. |
| 3     | low    | (DSR)  | Data set ready   |
| 4     | low    | (CTS)  | Clear to send  |
| 5     | low    | (CD)   | Carrier detect   |
| 6     | low    | (RTS)  | Ready to send  |
| 7     | low    | (DTR)  | Data terminal ready  |
| 8     | high   |        | Read overrun   |
| 9     | high   |        | Break sent   |
| 10    | high   |        | Break received   |
| 11    | high   |        | Transmit x-OFFed   |
| 12    | high   |        | Receive x-OFFed  |
| 13-15 | ----   |        | (reserved)   |

## SERIAL FLAGS (bit definitions for io\_SerFlags)

The following table shows the flags that can be set in the `io_SerFlags` field. The default state of all of these flags is zero. `SERF_SHARED`, `SERF_XDISABLED` and `SERF_7WIRE` must always be set before `OpenDevice()`.

Table 40-2: Serial Flags (`io_SerFlags`)

| Flag Name                    | Effect on Device Operation  |
|------------------------------|---|
| <code>SERF_XDISABLED</code>  | Disable the XON/XOFF feature. XON/XOFF <i>must</i> be disabled during XModem transfers.   |
| <code>SERF_EOFMODE</code>    | Set this bit if you want the serial device to check input characters against <code>io_TermArray</code> and to terminate the read immediately if an end-of-file character has been encountered. <i>Note:</i> This bit may be set and reset directly in the user's <code>IOExtSer</code> without a call to <code>SDCMD_SETPARAMS</code> .   |
| <code>SERF_SHARED</code>     | Set this bit if you want to allow other tasks to simultaneously access the serial port. The default is exclusive-access. Any number of tasks may have shared access. Only one task may have exclusive access. If someone already has the port for exclusive access, your <code>OpenDevice()</code> call will fail. This flag must be set before <code>OpenDevice()</code> .   |
| <code>SERF_RAD_BOOGIE</code> | <p>If set, this bit activates high-speed mode. Certain peripheral devices (MIDI, for example) require high serial throughput. Setting this bit high causes the serial device to skip certain of its internal checking code to speed throughput. Use <code>RAD_BOOGIE</code> only when you have:</p> <ul style="list-style-type: none"><li>- Disabled parity checking</li><li>- Disabled XON/XOFF handling</li><li>- Use 8-bit character length</li><li>- Do not wish a test for a break signal</li></ul> <p>Note that the Amiga is a multitasking system and has immediate processing of software interrupts. If there are other tasks running, it is possible that the serial driver may be unable to keep up with high data transfer rates, even with this bit set.</p> |
| <code>SERF_QUEUEDBRK</code>  | <p>If set, every break command that you transmit will be enqueued. This means that all commands will be executed on a FIFO (first in, first out) basis.</p> <p>If this bit is cleared (the default), a break command takes immediate precedence over any serial output already enqueued. When the break command has finished, the interrupted request will continue (if not aborted by the user).</p>   |
| <code>SERF_7WIRE</code>      | If set at <code>OpenDevice()</code> time, the serial device will use seven-wire handshaking for RS-232-C communications. Default is three-wire (pins 2, 3, and 7).  |
| <code>SERF_PARTY_ODD</code>  | If set, selects odd parity. If clear, selects even parity.  |

**SERF\_PARTY\_ON** If set, parity usage and checking is enabled. Also see the **SERF\_MSPON** bit described under **io\_ExtFlags** above.

## SETTING THE PARAMETERS

You change serial parameters by setting the flags and parameters as you desire, and then transmitting a **SDCMD\_SETPARAMS** command to the device. Here is an example:

```
SerialIO->IOSer.io_Command = SDCMD_SETPARAMS;
SerialIO->io_SerFlags      &= ~SERF_PARTY_ON;    /* off */
SerialIO->io_SerFlags      |= SERF_XDISABLED;    /* on */
SerialIO->io_Baud          = 9600;
if (DoIO(SerialIO))
    printf("Error setting parameters!\n");
```

The above fragment modifies two bits in **io\_SerFlags** and changes the baud rate.

### NOTE

A parameter change should not be performed while an IO request is actually being processed, because it might invalidate request handling already in progress. Therefore you should use **SDCMD\_SETPARAMS** only when you have no serial IO requests pending.

## Error codes from the Serial Device

|                              |    |                         |
|------------------------------|----|-------------------------|
| #define SerErr_DevBusy       | 1  | - device in use         |
| #define SerErr_InvBaud       | 3  | - invalid baud rate     |
| #define SerErr_BufErr        | 4  | - out of memory         |
| #define SerErr_InvParam      | 5  | - bad parameter         |
| #define SerErr_LineErr       | 6  | - hardware data overrun |
| #define SerErr_ParityErr     | 9  |                         |
| #define SerErr_TimerErr      | 11 |                         |
| #define SerErr_BufOverflow   | 12 |                         |
| #define SerErr_NoDSR         | 13 | - No Data Set Ready     |
| #define SerErr_NoCTS         | 14 | - No Clear To Send      |
| #define SerErr_DetectedBreak | 15 |                         |

## Multiple serial port support

Applications that use the serial port should provide the user with a means to select the name and unit number of the driver. The defaults will be "serial.device" and unit number 0. Typically unit 0 refers to the user-selected default. Unit 1 refers to the built-in serial port. Numbers above 1 are for extended units. The physically lowest connector on a board will always have the lowest unit number.

Careful attention to error handling is required to survive in a multiple port environment. Differing serial hardware will have different capabilities. The device will refuse to open non-existent unit numbers (symbolic

name mapping of unit numbers is not provided at the device level). The `SDCMD_SETPARAMS` command will fail if the underlying hardware cannot support your parameters. Some devices may use Quick I/O for read or write requests, others will not. Watch out for partially completed read requests; `io_Actual` may not match your requested read length.

If the tooltypes mechanism is used for selecting the device and unit, the defaults of "DEVICE=serial.device" and "UNIT=0" should be provided. The user should be able to permanently set the device and unit in a configuration file.

## Taking Over the Hardware

For some applications use of the device driver interface is not possible. By following the established rules, applications may take over the serial interface at the hardware level. This extreme step is not, however, encouraged. Taking over means losing the ability to work with additional serial ports, and will limit future compatibility.

Access to the hardware registers is controlled by the `misc.resource`. See the "Resources" chapter, and *exec/misc.i* for details. The `MR_SERIALBITS` and `MR_SERIALPORT` units control the serial registers.

One additional complication exists. The current `serial.device` will not release the `misc.resource` bits until after expunge. This code provides a work around:

```
/*
 * A safe way to expunge ONLY a certain device.
 * This code attempts to flush ONLY the named device out of memory and
 * nothing else. If it fails, no status is returned (the information
 * would have no valid use after the Permit()).
 */
#include <exec/types.h>
#include <exec/excbase.h>

void FlushDevice(char *);

extern struct ExecBase *SysBase;

void FlushDevice(name)
char *name;
{
    struct Device *devpoint;

    Forbid(); /* ugly */
    if( devpoint = (struct Device *)FindName(&SysBase->DeviceList,name) )
        RemDevice(devpoint);
    Permit();
}
```

# Chapter 41

## Timer Device

### Introduction

The Amiga timer device provides a general time-delay capability. It can signal you when *at least* a certain amount of time has passed. The timer device is very accurate under normal system loads. But because the Amiga is a multitasking system, the timer device cannot guarantee that exactly the specified amount of time has elapsed — processing overhead increases as more tasks are run. High-performance applications (such as MIDI time-stamping) should take over the 16-bit counters of the CIA B timer resource instead of using the timer device.

The timer device works the same as other Amiga IO devices. To use it, you must first open it. You send commands to it by filling in an IO request block with the amount of time you want and calling the Exec **SendIO()** or **DoIO()** functions. At the end of that time, the device returns a message to you stating that the time has elapsed.

### Timer Device Units

There are two different units in the timer device. One uses the vertical blank interrupt for its “tick” and is called **UNIT\_VBLANK**. The other uses a programmable timer in the 8520 CIA chip and is called **UNIT\_MICROHZ**.

The VBLANK timer unit is very stable and has a precision comparable to the vertical blanking time, that is,  $\pm 16.67$  milliseconds. When you make a timing request, such as "signal me in 21 seconds," the reply will come in  $21 \pm .017$  seconds. This timer has very low overhead and should be used for all long-duration requests.

The MICROHZ timer unit uses the built-in precision hardware timers to create the timing interval you request. It accepts the same type of command—"signal me in so many seconds and microseconds." The microhertz timer has the advantage of greater resolution than the vertical blank timer, but it has less accuracy over long periods of time. The microhertz timer also has much more system overhead, which means accuracy is reduced as the system load increases. It is primarily useful for short-burst timing for which critical accuracy is not required.

The primary means of specifying a requested time is via a **timerequest** structure. A **timerequest** consists of an **IORequest** structure followed by a **timeval** structure, as shown below.

```
struct timerequest
{
    struct IORequest tr_node;
    struct timeval tr_time;
};
```

#### NOTE

The timer driver does not use a standard extension **IORequest** block. It only uses the base **IORequest** structure with a **timeval** structure added on the end. The time data (seconds and microseconds) are stored in the **timeval** structure:

```
struct timeval
{
    ULONG tv_secs;
    ULONG tv_micro;
};
```

The time specified is measured from the time the request is posted. This means you must post a timer request for 30 minutes, rather than for a specific time such as 10:30 p.m. The **micro** field is the number of microseconds in the request. Seconds and microseconds are concatenated by the driver. The number of microseconds must be "normalized;" it should be a value less than one million. You must also take care to avoid posting a **timerequest** of less than 2 microseconds with the **UNIT\_MICROHZ** timer device. *In V1.3 and earlier versions of the Amiga system software, sending a timerequest for 0 or 1 microseconds can cause a system crash. Make sure all your timer requests are for 2 microseconds or more when you use the UNIT\_MICROHZ timer.*

When the specified amount of time has elapsed, the driver will send the **IORequest** back via **ReplyMsg()** (the same as all other drivers). This means that you must fill in the **ReplyPort** pointer of the **IORequest** structure if you wish to be signaled.

#### NOTE

You must *not* reuse an **IORequest** until the timer device has replied to it. When you submit a timer request, the driver destroys the values you have provided in the **timeval** structure. This means that you must reinitialize the time specification before reposting an **IORequest**.

Multiple requests may be posted to the timer driver. For example, you can make three timer requests in a row:

```
Signal me in 20 seconds (request 1)
Signal me in 30 seconds (request 2)
Signal me in 10 seconds (request 3)
```

As the timer queues these requests, it changes the time values and sorts the timer requests to service each request at the desired interval, resulting effectively in the following order:

(request 3) in now+10 seconds  
(request 1) 10 seconds after request 3 is satisfied  
(request 2) 10 seconds after request 1 is satisfied

A sample timer program is given at the end of this chapter.

## Opening a Timer Device

To gain access to a timer unit, you must first open that unit. This is done by using the system function **OpenDevice()**. A typical C-language call is shown below:

```
struct timereq *timer_request_block
error = OpenDevice(TIMERNAME,unit_number,timer_request_block,0);
if(error) cleanup();
```

The parameters shown above are as follows:

### **TIMERNAME**

This is a #define for the null-terminated string, currently "timer.device."

### **unit\_number**

This indicates which timer unit you wish to use, either UNIT\_VBLANK or UNIT\_MICROHZ. These are defined in the *devices/timer.h* and *.i* header files in the *ROM Kernel Manual:Includes and Autodocs*

### **timer\_request\_block**

This is the address of an **IORequest** data structure that will be used later to communicate with the device. The **OpenDevice()** command will fill in the unit and device fields of this data structure.

## Adding a Time Request

You add a timer request to the device by passing a correctly initialized IO request to the timer. The code fragment below demonstrates a sample request:

```
struct timerequest *timermsg;
struct MsgPort *treplyport;
set_timer(seconds,microseconds)
ULONG seconds, microseconds;
{
    /* Can't ask for 0 or 1 microseconds */
    if(seconds==0 && microseconds < 2) return;

    timermsg->tr_node.io_Command      = TR_ADDREQUEST;
    timermsg->tr_node.io_Message.mn_ReplyPort = treplyport;
    timermsg->tr_time.tv_secs  = seconds;
    timermsg->tr_time.tv_micro = microseconds;
    DoIO(timermsg);
}
```



## NOTE

Using DoIO() here puts your task to sleep until the timer request has been satisfied (see the sample program at the end of the chapter).

If you wish to send out multiple timer requests, you have to create multiple request blocks (referenced here as "timersgs"). You can do this by allocating memory for each **timerequest** you need and filling in the appropriate fields with command data. Some fields are initialized by the call to the **OpenDevice()** function. So, for convenience, you may allocate memory for the **timerequests** you need, call **OpenDevice()** with one of them, and then copy the initialized fields into all the other **timerequests**.

It is also permissible to open the timer device multiple times. In some cases this may be easier than opening it once and using multiple requests. When multiple requests are given, **SendIO()** should be used to transmit each one to the timer. The code fragment below illustrates this point:

```
/*-----*/
/* A code fragment showing multiple timer requests */
/*-----*/
struct timerequest *timersg[3];
struct MsgPort *treplyport;
struct Message *msg;

ULONG x, seconds[3], microseconds[3];
{
    /* Timer is already opened with timersg[0]! */
    timersg[0]->tr_node.io_Command      = TR_ADDREQUEST;
    timersg[0]->tr_node.io_Message.mn_ReplyPort = treplyport;

    /* Copy fields from the request used to Open the Timer */
    *timersg[1] = *timersg[0];
    *timersg[2] = *timersg[0];

    /* Initialize other fields */
    for(x=0; x<3; x++)
    {
        timersg[x]->tr_time.tv_secs      = seconds[x];
        timersg[x]->tr_time.tv_micro    = microseconds[x];
    }

    /* Send multiple requests asynchronously */
    /* Do not got to sleep yet... */
    SendIO(timersg[0]);
    SendIO(timersg[1]);
    SendIO(timersg[2]);

    /* There might be other processing done here */

    /* Now go to sleep with WaitPort() waiting for the requests */
    WaitPort(treplyport);

    /* Get the reply message */
    msg=GetMsg(treplyport);
    for(x=0; x<3; x++)
    {
        if(msg==timersg[x])
            printf("Request %ld finished first\n", x);
    }

    /* Be sure to clear the replies for the other 2 requests with GetMsg */
}
```

## Aborting a Timer Request

You call the **AbortIO()** function when you want to cancel a **timerequest** which you have already sent to the timer device:

```
x = AbortIO(timermsg);  
WaitIO(timermsg);
```

### NOTE

You must call **WaitIO()** after you abort the request in order to get the reply message back.

## Closing a Timer

After you have finished using a timer device, you should close it. Be sure to call the **CloseDevice()** function once for each call you have made to **OpenDevice()**:

```
CloseDevice(timermsg);
```

## Additional Timer Functions and Commands

There are two additional timer commands (accessed as standard device commands, using an **IORequest** block as shown above) and three additional functions (accessed as if they were library functions).

The additional timer commands are as follows:

- **TR\_GETSYSTIME** — get the system time
- **TR\_SETSYSTIME** — set the system time

The additional timer library-like functions are:

- **SubTime( Dest, Source )** — subtract the value of one timer request from another
- **AddTime( Dest, Source )** — add the value of one timer request to another
- **result = CmpTime( Dest, Source )** — compare the time in two time requests

## SYSTEM TIME

The "system time" is provided for the convenience of the developer and is also utilized by Intuition. It is not guaranteed to be the same as DOS time as it appears in the DOS **DateStamp** command since DOS time and system time are maintained separately by the Amiga. However, under normal conditions they are the same. The timer.device provides two commands to use with the system time.

The command TR\_SETSYSTIME sets the system's idea of what time it is. The system starts out at time "zero" so it is safe to set it forward to the "real" time. However, care should be taken when setting the time backwards. System time is specified as being monotonically increasing.

The command TR\_GETSYSTIME is used to get the system time. The timer device does not interpret system time to any physical value. By convention, it tells how many seconds have passed since midnight, 1 January 1978. Your program must calculate the time from this value.

The system time is incremented by a special power supply signal that occurs at the external line frequency. This signal is very stable over time, but it can vary by several percent over short periods of time. Hence, system time is stable to within a few seconds a day.

System time is also changed every time someone asks what time it is using TR\_GETSYSTIME. The system does this by incrementing the microsecond counter. This way the return value of the system time is unique and unrepeating so it can be used by applications as a unique identifier.

### NOTE

The timer device sets system time to zero at boot time. AmigaDOS will then reset the system time to the value specified on the boot disk. If the DOS SetClock command is given, this also resets system time.

Here is a program that can be used to determine the system time. Instead of using the Exec support function CreateStdIO() for the request block, the block is initialized "correctly" for use as a timeval request block. The command is executed by the timer device and, on return, the caller can find the data in his request block.

```
/* getsystime.c - get system time */
/* Lattice use -bl - cfist -v -y */
/* Link with lc.lib, amiga.lib */
#include <exec/types.h>
#include <devices/timer.h>
#include <proto/all.h>
#include <stdlib.h>
#include <stdio.h>

struct timerequest tr;
struct MsgPort *tport;
struct Message *msg;

void main(int argc, char **argv)
{
    LONG error;
    ULONG days, hrs, secs, mins, mics;

    /* Open the MICROHZ timer device */
    error = OpenDevice(TIMERNAME, UNIT_MICROHZ, (struct IORequest *) &tr, 0);
    if(error) return; /* If the timer will not open then just return */

    tport = CreatePort(0, 0);
    /* If we can't get a reply port then just quit */
    if(!tport)
```

```

    {
        CloseDevice((struct IORequest *) &tr);
        return;
    }

/* Fill in the IO block with command data */
tr.tr_node.io_Message.mn_Node.ln_Type = NT_MESSAGE;
tr.tr_node.io_Message.mn_Node.ln_Pri = 0;
tr.tr_node.io_Message.mn_Node.ln_Name = NULL;
tr.tr_node.io_Message.mn_ReplyPort = tport;
tr.tr_node.io_Command = TR_GETSYSTIME;

/* Issue the command and wait for it to finish, then get the reply */
DoIO((struct IORequest *) &tr);

/* Get the results and close the timer device */
mics=tr.tr_time.tv_micro;
secs=tr.tr_time.tv_secs;
DeletePort(tport);
CloseDevice((struct IORequest *) &tr);

/* Compute days, hours, etc. */
mins=secs/60;
hrs=mins/60;
days=hrs/24;
secs=secs%60;
mins=mins%60;
hrs=hrs%24;

/* Display the time */
printf("\nSystem Time (measured from Jan.1,1978)\n");
printf("  Days   Hours  Minutes Seconds Microseconds\n");
printf("%6ld %6ld %6ld %6ld %10ld\n",days,hrs,mins,secs,mics);
} /* end of main */

```

## USING THE TIME ARITHMETIC ROUTINES

As indicated above, the time arithmetic routines are accessed in the timer device structure as if they were a routine library. To use them, you create an **IORequest** block and open the timer. In the **IORequest** block is a pointer to the device's base address. This address is needed to access each routine as an offset—for example, **\_LVAddTime**, **\_LVSubTime**, **\_LVOCmpTime**—from that base address.

There are C-language interface routines in *amiga.lib* that perform this interface task for you. They are accessed through a variable called **TimerBase**. You prepare this variable by the following method:

```

/* Lattice use -bl - cfist -v -y */
/* Link with lc.lib, amiga.lib */
#include <exec/types.h>
#include <exec/memory.h>
#include <devices/timer.h>
#include <proto/all.h>
#include <stdio.h>
# ifdef LATTICE
int CXBRK(void) {return(0); } /* Disable Lattice CTRL-C */
# endif
struct Library *TimerBase; /* setup the interface variable (must be global) */

void main(int argc,char **argv)
{
    struct timeval      *time1, *time2, *time3;
    struct timerequest *tr;
    LONG                error,result;
    /*-----*/
    /* Get some memory for our structures */
    /*-----*/

```

```

time1=(struct timeval *)AllocMem(sizeof(struct timeval),
                                MEMF_PUBLIC | MEMF_CLEAR);
time2=(struct timeval *)AllocMem(sizeof(struct timeval),
                                MEMF_PUBLIC | MEMF_CLEAR);
time3=(struct timeval *)AllocMem(sizeof(struct timeval),
                                MEMF_PUBLIC | MEMF_CLEAR);
tr=(struct timerequest *)AllocMem(sizeof(struct timerequest),
                                MEMF_PUBLIC | MEMF_CLEAR);
/* Make sure we got the memory */
if(!time1 | !time2 | !time3 | !tr) goto cleanexit;
/*-----*/
/* Set up some values to test time arithmetic with */
/* In a real application these values might be filled */
/* in via the GET_SYSTIME command of the timer device */
/*-----*/
time1->tv_secs = 3;   time1->tv_micro = 0;           /* 3.0 seconds */
time2->tv_secs = 2;   time2->tv_micro = 500000;     /* 2.5 seconds */
time3->tv_secs = 1;   time3->tv_micro = 900000;     /* 1.9 seconds */

printf("Time1 is %ld.%ld\n", time1->tv_secs,time1->tv_micro);
printf("Time2 is %ld.%ld\n", time2->tv_secs,time2->tv_micro);
printf("Time3 is %ld.%ld\n\n",time3->tv_secs,time3->tv_micro);
/*-----*/
/* Open the MICROHZ timer device */
/*-----*/
error = OpenDevice(TIMERNAME,UNIT_MICROHZ,(struct IORequest *) tr, 0L);
if(error) goto cleanexit;

/* Set up to use the special time arithmetic functions */
TimerBase = (struct Library *)tr->tr_node.io_Device;
/*-----*/
/* Now that TimerBase is initialized, it is permissible */
/* to call the time-comparison or time-arithmetic routines */
/* Result of this example is -1 which means the first */
/* parameter has greater time value than second parameter */
/* +1 means the second parameter is bigger; 0 means equal. */
/*-----*/
result = CmpTime( time1, time2 );
printf("Time1 and 2 compare = %ld\n",result);

/* Add to time1 the values in time2 */
AddTime( time1, time2);
printf("Time1+time2 result = %ld.%ld\n",time1->tv_secs,time1->tv_micro);

/* Subtract values in time3 from the value */
/* currently in time1. Results in time1. */
SubTime( time2, time3);
printf("Time2-time3 result = %ld.%ld\n",time2->tv_secs,time2->tv_micro);
/*-----*/
/* Free system resources that we used */
/*-----*/
cleanexit:
if (time1) FreeMem(time1,sizeof(struct timeval));
if (time2) FreeMem(time2,sizeof(struct timeval));
if (time3) FreeMem(time3,sizeof(struct timeval));
if (tr) FreeMem(tr, sizeof(struct timerequest));
if (!error)CloseDevice((struct IORequest *) tr);
}

```

## WHY USE TIME ARITHMETIC?

As mentioned earlier in this section, because of the multitasking capability of the Amiga, the timer device can provide timings that are at least as long as the specified amount of time. If you need more precision than this, using the system timer along with the time arithmetic routines can at least, in the long run, let you synchronize your software with this precision timer after a selected period of time.

Say, for example, that you select timer intervals so that you get 161 signals within each 3-minute span. Therefore, the **timeval** you would have selected would be 180/161, which comes out to 1 second and 118,012 microseconds per interval. Considering the time it takes to set up a call to **set\_timer** and delays due to task-switching (especially if the system is very busy), it is possible that after 161 timing intervals, you may be somewhat beyond the 3-minute time. Here is a method you can use to keep in sync with system time:

1. Begin.
2. Read system time; save it.
3. Perform your loop however many times in your selected interval.
4. Read system time again, and compare it to the old value you saved. (For this example, it will be more or less than 3 minutes as a total time elapsed.)
5. Calculate a new value for the time interval (**timeval**); that is, one that (if precise) would put you exactly in sync with system time the next time around. **Timeval** will be a lower value if the loops took too long, and a higher value if the loops didn't take long enough.
6. Repeat the cycle.

Over the long run, then, your average number of operations within a specified period of time can become precisely what you have designed.

## Sample Timer Program

Here is an example program showing how to use a timer device.

```
/* Simple Timer Example Program:
 *
 * Includes dynamic allocation of data structures needed to communicate
 * with the timer device as well as the actual device IO
 * Lattice use lc -bl -cfist -v -y Link with lc.lib, amiga.lib
 */

#include <exec/types.h>          /* Some system header files we need */
#include <exec/memory.h>
#include <devices/timer.h>
#include <proto/all.h>
#include <stdio.h>

/* Our timer sub-routines */
void delete_timer (struct timerequest *);
LONG get_sys_time (struct timeval *);
LONG set_new_time (LONG);
void wait_for_timer(struct timerequest *, struct timeval *);
LONG time_delay ( struct timeval *, LONG );
struct timerequest *create_timer( ULONG );
void show_time (ULONG);

#ifdef LATTICE
int CXBRK(void) {return(0); } /* Disable Lattice CTRL-C handling */
#endif

struct Library *TimerBase;      /* to get at the time comparison functions */

/* manifest constants -- "never will change" */
```

```

#define SECSPERMIN (60)
#define SECSPERHOUR (60*60)
#define SECSPERDAY (60*60*24)

void main(int argc, char **argv)
{
    LONG seconds;
    struct timerequest *tr; /* IO block for timer commands */
    struct timeval oldtimeval; /* timevals to store times */
    struct timeval mytimeval;
    struct timeval currentval;

    printf("Timer test\n");

    /* sleep for two seconds */
    currentval.tv_secs = 2;
    currentval.tv_micro = 0;
    time_delay( &currentval, UNIT_VBLANK );
    printf( "After 2 seconds delay\n" );

    /* sleep for four seconds */
    currentval.tv_secs = 4;
    currentval.tv_micro = 0;
    time_delay( &currentval, UNIT_VBLANK );
    printf( "After 4 seconds delay\n" );

    /* sleep for 500,000 micro-seconds = 1/2 second */
    currentval.tv_secs = 0;
    currentval.tv_micro = 500000;
    time_delay( &currentval, UNIT_MICROHZ );
    printf( "After 1/2 second delay\n" );

    printf( "DOS Date command shows: " );
    (void) Execute( "date", 0, 0 );

    /* save what system thinks is the time....we'll advance it temporarily */
    get_sys_time( &oldtimeval );
    printf("Original system time is:\n");
    show_time(oldtimeval.tv_secs );

    printf("Setting a new system time\n");

    seconds = 1000 * SECSPERDAY + oldtimeval.tv_secs;

    set_new_time( seconds );
    /* (if user executes the AmigaDOS DATE command now, he will*/
    /* see that the time has advanced something over 1000 days */

    printf( "DOS Date command now shows: " );
    (void) Execute( "date", 0, 0 );

    get_sys_time( &mytimeval );
    printf( "Current system time is:\n");
    show_time(mytimeval.tv_secs);

    /* Added the microseconds part to show that time keeps */
    /* increasing even though you ask many times in a row */
    get_sys_time( &mytimeval );
    printf("TimeA %ld.%ld\n", mytimeval.tv_secs,
        mytimeval.tv_micro);
    get_sys_time( &mytimeval );
    printf("TimeB %ld.%ld\n", mytimeval.tv_secs,
        mytimeval.tv_micro);
    get_sys_time( &mytimeval );
    printf("TimeC %ld.%ld\n", mytimeval.tv_secs,
        mytimeval.tv_micro);

    printf( "\nResetting to former time\n" );
    set_new_time( oldtimeval.tv_secs );

    get_sys_time( &mytimeval );
    printf( "Current system time is:\n");

```

```

show_time(mytimeval.tv_secs);

/* just shows how to set up for using the timer functions, does not */
/* demonstrate the functions themselves. (TimerBase must have a */
/* legal value before AddTime, SubTime or CmpTime are performed. */
tr = create_timer( UNIT_MICROHZ );
TimerBase = (struct Library *)tr->tr_node.io_Device;

/* and how to clean up afterwards */
TimerBase = (struct Library *)(-1);
delete_timer( tr );
}

struct timerequest *create_timer( ULONG unit )
{
    /* return a pointer to a timer request. If any problem, return NULL */
    LONG error;
    struct MsgPort *timerport;
    struct timerequest *timermsg;

    timerport = CreatePort( 0, 0 );
    if( timerport == NULL )
    {
        return( NULL );
    }
    timermsg = (struct timerequest *)
        CreateExtIO( timerport, sizeof( struct timerequest ) );
    if( timermsg == NULL )
    {
        return( NULL );
    }
    error = OpenDevice( TIMERNAME, unit, (struct IORequest *) timermsg, 0L );
    if( error != 0 )
    {
        delete_timer( timermsg );
        return( NULL );
    }
    return( timermsg );
}

/* more precise timer than AmigaDOS Delay() */
LONG time_delay( struct timeval *tv, LONG unit )
{
    struct timerequest *tr;
    /* get a pointer to an initialized timer request block */
    tr = create_timer( unit );

    /* any nonzero return says timedelay routine didn't work. */
    if( tr == NULL ) return( -1L );

    wait_for_timer( tr, tv );

    /* deallocate temporary structures */
    delete_timer( tr );
    return( 0L );
}

void wait_for_timer( struct timerequest *tr, struct timeval *tv )
{
    /*-----*/
    /* With the UNIT_MICROHZ timer, it is illegal */
    /* to wait for 0 or 1 microseconds! */
    /*-----*/
    if( tv->tv_secs==0L && tv->tv_micro < 2L ) return;

    tr->tr_node.io_Command = TR_ADDREQUEST; /* add a new timer request */

    /* structure assignment */
    tr->tr_time = *tv;

    /* post request to the timer -- will go to sleep till done */
    DoIO((struct IORequest *) tr );
}

```



```

}

LONG set_new_time(LONG secs)
{
    struct timerequest *tr;
    tr = create_timer( UNIT_MICROHZ );

    /* non zero return says error */
    if( tr == 0 ) return( -1 );

    tr->tr_node.io_Command = TR_SETSYSTIME;
    tr->tr_time.tv_secs = secs;
    tr->tr_time.tv_micro = 0;
    DoIO((struct IORequest *) tr );

    delete_timer(tr);
    return(0);
}

LONG get_sys_time(struct timeval *tv)
{
    struct timerequest *tr;
    tr = create_timer( UNIT_MICROHZ );

    /* non zero return says error */
    if( tr == 0 ) return( -1 );

    tr->tr_node.io_Command = TR_GETSYSTIME;
    DoIO((struct IORequest *) tr );

    /* structure assignment */
    *tv = tr->tr_time;

    delete_timer( tr );
    return( 0 );
}

void delete_timer(struct timerequest *tr )
{
    struct MsgPort *tp;
    if( tr != 0 )
    {
        tp = tr->tr_node.io_Message.mn_ReplyPort;
        if(tp != 0)
        {
            DeletePort(tp);
        }
        CloseDevice( (struct IORequest *) tr );
        DeleteExtIO( (struct IORequest *) tr );
    }
}

void show_time(ULONG secs)
{
    ULONG days,hrs,mins;

    /* Compute days, hours, etc. */
    mins=secs/60;
    hrs=mins/60;
    days=hrs/24;
    secs=secs%60;
    mins=mins%60;
    hrs=hrs%24;

    /* Display the time */
    printf("    Hour Minute Second  (Days since Jan.1,1978)\n");
    printf("**%5ld:%5ld:%5ld      (%6ld )\n\n",hrs,mins,secs,days);
}
    /* end of main */

```

## Chapter 42

# Trackdisk Device

### Introduction

The Amiga trackdisk device directly drives the disk, controls the disk motors, reads raw data from the tracks, and writes raw data to the tracks. Normally, you use the AmigaDOS functions to write or read data from the disk. The trackdisk device is the lowest-level software access to the disk data and is used by AmigaDOS to access the disks. The trackdisk device supports the usual commands such as `CMD_WRITE` and `CMD_READ`. In addition, it supports an extended form of these commands to allow additional control over the trackdisk.device.

The trackdisk device can queue up command sequences so that your task can do something else while it is waiting for a particular disk activity to occur. If several sequenced write commands are queued to a disk, a task assumes that all such writes are going to the same disk. The trackdisk device itself can stop a command sequence if it senses that the disk has been changed, returning all subsequent `IORequest` blocks to the caller with an error ("disk changed").

## The Amiga Floppy Disk

The standard 3.5 inch Amiga floppy disk consists of a number of tracks that are NUMSECS (11) sectors of TD\_SECTOR (512) usable data bytes plus TD\_LABELSIZE (16) bytes of label area. There are usually 2 tracks per cylinder (2 heads) and 80 cylinders per disk. (The number of tracks can be found via a request to the trackdisk device.

### NOTE

The result is given in tracks and not cylinders.) On a standard 3.5" drive, this gives useful space of 880K bytes plus 28K bytes of label area per floppy disk.

Although the disk is logically divided up into sectors, all I/O to the disk is done a track at a time. This allows access to the drive with no interleaving and increases the useful storage capacity by about 20 percent. Normally, a read of a sector will only have to copy the data from the track buffer. If the track buffer contains another track's data, then the buffer will first be written back to the disk (if it is "dirty") and the new track will be read in. All track boundaries are transparent to the programmer (except for FORMAT, SEEK, and RAWREAD/RAWWRITE) because you give the device an offset into the disk in the number of bytes from the start of the disk. The device ensures that the correct track is brought into memory.

The performance of the disk is greatly enhanced if you make effective use of the track buffer. The performance of sequential reads will be up to an order of magnitude greater than reads scattered across the disk.

The trackdisk device uses the blitter to encode and decode the data to and from the track buffer. Because the blitter can access only chip memory (memory that is accessible to the special-purpose chips is known as MEMF\_CHIP to the memory allocator AllocMem()), all buffers sent to the trackdisk must be in chip memory. In addition, only full-sector writes on sector boundaries are supported.

### NOTE

The user's buffer must be word-aligned.

The trackdisk device is based upon a standard device structure. It has the following restrictions:

- All reads and writes must use an `io_Length` that is an integer multiple of TD\_SECTOR bytes (the sector size in bytes).
- The offset field must be an integer multiple of TD\_SECTOR.
- The data buffer must be word-aligned and in MEMF\_CHIP memory.

## Trackdisk Device Commands

The trackdisk device allows the following system interface functions and commands. In addition to the usual device commands, the trackdisk device has a set of extended commands.

The system interface functions are:

|                      |  |
|----------------------|--|
| <b>OpenDevice()</b>  | Obtain exclusive use of a particular disk unit |
| <b>CloseDevice()</b> | Release the unit to another task               |
| <b>Expunge()</b>     | Remove the device from the device list         |
| <b>BeginIO()</b>     | Dispatch a device command; queue commands      |
| <b>AbortIO()</b>     | Abort a device command                         |

The device-specific commands are:

|                        |  |
|------------------------|--|
| <b>CMD_READ</b>        | Read one or more sectors                           |
| <b>CMD_WRITE</b>       | Write one or more sectors                          |
| <b>CMD_UPDATE</b>      | Write out track buffer if dirty                    |
| <b>CMD_CLEAR</b>       | Mark track buffer as invalid                       |
| <b>TD_MOTOR</b>        | Turn the motor on or off                           |
| <b>TD_SEEK</b>         | Move the head to a specific track                  |
| <b>TD_FORMAT</b>       | Initialize one or more tracks                      |
| <b>TD_CHANGENUM</b>    | Discover the current disk-change number            |
| <b>TD_CHANGESTATE</b>  | See if there is a disk present in a drive          |
| <b>TD_PROTSTATUS</b>   | See if a disk is write-protected                   |
| <b>TD_RAWREAD</b>      | Read RAW sector data from disk (unencoded MFM)     |
| <b>TD_RAWWRITE</b>     | Write RAW sector data to disk                      |
| <b>TD_GETDRIVETYPE</b> | Discover the type of disk drive in use by the unit |
| <b>TD_GETNUMTRACKS</b> | Discover the number of tracks usable with the unit |
| <b>TD_ADDCHANGEINT</b> | Add a diskchange handler                           |
| <b>TD_REMCHANGEINT</b> | Remove a diskchange handler                        |

In addition to the device-specific commands listed above, the trackdisk device has a number of enhanced commands. These commands are similar to their normal counterparts but have additional features: they allow you to control whether a command will be executed if the disk has been changed and they allow you to read or write to the sector label portion of a sector.

Enhanced commands take a slightly larger I/O request block, which contains information that is needed only by the enhanced command and that is ignored by the standard form of that command. The extra information takes the form of two extra longwords at the end of the data structure. These commands are performed only if the change count is less than or equal to the one in the **iotd\_Count** field of the command's I/O request block. The enhanced commands are listed below:

|                     |  |
|---------------------|--|
| <b>ETD_WRITE</b>    | Write one or more sectors (plus sector labels)     |
| <b>ETD_READ</b>     | Read one or more sectors (plus sector labels)      |
| <b>ETD_MOTOR</b>    | Turn the motor on or off                           |
| <b>ETD_SEEK</b>     | Move the head to a specific track                  |
| <b>ETD_FORMAT</b>   | Initialize one or more tracks (plus sector labels) |
| <b>ETD_UPDATE</b>   | Write out track buffer if dirty                    |
| <b>ETD_CLEAR</b>    | Mark track buffer as invalid                       |
| <b>ETD_RAWREAD</b>  | Read RAW sector data from disk (unencoded MFM)     |
| <b>ETD_RAWWRITE</b> | Write RAW sector data to disk                      |

## Creating an I/O Request

The trackdisk device, like other devices, requires that you create an I/O request message that you pass to the device for processing. The message contains the command and several other items of control information.

Here is a program fragment that can be used to create the message block that you use for trackdisk communications. In the fragment, the routine **CreateExtIO()** is called to return a pointer to a message block. This is acceptable for the standard form of the commands since the enhanced structure contains two additional fields at the end of the standard structure. **CreateExtIO()** is in *amiga.lib* and is listed in the appendices of the *Amiga ROM Kernel Reference Manual: Includes & AutoDocs*.

```
struct IOExtTD *diskReq; /* IOResult block pointer for enhanced commands */
struct MsgPort *diskPort; /* a port at which to receive replies */

if (diskPort=CreatePort(NULL,NULL))
{
    if (diskReq=(struct IOExtTD *)CreateExtIO(diskPort,
                                              sizeof(struct IOExtTD)))
    {
        /* do your stuff here... */

        DeleteExtIO((struct IOResult *)diskReq);
    }
    else printf("Out of memory\n");
    DeletePort(diskPort);
}
else printf("Could not create diskReq port\n");
```

The routine **CreatePort()** is part of *amiga.lib*. It returns a pointer to a **MsgPort** structure that can be used to receive replies from the trackdisk device.

The routine **CreateExtIO()** is part of *amiga.lib*. It returns a pointer to an **IOResult** structure of the size requested. To create a standard **IOResult** structure, you would call this function with `sizeof(struct IOStdReq)`.

The data structure **IOExtTD** takes the form:

```
struct IOExtTD
{
    struct IOStdReq iotd_Req;
    ULONG iotd_Count;
    ULONG iotd_SecLabel;
};
```

where

### **IOStdReq**

is a standard **IOResult** block that contains fields used to transmit the standard commands (explained below).

### **iotd\_Count**

helps keep old I/O requests from being performed when the diskette has been changed. Any I/O request found with an **iotd\_Count** less than the current change counter value will be returned with a characteristic error (**TDERR\_DiskChange**) in the **io\_Error** field of the I/O request block. This allows stale I/O requests to be returned to the user after a disk has been changed. The current disk-change counter value can be obtained by **TD\_CHANGENUM**.

If the user wants enhanced disk I/O but does not care about disk removal, then **iotd\_Count** may be set to the maximum unsigned long integer value (0xFFFFFFFF).

#### **iotd\_SecLabel**

allows access to the sector identification section of the sector header.

Each sector has 16 bytes of descriptive data space available to it; the trackdisk device does not interpret this data. If **iotd\_SecLabel** is null, then this descriptive data is ignored. If it is not null, then **iotd\_SecLabel** should point to a series of contiguous 16-byte chunks (one for each sector that is to be read or written). These chunks will be written out to the sector's label region on a write or filled with the sector's label area on a read. If a **CMD\_WRITE** (the standard write call) is done, then the sector label area is left unchanged.

When the trackdisk device is requested to provide status information for commands such as **TD\_REMOVE** or **TD\_CHANGENUM**, the value is returned in the **io\_Actual** field of the **IORequest**.

## Opening a Trackdisk Device

To gain access to a disk unit, you must first open the unit by using the system command **OpenDevice()**. A typical C-language call is shown below:

```
error = OpenDevice(TD_NAME, unit_number, disk_request_block, flags);
```

where:

#### **TD\_NAME**

is a define for a null-terminated string, in this case "trackdisk.device."

#### **unit\_number**

is the disk unit you wish to use (defined below).

#### **disk\_request\_block**

is the address of an **IORequest** data structure that will later be used to communicate with the device. The **OpenDevice()** function will fill in the unit and device fields of this data structure.

#### **flags**

tell how the I/O device is to be opened. Currently, there are only two values for this flag: 0 to open the device with only standard 3.5" units as valid; and **TDF\_ALLOW\_NON\_3\_5** which will currently allow you to also open the 5.25" drive units.

#### **error**

0 signifies success, otherwise it is the error code.

The **unit\_number** can be any value from 0 to 3. Unit 0 is the built-in 3 1/2-inch disk drive. Units 1 through 3 represent additional disk drives that may be connected to an Amiga system.

The following are some common errors that may be returned from an **OpenDevice()** call.

#### **TDERR\_DriveInUse**

Some other task has already been granted exclusive use of this device.

### **TDERR\_BadUnitNum**

Either you have specified a unit number outside the range of 0-3 or a unit is not connected in the specified position.

### **TDERR\_BadDriveType**

You may be trying to use a device that is not compatible with the trackdisk device. This error will occur when you try opening a device that is a 5.25 inch device without specifying the TBF\_ALLOW\_NON\_3\_5 flag in the `OpenDevice()` call.

See `devices/trackdisk.h/i` for other trackdisk error codes.

## **Sending a Command to the Device**

You send a command to this device by initializing the appropriate fields of your `IOStdReq` or `IOExtTD` and then using `SendIO()`, `DoIO()`, or `BeginIO()` to transmit the command to the device. Here is an example:

```
/*
 * This turns the motor on
 */
VOID Motor_On(struct IOExtTD *diskReq)
{
    diskReq->iotd_Req.io_Length=1;
    diskReq->iotd_Req.io_Command=TD_MOTOR;
    DoIO((struct IORequest *)diskReq);    /* task will sleep till done */
}
```

## **Terminating Access to the Device**

As with all devices, you *must* close the trackdisk device when you have finished using it. To release the device, a `CloseDevice()` call is executed with the same `IORequest` used when the device was opened. This only closes the device and makes it available to the rest of the system. It does **not** de-allocate the `IORequest` structure.

## **Device-specific Commands**

The device-specific commands supported by the trackdisk.device are explained below. All commands of the `ETD_xxx` variety have the ability to check for diskchange and will return an error if the disk was changed. Some of the `ETD_xxx` commands also give access to the sector label area on the disk. Commands that do this are marked accordingly.

### **ETD\_READ and CMD\_READ**

`ETD_READ` and `CMD_READ` obey all of the trackdisk device restrictions noted above. They transfer data from the track buffer to the user's buffer. If the desired sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read. `ETD_READ` will read the sector label area if the `iotd_SecLabel` is non-NULL.

|                              |   |
|------------------------------|---|
| diskReq->iotd_Req.io_Command | ETD_READ or CMD_READ  |
| diskReq->iotd_Req.io_Length  | number of bytes to READ<br>(Must be a multiple of TD_SECTOR)        |
| diskReq->iotd_Req.io_Data    | pointer to buffer (of io_Length bytes)                              |
| diskReq->iotd_Req.io_Offset  | byte offset from start of disk<br>(Must be a multiple of TD_SECTOR) |

For the enhanced version of the request:

|                        |  |
|------------------------|--|
| diskReq->iotd_Count    | Change count number  |
| diskReq->iotd_SecLabel | NULL or sector label buffer pointer<br>(Size must be a multiple of TD_LABELSIZE) |

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|

## ETD\_WRITE and CMD\_WRITE

ETD\_WRITE and CMD\_WRITE obey all of the trackdisk device restrictions noted above. They transfer data from the user's buffer to track buffer. If the track that contains this sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read in for modification. ETD\_WRITE will write the sector label area if iotd\_SecLabel is non-NULL.

|                              |   |
|------------------------------|---|
| diskReq->iotd_Req.io_Command | ETD_WRITE or CMD_WRITE  |
| diskReq->iotd_Req.io_Length  | number of bytes to WRITE<br>(Must be a multiple of TD_SECTOR)       |
| diskReq->iotd_Req.io_Data    | pointer to buffer (of io_Length bytes)                              |
| diskReq->iotd_Req.io_Offset  | byte offset from start of disk<br>(Must be a multiple of TD_SECTOR) |

For the enhanced version of the request:

|                        |  |
|------------------------|--|
| diskReq->iotd_Count    | Change count number  |
| diskReq->iotd_SecLabel | NULL or sector label buffer pointer<br>(Size must be a multiple of TD_LABELSIZE) |

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|



## ETD\_UPDATE AND CMD\_UPDATE

The Amiga trackdisk device does not write data sectors unless it is necessary (you request that a different track be used) or until the user requests that an update be performed. This improves system speed by caching disk operations. The update commands ensure that any buffered data is flushed out to the disk. If the track buffer has not been changed since the track was read in, the update commands do nothing. As usual, the ETD\_UPDATE command checks for diskchange.

diskReq->iotd\_Req.io\_Command            ETD\_UPDATE or CMD\_UPDATE

For the enhanced version of the request:

diskReq->iotd\_Count                    Change count number

Result:

diskReq->iotd\_Req.io\_Error            Error return (see error table below)

## ETD\_CLEAR and CMD\_CLEAR

ETD\_CLEAR and CMD\_CLEAR mark the track buffer as invalid, forcing a reread of the disk on the next operation. ETD\_UPDATE or CMD\_UPDATE would be used to force data out to the disk before turning the motor off. ETD\_CLEAR or CMD\_CLEAR is usually used after having locked out the trackdisk.device via the use of the disk resource, when you wish to prevent the track from being updated, or when you wish to force the track to be re-read. ETD\_CLEAR or CMD\_CLEAR will not do an update, nor will an update command do a clear.

diskReq->iotd\_Req.io\_Command            ETD\_CLEAR or CMD\_CLEAR

For the enhanced version of the request:

diskReq->iotd\_Count                    Change count number

Result:

diskReq->iotd\_Req.io\_Error            Error return (see table below)

## ETD\_MOTOR and TD\_MOTOR

ETD\_MOTOR and TD\_MOTOR give you control of the motor. The **io\_Length** field contains the requested state of the motor. A 1 will turn the motor on; a 0 will turn it off. The old state of the motor is returned in **io\_Actual**. If **io\_Actual** is zero, then the motor was off. Any other value implies that the motor was on. If the motor is just being turned on, the device will delay the proper amount of time to allow the drive to come up to speed. Normally, turning the drive on is not necessary—the device does this automatically if it receives a request when the motor is off. However, turning the motor off is the programmer's responsibility. In addition, the standard instructions to the user are that it is safe to remove a diskette if and only if the motor is off (that is, if the disk light is off).

|                              |                          |
|------------------------------|--------------------------|
| diskReq->iotd_Req.io_Command | ETD_MOTOR or TD_MOTOR    |
| diskReq->iotd_Req.io_Length  | 1=motor on, 0=motor off. |

For the enhanced version of the request:

|                     |                     |
|---------------------|---------------------|
| diskReq->iotd_Count | Change count number |
|---------------------|---------------------|

Result:

|                             |                                      |
|-----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Actual | Boolean - previous motor state       |
| diskReq->iotd_Req.io_Error  | Error return (see error table below) |

## ETD\_FORMAT and TD\_FORMAT

ETD\_FORMAT and TD\_FORMAT are used to write data to a track that either has not yet been formatted or has had a hard error on a standard write command. TD\_FORMAT completely ignores all data currently on a track and does not check for disk change before performing the command. The `io_Data` field must point to at least one track worth of data. The `io_Offset` field must be track aligned, and the `io_Length` field must be in units of track length (that is, `NUMSECS*TD_SECTOR`). The device will format the requested tracks, filling each sector with the contents of the buffer pointed to by `io_Data` field. You should do a read pass to verify the data.

If you have a hard write error during a normal write, you may find it possible to use the TD\_FORMAT command to reformat the track as part of your error recovery process. ETD\_FORMAT will write the sector label area if the `ioid_SecLabel` is non-NULL.

|                              |  |
|------------------------------|--|
| diskReq->iotd_Req.io_Command | ETD_FORMAT or TD_FORMAT  |
| diskReq->iotd_Req.io_Length  | number of bytes to format<br>(Must be a multiple of <code>TD_SECTOR * NUMSEC</code> )      |
| diskReq->iotd_Req.io_Data    | pointer to buffer (of <code>io_Length</code> bytes)  |
| diskReq->iotd_Req.io_Offset  | byte offset from start of disk<br>(Must be a multiple of <code>TD_SECTOR * NUMSEC</code> ) |

For the enhanced version of the request:

|                        |   |
|------------------------|---|
| diskReq->iotd_Count    | Change count number   |
| diskReq->iotd_SecLabel | NULL or sector label buffer pointer<br>(Size must be a multiple of <code>TD_LABELSIZE * NUMSEC</code> ) |

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|

## Status Commands

The commands that return status on the current disk in the unit are TD\_CHANGENUM, TD\_CHANGESTATE, TD\_PROTSTATUS, TD\_GETDRIVETYPE, and TD\_GETNUMTRACKS. These calls may be done with QuickIO and thus may be called within interrupt handlers (such as the trackdisk disk change handler).

## **TD\_CHANGENUM**

**TD\_CHANGENUM** returns the current value of the disk-change counter (as used by the enhanced commands—see below). The disk change counter is incremented each time the disk is inserted or removed.

```
diskReq->iotd_Req.io_Command  TD_CHANGENUM
diskReq->iotd_Req.io_Flags    IOF_QUICK (Not required)
```

**Result:**

```
diskReq->iotd_Req.io_Actual    Disk change number
```

## **TD\_CHANGESTATE**

**TD\_CHANGESTATE** returns zero if a disk is currently in the drive, and nonzero if the drive has no disk.

```
diskReq->iotd_Req.io_Command  TD_CHANGESTATE
diskReq->iotd_Req.io_Flags    IOF_QUICK (Not required)
```

**Result:**

```
diskReq->iotd_Req.io_Actual    State: 0=Disk IN drive.
```

## **TD\_PROTSTATUS**

**TD\_PROTSTATUS** returns nonzero if the current diskette is write-protected.

```
diskReq->iotd_Req.io_Command  TD_PROSTATUS
diskReq->iotd_Req.io_Flags    IOF_QUICK (Not required)
```

**Result:**

```
diskReq->iotd_Req.io_Actual    Protection state: 0=non-protected
```

## **TD\_GETDRIVETYPE**

**TD\_GETDRIVETYPE** returns the drive type for the unit that was opened. The unit can be opened only if the device understands the drive type it is connected to.

diskReq->iotd\_Req.io\_Command    TD\_GETDRIVETYPE  
diskReq->iotd\_Req.io\_Flags    IOF\_QUICK (Not required)

Result:

diskReq->iotd\_Req.io\_Actual    Drive type: See trackdisk.h/i

## TD\_GETNUMTRACKS

TD\_GETNUMTRACKS returns the number tracks on that device. This is the number of tracks of TD\_SECTOR \* NUMSECS size. It is not the number of cylinders. With two heads, the number of cylinders is half of the number of tracks. The number of cylinders is equal to the number of tracks divided by the number of heads (surfaces). The standard 3.5" Amiga drive has two heads.

diskReq->iotd\_Req.io\_Command    TD\_GETNUMTRACKS  
diskReq->iotd\_Req.io\_Flags    IOF\_QUICK (Not required)

Result:

diskReq->iotd\_Req.io\_Actual    Number of tracks on the disk

## Being Notified of Disk Changes

Many programs will wish to be notified if the user has changed the disk in the active drive. While this can be done via the Intuition DISKREMOVED and DISKINSERTED messages, sometimes more tightly controlled testing is required.

## TD\_ADDCHANGEINT

TD\_ADDCHANGEINT lets you add a software interrupt handler to the disk device that will be Cause()'ed when a disk insert or remove occurs. The structure for the handler is identical to other handlers. See the software interrupt chapter for more information on the handler. Within the handler, you may only call the status commands that can use IOF\_QUICK.

To set up the handler, an Interrupt structure must be initialized. This structure is supplied as the io\_Data to the TD\_ADDCHANGEINT command. The handler will be linked into the handler chain and will execute whenever a disk change happens. You *must* remove the handler before you exit.

### WARNING

This command does *not* return when executed. It holds onto the IO request until the TD\_REMCHANGEINT command is executed with that same IO request. Hence, you must use SendIO() with this command.

|                              |                                |
|------------------------------|--------------------------------|
| diskReq->iotd_Req.io_Command | TD_ADDCHANGEINT                |
| diskReq->iotd_Req.io_Length  | sizeof(struct Interrupt)       |
| diskReq->iotd_Req.io_Data    | pointer to Interrupt structure |

## TD\_REMCHANGEINT

TD\_REMCHANGEINT removes the interrupt handler from the device's interrupt handler list. You *must* pass it the same Interrupt structure used to add the handler.

### WARNING

Under V1.3 and earlier versions of the Amiga system software, TD\_REMCHANGEINT does not work and should not be used.

|                              |                                |
|------------------------------|--------------------------------|
| diskReq->iotd_Req.io_Command | TD_REMCHANGEINT                |
| diskReq->iotd_Req.io_Length  | sizeof(struct Interrupt)       |
| diskReq->iotd_Req.io_Data    | pointer to Interrupt structure |
| diskReq->iotd_Req.io_Flags   | IOF_QUICK ( <i>Required</i> )  |

## Commands for Low-Level Access

The following commands may be used to read the raw flux changes on the disk. The data returned from the TD\_RAWREAD or sent to TD\_RAWWRITE should be encoded into some form of legal flux patterns. See the *Amiga Hardware Manual* and books on magnetic media recording and reading.

### WARNING

In V1.3 Kickstart and earlier these functions are unreliable even though under certain configurations the commands may appear to work.

## ETD\_RAWREAD and TD\_RAWREAD

ETD\_RAWREAD and TD\_RAWREAD perform a raw read from a track on the disk. They seek to the specified track and read it into the user's buffer.

*No processing of the track is done.* It will appear exactly as the bits come off the disk -- typically in some legal flux format (such as MFM, FM, GCR, etc; if you don't know what these are, you shouldn't be using this call). *Caveat Programmer.*

This interface is intended for sophisticated programming only. You must fully understand digital magnetic recording to be able to utilize this call. It is also important that you understand that the MFM encoding scheme used by the higher level trackdisk routines may change without notice. Thus, this routine is only really useful for reading and decoding other disks such as MS-DOS formatted disks.

LIMITATIONS for sync'ed reads and writes: There is a delay between the index pulse and the start of bits coming in from the drive (e.g. dma started). It is in the range of 135-200 microseconds. This delay breaks down as follows: 55 microseconds for software interrupt overhead (this is the time from interrupt to the write of the DSKLEN register); 66 microsecs for one horizontal line delay (remember that disk I/O is synchronized with Agnus' display fetches). The last variable (0-65 microseconds) is an additional scan line since DSKLEN is poked anywhere in the horizontal line. This leaves 15 microseconds unaccounted for. In short, You will almost never get bits within the first 135 microseconds of the index pulse, and may not get it until 200 microseconds. At 4 microsecs/bit, this works out to be between 4 and 7 bytes of user data delay.

### WARNING

Commodore-Amiga may make enhancements to the disk format in the future. Commodore-Amiga intends to provide compatibility within the trackdisk device. Anyone who uses this routine is bypassing this upward-compatibility and does so at their own risk.

|                              |  |
|------------------------------|--|
| diskReq->iotd_Req.io_Command | ETD_RAWREAD or TD_RAWREAD                        |
| diskReq->iotd_Req.io_Length  | number of bytes to read<br>(maximum size of 32K) |
| diskReq->iotd_Req.io_Data    | pointer to buffer (of io_Length bytes)           |
| diskReq->iotd_Req.io_Offset  | Track number (starting at 0)                     |
| diskReq->iotd_Req.io_Flags   | Set IOTDB_INDEX for index sync                   |

For the enhanced version of the request:

|                     |                     |
|---------------------|---------------------|
| diskReq->iotd_Count | Change count number |
|---------------------|---------------------|

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|

### ETD\_RAWWRITE and TD\_RAWWRITE

ETD\_RAWWRITE and TD\_RAWWRITE perform a raw write to a track on the disk. They seek to the specified track and write it from the user's buffer.

*No processing of the track is done.* It will be written exactly as the bits come out of the buffer -- typically in some legal flux format (such as MFM, FM, GCR; if you don't know what these are, you shouldn't be using this call). Caveat Programmer.

This interface is intended for sophisticated programming only. You must fully understand digital magnetic recording to be able to utilize this call. It is also important that you understand that the MFM encoding scheme used by the higher level trackdisk routines may change without notice. Thus, this routine is only really useful for encoding and writing other disk formats such as MS-DOS disks.

LIMITATIONS for sync'ed reads and writes: See TD\_RAWREAD.

## WARNING

Commodore-Amiga may make enhancements to the disk format in the future. Commodore-Amiga intends to provide compatibility within the trackdisk device. Anyone who uses this routine is bypassing this upward-compatibility and does so at their own risk.

|                              |   |
|------------------------------|---|
| diskReq->iotd_Req.io_Command | ETD_RAWWRITE or TD_RAWWRITE                       |
| diskReq->iotd_Req.io_Length  | number of bytes to write<br>(maximum size of 32K) |
| diskReq->iotd_Req.io_Data    | pointer to buffer (of io_Length bytes)            |
| diskReq->iotd_Req.io_Offset  | Track number (starting at 0)                      |
| diskReq->iotd_Req.io_Flags   | Set IOTDB_INDEX for index sync                    |

For the enhanced version of the request:

|                     |                     |
|---------------------|---------------------|
| diskReq->iotd_Count | Change count number |
|---------------------|---------------------|

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|

## Commands for Diagnostics and Repair

Currently TD\_SEEK and ETD\_SEEK are provided for internal diagnostics, disk repair, and head cleaning only.

### ETD\_SEEK and TD\_SEEK

TD\_SEEK and ETD\_SEEK will move the drive heads to the track specified. The io\_Offset field should be set to the (byte) offset to which the seek is to occur. TD\_SEEK and ETD\_SEEK do not verify their position until the next read. That is, they only move the heads; they do not actually read any data.

|                              |   |
|------------------------------|---|
| diskReq->iotd_Req.io_Command | ETD_SEEK or TD_SEEK   |
| diskReq->iotd_Req.io_Offset  | byte offset from start of disk<br>(Must be a multiple of TD_SECTOR * NUMSECS) |

For the enhanced version of the request:

|                     |                     |
|---------------------|---------------------|
| diskReq->iotd_Count | Change count number |
|---------------------|---------------------|

Result:

|                            |                                      |
|----------------------------|--------------------------------------|
| diskReq->iotd_Req.io_Error | Error return (see error table below) |
|----------------------------|--------------------------------------|

## Trackdisk Device Errors

Table 7-1 is a list of error codes that can be returned by the trackdisk device. When an error occurs, these error numbers will be returned in the `io_Error` field of your `IORequest` block.

Table 42-1: Trackdisk Device Error Codes

| Error Name           | Error Number | Meaning  |
|----------------------|--------------|--|
| TDERR_NotSpecified   | 20           | Error could not be determined                        |
| TDERR_NoSecHdr       | 21           | Could not find sector header                         |
| TDERR_BadSecPreamble | 22           | Error in sector preamble                             |
| TDERR_BadSecID       | 23           | Error in sector identifier                           |
| TDERR_BadHdrSum      | 24           | Header field has bad checksum                        |
| TDERR_BadSecSum      | 25           | Sector data field has bad checksum                   |
| TDERR_TooFewSecs     | 26           | Incorrect number of sectors on track                 |
| TDERR_BadSecHdr      | 27           | Unable to read sector header                         |
| TDERR_WriteProt      | 28           | Disk is write-protected                              |
| TDERR_DiskChanged    | 29           | Disk has been changed<br>or is not currently present |
| TDERR_SeekError      | 30           | While verifying seek position,<br>found seek error   |
| TDERR_NoMem          | 31           | Not enough memory to do this operation               |
| TDERR_BadUnitNum     | 32           | Bad unit number<br>(unit # not attached)             |
| TDERR_BadDriveType   | 33           | Bad drive type<br>(not an Amiga 3 1/2 inch disk)     |
| TDERR_DriveInUse     | 34           | Drive already in use<br>(only one task exclusive)    |
| TDERR_PostReset      | 35           | User hit reset; awaiting doom                        |

## Example Program

The following sample program makes a track-by-track copy of unit 0 onto unit 1.

```
/*
 * TrackDisk example code...
 *
 * This program does a track by track copy from DF0: to DF1:
 *
 * This program will only run from the CLI. If started from
 * the workbench, it will just exit...
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/trackdisk.h>
#include <libraries/dosextens.h>
```



```

#include <proto/all.h>

#include <stdio.h>

/* This prevents Lattice ctrl-C processing... */
int CXBRK(VOID) { return(0); }

#define TRACK_SIZE      ((LONG) (NUMSECS * TD_SECTOR))

/*
 * Turn the BUSY flag off/on for the drive
 * If onflag is TRUE, the disk will be marked as busy...
 *
 * This is to stop the validator from executing while
 * we are playing with the disks.
 */
VOID disk_busy(UBYTE *drive, LONG onflag)
{
    struct StandardPacket *pk;
    struct Process        *tsk;

    tsk=(struct Process *)FindTask(NULL);
    if (pk=AllocMem(sizeof(struct StandardPacket), MEMF_PUBLIC|MEMF_CLEAR))
    {
        pk->sp_Msg.mn_Node.ln_Name=(UBYTE *) &(pk->sp_Pkt);

        pk->sp_Pkt.dp_Link=&(pk->sp_Msg);
        pk->sp_Pkt.dp_Port=&(tsk->pr_MsgPort);
        pk->sp_Pkt.dp_Type=ACTION_INHIBIT;
        pk->sp_Pkt.dp_Arg1=(onflag ? -1L : 0L);

        PutMsg(DeviceProc(drive), (struct Message *)pk);
        WaitPort(&(tsk->pr_MsgPort));
        GetMsg(&(tsk->pr_MsgPort));
        FreeMem(pk, (long)sizeof(*pk));
    }
}

/*
 * This turns the motor off
 */
VOID Motor_Off(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Length=0;
    disk->iotd_Req.io_Command=TD_MOTOR;
    DoIO((struct IORequest *)disk);
}

/*
 * This turns the motor on
 */
VOID Motor_On(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Length=1;
    disk->iotd_Req.io_Command=TD_MOTOR;
    DoIO((struct IORequest *)disk);
}

/*
 * This reads a track, reporting any errors...
 */
SHORT Read_Track(struct IOExtTD *disk, UBYTE *buffer, SHORT track)
{
    SHORT All_OK=TRUE;

    disk->iotd_Req.io_Length=TRACK_SIZE;
    disk->iotd_Req.io_Data=(APTR)buffer;
    disk->iotd_Req.io_Command=CMD_READ;
    disk->iotd_Req.io_Offset=(ULONG) (TRACK_SIZE * track);
    DoIO((struct IORequest *)disk);
    if (disk->iotd_Req.io_Error)
    {

```

```

        All_OK=FALSE;
        printf("Error %u when reading track %d",disk->iotd_Req.io_Error,track);
    }
    return(All_OK);
}

/*
 * This writes a track, reporting any errors...
 */
SHORT Write_Track(struct IOExtTD *disk,UBYTE *buffer,SHORT track)
{
    SHORT All_OK=TRUE;

    disk->iotd_Req.io_Length=TRACK_SIZE;
    disk->iotd_Req.io_Data=(APTR)buffer;
    disk->iotd_Req.io_Command=TD_FORMAT;
    disk->iotd_Req.io_Offset=(ULONG)(TRACK_SIZE * track);
    DoIO((struct IORequest *)disk);
    if (disk->iotd_Req.io_Error)
    {
        All_OK=FALSE;
        printf("Error %d when writing track %d",disk->iotd_Req.io_Error,track);
    }
    return(All_OK);
}

/*
 * This function finds the number of TRACKS on the device.
 * NOTE That this is TRACKS and not cylinders. On a Two-Head
 * drive (such as the standard 3.5" drives) the number of tracks
 * is 160, 80 cylinders, 2-heads.
 */
SHORT FindNumTracks(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Command=TD_GETNUMTRACKS;
    DoIO((struct IORequest *)disk);
    return((SHORT)disk->iotd_Req.io_Actual);
}

/*
 * This routine allocates the memory for one track and does
 * the copy loop.
 */
VOID Do_Copy(struct IOExtTD *diskreq0,struct IOExtTD *diskreq1)
{
    UBYTE *buffer;
    SHORT track;
    SHORT All_OK;
    SHORT NumTracks;

    if (buffer=AllocMem(TRACK_SIZE,MEMF_CHIP|MEMF_PUBLIC))
    {
        printf(" Starting Motors\r");
        Motor_On(diskreq0);
        Motor_On(diskreq1);
        All_OK=TRUE;

        NumTracks=FindNumTracks(diskreq0);

        for (track=0;(track<NumTracks) && All_OK;track++)
        {
            printf(" Reading track %d\r",track);

            if (All_OK=Read_Track(diskreq0,buffer,track))
            {
                printf(" Writing track %d\r",track);

                All_OK=Write_Track(diskreq1,buffer,track);
            }
        }
        if (All_OK) printf(" * Copy complete *");
        printf("\n");
    }
}

```

```

        Motor_Off(diskreq0);
        Motor_Off(diskreq1);
    }
    else printf("No memory for track buffer...\n");
}

/*
 * Prompts the user to remove one of the disks.
 * Since this program makes an EXACT copy of the disks
 * AmigaDOS would get confused by them so one must be removed
 * before the validator is let loose. Also, note that the
 * disks may NEVER be in drives on the SAME computer at the
 * SAME time unless one of the disks is renamed. This is due
 * to a bug in the system. It would normally be prevented
 * by a diskcopy program that knew the disk format and modified
 * the creation date by one clock-tick such that the disks would
 * be different.
 */
VOID Remove_Disks(VOID)
{
    printf("\nYou *MUST* remove at least one of the disks now.\n");
    printf("\nPress RETURN when ready\n");
    while(getchar() != '\n');
}

/*
 * Prompts the user to insert the disks.
 */
VOID Insert_Disks(VOID)
{
    printf("\nPlease insert source disk in DF0:");
    printf("\n          and destination in DF1:\n");
    printf("\nPress RETURN when ready\n");
    while(getchar() != '\n');
}

/*
 * Open the devices and mark them as busy
 */
VOID Do_OpenDevice(struct IOExtTD *diskreq0, struct IOExtTD *diskreq1)
{
    if (!OpenDevice(TD_NAME, 0L, (struct IORequest *)diskreq0, 0L))
    {
        disk_busy("DF0:", TRUE);

        if (!OpenDevice(TD_NAME, 1L, (struct IORequest *)diskreq1, 0L))
        {
            disk_busy("DF1:", TRUE);

            Insert_Disks();
            Do_Copy(diskreq0, diskreq1);
            Remove_Disks();

            disk_busy("DF1:", FALSE);
            CloseDevice((struct IORequest *)diskreq1);
        }
        else printf("Could not open DF1:\n");

        disk_busy("DF0:", FALSE);
        CloseDevice((struct IORequest *)diskreq0);
    }
    else printf("Could not open DF0:\n");
}

VOID main(int argc, char *argv[])
{
    struct IOExtTD *diskreq0;
    struct IOExtTD *diskreq1;
    struct MsgPort *diskPort;

    if (argc) /* Check if started from the CLI... */
    {

```

```

if (diskPort=CreatePort(NULL,NULL))
{
    if (diskreq0=(struct IOExtTD *)CreateExtIO(diskPort,
                                                sizeof(struct IOExtTD)))
    {
        if (diskreq1=(struct IOExtTD *)CreateExtIO(diskPort,
                                                    sizeof(struct IOExtTD)))
        {
            Do_OpenDevice(diskreq0,diskreq1);
            DeleteExtIO((struct IORequest *)diskreq1);
        }
        else printf("Out of memory\n");
        DeleteExtIO((struct IORequest *)diskreq0);
    }
    else printf("Out of memory\n");
    DeletePort(diskPort);
}
else printf("Could not create diskReq port\n");
}

```

### NOTE

Since this example program makes an exact track-for-track duplicate, AmigaDOS will get confused if both disks are in drives on the system at the same time. While the disks are inhibited, this does not cause a problem, but during normal operation, this will cause a system hang. To prevent this, you can relabel one of the disks. A commercial diskcopy program would have to understand the disk format and either relabel the disk or modify the volume creation date/time by a bit in order to make the disks look different to the system.



# Chapter 43

## Resources

### Introduction

This section contains a description of the system resource routines. “Resources” refers to the Amiga’s low-level hardware control functions. Most applications will never need to use Amiga hardware at the resource level — the Amiga’s device interface is much more convenient and provides for multi-tasking. However, some high performance applications, such as MIDI time stamping, require direct access to Amiga hardware registers.

In order to get direct access to the hardware in a way that is compatible with multi-tasking, you use the system resource functions. These routines let you temporarily bar other tasks from using the resource. You may then use the associated hardware directly for your special purposes. Return the resource back to the system for other tasks to use when you are finished with it.

There are currently four standard resources in the Amiga system; `disk.resource`, `cia.resource`, `misc.resource` and `potgo.resource`:

#### **`disk.resource`**

grants temporary exclusive access to the disk hardware.

**cia.resource**

grants access to the interrupts and timer bits of the 8520 CIA (Complex Interface Adapter) chips.

**misc.resource**

grants exclusive access to functional blocks of chip registers. At present, definitions have been made for the serial and parallel hardware only.

**potgo.resource**

manages the bits of the POTGO and POTINP registers.

See the *Amiga Hardware Reference Manual* for detailed information on the actual hardware involved. This section covers how to properly arbitrate for the hardware.

**WARNING**

Resources are just one step above direct hardware manipulation. You are advised to try the higher level device and library approach before resorting to the hardware.

## Disk Resource

Whenever using floppy disk hardware, it must be acquired from the disk resource. There are up to four possible disk/MFM units available. The disk resource provides both a gross and a fine unit allocation scheme. **AllocUnit()** and **FreeUnit()** are used to claim a unit for long term use, and **GetUnit()** and **GiveUnit()** are used to claim a unit for shorter periods.

The trackdisk.device uses and abides by both allocation schemes. Because a trackdisk unit is never closed for Amiga 3.5" drives (the file system keeps them open) the associated resource units will always be allocated for these drives. **GetUnit()** and **GiveUnit()** can still be used, however, by other applications that have not succeeded with **AllocUnit()**.

It is therefore possible to prevent the trackdisk device from using units that have not yet been mounted by successfully performing an **AllocUnit()** for that unit. It is also possible to starve trackdisk usage by performing a **GetUnit()**. The appropriate companion routine (**FreeUnit()** or **GiveUnit()**) should be called to restore the resource at the end of its use.

## CIA Resource

For high performance timing applications such as MIDI time stamping or SMPTE time coding, the CIA resource should be used. The CIA-B timers are available to be allocated and are associated with interrupt 6. The timer is therefore allocated by successfully adding a vector for interrupt 6 with **AddICRVector()**. Ownership of the interrupt bit means you own the related timer or chip function. **SetICR()** and **AbleICR()** are used to read and write to the interrupt hardware registers. **RemICRVector()** is used to remove your vector when your program is finished.

**NOTE**

You should not use the CIA-A timers. These are reserved for the system. The table below summarizes system use of the CIA timers:

## Amiga CIA Timer Allocation

### CIAA (int 2)

|        |                             |
|--------|-----------------------------|
| timerA | Used for keyboard handshake |
| timerB | Used for uSec timer.device  |
| TOD    | Used for 60Hz timer.device  |

### CIAB (int 6)

|        |  |
|--------|--|
| timerA | Not used                               |
| timerB | Not used                               |
| TOD    | Used for graphics.library beam counter |

Your program should first try to allocate CIA-B, timer B. If it is not available then try to allocate CIA-B, timer A. If neither timer can be allocated, inform the user and abort the operation. The `cia.resource` provides the name of the interrupt owner in that case.

## Misc Resource

Before using serial or parallel port hardware, it first must be acquired from the `misc.resource`. The `misc.resource` oversees usage of the serial data port, the serial communication bits, the parallel data and handshake port, and the parallel communication bits. The parallel communication bits double as the Commodore serial bus interface bits for those who want to connect a 1541 to the Amiga.

The `misc.resource` consists of two routines, `MR_GETMISCRESOURCE` and `MR_FREEMISCRESOURCE`. Since these two routines do not have a library base pointer name, they can only be called from an assembly language program. See the include file `resources/misc.i` for a description of what ownership rights each unit grants. Here's a working example that gets one of the two miscellaneous resources:

```
* OPT L+
*
* Assembly language fragment that grabs the two parts of the serial
* resource (using misc.resource). If it gets the resource, it will
* wait for CTRL-C to be pressed before releasing.
*
* When a task has successfully obtained the serial resource, it "owns"
* the hardware registers that control the serial port. No other tasks
* are allowed to interfere.
*
* This example must be linked with "amiga.lib"
*
* Wednesday 07-Dec-88 19:35:13
*
*          INCDIR "inc:"
*          INCLUDE "exec/types.i"
*          INCLUDE "resources/misc.i"
*          INCLUDE "libraries/dos.i"

_AbsExecBase    EQU 4

JSRLIB  MACRO                                ;Macro for easy use of system functions
XREF    _LVO\1
JSR     _LVO\1(A6)
ENDM

;
; Open Exec and the misc.resource, check for success
;
        move.l _AbsExecBase,a6 ;Prepare to use exec
```



```

        lea.l    MiscName(pc),a1
        JSRLIB   OpenResource    ;Open "misc.resource"
        move.l   d0,d7           ;Stash resource base
        bne.s    resource_ok
        moveq     #RETURN_FAIL,d0
        rts

resource_ok    exg.l    d7,a6            ;Put resource base in A6

;
; We now have a pointer to a resource.
; Call one of the resource's library-like vectors.
;
        move.l   #MR_SERIALBITS,d0    ;We want these bits
        lea.l    MyName(pc),a1        ;This is our name
        jsr      MR_ALLOCMISCRESOURCE(a6)
        tst.l    d0
        bne.s    no_bits              ;Someone else has it...
        move.l   #MR_SERIALPORT,d0
        lea.l    MyName(pc),a1
        jsr      MR_ALLOCMISCRESOURCE(a6)
        tst.l    d0
        bne.s    no_port              ;Someone else has it...

;
; We just stole the serial port registers; wait.
; Nobody else can use the serial port, including the serial.device!
;
        exg.l    d7,a6                ;use exec again
        move.l   #SIGBREAKF_CTRL_C,d0
        JSRLIB   Wait                 ;Wait for CTRL-C
        exg.l    d7,a6                ;Get resource base back

;
; Free 'em up
;
        move.l   #MR_SERIALPORT,d0
        jsr      MR_FREEMISCRESOURCE(a6)
no_port
        move.l   #MR_SERIALBITS,d0
        jsr      MR_FREEMISCRESOURCE(a6)
no_bits
        moveq     #RETURN_FAIL,d0
        rts

;
; Text area
;
MiscName    dc.b    'misc.resource',0
MyName      dc.b    'Serial Port hog',0
            dc.w    0
            END

```

#### NOTE

There are two serial.device resources to take over, MR\_SERIALBITS and MR\_SERIALPORT. You should get both resources when you take over the serial port to prevent other tasks from using them. The parallel.device also has two resources to take over. See the *resources/misc.h* include file for the relevant C definitions and structures.

Under V1.3 and earlier versions of the Amiga system software the MR\_GETMISCRESOURCE routine will always fail if the serial.device has been used at all by another task — even if that task has finished using the resource. In other words, once a printer driver or communication package has been activated, it will keep the associated resource locked up preventing your task from using it. Under these conditions, you must get the resource back from the system yourself.

You do this by calling the function `FlushDevice()`:

```
/*
 * A safe way to expunge ONLY a certain device. The serial.device holds
 * on to the misc serial resource until a general expunge occurs.
 * This code attempts to flush ONLY the named device out of memory and
 * nothing else. If it fails, no status is returned since it would have
 * no valid use after the Permit().
 */
#include <exec/types.h>
#include <exec/execbase.h>
#include <proto/all.h>

void FlushDevice(char *);

extern struct ExecBase *SysBase;

void FlushDevice(char *name)
{
    struct Device *devpoint;

    Forbid();
    if( devpoint=(struct Device *)FindName(&SysBase->DeviceList,name) )
        RemDevice(devpoint);
    Permit();
}
```

## POTGO Resource

The POTGO resource is used to get control of the hardware POTGO register connected to the proportional IO pins on the game controller ports. There are two registers, POTGO (write-only) and POTINP (read-only). These pins could also be used for digital IO. Intuition uses POTGO1 for reading the right and (optional) middle mouse buttons.

The resource consists of 3 functions `AllocPotBits()`, `FreePotBits()` and `WritePotgo()`. The example program shown below demonstrates how to use the POTGO resource to track mouse button presses on port 1.

```
/* An example of using the potgo.resource to read pins 9 and 5 of
 * port 1 (the non-mouse port). This bypasses the gameport.device.
 * When the right button on a mouse plugged into port 1 is pressed,
 * the read value will change.
 *
 * Use of port 0 (mouse) is unaffected.
 *
 * Lattice use lc -bl -cfist -v -y. Link with amiga.lib and lc.lib.
 */
#include <exec/types.h>
#include <libraries/dos.h>
#include <proto/all.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) {return(0);} /* Disable Lattice Ctrl-C checking */
#endif

struct PotgoBase *PotgoBase;
ULONG potbits;
UWORD value;

#define UNLESS(x) if(!(x))
#define UNTIL(x) while(!(x))

#define OUTRY 1L<<15
```

```

#define DATRY 1L<<14
#define OUTRX 1L<<13
#define DATRX 1L<<12

void main(int argc, char **argv)
{
    UNLESS(PotgoBase=(struct PotgoBase *)OpenResource("potgo.resource"))
        return;
    printf("PotgoBase is at %lx\n",PotgoBase);

    potbits=AllocPotBits(OUTRY|DATRY|OUTRX|DATRX);
    /* Get the bits for the right and middle mouse buttons
       on the alternate mouse port. */

    if(potbits != (OUTRY|DATRY|OUTRX|DATRX))
    {
        printf("Pot bits are already allocated! %lx\n",potbits);
        FreePotBits(potbits);
        return;
    }

    WritePotgo(0xFFFFFFFFL,potbits);
    /* Set all ones in the register (masked by potbits) */

    UNTIL(SIGBREAKF_CTRL_C & SetSignal(0L,0L))
        /* until CTRL-C is pressed */
        {
            value=(UWORD *)0x00DFF016;
            /* Read word at $DFF016 */
            printf("POTINP = %lx\n",value & potbits);
            /* Show what was read (restricted to our allocated bits) */
        }

    FreePotBits(potbits);
}

```

# Appendix A

## Troubleshooting Your Software

Many Amiga programming errors have classic symptoms. This guide will help you to eliminate or avoid these problems in your software.

### Audio - Corrupted Samples

The bit data for audio samples **MUST** be in CHIP RAM. Check your compiler manual for directives or flags which will place your audio sample data in CHIP RAM. Or dynamically allocate CHIP RAM and copy or load the audio sample there.

### Character Input/Output Problems

RAWKEY users must be aware that RAWKEY codes can be different letters or symbols on national keyboards. If you need to use RAWKEY, run the codes through RawKeyConvert (see "Intuition" manual chapters) to get proper translation to correct ASCII codes. Improper display or processing of high-ASCII international characters can be caused by incorrect `tolower()`/`toupper()`, or by sign extension of character values when switched on or assigned into larger size variables. Use unsigned variables such as `UBYTE` (not `char`) for strings and characters whenever possible.

### CLI Error Messages

This is caused by calling `exit(n)` with an invalid or missing return value `n`. Assembler programmers using startup code should jump to the startup code's `_exit` with a valid return value on the stack. Programs without startup code should return with a valid value in `D0`. Valid return values are defined in `libraries/dos.h` and `i`. Other values (-1 for instance) can cause CLI error messages such as "not an object module". Useful hint - if your program is called from a script, your valid return value can be conditionally branched on in the script (ie.

call program, then perform actions based on IF WARN or IF NOT WARN). RETURN\_FAIL will cause the script to stop if a normal FAILAT value is being used in script.

### **CLI Won't Close on RUN**

A CLI can't close if a program has a Lock on the CLI input or output stream ("\*"). If your program is RUN >NIL: from a CLI, that CLI should be able to close unless your code or your compiler's startup code explicitly opens "\*".

### **Crashes and Memory Corruption**

Memory corruption, address errors, and illegal instruction errors are generally caused by use of an uninitialized, incorrectly initialized, or already freed/closed pointer or memory. You may be using the pointer directly, or it may be one that you placed (or forgot to place) in a structure passed to system calls. Or you may be overwriting one of your arrays, or accidentally modifying or incrementing a pointer later used in a free/close. Be sure to test the return of all open/allocation type functions before using the result, and only close/free things that you successfully opened/allocated. Use watchdog/torture utilities such as MemWatch and MemMung to catch use of uninitialized pointers or freed memory. (MemMung sets freed memory areas and location \$0 to an odd value). You may also be overflowing your stack - your compiler's stack checking option may be able to catch this. Cut stack usage by dynamically allocating large structures, buffers, and arrays which are currently defined inside your functions.

Corruption or crashes can also be caused by passing wrong or missing arguments to a system call (for example SetAPen(3) or SetAPen(win,3), instead of SetAPen(rp,3)). If using short integers be sure to explicitly type long constants as long (eg. 42L). (For example, with short ints,  $1 << 17$  may become zero). If corruption is occurring during exit, use printf (or kprintf, etc.) with Delay(n) to slow down your cleanup and broadcast each step. See exec/alerts.h for Amiga-specific alert numbers. Also see "Crashes - After Exit".

### **Crashes - After Exit**

If this only happens when you start your program from Workbench, then you are probably UnLocking one of the WBSStartup message wa\_Locks, or UnLocking the Lock returned from an initial CurrentDir() call. If you CurrentDir(), save the lock returned initially, and CurrentDir() back to it before you exit.

If you are crashing from both Workbench and CLI, and you are only crashing AFTER exit, then you are probably either freeing/closing something twice, or freeing/closing something you did not actually allocate/open, OR you are leaving an outstanding device IO request or other wakeup request. You must abort and WaitIO any outstanding IO requests before you free things and exit (see the autodocs for your device, and for Exec AbortIO and WaitIO). Similar problems can be caused by deleting a subtask that might be in a WaitTOF(). Only delete subtasks when you are sure they are in a safe state such as Wait(OL);

### **Crashes - Subtasks, Interrupts**

If part of your code runs on a different stack or the system stack, you must turn off compiler stack-checking options. If part of your code is called directly by the system or by other tasks, you must use long code/long data OR use special compiler flags or options to assure that the correct base registers are set up for your subtask or interrupt code.

### **Crashes - Window Related**

Be careful not to CloseWindow() a window during a while(msg=GetMsg(...)) loop on that window's port (next GetMsg would be on freed pointer). Also, use ModifyIDCMP(NULL) with care, especially if using one port with multiple windows. Be sure to ClearMenuStrip() any menus before closing a window, and do not free items such as dynamically allocated gadgets and menus while they are attached to a window.

### **Crashes - Workbench Only**

If your program crashes during execution or during your exit procedure only when started from WB, and your startup opens no stdio window or NIL: file handles for WB programs, then make sure you are not writing anything to stdout (ie. printf, etc.) when started from WB (argc==0). See also "Crashes - After Exit".

### **Disk Icon Won't Go Away**

This occurs when a program has a Lock or leaves a Lock on one or more of a disk's files. A memory loss of exactly 24 bytes is usually Lock which has not been UnLocked.

### **Fails only on 68020/30**

The following programming practices can be the cause of this problem: using upper byte of addresses as flags; doing signed math on addresses; self-modifying code; using the MOVE SR assembler instruction (use Exec GetCC() instead); software delay loops; assumptions about the order in which asynchronous tasks will finish. The following differences in 68020/30 can cause problems: invalid cache entry due to DMA or other non-processor modification of the data's actual location; different exception stack frame; interrupt autovectors may be moved by VBR; 68020/30 CLR instruction does a single write access as opposed to 68000 CLR instruction's read then write access (use MOVE instead).

### **Fails only on 68000**

The following programming practices can be the cause of this problem: software delay loops; word or longword access of an odd address (illegal on the 68000); use of the assembler CLR instruction on a hardware register which is triggered by any access. 68000 CLR instruction performs two accesses (read and write) while 68020/30 CLR does a single write access. Use MOVE instead; assumptions about the order in which asynchronous tasks will finish; use of compiler flags which have generated inline 68881/68882 math coprocessor instructions or 68020/30 specific code.

### **Fails only on Older ROMs or Newer WB**

This can be caused by asking for a library version higher than you need (DO NOT use the #define LIBRARY\_VERSION!!!!). Can also be caused by calling functions or using structures which do not exist in the older version of the operating system. Ask for the lowest version which provides the functions you need (usually 33), and exit gracefully and informatively if an OpenLibrary fails.

### **Fails only on Newer ROMs or Newer WB**

This should not happen with proper programming. Possible causes: running too close to your stack limits or the memory limits of a base machine (newer versions of the operating system may use slightly more stack in system calls, and usually use more free memory); using system functions improperly; not testing function return values; using improperly initialized pointers; trashing memory; assuming something (such as a flag) is B if it is not A; failing to initialize formerly reserved structure fields to zero; violating Amiga programming guidelines (for example: depending on or poking private system structures, jumping into ROM, depending on undocumented or unsupported behaviors); failure to read the function autodocs.

### **Fails only on CHIP-RAM-Only Machines**

Caused by specifically asking for or requiring MEMF\_FAST memory. If you don't need CHIP RAM, ask for memory type 0L, or MEMF\_CLEAR, or MEMF\_PUBLIC\MEMF\_CLEAR as applicable. If there is FAST memory available, you will be given FAST memory. If not, you will get CHIP RAM.

### **Fails only on machines with FAST RAM**

Data and buffers which will be accessed directly by the custom chips MUST be in CHIP RAM. This includes bitplanes (use OpenScreen() or AllocRaster()), audio samples, trackdisk buffers, and the graphic image data for sprites, pointers, bobs, images, gadgets, etc. Use compiler or linker flags to force CHIP RAM loading of any initialized data needing to be in CHIP RAM, or dynamically allocate CHIP RAM and copy any initialization data there.

### **Fails only with Enhanced Chips**

Usually caused by writing or reading addresses past the end of older custom chips, or writing something other than 0 (zero) to bits which are undefined in older chip registers, or failing to mask out undefined bits when interpreting the value read from a chip register. See also "Fails only on CHIP-RAM-Only Machines".

### **Fireworks**

A dazzling pyrotechnic video display is caused by trashing or freeing a copper list which is in use, or trashing the pointers to the copper list. If you aren't messing with copper lists, see "Crashes and Memory Corruption".

### **Graphics - Corrupted Images**

The bit data for graphic images such as sprites, pointers, bobs, and gadgets MUST be in CHIP RAM. Check your compiler manual for directives or flags which will place your graphic image data in CHIP RAM. Or dynamically allocate CHIP RAM and copy them there.

### **Hang - One Program Only**

Program hangs are generally caused by Waiting on the wrong signal bits, wrong port, wrong message, or by never having actually asked for what you are waiting for. They can also be caused by Verify deadlocks. Be sure to turn off all Intuition VERIFY messages (such as MENUVERIFY) before calling AutoRequest() or doing disk access.

### **Hang - Whole System**

This is generally caused by a Disable() without a corresponding Enable(). It can also be caused by memory corruption, especially corruption of low memory. See "Crashes and Memory Corruption".

### **Memory Loss**

First determine that your program is actually causing a memory loss. Boot with a normal Workbench disk whose s:startup-sequence LoadWB line has been changed to "LoadWB -debug". It is important to boot with a standard Workbench because a number of third party items such as some background utilities, shells, and network handlers dynamically allocate and free pieces of memory. Arrange all necessary shell or other windows so that part of the backdrop window is accessible, and so that no window rearrangement will be needed to run your program.

Select "flushlibs" from the rightmost Workbench menu. This will flush all non-open disk-loaded fonts, devices, etc. from memory. Wait a few seconds, then click on the Workbench backdrop and write down the amount of free memory. Now without rearranging any windows, start your program and use all of your program features. Exit your program, wait a few seconds, then click on the Workbench backdrop and write down the Free value. Now select "flushlibs", wait a few seconds, then write down this final Free amount. If this matches the first value you wrote down, then your program is fine, and is not causing a memory loss.

If memory was actually lost, and your program can be run from CLI or Workbench, then try the above procedure with both methods of starting your program. See "Memory Loss - CLI Only" and "Memory Loss - WB Only" if appropriate. If you lose memory from both WB and CLI, then check all of the open/alloc/get/create/lock type calls in your code, and make sure that there is a matching close/free/delete/unlock type call for each of them (note - there are a few system calls that have or require no corresponding free - check the autodocs). Generally, the close/free/delete/unlock calls should be in opposite order of the allocations.

If you are losing a fixed small amount of memory, look for a structure of that size in the Structure Offsets listing in the "Includes and Autodocs" manual. For example, a loss of exactly 24 bytes is probably a Lock which has not been UnLocked. If you are using ScrollRaster(), be aware that ScrollRaster() left or right in a Superbitmap window with no TmpRas will currently lose memory (workaround - attach a TmpRas). If you lose much more memory when started from Workbench, make sure your program is not using Exit(n). This would bypass startup code cleanups and prevent a Workbench-loaded program from being unloaded. Use exit(n) instead.

### **Memory Loss - CLI Only**

Make sure you are testing in a standard environment. Some third-party shells dynamically allocate history buffers, or cause other memory fluctuations. Also, if your program executes different code when started from CLI, check that code and its cleanup. And check your startup.asm if you wrote your own.

**Memory Loss - CTRL-C Exit Only**

You have Amiga-specific resources opened or allocated and you have not disabled your compiler's automatic CTRL-C handling (causing all of YOUR program cleanups to be skipped). Disable compiler CTRL-C handling and handle CTRL-C (SIGBREAKF\_CTRL\_C) yourself.

**Memory Loss - During Execution**

A continuing memory loss during execution can be caused by failure to keep up with voluminous IDCMP messages such as MOUSEMOVE messages. Intuition can not re-use IDCMP message blocks until you ReplyMsg() them. If your window's allotted message blocks are all in use, new sets will be allocated and not freed till the window is closed. Continuing memory losses can also be caused by a program loop containing an allocation-type call without a corresponding free.

**Memory Loss - Workbench Only**

Commonly, this is caused by a failure of your code to unload after you exit. Make sure that your code is being linked with a standard correct startup module, and do NOT use the Exit(n) function to exit your program. This function will bypass your startup code's cleanup, including its ReplyMsg() of the WBStartup message (which would signal Workbench to unload your program from memory). You should exit via either exit(n) where n is a valid DOS error code such as RETURN\_OK (dos/libraries.h), or via final "]" or return. Assembler programmers using startup code can JMP to \_exit with a long return value on stack, or RTS.

**Menu Problems**

Flashing while browsing is caused by leaving a pixel or more space between menu subitems when designing your menu. Crashing after browsing a menu (looking at menu without selecting any items) is caused by not properly handling MENUNULL select messages. Multiple selection not working is caused by not handling NextSelect properly. See the Intuition "Menus" chapter.

**Out-of-Sync Response to Input**

Caused by failing to handle all received signals, or all possible messages on a wakeup. More than one event or message may have caused your program to awakened. Check the signals returned by Wait and act on every one that is set. At ports which may have more than one message (for instance, a window's IDCMP port), you must handle the messages in a while(msg=GetMsg(...)) loop.

**Performance Loss in Other Processes**

This is often caused by a one program doing one or more of the following: busy waiting or polling; running at a higher priority; doing lengthy Forbids or Disables.

**Trackdisk Data not Transferred**

Make sure your trackdisk buffers are in CHIP RAM.

**Windows - Borders Flicker after Resize**

Set the NOCAREREFRESH flag. Even SMART\_REFRESH windows may generate refresh events if there is a sizing gadget. If you don't have specific code to handle this, you must set the NOCAREREFRESH flag. If you do have refresh code, be sure to use the Begin/EndRefresh() calls. Failure to do one or the other will leave Intuition in an intermediate state, and slow down operation for all windows on the screen.



## GENERAL DEBUGGING TECHNIQUES

### Narrow the search

Use methodical testing procedures, and debugging messages if necessary, to locate the problem area. Low level code can be debugged using `kprintf` serial (or `dprintf` parallel) messages (see Linker Library documentation). Check the initial values, allocation, use, and freeing of all pointers and structures used in the problem area. Check that all of your system and internal function calls pass correct initialized arguments, and that all possible error returns are checked for and handled.

### Isolate the problem

If errors can not be found, simplify your code to the smallest possible example that demonstrates the problem. Often you will find that this smallest example will not have the problem. If so, add back the other features of your code until the problem reappears, then debug that section.

### Use debugging tools

A variety of debugging tools are available to help locate faulty code. Some of these are source level and other debuggers, crash interceptors, vital memory watchdogs like MemWatch and WatchMem, and free memory invalidation tools such as MemMung.

## A FINAL WORD ABOUT TESTING

Test your program with memory watchdog and invalidation tools on a wide variety of systems and configurations. Programs with coding errors may appear to work properly on one or more configurations, but may fail or cause fatal problems on another. Make sure that your code is tested on both a 68000 and a 68020/30, on machines with and without FAST RAM, and on machines with and without enhanced chips. Test all of your program functions on every machine.

Test all error and abort code. A program with missing error checks or unsafe cleanup might work fine when all of the items it opens or allocates are available, but may fail fatally when an error or problem is encountered. Try your code with missing files, filenames with spaces, incorrect filenames, cancelled requesters, CTRL-C, missing libraries or devices, low memory, missing hardware, etc.

Test all of your text input functions with high-ASCII characters (such as the character produced by pressing ALT-F then "A"). Note that RAWKEY codes can be different keyboard characters on national keyboards (higher levels of keyboard input are automatically translated to the proper characters). If your program will be distributed internationally, support and take advantage of the additional screen lines available on a PAL system. Enhanced Agnus chip A2000's may be switched to PAL via motherboard jumper J102. Note that a base PAL machine will have less memory free due to the larger display size.

Write good code. Test it. Then make it great.

# Appendix B

## Linker Libraries

### Introduction

This section describes the *amiga.lib* and *debug.lib* libraries. Unlike the libraries described in the other chapters of the manual, these are not shared run-time libraries. Code from the Linker Libraries is inserted by the linker into your final program. These libraries are typically supplied by your language or compiler vendor. Only the functions you use are pulled into your code.

The libraries described here are:

## AMIGA.LIB

This is the main Amiga scanned linker library, generally linked with most programs for the Amiga. The major components of `amiga.lib` are:

|                     |   |
|---------------------|---|
| <i>stubs</i>        | Individual interface stubs for each Amiga ROM routine that enable stack-based C compilers to call register-based Amiga ROM routines. Some compilers have a feature that bypasses the stubs for smaller code size. |
| <i>offsets</i>      | The negative offset from the library base for each Amiga function. These are called <i>Library Vector Offsets</i> ( <code>_LVO</code> ). Some assemblers provide a faster way to obtain the same offsets.         |
| <i>exec_support</i> | C functions which simplify many exec procedures such as the creation and deletion of tasks, ports, and IO request structures.   |
| <i>clib</i>         | C support functions including pseudo-random number generation and a limited set of file and stdio functions designed to work directly with AmigaDOS file handles.   |
| <i>other</i>        | Miscellaneous handy functions, callable from any language.  |

## DEBUG.LIB

Contains standard I/O (stdio) style functions for communicating with a serial terminal connected to the Amiga via its built-in serial port. Typically this terminal will be a 9600-baud, 8 data-bits, one stop-bit connection to an external terminal or an Amiga running a terminal package. The `debug.lib` functions allow you to output messages and prompt for input, even from within low level task or interrupt code, without disturbing the Amiga's display and or current state (other than the state of the serial hardware itself). No matter how badly the system may have crashed, these functions can usually get a message out. A similar debugging library (currently called *ddebug.lib* is available for sending debugging output to the parallel port. This is useful for debugging serial applications. `Ddebug.lib` is not documented here. It contains functions similar to `debug.lib` but with names starting with 'D' instead of 'K'.

Please refer to the *ROM Kernel Reference Manual: Includes and Autodocs* for a detailed description of the functions.

# Amiga.lib

Amiga.lib has the following functions:

## EXEC\_SUPPORT

### AddTOF() and RemTOF()

**AddTOF()** adds a task to the vertical-blanking interval interrupt server chain. This frees C programmers from the burden of having to write an assembly language stub to perform this function. The task can be removed with **RemTOF()**.

### BeginIO()

This function takes an **IORequest** and passes it directly to the **BEGINIO** vector of the proper device. This works exactly like **SendIO()**, but does not clear the **io\_Flags** field first. This function does not wait for the I/O to complete.

### CreateExtIO() and DeleteExtIO()

**CreateExtIO()** allocates memory for and initializes a new IO request block of a user-specified number of bytes. The number of bytes *must* be the size of a legal **IORequest** (or extended request) or very nasty things will happen. **DeleteExtIO()** frees up an IO request as allocated by **CreateExtIO()**. The **mn\_Length** field determines how much memory to deallocate.

### CreatePort() and DeletePort()

**CreatePort()** allocates and initializes a new message port. The message list of the new port will be prepared for use via **NewList()**. The port will be set to signal your task when a message arrives (**PA\_SIGNAL**). **DeletePort()** deletes the port created by **CreatePort()**. All messages that may have been attached to that port must already have been replied to.

### CreateTask() and DeleteTask()

These functions simplify creation and deletion of subtasks by dynamically allocating and initializing the required structures and stack space. They also add the task to Exec's task list with the given name and priority. A **tc\_MemEntry** list is provided so that all stack and structure memory allocated by **CreateTask()** is automatically deallocated when the task is removed. Before deleting a task with **DeleteTask()**, you must first make sure that the task is not currently executing any system code which might try to signal the task after it is gone.

### NewList()

Prepares a List structure for use; the list will be empty and ready to use.

## CLIB

### FastRand()

Generates a pseudo-random number. The seed value is taken from stack, shifted left one position, exclusive-or'ed with hex value \$1D872B41 and returned.

**RangeRand()**

**RangeRand()** accepts a value from 1 to 65535, and returns a value within that range (a 16-bit integer). Note that this function is implemented in C.

**fclose()**

Closes a file.

**fgetc()**

Gets a character from a file.

**fprintf()**

Prints a formatted output line to a file.

**fputc()**

Puts character to file.

**fputs()**

Writes a string to file.

**getchar()**

Gets a character from stdin.

**printf()**

Puts format data to stdout.

**putchar()**

Puts character to stdout.

**Bfputs()**

Puts a string to stdout, followed by newline.

**sprintf()**

Formats data into a string (see `exec.library/RawDoFmt`).

**OTHER****afp()**

Converts ASCII string variable into fast floating-point.

**arnd()**

ASCII round-off of the provided floating-point string.

**dbf()**

Accepts a dual-binary format floating-point number and converts it to an FFP format floating-point number.

**fpa()**

Accepts an FFP number and the address of the ASCII string where its converted output is to be stored. The number is converted to a NULL terminated ASCII string and stored at the address provided. Additionally, the base ten (10) exponent in binary form is returned.

**fpbcd()**

Accepts a floating-point number and the address where the converted BCD data is to be stored. The FFP number is converted and stored at the specified address in an ASCII form.

## **Debug.lib**

Debug.lib has the following functions:

**KCmpStr()**

Compare two null-terminated strings.

**KGetChar()**

Get a character from the console.

**KGetNum()**

Get a number from the console.

**KMayGetChar()**

Return a character if present, but don't wait.

**KPrintf()**

Print formatted data to the console.

**KPutChar()**

Put a character to the console.

**KPutStr()**

Put a string to the console.



## Appendix C

# Floppy Boot Process and Physical Layout

### The Floppy Disk Boot Process

The first two sectors on each floppy disk contain special boot information. These sectors are read into the system at an arbitrary position; therefore, the code *must* be position independent. The first three longwords come from the include file *devices/bootblock.h*. The type must be BBID\_DOS; the checksum must be correct (an additive carry wraparound sum of 0xffffffff). Execution starts at location 12 of the first sector read in.

The code is called with an open trackdisk.device IO request pointer in A1 (see the “Trackdisk” chapter for more information). The boot code is free to use the IO request as it wishes (the code may trash A1, but must not trash the IO request itself).

The boot code must return values in two registers: D0 and A0. D0 is a failure code -- if it is non-zero then a system alert will be called, and the system will reboot.

If D0 is zero then A0 must contain the start address to jump to. The strap module will free the boot sector memory, free the boot picture memory, close the trackdisk.device IO request, do any other cleanup that is required, then jump to the location pointed to by A0.

Boot code may allocate memory, use trackdisk.device to load relocatable information into the memory, then return with D0=0 and A0 pointing to code. The system will clean up, then call the code.



## Commodore-Amiga Disk Format

The following are details about how the bits on the Commodore-Amiga disk are actually written.

### Gross Data Organization:

3 1/2 inch (90mm) disk  
double-sided  
80 cylinders/160 tracks

### Per-track Organization:

Nulls written as a gap, then 11 sectors of data.  
No gaps written between sectors.

### Per-sector Organization:

All data is MFM encoded. This is the pre-encoded contents of each sector:

two bytes of 00 data (MFM = \$AAAA each)  
two bytes of A1\* ("standard sync byte" -- MFM  
encoded A1 without a clock pulse)  
(MFM = \$4489 each)  
one byte of format byte (Amiga 1.0 format = \$FF)  
one byte of track number  
one byte of sector number  
one byte of sectors until end of write (NOTE 1)  
[above 4 bytes treated as one longword  
for purposes of MFM encoding]  
16 bytes of OS recovery info (NOTE 2)  
[treated as a block of 16 bytes for encoding]  
four bytes of header checksum  
[treated as a longword for encoding]  
four bytes of data-area checksum  
[treated as a longword for encoding]  
512 bytes of data  
[treated as a block of 512 bytes for encoding]

### NOTE 1

The track number and sector number are constant for each particular sector. However, the sector offset byte changes each time we rewrite the track.

The Amiga does a full track read starting at a random position on the track and going for slightly more than a full track read to assure that all data gets into the buffer. The data buffer is examined to determine where the first sector of data begins as compared to the start of the buffer. The track data is block moved to the beginning of the buffer so as to align some sector with the first location in the buffer.

Because we start reading at a random spot, the read data may be divided into three chunks: a series of sectors, the track gap, and another series of sectors. The sector offset value tells the disk software how many more sectors remain before the gap. From this the software can figure out the buffer memory location of the last byte of legal data in the buffer. It can then search past the gap for the next sync byte and, having found it, can block move the rest of the disk data so that all 11 sectors of data are contiguous.

Example:

The first-ever write of the track from a buffer looks like this:

<GAP> |sector0|sector1|sector2|.....|sector10|

sector offset values:

11 10 9 ..... 1

(If I find this one at the start of my read buffer, then I know there are this many more sectors with no intervening gaps before I hit a gap). Here is a sample read of this track:

<junk>|sector9|sector10|<gap>|sector0|...|sector8|<junk>

value of 'sectors till end of write':

2 1 .... 11 ... 3

result of track re-aligning:

<GAP>|sector9|sector10|sector0|...|sector8|

new sectors till end of write:

11 10 9 ... 1

so that when the track is rewritten, the sector offsets are adjusted to match the way the data was written.

#### NOTE 2 - Sector Label Area

This is operating system dependent data and relates to how AmigaDOS assigns sectors to files. Reserved for future use.

#### MFM Track Encoding

When data is MFM encoded, the encoding is performed on the basis of a data block-size. In the sector encoding described above, there are bytes individually encoded; three segments of 4 bytes of data each, treated as longwords; one segment of 16 bytes treated as a block; two segments of longwords for the header and data checksums; and the data area of 512 bytes treated as a block.

When the data is encoded, the odd bits are encoded first, then the even bits of the block.

The procedure is: Make a block of bytes formed from all odd bits of the block, encode as MFM. Make a block of bytes formed from all even bits of the block, encode as MFM. Even bits are shifted left one bit position before being encoded.

The raw MFM data that must be presented to the disk controller will be twice as large as the unencoded data. The following table shows the relationship:

|   |   |                      |
|---|---|----------------------|
| 1 | → | 01                   |
| 0 | → | 10 ;if following a 0 |
| 0 | → | 00 ;if following a 1 |

With clever manipulation, the blitter can be used to encode and decode the MFM.

# GLOSSARY

|                    |   |
|--------------------|---|
| Active screen      | The screen containing the active window.  |
| Active window      | The window receiving user input. Only one window is active at a time.   |
| Agnus              | One of the Amiga custom chips. Contains the blitter, copper, controls RAM addressing, DMA and other timing. The A500 contains a square version of this chip called Fat Angus. Regular and Fat Agnus can address 512K of chip memory. Newer versions of Agnus are able to address 1 megabyte of memory. The name stands for "Address Generator". |
| Alert              | A large flashing box displayed when there is a serious application or system problem.   |
| Alternate          | An image or border used in gadget highlighting. When the gadget is selected, the alternate image or border is substituted for the original image or border.   |
| Amiga™ keys        | Two command keys on the keyboard to the left and right of the space bar.  |
| amiga.lib          | A collection of functions and code stubs collected into a linker-format library. See the "Linker Libraries" appendix for more information.  |
| AmigaDOS™          | Amiga Disk Operating System. Controls file & CLI operations.  |
| Application gadget | A custom gadget created by a developer.   |
| AUTOCONFIG™        | The Amiga feature that automatically assigns memory addresses to expansion boards without the use of the switches or jumpers seen on other computers. See the "Expansion" chapter.  |
| Autodocs™          | Programmer's documentation that lists each Amiga function and command on a separate page. The documents have been automatically extracted from the system source code, thus the name AutoDocs. Available in the Addison-Wesley <i>ROM Kernel Manual, Includes &amp; Autodocs</i> (ISBN 0-201-18177-0).  |
| Auto-knob          | The special automatic knob for proportional gadgets; changes its shape according to the current proportional settings.  |
| Backdrop window    | A window that stays anchored to the back of the display.  |

|                          |  |
|--------------------------|--|
| <b>Bitmap</b>            | The complete definition of a display in memory, consisting of one or more bit-planes and information about how to organize the rectangular display.  |
| <b>Bitplane</b>          | A contiguous series of memory words, treated as if it were a rectangular shape.  |
| <b>Blitter</b>           | A graphics engine that is part of the custom chips. It can do BLITs (Block Image Transfers) in hardware. Sometimes called a BIMMER (Bitmap Image Manipulator) because Amiga blitter can do logic operations during the transfer, line draws, hardware fills, and more. |
| <b>Bob</b>               | Blitter Object Block. A graphic image prepared for processing by the blitter.  |
| <b>Body variables</b>    | Proportional gadget variables that contain the increment by which the pot variables may change.  |
| <b>Boolean gadget</b>    | A simple yes-or-no gadget.   |
| <b>Border area</b>       | The window perimeter, inside the border line. This area may contain border gadgets.  |
| <b>Border line</b>       | The default double-line drawn around the perimeter of all windows, except Borderless windows.  |
| <b>Borderless window</b> | A window with no drawn border lines.   |
| <b>CHIP memory</b>       | Memory accessible by the Amiga custom chips.   |
| <b>Checkmark</b>         | A small image that appears next to a menu item showing that the user has selected that item. By default, the checkmark is $\checkmark$ , but a custom image can be substituted.  |
| <b>CLI</b>               | <i>See</i> Command Line Interface.   |
| <b>Click</b>             | To quickly press and release a mouse button.   |
| <b>Clipboard</b>         | A device and convention used to store the last data cut or copied from a project.  |
| <b>Clipping</b>          | Causing a graphical rendering to appear only in some bounded area, such as only within the non-concealed areas of a window.  |
| <b>Close gadget</b>      | Gadget in the upper left corner of a window that may be selected to request that a window be closed.   |
| <b>Color indirection</b> | The method used by Amiga for coloring individual pixels, in which the binary number formed from all the bits that define a given pixel refers to one of the 32 color registers. Each of the 32 color registers can be set equal to any of 4,096 colors.                |
| <b>Color palette</b>     | The set of colors available in a screen.   |
| <b>Color register</b>    | One of 32 hardware registers containing colors that you can define.  |

|                        |  |
|------------------------|--|
| Column                 | A set of adjoining pixels that forms a vertical line on the video display.   |
| Command keys           | Keys that combine with alphanumeric keys to create command key sequences, which substitute for making selections with the mouse buttons.   |
| Command Line Interface | The conventional console-style interface to system commands, files and programs. Also called the CLI.  |
| Complement             | The binary complement of a color, used as a method of gadget highlighting and in flashing the screen. To complement a binary number means to change all the 1s to 0s and all the 0s to 1s. |
| Console device         | A communication path for both user input and program output. Especially recommended for input/output of text-only applications.  |
| Container              | Part of a proportional gadget; the area within which the knob or slider can move; the select box of the gadget.  |
| Control sequence       | A sequence of characters used to communicate with the console device. Sequences start with the Control Sequence Introducer or <CSI>. <CSI> may be \$9B or \$1B \$5B ( <ESC>[ ).            |
| Controller             | A hardware device, such as a mouse or a light pen, used to move the pointer or furnish some other input.   |
| Coordinates            | A pair of numbers shown in the form (x,y), where x is an offset from the left side of the display or display component and y is an offset from the top.                                    |
| Copper                 | Display synchronized coprocessor that handles the Amiga video display.   |
| DAC                    | Digital-to-analog converter. Converts a binary number to an analog voltage. Used in the audio output.  |
| Dead Key               | A key that produces no output when pressed, but modifies the next keystroke. For example, on a USA keyboard pressing ALT-K, then the letter a, produces umlaut-a.                          |
| Denise                 | One of the Amiga custom chips. Contains the video output signals, mouse input, etc. Name stands for "Display Enable".  |
| Depth                  | Number of bitplanes in a display.  |
| Depth gadgets          | Gadgets in the title bar of a screen or window used to send the screen or window to the back of the display or bring it up front.  |
| Device                 | Exec devices control input and output to hardware. Example: serial.device. Disk-based devices are stored in either the DEVS:, or SYS:Expansion directories.                                |
| Disable                | In Intuition, to make something unavailable to the user. In Exec, to lock out all interrupts.  |

|                       |  |
|-----------------------|--|
| Display               | To put up a screen, window, requester, alert, or any other graphics object on the video display.   |
| Display field         | One complete scanning of the video beam from top to bottom of the video display screen.  |
| Display memory        | The RAM that contains the information for the display imagery; the hardware translates the contents of the display memory into video signals. Also called CHIP memory.   |
| Display modes         | Common modes supported by the hardware are: high or low horizontal resolution, interlaced or non-interlaced vertical resolution, sprite mode, dual-playfield mode, Hold-And-Modify (HAM), and Extra Half-Bright (EHB). |
| DMA                   | Direct Memory Access. The transfer of data to memory by hardware, without intervention of the processor.   |
| Double-click          | To quickly press and release a mouse button twice.   |
| Double-menu requester | A requester that the user can open by double-clicking the mouse menu button.   |
| Drag                  | To move an icon, gadget, window, or screen by placing the pointer over the object to be moved and holding down the select button while moving the mouse.   |
| Drag gadget           | The portion of a window or screen title bar used for moving the window or screen around on the video display.  |
| Dual-playfield mode   | A display mode that allows you to manage two separate display memories, giving you two separately controllable displays at the same time.  |
| Enable                | In Intuition, to make something available to the user; a menu item or gadget that is enabled can be selected by the user. In Exec, to restart interrupts after a Disable.  |
| Exec                  | Core of the Amiga multitasking operating system.   |
| Extended selection    | A technique for selecting more than one menu item at a time.   |
| Extra Half-Bright     | A video mode with an extra bitplane. Where bits are set, the color intensity is halved.  |
| FAST memory           | Memory not accessible by the Amiga custom chips.   |
| Fill                  | To put a color or pattern within an enclosed area.   |
| Flag                  | A mechanism for selecting an option or detecting a state; a name representing a bit to be set or cleared.  |
| Font                  | A set of letters, numbers, and symbols that share the same basic design.   |
| Gadget                | Any of the control devices provided within a window, screen, or requester; employed by users to change what is being displayed or to communicate with an application or with Intuition.                                |

|                             |  |
|-----------------------------|--|
| <b>Gel</b>                  | Graphics Element. A generic term for graphics structures supported by the graphics library.  |
| <b>Ghost</b>                | Display less distinctly (overlay an area with a faint pattern of dots) to indicate that something, such as a gadget or a window, is not available or not active.   |
| <b>Ghost shape</b>          | The new outline of a window that shows briefly when the user is dragging or sizing a window.   |
| <b>Gimmezerozero window</b> | A window with a separate bitmap for the window border.   |
| <b>Handler</b>              | An AmigaDOS construct. Handlers are in the DOS namespace, but do not support files. Examples: SER: & PAR:. Disk-based handlers are stored in the l: directory.   |
| <b>Header file</b>          | A file that is included at the beginning of a C program and contains definitions of data types and structures, constants, and macros. <i>See</i> Include file and Autodocs.  |
| <b>High-resolution mode</b> | A horizontal display mode in which 640 pixels are displayed across a horizontal line.  |
| <b>Highlight</b>            | To modify the display of a selected menu item or gadget in a way that distinguishes it from its non-selected state.  |
| <b>Hit select</b>           | A method of gadget selection in which the gadget is unselected as soon as the select button is released.   |
| <b>Hold-and-modify mode</b> | A display mode that gives you extended color selection — up to 4,096 colors on the screen at one time. Also called HAM mode.   |
| <b>Hue</b>                  | The characteristic of a color that is determined by the color's position in the color spectrum.  |
| <b>Icon</b>                 | A visual representation of an object in the Workbench, such as a program, file, or disk.   |
| <b>IDCMP</b>                | “Intuition Direct Communications Message Port”; the primary communication path for user input to an application. Gives mouse and keyboard events and Intuition events in raw form. Provides a path for communicating to Intuition. |
| <b>Include files</b>        | Files providing system structure definitions and constants. Generally supplied by compiler and language vendors. <i>See</i> Header file and Autodocs.  |
| <b>Initialize</b>           | To set up a structure with certain default parameters.   |
| <b>Input event</b>          | The message created by the input device whenever a signal is detected at one of the Amiga input ports.   |
| <b>Interlaced mode</b>      | A vertical display mode in which twice the number of lines are displayed from top to bottom of the video display.  |
| <b>IntuiMessage</b>         | The input message created by Intuition for application programs; the message is the medium in this case.   |

|                            |   |
|----------------------------|---|
| <b>Intuition™</b>          | The Amiga user interface. Intuition is implemented as a library.  |
| <b>KeyMap</b>              | Translation table used by the console device to translate keycodes into normal characters.  |
| <b>Kickstart™</b>          | The Amiga Operating System ROM. Kickstart is also provided on a floppy disk for A1000 systems.  |
| <b>Knob</b>                | Part of a proportional gadget; the user manipulates the knob to set a proportional value.   |
| <b>Library</b>             | A collection of predefined functions that can be used by any program. Exec libraries are found at run time. Disk based Exec libraries are stored in LIBS: or SYS:Expansion. Linker libraries are combined directly into the code of applications.   |
| <b>Linked list</b>         | A chain of objects linked together with pointers. See the Exec "Lists" chapter for information on Exec doubly-linked lists.   |
| <b>Lock</b>                | An AmigaDOS structure that arbitrates access to shared files.   |
| <b>Low-resolution mode</b> | A horizontal display mode in which 320 pixels are displayed across a horizontal line.   |
| <b>Menu bar</b>            | A strip in the screen title bar that shows the menu list when the user holds down the menu button.  |
| <b>Menu button</b>         | The right-hand button on the mouse.   |
| <b>Menu item</b>           | One of the choices in a menu; the options presented to the user.  |
| <b>Menu list</b>           | List of menus displayed in the screen title bar when the user holds down the menu button.   |
| <b>Menu shortcut</b>       | An alternate way of choosing a menu item by pressing a key on the keyboard while holding down the right AMIGA key.  |
| <b>Message ports</b>       | A software mechanism managed by the Amiga Exec that allows intertask communications.  |
| <b>Microsecond (us)</b>    | One millionth of a second (1/1,000,000).  |
| <b>MIDI</b>                | Musical Instruments Digital Interface. A standard serial interface used by many musical instruments and music keyboards.  |
| <b>Millisecond (ms)</b>    | One thousandth of a second (1/1,000).   |
| <b>MMU</b>                 | A device for arbitrating and protecting against a task damaging the memory of another task, and/or for extending memory capacity with "virtual" swap memory on disk. Also has lots of other uses that are well beyond the scope of this discussion. Some newer Amiga models have MMU chips. |



|                         |   |
|-------------------------|---|
| Mouse                   | A controller device used to move the pointer and make selections.   |
| Multitasking            | A system in which many tasks can be operating at the same time, with no task forced to be aware of any other task.  |
| Mutual exclusion        | The principle that says that selecting a menu item (or gadget) can cause other menu items (or gadgets) to become deselected.  |
| Nanosecond (ns)         | One billionth of a second (1/1,000,000,000).  |
| Non-interlaced mode     | A display mode in which 200 lines are displayed from top to bottom of the video display (256 lines in PAL).   |
| NTSC                    | National Television Systems Committee. The North American specification for television. The base Amiga crystal frequency for NTSC is 28.63635 Mhz.  |
| Null-terminated         | A string that ends with a byte of zero; text strings must be null-terminated.   |
| Nuke                    | To destroy, demolish, obliterate, wipe out, mung, hash into little bits, waste, screw up, or make FUBAR, by means of atomic weapons, or with a computer.  |
| Offset                  | A position in the display that is relative to some other position.  |
| Overscan                | The portion of the video display beyond the normal image display area. Differing amounts of overscan will be visible on different monitors. Overscan capability is important for video applications that must fill the entire screen. |
| Packet, AmigaDOS        | An AmigaDOS specific message. All DOS activity is actually carried out with a packet sent to the proper DOS handler process. Packets are available to open files, read sections of a file, seek to a new file position, etc.          |
| PAL                     | Two definitions. Phase Alternate Line, a European television transmission standard. The Amiga crystal frequency for PAL is 28.37516 Mhz. Also Programmable Array Logic, an easily customizable logic chip.                            |
| Parallel port           | A Centronics-compatible connector on the back of the Amiga used to attach printers and other add-ons.   |
| Paula                   | One of the Amiga custom chips. Includes the audio DAC, interrupt chip, custom serial chip, disk controller, analog counters, etc. The name stands for "Ports, Audio, Uart, Logic Array".  |
| Pen                     | A variable containing a color register number used for drawing lines or filling background.   |
| Pixel                   | Short for "picture element." The smallest addressable element in the video display. Each pixel is one dot of color.   |
| 16-inch rotary debugger | A highly effective tool for locating problems in computer software. Available for delivery in most major metropolitan areas. Anchovies contribute to poor coding style.   |

|                            |   |
|----------------------------|---|
| <b>Playfield</b>           | One of the basic elements in Amiga graphics; the background for all the other display elements.   |
| <b>Pointer</b>             | A small object, usually an arrow, that moves on the display when the user moves the mouse (or the cursor keys). It is used to choose menu items, open windows, and drag and select other objects. |
| <b>Pot variables</b>       | Proportional gadget variables that contain the actual proportional values.  |
| <b>Preferences</b>         | A program that allows the user to change various global parameter settings.   |
| <b>Preserve</b>            | To keep overlapped portions of the display in hidden memory buffers.  |
| <b>Primitives</b>          | Amiga low-level library functions.  |
| <b>Process</b>             | An extension to a task. Created by AmigaDOS. Only processes may call DOS library functions.   |
| <b>Project</b>             | A Workbench term for the output of a tool. A data file.   |
| <b>Project menu</b>        | A menu for opening and saving project files.  |
| <b>Proportional gadget</b> | A gadget used to display a proportional value or get a proportional setting from the user. Consists of a knob or slider and a container.  |
| <b>RAM</b>                 | Random access (volatile) memory.  |
| <b>Raster</b>              | The area in memory where the bitmap is located.   |
| <b>RastPort</b>            | The data structure that defines the general parameters of display memory.   |
| <b>Refresh</b>             | To recreate a display that was hidden and is now revealed.  |
| <b>Render</b>              | To draw or write into display memory.   |
| <b>Requester</b>           | A rectangular information exchange region in a window. When a requester appears, the user must select a gadget in the requester to close the requester before doing anything else in the window.  |
| <b>Resolution</b>          | On a video display, the number of pixels that can be displayed in the horizontal and vertical directions.   |
| <b>ROM</b>                 | Read-Only (non-volatile) Memory. Used to store unchanging system data. The A500 and A2000 have Kickstart in ROM.  |
| <b>Screen</b>              | A full-width area of the display with a set color palette, resolution, and other display modes. Windows open in screens.  |
| <b>Scroll</b>              | To move the contents of display memory within a window.   |
| <b>Scroll bar</b>          | A proportional gadget with which the user can display different parts of the display memory.  |

|                     |   |
|---------------------|---|
| Select              | To pick a gadget or menu item.  |
| Select box          | The sensitive area of a gadget or menu item. When the user clicks the pointer within a gadget's select box, the gadget becomes selected.  |
| Select button       | The left-hand button on a mouse.  |
| Semaphore           | A structure provided by Exec. An efficient method for arbitrating access to an item in a multitasking environment.  |
| Serial port         | The RS-232C compatible connector on the back of the Amiga. Used to attach modems, other computers, and serial add-ons.  |
| Shortcut            | A quick way, from the keyboard, to choose a menu item or select a gadget.   |
| Simple Refresh      | A method of refreshing window display in which concealed areas are redrawn by the program when they are revealed  |
| Sizing gadget       | A gadget for the user to change the size of a window.   |
| Slider              | Part of a proportional gadget; used to pick a value within a range by dragging the slider or by moving the slider by increments with clicks of a mouse button.  |
| :~)                 | A standard smiley. Used to indicate satire or humor.  |
| Smart Refresh       | A method of refreshing window display in which the system keeps information about concealed areas in off-display buffers and refreshes the display from this information. If the window is sized, the program may have to recreate the display. |
| Sprite              | A small, easily movable graphic object. You can have multiple sprites in a window at the same time.   |
| Sprite mode         | A display mode that allows you to have sprites in your windows.   |
| String gadget       | A gadget that prompts the user to enter a text string or an integer.  |
| Structure           | A specific layout of fields in memory. Structures are defined in <i>include files</i> . Each member of a structure has a positive offset from the structure base.   |
| Submenu             | An additional menu that appears when some menu items are chosen by the user.  |
| SuperBitMap Refresh | A method of window refresh where the display is recreated from a separate bitmap area.  |
| SuperBitMap window  | A window with a bitmap which may be larger than the screen's bitmap.  |
| System gadgets      | Predefined gadgets for windows and screens; for screens, dragging and depth arranging; for windows, dragging, depth arranging, sizing, and closing.   |
| Task                | The basic unit of multitasking. Each task appears to have full control over its own virtual 68000 machine.  |

|                  |  |
|------------------|--|
| Text cursor      | In programs containing text and in string gadgets, a marker that indicates your position in the text.  |
| Title bar        | A strip at the top of a screen or window that contains gadgets and an optional name for the screen or window.  |
| Toggle select    | A method of gadget selection in which the gadget remains selected when the user releases the select button and does not become deselected until the user picks it again. |
| Tool             | An application program.  |
| TOOLTYPES        | An array of parameters passed to Workbench tools. TOOLTYPES are stored in icons; use the Workbench Info menu to view or change them.                                     |
| Topaz            | The default system font. It is a fixed-width font in two sizes: 60 columns wide and 8 lines tall; 80 columns wide and 9 lines tall.                                      |
| Transparent      | A special color register definition that allows a background color to show through. Used in dual-playfield mode.   |
| Type style       | A variation of a typeface, such as italic or bold.   |
| Typeface         | <i>See</i> Font.   |
| Undo             | An option that reverses the previous editing action. Very popular with users.  |
| UserPort         | The message port created for you by Intuition when you request an IDCMP. Your program receives messages from Intuition via this port.                                    |
| Vector           | A line segment.  |
| Video display    | Everything that appears on the screen of a video monitor or television.  |
| View             | The graphics library data structure used to create the Intuition display.  |
| ViewPort         | The graphics library data structure used to create and manage the Intuition screen.  |
| Window           | Rectangular display in a screen that accepts input from the user and displays output from the application.   |
| WindowPort       | The message port created for you when you request an IDCMP. You respond to messages from Intuition via this port.  |
| Workbench™       | The graphical user interface to file and programs.   |
| Workbench disk   | The bootable disk inserted at the hand prompt. Contains all of your system files, libraries, devices, etc. A hard disk drive will usually have a Workbench partition.    |
| Workbench screen | The primary Intuition screen.  |

# INDEX

- 4703, 306
- 68020, 911
- 68881, 563
- AbortIO(), 292, 295, 609, 611, 616-617, 619, 638
- ABORTIO macro, 292
- ActivateGadget(), 99
- ActivateWindow(), 53
- Active screen, 13
- Active window, 36
- active window, 36
- AddAnimOb(), 479
- AddBob(), 459
- AddDosNode(), 530-531
- AddFont(), 410
- AddGadget(), 79
- AddGadgets(), 82
- AddGlist(), 82
- AddGList(), 98
- AddHead(), 251
- AddIntServer(), 314
- AddLibrary(), 234
- AddPort(), 281
- Address error, 275
- AddTail(), 251
- AddTask(), 265
- AddTime(), 875
- AddTOF(), 917
- AddVSprite(), 439
- afp(), 918
- After pointer
  - changing Bob priority, 461
  - in animation precedence, 476
  - in Bob priority, 458
- agateWindow.h, 59
- Alerts, 146
  - Application, 146
  - creating, 147
  - display, 146
  - severity of, 146
  - System, 146
- Allocate(), 246
- Allocating memory, 239
- AllocEntry(), 243, 246
- AllocMem(), 200, 210, 241, 265
- AllocRaster(), 57, 347
  - allocating memory, 353
- AllocRemember(), 200, 210
  - example code, 201
- AllocSignal(), 167, 269, 277
- AllocTrap(), 277
- ALT, 192
- Alt Key, 695
- Alternate (ALT) keys, 192
- Alternate Filesystems, 538
- AMIGA keys, 657
  - as command keys, 192
  - in command-key sequences, 115
  - Workbench shortcuts, 193
- AmigaDOS, 538
- AmigaDOS Partition, 538
- Amiga.lib, 916
- Amplitude, 608
- AndRectRegion(), 517
- AndRegionRegion(), 518
- ANFRACSIZE, 482
- Animate(), 478, 481
- animation
  - acceleration, 478
  - AnimCRoutine, 477
  - AnimORoutine, 477
  - introduction, 421
  - motion control, 478
  - Ring Motion Control, 478
  - sequenced drawing, 473
  - types, 423
  - velocity, 478
- AnimComp
  - BobComp, 453
  - BOBISCOMP flag, 453
  - position, 477
  - TimeSet member, 478
- AnimCRoutine
  - in creating animation, 481
  - with Animate(), 481

- ANIMHALF, 482
- AnimOb
  - definition, 479
  - position, 479
- AnimORoutine
  - in creating animation, 481
  - with Animate(), 481
- AnX AnimOb member
  - specifying registration point, 479
- AnX variable
  - in ring processing, 479
  - in velocity and acceleration, 482
  - moving registration point, 478
- AnY AnimOb member
  - specifying registration point, 479
- AnY variable
  - in ring processing, 479
  - in velocity and acceleration, 482
  - moving registration point, 478
- AOIPen variable
  - in filling, 361
  - in RastPort, 361
- A-Pen
  - see FgPen, 361
- Application gadgets
  - (also see gadgets), 41
- area buffer, 359
- area pattern, 362
- AreaDraw()
  - adding a vertex, 367
  - in area fill, 359
- AreaEnd()
  - drawing and filling shapes, 367
  - in area fill, 359
- AreaInfo pointer, 359
- AreaMove()
  - beginning a polygon, 367
  - in area fill, 359
- arnd(), 918
- AskFont(), 397
- AskKeyMap(), 663
- AskSoftStyle(), 402
- Assembly language, 223
- AUD0-AUD3 interrupts, 307
- Audio, 607
  - A Simple Audio Example, 611
  - AbortIO(), 611
  - allocation key, 616
  - allocation/arbitration commands, 615
  - BeginIO(), 611
  - CloseDevice(), 611
  - double-buffering, 618
  - Free, 609
  - hardware control commands, 618
  - IORequest block, 609
  - Lock, 609, 615
  - multi-channel, 616
  - OpenDevice(), 610
  - playing the sound, 618
  - Precedence, 609
  - Reserve, 609
  - scope of commands, 610
  - starting the sound, 620
  - steal channel, 615
  - stopping the sound, 619
  - Wait(), 611
  - WaitPort(), 611
- Audio Channels, 607
  - allocation, 614-615
  - changing the precedence, 616
  - freeing, 616-617
- Audio Device, 607
- audio device
  - precedence of users, 614
- Audio Hardware, 615
- Autoboot, 531, 538
- AUTOCONFIG, 527, 529, 532
  - hardware manufacturer number, 527
- AutoRequest(), 139, 144
- autorequester.c, 147
- autovector address, 306
- AvailFonts(), 404
- AvailMem(), 243
- Backdrop window, 38
- backdrop window, 59
- background pen, 361
- background playfield, 354
- BadBlock, 539
- BadBlockBlock, 538, 541
- BDRAWN flag, 453
- beam synchronization, 379
- Beeping, 210
- Before pointer
  - changing Bob priority, 461
  - in animation precedence, 476
  - in Bob priority, 458
- BeginIO(), 292, 295, 609, 611, 917
- BeginRefresh(), 45, 55
- BeginUpdate(), 510
- BehindLayer(), 509
- BgPen, 400
- BindDrivers, 530
- BitMap
  - address, 348
  - in SuperBitMap layers, 512
  - software clipping, 368
  - with write mask, 360
- BitMap structure

- in dual-playfield display, 355
  - preparing, 347
- bit-planes
  - extracting a rectangle from, 376
  - in dual-playfield display, 354
- Bit-planes
  - in image display, 183
  - in screens, 17
- blank pointer, 59
- BLIT interrupts, 307
- Blitter
  - in Bob animation, 423
- blitter
  - in copying data, 378
- Blitter
  - in trackdisk device, 884
- blitter
  - VBEAM counter, 381
- BltBitMap(), 376
- BltBitMapRastPort(), 376
- BltClear(), 372
- BltMaskBitMapRastPort(), 376
- bltnode structure
  - creating, 380
  - linking blitter requests, 379
- BltPattern(), 374
- BltTemplate(), 375
- BNDRYOFF() macro, 368
- Bob
  - colors, 455
- BobComp pointer, 453
- BOBISCOMP flag, 453
- BOBNIX flag, 454
- Bobs
  - adding new features, 471
  - as a paintbrush, 453
  - as part of AnimComp, 453
  - Before, After pointers, 476
  - bit-planes, 456-457
  - changing, 460
  - colors, 456-457
  - defining, 448
  - displaying, 460
  - double-buffering, 454, 464
  - drawing order, 458
  - introduction, 421
  - list, 458
  - priorities, 458
  - removing, 454
  - shadow mask, 449, 456
  - shape, 454
  - simple definition, 423
  - size, 454
  - sorting the list, 460

- structure, 450
  - transparency, 449
- BOBSAWAY flag, 454
- Boolean Gadget, 100
- BoolInfo Structure, 94
- Boot, 531
- Boot Disk, 538
- BootNode, 533
- Border Structure, 100
- Border structure, 177
- Border variables, 38
- BORDERHIT flag, 468
- Borderless window, 37
- borderless window, 59
- borderless.c, 59
- BorderLine pointer, 467
- Borders
  - in Borderless windows, 37
  - in Gimmezerozero windows, 38
  - window, 41
  - window variables, 41
- BOTTOMHIT flag, 470
- bottommost member
  - GelsInfo structure, 436
  - in Bob/VSprite collisions, 470
- B-Pen
  - see BgPen, 361
- BufBuffer DBufPacket member, 464
- BufPath DBufPacket member, 464
- BufY, BufX DBufPacket members, 464
- BuildSysRequest(), 140
- Bus error, 275
- BUSWIDTH, 533
- Busy wait, 263
- BWAITING flag, 453
- bytecnt variable, 375
- bytecount pointer, 372
- Caps Lock key, 657
- CAPS LOCK key, 667
- CAT, 629
- CATS, 527
- Cause(), 316
- Cause, 893
- CD\_ASKDEFAULTKEYMAP, 662
- CD\_SETDEFAULTKEYMAP, 662
- ChangeSprite(), 430
- Channel, 608
  - steal, 615
- Channels, 607
- CHECKED and CHECKIT
  - checkmark, 113
  - mutual exclusion, 114
- CheckIO(), 292, 295, 638
- CHECKIT, 114

- Chip memory, 57
- CHIP memory, 210
- Chip memory, 240
- CHK instruction, 275
- CIA Resource, 904
- cia.resource, 904
- cleanup variable, 381
- ClearDMRequest(), 54
- ClearMenuStrip(), 53, 111, 116, 124
- ClearMenuStrip, 125
- ClearPointer(), 46, 54, 59
- ClearRectRegion(), 518
- ClearRegion(), 518
- CLIP, 629
- Clip
  - Identification, 629
- ClipBlit(), 376
- Clipboard, 627
  - clear, 628
  - data, 629
  - disk file, 629
  - flush, 628
  - IFF, 629
  - invalid, 628
  - Multiple Clips, 630
  - post, 628, 630
  - read, 628
  - read ID, 628
  - start, 628
  - stop, 628
  - unit number, 630
  - update, 628
  - write, 628, 630
  - write ID, 628
- Clipboard Tool, 629
- clipboard.device, 630
- ClipID, 630
- clipping
  - in area fill, 368
  - in line drawing, 366
- clipping rectangles
  - in layer operations, 510
  - in layers, 506, 514
  - modifying regions, 517
- clipping region
  - in VSprites with GELGONE, 436
- ClipRect structure, 514
- ClipReq, 629
- CLIPS:, 629
- Close(), 745
- Close Gadget, 74
- CloseDevice(), 296, 609, 611, 628, 648
- CloseFont(), 403
- CloseLibrary(), 234
- CloseScreen(), 15, 24
- CloseWindow(), 36, 49, 74, 168
- CloseWorkBench(), 14, 25, 218
- CMD\_CLEAR Command, 890
- CMD\_CLEAR commands, 293
- CMD\_FLUSH commands, 293
- CMD\_READ Command, 888
- CMD\_READ commands, 293
- CMD\_RESET commands, 293
- CMD\_START commands, 293
- CMD\_STOP commands, 293
- CMD\_UPDATE Command, 889
- CMD\_UPDATE commands, 293
- CMD\_WRITE Command, 889
- CMD\_WRITE commands, 293
- CmpTime(), 875
- collisions
  - between GEL objects, 465
  - boundary, 468
  - boundary hits, 470
  - collision mask, 466
  - detection in hardware, 465
  - fast detection, 467
  - GEL-to-GEL, 470
  - in animation, 465
  - multiple, 471
  - sensitive areas, 467
  - user routines, 468
- CollMask Member
  - in Bobs, 457
- CollMask member
  - of VSprite structure, 466
- Color, 17
- color
  - affect of display mode on, 332
  - ColorMap structure, 348
  - flickering, 442
- Color
  - in Borders, 176
- color
  - in dual playfield mode, 340
  - in flood fill, 369
  - in hold-and-modify mode, 357
- Color
  - in Images, 181, 183
  - in IntuiText, 178
  - in windows, 47
- color
  - mode in flood fill, 369
  - of Bobs, 455
  - of individual pixel, 365
  - PlayField and VSprites, 442
  - relationship to bit-planes, 334
  - relationship to depth of BitMap, 339



- Simple Sprites, 428
- single-color raster, 372
- sprites, 341
- transparency, 438
- VSprite, 438
- ColorMap structure, 348
- Command key style, 216
- Command-key sequence events, 192
- Command-key sequences, 115
- compatibility
  - international, 914
- compatibility problems, 911
- COMPLEMENT, 400
- complement mode, 362
- ConfigDev, 528
- ConfigDev Structure, 532
- CONFIGME, 530, 537
- CONFIGTIME, 532
- ConMayGetChar(), 647
- ConPutChar(), 642
- console
  - alternate key maps, 667
  - capsable keys, 667
  - character output, 638
  - closing, 648
  - control sequence introducer, 656
  - control sequences, 642
  - high key map, 664
  - input event qualifiers, 656
  - input stream, 653
  - key map standards, 668
  - keyboard input, 638
  - keymapping, 659, 663
  - keymapping qualifiers, 664-665
  - keytypes, 666
  - low key map, 664
  - mouse button events, 662
  - raw events, 655
  - raw input types, 655
  - reads, 647
  - repeatable keys, 668
  - string output keys, 666
  - window bounds, 654
- Console Device, 695
- Control (CTRL) key, 192
- ConWrite(), 642
- cookie cut, 378
- COPER interrupts, 307, 314
- Copper
  - changing colors, 348
  - display instructions, 349
  - in custom screens, 15
  - in drawing VSprites, 442
  - in interlaced displays, 356
  - long-frame list, 356
  - MakeVPort(), 353
  - MrgCop(), 349, 356
  - short-frame list, 356
- copying
  - data, 378
  - rectangles, 376
- CopyMem(), 210, 243
- CopyMemQuick(), 243
- count variable, 366
- CPU priority level, 307
- cp\_x variable
  - in drawing, 364
- cp\_y variable
  - in drawing, 364
- crashes, 910
- crashing
  - with drawing routines, 366
  - with fill routines, 368
- CreateBehindLayer(), 508
- CreateExtIO(), 641, 647, 745, 886, 917
- CreatePort(), 281, 746, 886, 917
- CreateTask(), 266, 917
- CreateUpfrontLayer(), 508
- Critical section, 271
- CTRL, 192
- Current Clip, 630
- CurrentBinding Structure, 530
- CurrentTime(), 209
- Custom gadgets
  - in screens, 20
  - in windows, 34
- Custom pointer, 57
- Custom screens
  - closing, 15
  - managed by applications, 15
  - rendering in, 15
- Cut, 627
- Cycle, 608
- DamageList structure
  - in layers, 510
- dbf(), 918
- DBuffer pointer, 464
- DBufPacket structure, 464
- Deallocate(), 246
- deallocation
  - Copper list, 353
- Deallocation
  - memory, 239
- deallocation
  - memory, 353, 360
- Debug(0), 322
- Debugger, 321
- debugging, 914

- Debug.lib, 916
- DeleteExtIO(), 648, 917
- DeleteLayer(), 508
- DeletePort(), 282, 917
- DeleteTask(), 266, 917
- depth, 339
- Depth
  - in Images, 184
  - screen, 17
- Depth member, 455
- Depth variable
  - in VSprite structure, 437
- Depth-Arrangement Gadgets, 74
- destRastPort variable, 376
- destX variable, 376
- destY variable, 376
- Device
  - romtag, 234
- Device Partitions, 538
- Devices
  - definition, 289
  - driver, 289
  - input/output, 289
- devices
  - standard, 298
- Devices
  - task structure fields for, 265
  - unit, 289
- devices/audio.h, 609
- devices/clipboard.h, 629-630
- devices/hardblocks.h, 545
- DEVS:Mountlist, 539
- DF0:, 538
- DHeight variable
  - in ViewPort, 345
  - in ViewPort display memory, 343
- DiagArea Structure, 532
- DIAGVALID, 537
- Digital-To-Analog, 608
- Disable(), 272
- DISABLE, 319
- Disable(), 319
- DISABLE macro, 272
- Disabling
  - interrupts, 272
- disabling
  - interrupts, 309
- Disabling
  - maximum disable period, 272
- disabling interrupts, 319
- Disk Resource, 904
- diskfont.h, 408, 410
- DiskFontHeader structure, 410
- DISKINSERTED, 893
- DISKREMOVED, 893
- disk.resource, 904
- DisownBlitter(), 379
- Display Element, 174
- display fields, 330
- Display memory
  - RastPort, 12
  - screen, 12
- display modes, 340
- Display modes
  - in custom screens, 16
  - set by screen, 11
- display width
  - affect of overscan on, 329
  - effect of resolution on, 342
- DisplayAlert(), 147
- displayalert.c, 147
- DisplayBeep(), 210
- DisposeFontContents(), 409
- DisposeRegion(), 515
- DMA
  - displaying the View, 350
  - playfield, 339
- DoCollision()
  - purpose, 465
  - with collision masks, 469
- DoIO(), 291, 294, 539, 611, 618, 628, 638
- DoIO, 647
- DOS Device Node, 530
- DOS Node, 531
- dotted lines, 362
- double-buffering
  - allocations for, 355
  - Copper in, 356
  - Copper lists, 444
  - with Bobs, 464
- DoubleClick(), 205
- doublemenreq.c, 147
- doublemenreq.h, 147
- Dragging gadget, 12
- Dragging Gadget, 74
- Draw()
  - in line drawing, 365
  - multiple line drawing, 366
- DrawBorder(), 174
- DrawerData structure, 583
- DrawGList(), 210
  - and BDRAWN flag, 453
  - and BOBNIX flag, 454
  - and BOBSAWAY flag, 454
  - and BWAITING flag, 453
  - animation, 481
  - changing Bobs, 461
  - displaying Bobs, 460

- moving registration point, 478
- preparing the GELS list, 443
- removing Bobs, 460
- with DoCollision(), 481
- DrawImage(), 174
- drawing
  - changing part of drawing area, 374
  - clearing memory, 372
  - colors, 361
  - complement mode, 362
  - lines, 365
  - memory for, 358
  - modes, 362
  - moving source to destination, 375
  - pens, 361
  - pixels, 365
  - shapes, 369
  - turning off outline, 368
- drawing pens
  - color, 361
  - current position, 364
- DrawMode variable, 400
  - in area drawing and filling, 367
  - in flood fill, 369
  - in stencil drawing, 374
  - with BltTemplate, 376
- Drive Initialization, 539
- DriveInit(), 539
- DSKBLK interrupts, 307
- DSKSYNC interrupts, 307
- dual playfields
  - bit-planes, 354
  - color map, 349
  - colors, 340
  - priority, 354
- DUALPF flag
  - in dual playfield display, 354
  - in ViewPort, 340
- Dual-playfield mode, 17
- DumpRPort(), 752
- DWidth variable
  - in ViewPort, 337, 345
  - in ViewPort display memory, 343
- DxOffset variable
  - effect on display window, 345
  - in ViewPort display memory, 343
- DyOffset variable
  - effect on display window, 345
  - in ViewPort display memory, 343
- Enable(), 272
- ENABLE, 319
- Enable(), 319
- ENABLE macro, 272
- End-of-Clip, 630
- End-of-File, 630
- EndRefresh(), 45, 55
- EndRequest(), 54
- EndUpdate(), 510
- Enqueue(), 251, 531
- EPROM, 528
- EpsonQ
  - Example source files, 804
- EQUAL status code, 380
- errors, 909
- ESC, 192
- Escape (ESC) key, 192
- ETD\_CLEAR Command, 890
- ETD\_FORMAT Command, 891
- ETD\_MOTOR Command, 890
- ETD\_RAWREAD Command, 894
- ETD\_RAWWRITE Command, 895
- ETD\_READ Command, 888
- ETD\_SEEK Command, 896
- ETD\_UPDATE Command, 889
- ETD\_WRITE Command, 889
- Events, 269
- example
  - text
    - FontParade, 404
    - ShowDefaultFont, 397
    - ShowDrawModes, 401
    - ShowOpenFont, 399
    - suits8, 414
    - Wrapper, 417
- Exception signal, 274
- Exceptions
  - synchronous, 274
- Exceptions (TRAPS), 323
- Exclusion, 271
- Exec Devices, 529
- Exec Lists, 249
- exec/libraries.h, 236
- expansion
  - device drivers, 529
- Expansion Board Drivers
  - disk, 529
  - ROM, 529, 531
- Expansion Peripherals, 528
- ExpansionBase, 531
- expansion.library, 527, 530
- ExpansionRom, 528, 532
- ExpansionRom Structure, 532
- extended selection, 116, 125
- EXTER interrupts, 307, 314
- Extra-halfbright mode, 17
- Extra-Half-Brite
  - Clearing Plane 6, 360
  - Setting Plane 6, 360

- EXTRA\_HALFBRITE flag, 340-341
- Extra-Half-Brite mode, 356
- fast floating-point library, 547
- Fast memory, 240
- FastRand(), 917
- Fatal system error, 321
- CommandTable, 772
- DISABLE mutual-exclusion mechanism, 308
- DoSpecial(), 772-773
- InitRequester(), 141
- io\_TermArray, 740
- LockIBase(), 199
- PDCMD\_SETPARAMS, 740
- Troubleshooting Guide, 909
- UnlockIBase(), 199
- fclose(), 918
- fgetc(), 918
- FgPen, 400
- FgPen variable
  - in area drawing and filling, 367
  - in complement mode, 362
  - in flood fill, 369
  - in JAM1 mode, 361
  - in line drawing, 366
  - in RastPort, 361
  - in rectangle fill, 370
  - with BltTemplate, 376
- inputevent.h, 695
- FileSysEntry, 539, 544
- FileSysHeaderBlock, 538-539, 543
- filesysres.h, 545
- FileSysResource, 538, 545
- FileSystem.resource, 544
- mathieeedoubbas\_lib.lib, 565
- FinalPC, 248
- FindBoards.c, 528
- FindConfigDev(), 528
- FindName(), 252
- FindPort(), 282
- FindToolType(), 587
- First-In-First-Out (FIFO), 251, 279
- Flags variable, 478
  - in layers, 511
  - with BNDYOFF() macro, 368
- Flashing the display, 210
- flicker, 112, 379, 381
- Flood(), 369
- Floppy Boot Process, 921
- Floppy Disk, 884
- Floppy Drive, 538
- Floppy Format, 921
- Floppy Physical Layout, 921
- Floppy Technical, 921
- font accessors, 412

- font flags
  - FPF\_DESIGNED, 411
  - FPF\_DISKFONT, 411
  - FPF\_PROPORTIONAL, 411
  - FPF\_REMOVED, 411
  - FPF\_REVPATH, 411
  - FPF\_ROMFONT, 411
- font height, 410
- font style
  - BOLD, 411
  - EXTENDED, 411
  - ITALIC, 411
  - NORMAL, 411
- font width, 411
- FontContents structure, 408
- FontContentsHeader structure, 408
- Fonts, 100
- fonts, 395
- Fonts
  - custom, 19
  - default, 19
  - in creating text, 179
  - Topaz, 19
- Forbid(), 271, 593
- Forbidding, 271
- foreground pen, 361
- FOREVER loop, 353
- fpa(), 919
- fprintf(), 918
- fputc(), 918
- fputs(), 918
- Free memory, 248
- FreeColorMap(), 353
- FreeCopList(), 353
- FreeCprList(), 353
- FreeDiskObject(), 581
- FreeEntry(), 243, 246
- FreeMem(), 241
- FreeRaster(), 353
- FreeRemember(), 200
  - example code, 201
- FreeSignal(), 270, 277
- FreeSprite(), 431
- FreeTrap(), 277
- FreeVPortCopLists(), 353
- Frequency, 608
- Frequency modulation, 608
- FTXT, 629
- Function keys, 192
- Gadget Structure, 90
- Gadget structure, 583
- Gadget style, 214
- Gadgets
  - boolean type

- BoolInfo structure, 94
  - hit select, 82
  - toggle select, 82
- combining types, 89
- enabling and disabling, 81
- gadget structure, 90
- hand-drawn, 75
- highlighting, 80
- in window borders, 79
- integer type, 88
- line-drawn, 76
- proportional, 83
- PropInfo structure, 95
  - refreshing, 81
  - select box, 77
  - selection of, 77
  - string
    - description, 87
    - StringInfo structure, 96
  - without imagery, 76
- gadgets.c, 100
- gadgets.h, 100
- Gameport Connectors, 694
- Gameport Device
  - connectors, 683
  - system functions, 684
  - triggering events, 686
  - units, 683, 694
- GameTrigger structure, 686
- GELGONE Flag
  - in VSprite structure, 436
- GELS
  - introduction, 421
  - types, 422
- GelsInfo pointer, 360
- GelsInfo structure, 440
- getchar(), 918
- GetColorMap(), 353
- GetCurrentBinding(), 530
- GetDefPrefs(), 204, 209
- GetDiskObject(), 581
- GetMsg(), 286, 295
- GetPrefs(), 166, 204, 208
- GetRGB4(), 348
- GetScreenData(), 24
- GetSprite(), 429
- GfxBase variable, 346
- Gimmezerozero window
  - gadgets in, 38
  - requesters in, 38
- Gimmezerozero window type, 37
- GPD\_ASKCTYPE command, 684
- GPD\_ASKTRIGGER command, 688
- GPD\_READEVENT command, 688

- GPD\_SETCTYPE command, 685
- GPD\_SETTRIGGER command, 686
- graniteWindow.h, 7
- Graphics
  - Amiga primitives, 187
  - special Intuition functions, 187
- graphics library, 346
- graphics/text.h, 399
- HAM flag, 340-341, 357
- Hard Disk, 538
- HardBlocks, 538
- hardware interrupts, 305
- Hardware Manufacturer Number, 527
- hardware sprites
  - in animation, 349
- hardware Sprites
  - reserving, 440
- hardware/intbits.h, 310
- Harmony, 607
- Height variable
  - in ViewPort, 337
  - in VSprite structure, 437
- Height VSprite member
  - and Bobs, 454
- hellogoodbye.h, 59
- hellotext.h, 59
- High-resolution mode, 16
- HIRES flag, 340
- hires.h, 7
- HitMask member, 468
- Hold-and-modify mode, 17
- hold-and-modify mode, 357
- HP\_LaserJet
  - Example source files, 822
- icon library, 580
- IDCMP, 700
  - closing, 162
  - flags, 163
  - IntuiMessages, 162
  - message ports, 161
  - monitor task, 167
  - opening, 161
  - Requester features, 139
  - UserPort, 167
  - verification functions, 166
  - WindowPort, 167
- IDCMPFlags, 125
- IDNestCnt counter, 319
- IECLASS\_POINTERPOS, 698
- IFF, 629
- ILBM, 629
- Illegal instruction, 275
- Illustration Data Types, 173
- Image Structure, 100

- Image structure, 584
- ImageData pointer
  - changing Bobs, 460
  - changing VSprites, 440
  - in Bobs, 455
  - in VSprite structure, 437
- Images
  - data, 181
  - defining, 180
  - displaying, 174, 183
  - example, 185
  - Image structure, 184
  - location, 181
- ImageShadow member
  - in Bobs, 456
  - with OVERLAY flag, 449
- IND\_ADDHANDLER Command, 695
- IND\_REMHANDLER Command, 697
- IND\_SETPERIOD Command, 699
- IND\_SETTHRESH Command, 699
- IND\_WRITEEVENT Command, 697
- “.info” file, 581
- InitArea(), 359
- InitBitMap(), 56, 347
- InitialPC, 265
- InitMasks()
  - changing Bob image shadow, 461
  - defining collision mask, 467
  - with Borderline, 468
- InitRastPort(), 358
- InitResident(), 530
- InitStruct(), 246
- Inner window
  - in Gimmezerozero windows, 37
  - with the console device, 38
- Input device, 156
- Input Device
  - adding a handler, 695
  - commands, 694
  - designing an input handler, 696
  - event handler, 696
  - IOStdReq block, 695
  - generating input events, 697
  - key repeat events, 699
  - memory deallocation, 697
  - opening, 693
  - removing a handler, 697
  - setting key repeat interval, 699
  - setting key repeat timing, 699
- Input event, 156
- Input Event Chain, 696
- Input Event Structure, 694
- Input Events
  - generators of, 697
  - Intuition handling of, 695
- Input handler, 227
- Input Request Block, 697
- Input stream, 156, 227
- InputHandler(), 662
- Input/output
  - IDCMP, 161
  - Input device, 156
  - input stream, 156
  - paths, 157
- Insert(), 250
- INTEN interrupts, 307
- INTENA register, 306
- Interchange File Format, 629
- Interlaced mode, 16
- international compatibility, 914
- Interrupt, 893
- Interrupt structure, 308
- interrupts
  - 68000 interrupt request signals, 306
  - 68000 priority levels, 306
  - autovectors, 306
  - deferred, 307
  - disable, 309
  - disabling, 319
  - handlers, 308-309
  - hardware registers, 306
  - non-maskable (NMI), 307
  - priorities, 306
  - server return value, 314
  - servers, 308, 314
  - software, 316
- Inter-system communication, 279
- INTREQ register, 306
- IntuiMessage, 89
- IntuiMessage structure, 162
- IntuiMessages, 162
- IntuiText, 59
- IntuiText Structure, 100
- IntuiText structure, 179
- Intuition, 429
  - as input device handler, 695
  - mouse input, 694
- IntuitionBase, 199
  - and other System locks, 199
  - private fields of, 199
- INVERSVID, 401
- INVERSVID mode
  - in drawing, 362
- invisible pointer, 59
- I/O
  - asynchronous, 292, 295
  - performing, 294
  - quick I/O, 296

- synchronous, 291, 294
- I/O commands
  - abort all I/O requests, 293
  - clear internal buffers, 293
  - continue after a stop, 293
- I/O Commands
  - definition of, 290
- I/O commands
  - force out internal buffers, 293
  - non-standard, 291
  - read from a device unit, 293
  - reset the device unit, 293
  - standard, 293
  - stop device unit, 293
  - when errors occur, 295
  - write to a device unit, 293
- IO Request Block, 539
- I/O requests
  - completion, 295
- I/O Requests
  - definition of, 290
- I/O requests
  - multiple, 295
- IOAudio, 609
- IOClipReq, 629
- IODRPREq structure, 745
- IOExtSer Structure, 860
- IOExtTD Structure, 886
- IOPrtCmdReq structure, 745
- IORequest structures, 290
- IOStdReq structure, 291, 641
- io\_TermArray, 864
- ItemAddress(), 117, 125
- JAM1, 400
- JAM1 mode
  - in drawing, 361
  - with INVERSVID, 362
- JAM2, 400
- JAM2 mode
  - in drawing, 361
- joystick controller, 685
- KBD\_ADDRESETHANDLER command, 708
- KBD\_READEVENT command, 712
- KBD\_READMATRIX command, 706
- KBD\_REMRESETHANDLER command, 709
- KBD\_RESETHANDLERDONE command, 709
- KCmpStr(), 919
- Keyboard
  - as alternate to mouse, 193
  - command keys, 192
- keyboard device
  - keyboard events, 705
  - system functions, 706
- Keyboard Layout, 658

- keymap
  - dead-class keys, 669
- KeyMap structure, 663
- keymap.h, 665
- keymap.i, 665
- KGetChar(), 919
- KGetNum(), 919
- KMayGetChar(), 919
- KPrintF(), 919
- KPutChar(), 919
- KPutStr(), 919
- LACE flag
  - in View and ViewPort, 343
  - in ViewPort, 340
- Last-In-First-Out (LIFO), 251
- layer refresh
  - simple refresh, 511
  - smart refresh, 512
  - SuperBitMap, 512
- LAYERBACKDROP flag, 512
- Layer\_Info structure, 507, 513
- layers
  - accessing, 509-510
  - backdrop, 512
  - blocking output, 510
  - clipping rectangle list, 514
  - creating, 508, 513
  - creating the workspace, 513
  - deleting, 508
  - moving, 508
  - order, 509
  - redrawing, 510
  - scrolling, 508
  - sizing, 508
  - sub-layer operations, 510
  - updating, 510
- Layers Library
  - introduction, 505
  - opening, 512
- LAYERSIMPLE flag, 511
- LAYERSMART flag, 511
- LAYERSUPER flag, 511
- LEFTHIT flag, 470
- leftmost member
  - GelsInfo structure, 436
  - in Bob/VSprite collisions, 470
- libraries
  - adding, 234
  - caching a pointer, 233
  - calling a library function, 232
  - CLOSE vector, 235
  - definition of, 229
  - EXPUNGE vector, 235
  - OPEN vector, 235

- relation to devices, 237
- RESERVED vector, 235
- libraries/configvars.h, 528
- libraries/filehandler.h, 545
- Library
  - romtag, 234
- library
  - version, 231
- Library structure, 236
- Line 1010 emulator, 275
- Line 1111 emulator, 275
- line drawing, 365
- line pattern, 362
- LinePtrn variable, 367
- Lines
  - Border structure, 177
  - colors, 176
  - coordinates, 175
  - defining, 175
  - displaying, 174
  - drawing modes, 176
  - linking, 176
- lines
  - multiple, 366
  - patterned, 366
- List structure, 255
- List Structures, 249
- Lists
  - empty lists, 257
  - prioritized insertion, 251
  - scanning a list, 258
  - searching by name, 252
  - shared lists, 260
- LoadRGB4(), 348
- LoadSeg, 539
- LoadSegBlock, 538-539, 544
- LoadSegment(), 410
- LoadView(), 209
  - effect of freeing memory, 353
  - in display ViewPorts, 350
- Locking, 273
- LockLayer(), 510
- LockLayerInfo(), 509
- LockLayers(), 510
- LOFCprList variable, 356
- logic equations, 377
- long-frame Copper list, 356
- Low-resolution mode, 16
- main.c, 7
- MakeDosNode(), 530, 533, 539
  - parameter packet, 531
- MakeLibrary(), 234
- MakeScreen(), 25, 209
- MakeViewPort()
  - and Simple Sprites, 429
- MakeVPort(), 209, 349
  - allocating memory, 353
  - in double-buffering, 356
  - in dual playfield display, 354
- Manufacturer ID, 528
- Mask variable, 360, 376
- Masked Boolean Gadget, 100
- Masking interrupts, 272
- MatchToolValue(), 587
- math library, 547
- mathffp.library, 549
- mathieeedoubbas\_lib.lib, 563
- mathieeedoubbas.library, 563
- MathIeeeDoubTransBase, 568
- mathieeedoubtrans\_lib.lib, 569
- mathieeedoubtrans.library, 568
- mathtrans.library, 554
- maxx variable, 374
- maxy variable, 374
- MeMask member, 468
- memblock pointer, 372
- MemChunk structure, 248
- MemEntry structure, 244-245
- MEMF\_CHIP, 240
- MEMF\_CLEAR, 240
- MEMF\_FAST, 240
- MEMF\_PUBLIC, 240
- MemHeader structure, 247
- MemList structure, 243, 245
- memory
  - allocation, 200
  - allocation for BitMap, 347
- Memory
  - allocation within interrupt code, 241
  - clearing, 240
- memory
  - clearing, 372
- Memory
  - custom chips, 57
- memory
  - deallocation, 200
  - deallocation of, 360
- Memory
  - deallocation within interrupt code, 241
  - fast, 240
- memory
  - for area fill, 359
- Memory
  - free, 239, 248
- memory
  - freeing, 353
- Memory
  - location of, 240



- memory
  - loss, 912
  - problems, 910
- Memory
  - public, 240
- memory
  - Remember structure, 201
  - RememberKey, 201
- Memory
  - size
    - allocation, 239
    - deallocation, 239
    - special-purpose chip, 240
- memory.h, 243, 247
- memory.i, 247
- Menu boxes
  - item, 112
  - subitems, 112
- Menu checkmark, 113
- Menu commands
  - actions, 110, 113
  - attributes, 110, 113
- Menu Items
  - command key shortcuts, 193
- Menu items
  - enabling and disabling, 116
- Menu numbers
  - how they work, 117
  - how to get them, 116
- Menu selection
  - by user, 111
- Menu strips
  - changing, 116
  - removing, 111
  - submitting, 111
- Menu structure, 120
- Menu style
  - edit menus, 213
  - project menus, 213
- Menu system
  - activating, 110
- MenuItem structure, 121
- Menus
  - command-key sequences, 115
  - enabling and disabling, 116
  - intercepting operations
    - MenuVerify, 119
    - RMBTRAP, 120
  - menu numbers, 116
  - Menu structure, 120
  - MenuItem structure, 121
  - mutual exclusion, 113
  - requesters, 120
- menus.c, 125
- menus.h, 125
- MENUTOGGLE, 114
- Message, 279
- Message arrival action, 280
- Message ports
  - creation, 281
  - deletion, 282
  - public, 281
- Messages
  - getting, 286
  - IDCMP, 162
  - menu selection, 116
  - mouse, 191
  - putting, 283
  - replying, 286
  - waiting for, 284
- MICROHZ Timer Unit, 872
- MinList structure, 254
- MinNode structure, 253
- minterm variable, 377
- Misc Resource, 905
- misc.resource, 905
- Modes variable
  - in View structure, 345
  - in ViewPort, 339-340
- ModifyIDCMP(), 139, 161, 163, 167-168
- ModifyProp(), 82, 87
- modulo, 375
- Mount, 538-539
- Mouse
  - basic activities, 190
  - combining buttons and movement, 190
  - dragging, 191
  - keyboard as alternate, 193
  - left (select) button, 190
  - messages, 191
  - right (information transfer) button, 190
  - style, 218
- Mouse Button, 695
- mouse button events, 662
- Mouse Button Events, 693
- Mouse Button Philosophy, 190
- mouse controller, 684
- Mouse Movement Events, 693
- mouth structure, 719
- Move(), 365, 396
- MoveLayer(), 508
- MoveLayerInFrontOf(), 509
- MoveScreen(), 24
- MoveSprite(), 210, 430
- MoveWindow(), 56
- MrgCop(), 209, 356
  - in graphics display, 349
  - installing VSprites, 443

- merging Copper lists, 353
- MsgPort, 611
- MsgPort structure, 280
- Multiple Clips, 630
- Multitasking, 261
- Mutual exclusion, 273
- mutual exclusion, 309
- Mutual Exclusion
  - in gadgets, 79
- Mutual exclusion
  - in menus, 113
- narrator device
  - Arpabet, 721
  - consonants, 722
  - content words, 723
  - contractions, 722
  - controlling speech characteristics, 718
  - function words, 723
  - improving intelligibility, 725
  - introduction, 715
  - mouth shape, 719
  - opening, 717
  - phonemes, 721
  - phonetic spelling, 721
  - punctuating phonetic strings, 721
  - punctuation, 724
  - reading and writing, 719
  - recommended stress values, 724
  - special symbols, 722
  - speech synthesis system, 726
  - stress and intonation, 723
  - stress marks, 723
  - translator library, 716
  - vowels, 721
- narrator.h, 719
- narrator.i, 719
- narrator\_rb structure, 718
- nested disabled sections, 319
- NewFontContents(), 409
- NewLayerInfo(), 507
- NewList(), 256, 917
- NewModifyProp(), 82, 87, 99
- NewRegion(), 515
- NewScreen
  - SPRITE flag, 429
- NewScreen structure, 20
- NewWindow structure, 47, 583
- Next variable, 347
- NextComp pointer
  - in sequenced drawing, 475
- NextSeq pointer
  - in sequenced drawing, 476
- NMI interrupts, 307, 314
- Node structure, 253

- Nodes
  - initialization, 254
  - inserting, 250
  - priority, 254
  - removing, 250
  - successor and predecessor, 253
  - text names, 254
  - type, 254
- NO\_ICON\_POSITION, 584
- Non-interlaced mode, 16
- NOTEQUAL status code, 380
- ObtainConfigBinding(), 530
- OffGadget(), 81-82, 93, 214
- OffMenu(), 116, 121, 123-124, 212
- ON\_DISPLAY macro, 443
- OnGadget(), 81-82, 93
- OnMenu(), 116, 121, 123-124
- ON\_SPRITE macro, 443
- Open(), 744
- O-Pen
  - see AOIPen, 361
- OpenConsole(), 640-642
- OpenDevice(), 294, 296, 609-610, 628, 640, 642, 662
- OpenDiskFont(), 398, 403, 412
- OpenFont(), 398, 412
- OpenLibrary(), 230, 234
- OpenScreen(), 23, 35
- OpenWindow(), 35, 167
- OpenWorkBench(), 14, 25, 218
- OrRectRegion(), 517
- OrRegionRegion(), 518
- outline mode, 369
- outline pen, 361
- OVERLAY flag
  - in VSprite structure, 449
- Overscan
  - Effect on the Viewing Area, 329
- OwnBlitter(), 379
- PAL, 528
- parallel device
  - errors, 742
  - PDCMD\_SETPARAMS, 740
  - flags, 741
  - opening, 738
  - termination of reads, 740
- PartitionBlock, 538-539, 542
- Partitions, 538
- Paste, 627
- Paula, 306
- Pending Post, 630
- Period, 608
- Permit(), 271
- PFBA flag
  - in dual playfield mode, 342

- in ViewPort, 340
- Philosophy, 1, 5
- Pixel, 16
- pixel width, 342
- PlaneOnOff member
  - changing Bob color, 461
  - in Bobs, 457
- PlanePick member
  - changing Bob color, 461
  - in Bobs, 456
- PLANEPTR, 347
- PlayField Animation
  - introduction, 423
- Pointer
  - custom, 57
  - messages, 46
  - position in gadgets, 78
  - position in Gimmezerozero windows, 38
  - position in windows, 46
  - variables, 46
- Polling, 263
- PolyDraw(), 366
- polygons, 367
- Port, 279
- PORTS interrupts, 307, 314
- ports.h, 280, 283
- ports.i, 280, 283
- Post, 627, 629-630
- POTGO Resource, 907
- potgo.resource, 907
- power\_of\_two variable, 363
- PRD\_DUMPRT, 752
- Precedence, 608
- preemptive task scheduling, 306
- Preferences
  - getting user settings, 204
  - structure, 205
- Prep, 538
- Preserving the display
  - Simple Refresh, 42
  - Smart Refresh, 43
  - SuperBitMap, 44
- PrevComp pointer
  - in sequenced drawing, 475
- PrevSeq pointer
  - in sequenced drawing, 476
- PrintCommand(), 749
- PrintDevCommand(), 747
- printer device
  - alphanumeric drivers, 772
  - buffer deallocation, 769
  - changing printer preferences, 762
  - closing DOS printer device, 745
  - command definitions, 749
  - CommandTable, 772
  - creating an I/O request, 746
  - creating drivers, 763
  - data structures, 745
  - direct use, 744
  - double buffering, 767
  - dumping a RastPort, 752
  - dumping buffer, 768
  - Exec printer I/O, 745
  - getting status, 760
  - graphics printer drivers, 766
  - handling printer error codes, 755
  - opening, 746
  - opening AmigaDOS printer device, 744
  - Preferences, 765, 773
  - printing with corrected aspect ration, 753
  - processes and tasks, 745
  - PRT:, 749
  - sending I/O commands, 747
  - sending printer commands, 749
  - strip printing, 759
  - timeout, 765
  - using directly, 745
  - writing text, 748
- printer driver, 743
  - character conversion routine, 775
  - DoSpecial(), 772
  - example source code, 777
  - extended character table, 774
  - printertag.asm, 771
  - Render(), 766
  - SetDensity(), 771
  - testing, 776
  - Transfer(), 769
- printer special flags, 753
- PrinterData structure, 765
- printerIO structure, 745
- printf(), 918
- PrintIText(), 174
- Privilege violation, 275
- Process, 262
- Processes, 265
- Processor
  - halting, 263
  - interrupt priority levels, 272
- Product ID, 528
- PropInfo Structure, 95
- Proportional Gadget, 100
- Public memory, 240
- putchar(), 918
- PutDiskObject(), 581
- PutMsg(), 283
- QBlit()
  - linking bltnodes, 380

- waiting for the blitter, 379
- QBSBlt()
  - avoiding flicker, 379
  - linking bltnodes, 380
  - waiting for the blitter, 379
- Quantum, 262
- quartzWindow.h
  - IDCMPDemo.c, 168
- QueueRead(), 647
- Queues, 251
- RangeRand(), 918
- RasInfo structure, 345
- RASSIZE macro, 345
- raster
  - depth, 339
  - dimensions, 344
  - in dual-playfield mode, 340
  - memory allocation, 345
  - one color, 372
  - RasInfo structure, 345
  - scrolling, 373
- RastPort
  - in layers, 511
  - pointer to, 364
- rastport variable, 374
- rastport.h, 358
- rastport.i, 358
- RawKeyConvert(), 662
- RBF interrupts, 307
- ReadPixel(), 365
- rectangle fill, 370
- rectangle scrolling, 373
- RectFill(), 370
- RefreshGadgets(), 82
- RefreshGadgets, 99
- RefreshGList(), 82, 99
- RefreshWindowFrame(), 55
- regions
  - changing, 517
  - clearing, 518
  - creating, 515
  - removing, 515
- register parameters, 309
- registration point, 478
- ReleaseConfigBinding(), 530
- RemakeDisplay(), 26, 209
- RemBob(), 459
- Remember structure, 201
- Remembering, 200
- RemFont(), 410
- RemHead(), 251
- RemIBob(), 459
- RemIntServer(), 314
- Remove(), 251

- REMOVE macro, 257
- RemoveGadget(), 79
- RemoveGList(), 98
- RemPort(), 282
- RemTail(), 251
- RemTask(), 268
- RemTOF(), 917
- RemVSprite(), 439
- Rendezvous, 282
- Reply(), 162, 165
- Replying, 279, 286
- ReplyMsg(), 166, 286
- ReplyPort, 641
- ReportMouse(), 46, 54
- Request(), 54
- Requester structure, 141
- Requesters, 135
  - an alternative to, 137
  - as menus, 120
  - as Super Menus, 135
  - displaying, 136, 138
  - Double-Menu, 138
  - gadgets in, 138
  - IDCMP features, 139
  - keyboard equivalents in, 139
  - low-memory situations, 143
  - nesting, 137
  - on custom screens, 138
  - positioning, 136
    - POINTREL, 138
  - rendering, 137
  - simple, 139, 144
  - structure, 141
  - “verify” dead-lock, 139
- Resident
  - structure, 234
- resources/filesysres.h, 545
- RethinkDisplay(), 25, 209
- RHeight, 344
- RIGHTHIT flag, 470
- rightmost member
  - GelsInfo structure, 436
  - in Bob/VSprite collisions, 470
- RigidDiskBlock, 538
- RINGTRIGGER flag
  - in AnimComps, 478
  - in linking AnimComps, 479
  - moving registration point, 478
- RingXTrans variable
  - in ring processing, 479
  - moving registration point, 478
- RingYTrans variable
  - in ring processing, 479
  - moving registration point, 478

- ROM Diagnostic Routine, 533
- ROM Image, 533
- ROM Protocol, 531
- romtag, 234
- ROMTAG Init Time, 537
- ROM-Wack, 321
- RTE instruction, 309
- RTS instruction, 265, 274, 309, 314
- RWidth, 344
- RxOffset variable
  - effect on display, 344
  - in RasInfo structure, 345
  - in ViewPort display memory, 343
- RyOffset variable
  - effect on display, 344
  - in RasInfo structure, 345
  - in ViewPort display memory, 343
- Sample, 608
- SatisfyMsg, 629
- SAVEBACK flag
  - in VSprite structure, 449
- SAVEBOB flag
  - changing Bobs, 461
  - in Bobs, 453
- SaveBuffer member
  - in saving background, 449
  - with SAVEBACK, 449
- SAVEPRESERVE flag, 454
- Scheduling, 262
- screen display memory, 15
- Screen structure, 23
- Screens
  - active, 13
  - color, 17
  - custom, 15
  - depth, 17
  - display modes, 16
  - gadgets
    - custom, 20
    - system, 12
  - height and width, 19
  - position
    - dimensions, 19
  - Screen structure, 23
  - title
    - current, 20
    - default, 20
    - effect of Backdrop window on, 20
  - Workbench, 13
- ScreenToBack(), 24
- ScreenToFront(), 24
- scrolling, 373
- ScrollLayer(), 508, 512
- ScrollRaster(), 373
- ScrollVPort(), 348
- SCSI, 539
- SCSI Identify Command, 541
- SDCMD\_SETPARAMS, 866
- Semaphore, 530
- Semaphores, 273, 300
- SendIO(), 292, 295, 611, 618, 628, 638
- SendIO, 647
- Serial Device
  - break conditions, 864
  - buffers, 861
  - closing, 861
  - EOF mode, 864
  - error codes, 869
  - flags on open, 860
  - introduction, 859
  - IO request structures, 860
  - io\_TermArray, 864
  - multiple ports, 869
  - parameter changes, 866
  - Quick IO, 863
  - reading, 862
  - SDCMD\_SETPARAMS, 866
  - separate tasks, 865
  - serial flags, 868
  - terminating the read, 864
  - writing, 861
- SetAPen(), 400
- SetBPen(), 400
- SetCollision(), 466
- SetCurrentBinding(), 530
- SetDMRequest(), 54
- SetDrPt(), 366
- SetFont(), 398
- SetFunction(), 236
- SetIntVector(), 306, 310
- SetKeyMap(), 662-663
- SetMenuStrip(), 53, 111, 116, 124-125
- SetPointer(), 46, 54, 57, 59
- SetRast(), 372
- SetRGB4(), 17, 46, 219
- SetRGB4CM(), 348
- SetSignal(), 270
- SetSoftStyle(), 402
- SetTaskPri(), 264
- SetWindowTitles(), 20, 55
- SHFCprlist variable, 356
- Shortcuts
  - menu, 217
  - selection, 216
- short-frame Copper list, 356
- ShowTitle(), 16, 20, 23, 39
- SigExcept(), 274
- Signal(), 271

- Signal bit number, 280
- Signal Semaphore, 300
- Signals
  - allocation, 269
  - coordination, 269
  - exception, 274
  - on arrival of messages, 280
  - waiting for, 270
- Simple Refresh, 42
- Simple Sprite
  - allocation, 429
  - colors, 428
- Simple Sprite colors
  - and ViewPorts", 428
- Simple Sprites
  - functions, 429
  - GfxBase, 441
  - in Intuition, 429
  - position, 428
  - simple definition, 422
- SimpleSprite structure, 428
- single-buffering, 345
- SizeLayer(), 508, 512
- SizeWindow(), 56
- Sizing Gadget, 74
- Smart Refresh, 43
- SOFTINT interrupts, 307
- software clipping
  - in filling, 368
  - in line drawing, 366
- Software interrupts, 280-281
- software interrupts, 305, 307, 316
- SortGList()
  - changing Bobs, 461
  - ordering GEL list, 442
  - with DoCollision(), 481
- Sound, 607
- Sound Synthesis, 608
- source variable, 376
- SpecialInfo Structures, 94
- speech
  - see narrator device, 716
- speech output
  - introduction, 715
- SprColors pointer
  - changing VSprites, 440
  - in VSprite structure, 438
  - in VSprite troubleshooting, 441
- sprintf(), 918
- Sprite
  - pairs, 428
- Sprite Animation
  - introduction, 423
- sprite data, 210
- Sprite DMA, 441
- Sprite mode, 17
- Sprite pointer
  - colors, 219
- sprites, 210
- Sprites
  - as pointer, 46
- sprites
  - color, 341
- Sprites
  - data memory location, 57
- sprites
  - display, 338
- Sprites
  - in Intuition windows & screens, 210
  - reserving, 440
- sprRsrvd GelsInfo member
  - in reserving Sprites, 440
- srcMod variable, 376
- srcX variable, 376
- stack
  - overflow, 910
- Stack overflows, 268
- stencil drawing, 374
- Stereo, 615
- String Gadget, 100
- StringInfo Structure, 96
- Structure
  - CurrentBinding, 530
  - DiagArea, 532
  - ExpansionRom, 532
  - IOAudio, 609
  - IOClipReq, 629
  - Keymap, 663
  - SatisfyMsg, 629
- Structures
  - access to global system structures, 271
  - BoolInfo, 94
  - Border, 177
  - Gadget, 90
  - Image, 184
  - IntuiMessage, 162
  - IntuiText, 179
  - Menu, 120
  - MenuItem, 121
  - NewScreen, 20
  - NewWindow, 47
  - Preferences, 205
  - PropInfo, 95
  - Requester, 141
  - Screen, 23
  - shared, 271
  - SpecialInfo, 94

- StringInfo, 96
- Window, 52
- Submenu, 212
- SubTime(), 875
- SuperBitMap refresh, 44
- SuperBitMap window
  - description, 39
  - setting up the BitMap, 56
- Supervisor modes, 275
- supervisor modes, 306, 308
- SwapBitsRastPortClipRect(), 510
- SYS:Expansion, 529
- sysgads.h, 7
- System Gadgets
  - placement, 73
- System gadgets
  - screens, 12
  - windows, 39
- System Initialization, 528
- System stack, 275
- system stack, 308
- System Time, 876
- Task, 261
- Task signal, 280
- Task structure, 264
- Task-private interrupts, 274
- task-relative interrupts, 305
- Tasks
  - cleanup, 268
  - communication, 269
  - coordination, 269
  - creation
    - initialPC, 265
    - stack, 265
  - deallocation of system resources, 268
  - finalPC, 268
  - forbidding, 271
  - initialPC, 268
  - non-preemptive, 271
  - priority, 264
  - queues
    - ready queue, 264
    - waiting queue, 264
  - scheduling
    - non-preemptive, 262
    - preemptive, 262
  - stack
    - minimum size, 268
    - overflows, 268
    - supervisor mode, 267
    - user mode, 267
  - states
    - added, 263
    - exception, 263
    - removed, 263
    - running, 263
    - waiting, 263
    - termination, 268
- tasks.h, 264
- tasks.i, 264
- TBE interrupts, 307
- tc\_MemEntry, 246
- TD\_ADDCHANGEINT Command, 893
- TDB\_ALLOW\_NON\_3\_5, 887-888
- TD\_CHANGENUM Command, 892
- TD\_CHANGESTATE Command, 892
- TDF\_ALLOW\_NON\_3\_5, 887-888
- TD\_FORMAT Command, 891
- TD\_GETDRIVETYPE Command, 892
- TD\_GETNUMTRACKS Command, 893
- TD\_MOTOR Command, 890
- TD\_PROTSTATUS Command, 892
- TD\_RAWREAD Command, 894
- TD\_RAWWRITE Command, 895
- TD\_REMCHANGEINT Command, 894
- TD\_SEEK Command, 896
- testing, 914
- Text, 395
- Text(), 397, 403
- text
  - baseline, 396
  - BOLD, 402
  - changing font style, 402
  - character data, 412
  - character kern, 413
  - character location, 413
  - character space, 413
  - color, 400
- Text
  - colors, 178
- text
  - cp\_x variable in, 396
  - cp\_y variable in, 396
  - cursor position, 396
  - default fonts, 398
- Text
  - defining, 178
- text
  - defining fonts, 410
  - disk fonts, 410
- Text
  - displaying, 174, 179
  - drawing modes, 178
- text
  - DrawMode
    - COMPLEMENT, 400
    - INVERSVID, 401
    - JAM1 mode, 400

- JAM2 mode, 400
- example
  - FontParade, 404
  - ShowDefaultFont, 397
  - ShowDrawModes, 401
  - ShowOpenFont, 399
  - suits8, 414
  - Wrapper, 417
- EXTENDED, 402
- Text
  - fonts, 179
- text
  - InitRastPort() in, 399
  - inter-character spacing, 403
- Text
  - IntuiText structure, 179
- text
  - ITALIC, 402
- Text
  - linking, 179
- text
  - modulo, 412
  - NORMAL, 402
  - printing, 397
  - range of characters, 412
  - RastPort structure in, 395
  - relative font path, 403
  - selecting a font, 398
- text to speech
  - introduction, 715
- TextAttr structure, 19, 399
- TextFont structure, 410
  - Node structure in, 410
- TextLength(), 408
- Time
  - getting current values, 209
- Time Events, 693
- Timer Device
  - arithmetic routines, 877
  - OpenDevice(), 873
  - units, 871
- Timer variable, 478
- TimerBase Variable, 877
- timerequest Structure, 872
- TimeSet member
  - of AnimComp structure, 478
- Time-slicing, 262
- timeval Structure, 872
- Title
  - screen, 20
  - window, 37, 41
- Title bar
  - screen, 12
  - window, 41
- toggle selection, 113
- toggle-select, 114
- ToolType
  - expansion, 530
  - PRODUCT, 530
- ToolTypes array, 586
- Topaz font, 18
- TOPHIT flag, 470
- topmost member
  - GelsInfo structure, 436
  - in Bob/VSprite collisions, 470
- Trace, 275
- Trackdisk Device
  - status commands, 891
- trackdisk.device
  - diagnostic commands, 896
  - diskchange, 893
  - diskcopy code, 897
  - error codes, 897
  - low-level access, 894
  - OpenDevice(), 887
- Translate(), 575, 716
  - output buffer, 576
- translator library, 575
  - exception rules, 576
- TRAP
  - address error, 275
  - bus error, 275
  - CHK instruction, 275
  - illegal instruction, 275
  - line 1010 emulator, 275
  - line 1111 emulator, 275
  - privilege violation, 275
  - trace, 275
  - trap instructions, 275
  - TRAPV instruction, 275
  - zero divide, 275
- TRAP instruction, 268
- Traps, 274
  - instructions, 277
  - supervisor mode, 275
  - trap handler, 275
- TRAPS (68000 exceptions), 323
- TRAPV instruction, 275
- TR\_GETSYTIME, 876
- TR\_GETSYTIME Command, 875
- TR\_SETSYTIME Command, 875-876
- twowindows.c, 59
- Type styles, 18-19
- TypeOfMem(), 243
- Unit Number, 630
- UnloadSegment(), 410
- UnlockLayer(), 510
- UnlockLayers(), 510



- UpfrontLayer(), 509
- User settings
  - Preferences, 204
- UserExt member, 471
- VBEAM counter, 381
- VBLANK Timer Unit, 872
- Verification functions
  - IDCMP, 166
- VERTB interrupts, 307, 314
- Vertical Blank Frequency
  - find current VB frequency, 687
- video priority
  - in dual-playfield mode, 340
- View
  - remaking, 209
- View structure
  - Copper lists in, 356
  - function, 335
  - preparing, 346
- ViewAddress(), 188
- ViewPort
  - colors, 339, 348
  - display instructions, 349
  - display memory, 343
  - displaying, 336
  - function, 335
  - height, 338
  - interlaced, 343
  - low-resolution, 347
  - modes, 339-340
  - multiple, 347
  - parameters, 337
  - remaking, 209
  - width, 338
  - width of and sprite display, 339
- ViewPort structure, 346
- ViewPortAddress(), 188
- ViewPorts
  - and Simple Sprite colors, 428
- Virtual display memory, 42
- Virtual terminal, 2, 33
- Virtual terminal windows, 34
- VP\_HIDE flag, 341
- VSOVERFLOW Flag
  - in VSprite structure, 436
- VSOVERFLOW flag
  - reserving Sprites, 440
- VSPRITE flag
  - in VSprite structure, 449
- VSprite Flags
  - and True VSprites, 436
- VSprites
  - adding new features, 471
  - building the Copper list, 443
  - changing, 440
  - color, 438
  - hardware Sprite assignment, 441-442
  - merging instructions, 444
  - PlayField colors, 442
  - position, 436
  - shape, 437
  - simple definition, 422
  - size, 437
  - sorting the GEL list, 442
  - troubleshooting, 441
  - turning on the display, 443
- Wack, 321
- Wait(), 165-166, 263, 270-272, 285, 295, 609, 611
- WaitBlit(), 364, 372, 379
- WaitBOVP(), 353
- WaitIO(), 292, 295
- WaitPort(), 285, 609, 611
- WaitTOF(), 353, 444
- Waveform, 608
- WBENCHSCREEN, 125
- WBenchToBack(), 25
- WBenchToFront(), 25
- WhichLayer(), 509
- Width variable
  - in VSprite structure, 437
- Width VSprite member
  - and Bobs, 454
- Window structure, 52, 637
- WindowLimits(), 47, 51, 55
- Windows
  - activating, 36
  - application gadgets in, 41
  - Backdrop, 38
  - Borderless, 37
  - closing, 36
  - colors, 47
  - dimensions
    - limits, 47
    - starting dimensions, 47
  - Gimmezerozero, 37
  - graphics and text in, 47
  - input/output, 35
  - NewWindow structure, 47
  - opening, 35
  - pens, 47
  - pointer, 46
  - preserving the display, 42
  - refreshing the display
    - NOCAREREFRESH, 46
    - Simple Refresh, 42
    - Smart Refresh, 43
    - SuperBitMap, 44
  - screen title, 20

- special types, 37
- SuperBitMap, 39
- system gadgets in, 39
- Window structure, 52
- WindowToBack(), 56
- WindowToFront(), 56
- Workbench
  - application program, 14
  - “.info” file, 581
  - library, 14
  - sample startup code, 594
  - screen, 13
  - shortcut key functions, 193
  - start-up code, 593
  - start-up message, 585, 593
  - ToolTypes, 586
- Workbench object, 579
- WritePixel(), 365
- XAccel variable, 482
- Xerox\_4020
  - Example source files, 838
- x1 variable, 374
- xmax variable, 370
- xmin variable, 370
- XorRectRegion(), 518
- XorRegionRegion(), 518
- XTrans AnimComp member, 477
- XVel variable, 482
- YAccel variable, 482
- yl variable, 374
- ymax variable, 370
- ymin variable, 370
- YTrans AnimComp member, 477
- YVel variable, 482
- Zero divide, 275



**Amiga® Technical Reference Series**

# **AMIGA® ROM KERNEL REFERENCE MANUAL: LIBRARIES & DEVICES REVISED & UPDATED**

The Amiga computers are exciting high-performance microcomputers with superb graphics, sound, multiwindow and multitasking capabilities. Their technologically advanced hardware is designed around the Motorola 68000 microprocessor family and sophisticated custom chips that control graphics, audio, peripherals, and input/output to other equipment. The Amiga's unique operating system software provides programmers with unparalleled power, flexibility, and convenience in designing and creating programs.

Written by the technical experts who design the Amiga hardware and system software, **AMIGA ROM KERNEL REFERENCE MANUAL: LIBRARIES AND DEVICES** is now revised and updated for system software versions 1.2 and 1.3. It is the essential reference tool for all Amiga programmers. It gives an in-depth and thorough explanation of the powerful graphics, animation, text, math, and audio routines that make up the Amiga's ROM.

This volume includes:

- a tutorial and comprehensive reference to the Amiga's libraries and devices, now including both the multitasking Exec, and Intuition, the Amiga's graphical user interface
- numerous working example programs illustrating the use of the Amiga's ROM routines
- all new coverage of the system routines added in the V1.2 and V1.3 revisions of the Amiga's operating system
- new material on Exec semaphores, CIA resources, the Amiga's expansion library, ROM tags, support libraries, PAL-NTSC compatibility, and a user interface style guide.

For the serious programmer working in assembly language or C who wants to take full advantage of the Amiga's impressive capabilities, **AMIGA ROM KERNEL REFERENCE MANUAL: LIBRARIES AND DEVICES** is the definitive source of information on the internal design and workings of the Amiga's powerful system software.

The revised Amiga Technical Reference Series includes two other volumes of vital information for Amiga programmers and developers: **Amiga ROM Kernel Reference Manual: Includes and Autodocs** contains the Autodocs for Library, Device, and Resource calls and the C and assembly language Amiga Include Files. **Amiga Hardware Reference Manual** is an in-depth description of the Amiga's hardware and how it works.

Cover design by Mike Fender



Addison-Wesley Publishing Company, Inc.



TSBN 0-201-18187-A