

Tutorial

Mastering

AmigaDOS 3

Covers 2 and 2.1 too!

Complete Beginners Tutorial

Covers all AGA Amigas

BSB

Bruce Smith Books

Mastering AmigaDOS 3

Tutorial

Mark Smiddy and Bruce Smith



Mastering AmigaDOS 3

Volume One

Tutorial

© Mark Smiddy and Bruce Smith 1993

ISBN: 1-873308-20-5 First Edition: December 1993.

Editors: Peter Fitzpatrick, Mark Webb

Typesetting: Bruce Smith Books Ltd

Workbench, Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc. UNIX is a trademark of AT&T. MS-DOS is a trademark of Microsoft Corporation. Designer Mouseware is a trademark of Mark Smiddy. All other Trademarks and Registered Trademarks used are hereby acknowledged.

All rights reserved. No part of this publication may be reproduced or translated in any form, by any means, mechanical, electronic or otherwise, without the prior written consent of the copyright holder(s).

Disclaimer: While every effort has been made to ensure that the information in this publication (and any programs and software) is correct and accurate, the Publisher can accept no liability for any consequential loss or damage, however caused, arising as a result of using the information printed in this book.

E. & O.E.

The rights of Mark Smiddy and Bruce Smith to be identified as the Authors of the Work has been asserted by them in accordance with the *Copyright, Designs and Patents Act 1988*.

Bruce Smith Books is an imprint of Bruce Smith Books Limited.

Published by:

Bruce Smith Books Limited. PO Box 382, St. Albans, Herts, AL2 3JD.
Telephone: (0923) 894355, Fax: (0923) 894366

Registered in England No. 2695164.

Registered Office: Worplesdon Chase, Worplesdon, Guildford, Surrey, GU1 3UA.

Printed and bound in the UK by Ashford Colour Press, Gosport.

*"Experts are not born.
They are hewn from the bedrock of endeavour,
and the granite of experience."*

The Authors

MARK SMIDDY is a founding Consultant Editor on Future Publishing's, acclaimed *Amiga Shopper* and the world's best known AmigaDOS author. He has worked with and written about, a variety of computers over the last 12 years and thinks intellectual bores should be lined up against a wall and shot: not necessarily in that order. Mark currently lives in a sleepy Cleveland backwater with his girlfriend, four children, two hamsters, one bird and a migraine. Ear plugs are offered to personal callers at no extra cost.

BRUCE SMITH is an award winning journalist and author of over 50 books, many of which have been translated into a variety of languages. Author of the best-selling Amiga books in the world, his style is renowned for being easy going and highly readable, leading one reviewer to comment *"This is the first computer book I have read for pleasure rather than to cure insomnia."*

He is the founder of Bruce Smith Books, a publishing house dedicated to the support of the Amiga and lives in rural Hertfordshire. He is married with two children and his hobbies include scuba diving and amateur dramatics.

MAD3 – The Disk

Mastering AmigaDOS 3 – Tutorial contains a large number of programs. If you have an aversion to typing these in – or find that you can't get them to run correctly – then you may be interested to know that they are available on a companion disk we have compiled to go along with this book. In addition, the MAD3 Disk also contains a good selection of interesting and useful PD/Shareware software. The disk may also contain information that has come to light since this book was published.

The disk is available at a nominal charge of £1.50 to cover the cost of P&P and to obtain it simply fill in and return the tear-out form you will find towards the end of this book.

When you get your disk you will find that it contains a ReadMe file when the disk window is opened. Simply double-click on this for a full description of the files on the disk.

Contents

Introduction	15
Important.....	16
1: Strategy for Success	17
2: AmigaDOS and Workbench	19
Amiga Scripts.....	20
Version 2.....	21
Version 2.1.....	21
Version 3.....	21
3: First Principals, First Words	23
4: First AmigaDOS Commands	25
Letter Case	26
More Errors	27
5: Formatting and Copying Disks	29
Formatting a Disk	29
What the K?	31
Backing Up Disks	31
Single Drive Backup	32
Dual Drive Backup	32
Renaming Disks	33
Formatting from AmigaDOS.....	33
Single Drive AmigaDOS Formatting	35
Dual Drive AmigaDOS Formatting	35
Renaming Disks	35
Backing Up Disks	35
Single Drive DISKCOPY	36
Dual Drive DISKCOPY.....	36
6: Drawers and Directories	37
Drawers and Other Bloomers.....	38
The Tree Structure	39
Directory Paths	40
7: Drives and Filenames	41
Disk Drive Names	41
Filenames.....	42
8: Parsing and Synopses	45
Parsing	45
Modifiers.....	46
Split Decisions	47
Duality and Multiplicity.....	48

9: The Shell	51
The Arrow Keys.....	52
10: Shell Commands	55
The Problem.....	55
The Solution.....	56
Resident Commands	58
Where's Wally Now?	59
Whose Fault?	59
11: Shell's Special Features	61
Introducing Echo.....	61
Echoing Linefeeds.....	61
Starry, Starry Nights	62
Cue -> Shell	63
Alas, Poor AmigaDOS.....	64
ALIAS Effects	65
12: The Ram Disk	67
The Pros and Cons	67
Size and Use.....	68
Workbench Ram Disk	68
Amiga Reset	69
Ram Disk and Shell	70
13: Wildcards and Pattern Matching	71
14: Managing Directories	75
Using CD	76
New Directories	77
Directory Structure	78
Sub directories.....	79
Relative Paths	81
15: Managing Objects	83
Backups and Cock-ups	83
Duplicating Objects (COPY)	84
Simultaneous COPY and RENAME (at Destination).....	84
Multiple Copies, No Patterns	85
Simultaneous COPY and MAKEDIR.....	85
Copying Directories with Subdirectories.....	86
Changing an Object's Name (RENAME).....	87
Moving an Object (RENAME).....	87
Removing Objects (DELETE).....	89
16: Advanced Use of DIR	91
Combined Patterns	92
DIR Options.....	92
Interactive DIR	93

17: An Introduction to LIST	97
DIR and LIST	97
Using LIST	98
Limiting Output	99
Patterns in LIST	101
18: Basic System Control	103
Adding Buffers	104
Device Information	105
Which Version	106
19: Advanced Directories	107
Path Override	108
Locating Files	109
Assigned Directories	110
The Rules	110
Long Directory Names	112
Volume Names	113
Assign with Drives	115
20: Advanced Object Management	117
Protection Racket	117
A Practical Example	118
Linking Disk Objects	121
Time and Dates	122
Changing Dates	123
File Date	124
21: Advanced File Management	125
More Commands	126
More Piping	126
Examining Text Files	127
Examining Binary Files	128
Searching for Files	129
Searching in Files	129
Attaching Comments	130
Sorting to Order	131
Joining Files	132
22: Advanced System Control	133
Font Control	133
Keyboard Control	134
Printer Control	134
Processor Control	136
Memory Control	136
23: Introducing Re-direction	139
Output Re-direction	140
Hush Now	140

Output Re-direction (Append Mode).....	141
Input Redirection.....	141
24: Advanced LIST	143
The LIST Options	143
KEYS	143
KEYS	144
BLOCK	144
DATES	145
QUICK.....	145
ALL	145
LFORMAT	145
More Ss.....	146
Advanced LFORMAT.....	148
25: Multi-tasking Amiga	151
Pre-emptive What?	151
Who Does What?	152
Types of Multi-tasking.....	152
Co-operative Multi-tasking	153
A New Shell	154
Running Single Window Multi-tasking.....	154
26: Advanced Multi-tasking.....	157
Process Status.....	158
Stack and Priority.....	159
Priority	159
Quitting Processes	160
27: Environmental Variables	161
A Useful Variable	162
A Fistful of Dollars.....	163
For a Few Dollars More	163
ENV:	164
Writing Variables Directly.....	166
Using Environmental Variables in Calculations.....	166
Resetting Environmental Variables	168
Pro Bono Publico.....	168
28: Devices	171
CONSOLE:.....	172
RAM: and RAD:.....	172
NIL:.....	172
PIPE:	173
SPEAK:	175
PRT:.....	176
PAR:.....	176
SER:	177
RAW:.....	177

ENV:	178
AUX:	178
29: The Console Device.....	179
Copy and Paste	181
The Console and ANSI.....	182
30: ED – The Screen Editor	185
Creating a File.....	185
A Simple Script File.....	186
Executing Script Files.....	186
Editing ED Files	187
31: Introduction to Scripts	189
Executing Scripts	189
Some Simple Scripts	190
Script File Parameters	191
The Unbalanced Key	193
The Key Options	194
Defining Key Variables	194
Changing the Notion.....	196
Multi-tasking Scripts.....	196
32: Flow Control in Scripts	199
SKIP and LAB	199
IF Whats?	201
ELSE What?	202
Error Numbers and WARN, ERROR and FAIL	203
A Better Example.....	205
Asking Questions	206
Numbers With IF	206
33: Workbench and IconX	209
Workbench and Scripts	209
IconX	209
34: Evaluating and Manipulating	213
FORMATEd Input and Output.....	215
EVAL in Scripts.....	217
Program: LOOPER.....	220
Program: DUAL.....	221
Program: MULT	221
35: Startup-Sequences: An Introduction	223
Starting the Engine	225
Guru's Meditation	225
Ketchup on your Startup	226

36: AmigaDOS 3.0 Startup-Sequence	227
Explanation	227
37: AmigaDOS 2.1 Startup-Sequence	233
Explanation	233
38: AmigaDOS 2.04 Startup-Sequence	239
Explanation	239
39: Customising the Startup	245
Assigning Extra Paths	245
The Auto-Startup Directory	247
A Complete Example	247
Modifying the Workbench Startup-Sequence	248
ED-Startup File	248
ED Menu Examples	249
Menu Information	250
Designer Menus	251
40: Customising Disks	253
Creating System (Bootable) Disks	255
The Startup-Sequence File	255
An Example Startup Disk	255
Required Directories	256
The Root Directory	257
The Command Directory	258
The Devices Directory	259
The Libraries Directory	260
The Fonts Directory	260
The Handlers Directory	261
The Env-Archive Directory	261
The Sequences or Scripts Directory	261
Shell-Startup	262
Debugging	263
Inside the AmigaDOS Disk	264
41: Multi-User Machine	269
Remote Machine (Parasite)	269
Host Machine (File Server)	269
Getting Started	270
Testing 1-2-3	271
Problems, Problems	271
What Next?	272
Opening the Mail	274
Using HostRead	275
Commanding Scripts	276
Listing 1 Mail-2-Remote	276
Listing 2 Mail-2-Host	276
Listing 3 HostRead	277

Listing 4 RemoteRead	278
Debugging Chart	279
42: Pipes	281
Using Pipes	281
Paying the Piper	282
Naming Pipes	285
Program: Host-Chat.....	287
More Pipes	287
Program: CHATTER	288
Program: REMT-CHAT	290
Program: CATTY.....	290
43: The Recoverable Ram Disk.....	291
The Use of RAD:	292
Changing Size	292
Removing and Formatting	293
Booting from RAD:	294
Multiple RAD Disks	294
Wot No RAD Disk!.....	295
44: Commodities.....	297
Exchange	297
AutoPoint	299
Blanker.....	299
ClickToFront	300
CrossDOS	301
FKey	302
MouseBlanker.....	304
NoCapsLock	304
Using the Exchange Window	304
Commodity Tools	305
Last Words	305
45: Intellifont and Fountain.....	307
Fonts	307
Using Intellifont.....	308
Installing Outline Fonts	309
Font Installation.....	310
Bitmap Installation	312
Intellifonts Environmental Variables	313
A: MEMACS.....	315
Up and Running	316
Editing with MEMACS.....	317
The Project Menu	318
The Edit Menu	321
The Window Menu	325
The Move Menu.....	327

The Line Menu	328
The Word Menu	330
The Search Menu	331
The Extras Menu	334
Miscellaneous Commands	337
MEMACS Command Files	339
Macros	340
Macros and Function Keys	341
Startup Files	342
B: Original AmigaDOS 2 Startup-Sequence	343
The AmigaDOS 2 Startup-Sequence	343
C: ED's Commands	351
Cursor Control	351
ED's Extended Commands	353
Marking Blocks of Text	358
ED Menus	359
Default Menus	359
D: Mastering Amiga Guides	361
Command Index	375
Index	377

Preface

This is a story of chance. A chance meeting in a none too salubrious London pub that brought together two old colleagues. A chance remark by one revealed that he had been working for several months on a new AmigaDOS book but, since his time was now being taken with other projects, he needed a co-author. Before the other could so much as get his brain engaged, a third-party, Nigel Streeter (a colleague from Amiga User International) said "That's no problem Bruce, Mark's an expert on AmigaDOS – look no further." Before Mark could stop it, his chin landed with an almighty splash in the pint of Guinness he was embracing. And so our story opens...

Fireside tales aside, that was the start of a journey, one on which you too, are about to embark. But this is not a flight of fantasy, it's a journey of discovery. We are going to open your eyes and show you things which few people outside Commodore-Amiga Inc. ever dared contemplate.

This book cannot make you an expert, only you can do that – through effort and hard work – but if we can point you in the right direction, and guide you when the going gets tough, then we have achieved what we intended. There are no such things as experts in the imaginary sense. Expertise is relative to learning – and some people have learnt more than others. That's life: but at the end of the day, who cares? As long as you know what you need to know, that's all that counts. If you'd like to know more, then join us.

On a purely personal note we'd like to add a few words which may surprise some of you. As you may have gathered from the biographies on page three we do not possess university degrees. There isn't a particularly euphemistic way of saying what we're about to – so sensitive high-brows and graduates can progress to the rest of the book.

After more than eight years of working with and around graduates, we don't hold a very high opinion of conventional computer education. All too often computers are taught as some serious (or should that be mysterious) science. For us this conjures up images of strange little men huddled over keyboards in rooms approached by dark, forbidding corridors. Or brightly lit, air conditioned rooms, where the environment is created purely for the computers.

The sort of people churned out *en-masse* by such establishments are almost as sterile as the atmosphere they existed in. Almost as defunct of common sense as they are lacking in imagination; useless to the vast majority of the industry until we can break through the brain-washing. In serious cases, they're more like computers than the computers themselves.

Programming is an art and should be taught as such. A good program should be elegant, it should flow like a silk scarf on the summer breeze, but most of all, it must show imagination. This encompasses all the other considerations of "user-friendliness", "WSYIWYG" and such like. These are the sort of things which must go on under the surface and the user must be unaware of them.

Modern software exhibits too much of the sterility and cleverness bred into its authors, we suspect, when they were taught. Male chauvinists should look to women for the answer, for they excel in imagination in a way few men ever can – and some of the finest programmers and designers we know of are women! Commodore guru, Carolyn Sheppner and Gail Wellington, the co-inventor of original CDTV (in its latest form, the CD³²) are perfect examples.

We both learnt the art by scouring the manuals and experimenting with new ideas. During this quest we discovered (not surprisingly) a lot of Amiga manuals are terse, uninteresting, unbalanced, difficult to read or just plain inaccurate! This book attempts to redress the balance.

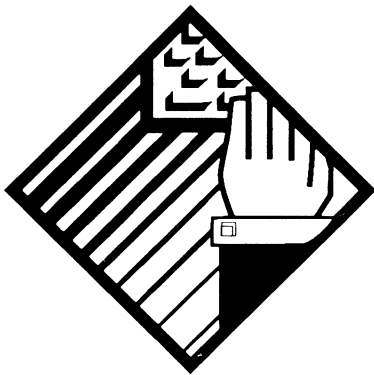
In writing this book we spent as much time experimenting with AmigaDOS and testing the examples, as we did describing them. Nevertheless, we'd be the first to admit we may have overlooked something – no one is perfect. Since AmigaDOS is constantly evolving, it's quite possible something in your system won't work precisely as we have described – this is the price we all pay for having a machine which never ceases to amaze.

In closing, just remember this: we were all beginners once. The key to learning is not to be afraid to experiment for yourself – that way, by your own mistakes you will learn the correct way. On this point we speak from much experience.

Life is an adventure and AmigaDOS is a fast train to the foot of a technicoloured rainbow. You're holding the tickets right now so what are you waiting for? Jump on and enjoy the ride...

Mark Smiddy and Bruce Smith

England, November 1993



Introduction

Mastering AmigaDOS 3 is based on the original texts of the critically acclaimed *Mastering AmigaDOS 2*. It's not just a simple re-hash – although that was the easy option – the whole book has been re-designed with more examples, more tutorial text and a more modular approach.

When we embarked on the project, we set ourselves the task of writing the best and most informative reference on the topic – in a field where the competition is pretty stiff (there's a lot of it), how good it is, is up to you to decide.

The similarity across versions means that many of the examples quote AmigaDOS 2.0 (or even 1.3) as a minimum requirement. This just means the command is good for use on a Workbench 2.0 machine too.

This two volume set does not just cover AmigaDOS 3 – it details versions 2.0 and 2.1 as well and almost all those commands supplied by Commodore which can be usefully executed from the Shell.

At this stage, that statement may not mean a lot but, if you already have our *Mastering AmigaDOS 3 – Reference*, just look at how many commands it covers! If it's relevant we've included it.

AmigaDOS is a very powerful, and at times complex, operating system. This, in part, is the reason the subject has been split into two volumes.

A single volume would be too big and unwieldy to be useful. Secondly – and more to the point

– it would be too expensive to be easily affordable. The division of the books is roughly like this:

Mastering AmigaDOS3 – Tutorial (this one) is here to get you started. It has all the tutorials and step-by-step instructions to lead you through some of the tricky bits. AmigaDOS by its very nature is highly complex so we have left more involved concepts and technical descriptions to the second volume. Never underestimate AmigaDOS until you see what can be done.

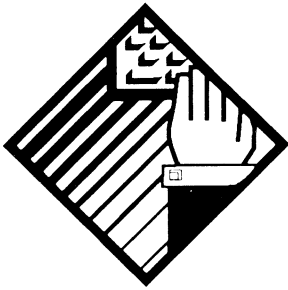
Mastering AmigaDOS3 – Reference (the other one) is primarily a reference work. It covers every single AmigaDOS command, plus all the System commands and Utilities, Preferences editors, Commodities and other goodies you probably won't find described anywhere else. It describes each command *in detail* with examples of usage – ideal if you know what you want but need clarification on how to do it. Due to the way these books are compiled (Reference first) some late breaking news is embedded within these pages too, so don't underestimate the value of the tutorial.

Important

Mastering AmigaDOS3 – Reference explains *every* option of every command it covers for each of the three versions, whereas this volume does *not*. Some experience is assumed – so if you're still new to this game, you should leave it for now. It will come in handy when you reach the later chapters on advanced topics.

Both books are fully compatible with all current (and some past) Amigas including the A500, A500 Plus, A600, A1200, CDTV computer, A1500, A2000, A3000 and A4000 provided they are running Kickstart 2 or higher with Workbench 2.x or 3.0. Fat women, extra memory and so forth are not required; but a second disk drive would be a bonus.

Mark Smiddy & Bruce Smith
England, October 1993



1: Strategy for Success

Learning AmigaDOS, like any new experience, is going to be difficult. So we're going to give you a couple of simple rules to help get you started:

- 1. Skip what you don't understand and**
- 2. Go back to it later.**

AmigaDOS is a fairly heavy subject and there's no getting away from the fact you have to know some things in advance.

However, just like you can't build a house without foundations; you cannot lay foundations without digging a hole; you can't dig a hole without a spade; and you can't use a spade unless you know what one looks like...

But what if the spade hadn't been invented?

Learning AmigaDOS is a little like that. It's so inextricably interwoven it's impossible to learn about one thing without previous knowledge of another. Fortunately we are here to help you.

Imagine what it would be like learning to run if you didn't know how to walk. Consider this:

"The cat sat on the mat"

No doubt it did – but how could you even start to understand a sentence such as this without knowing what a cat is? Don't believe us? Try this one:

"Oh sweet bourne – quench me, I beg and bring forth fish for my platter "

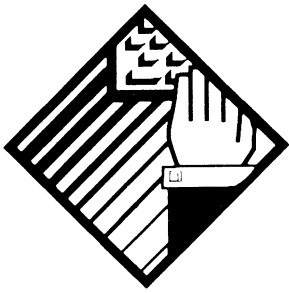
What is the writer saying? Follow the rules at the head of this page and you'll see...

A bourne is an old English word for a river. So this writer was poetically suggesting his thirst and very probably sat by the banks of the river with rod in hand. Of course if you are not forearmed with that knowledge you're going to stay in the dark!

Douglas Adams told of such a predicament in, *The Hitchhiker's Guide to the Galaxy*: "When you know what the question is, you'll know what the answer means."

He was relating a conversation between a Supercomputer and a, well, a "life form" who had posed the question of Life, the universe and everything. To which the computer had replied: "42".

It took Adams' fictional computer 7.5 million years to arrive at that solution. Fortunately, you should master the rudiments of AmigaDOS in a somewhat shorter space of time.



2: AmigaDOS and Workbench

Overview

- > AmigaDOS and Workbench
- > Using Scripts
- > About 2.04
- > About 2.1
- > About 3.0

Workbench is a graphical user interface – a WIMP system. Which is just another way of saying that it's an easy way to access AmigaDOS. Many of the features we all take for granted in Workbench are actually trickier if attempted from AmigaDOS. For instance, every time we start a Workbench "Tool" – we effectively run another program.

Similarly, we can throw away loads of "projects" simply by selecting them and either dragging them to the Trashcan or selecting Discard. Point of fact: Workbench uses AmigaDOS, albeit at a very low-level. So why on earth would you want to use AmigaDOS if everything is so easy from Workbench?

Actually there are hundreds of reasons for using AmigaDOS rather than Workbench; indeed many purists refuse to use Workbench at all! This is a bit extreme – if Workbench can do what you want there's no need to use AmigaDOS at all. We use Workbench for performing bland operations like formatting and disk copying more conveniently.

Imagine you want to move a program between disks. From Workbench you simply drag its icon from wherever it is – to where you want to put it. However, if you want to simply move the program somewhere else on the same disk – say from one drawer to another – you do exactly the same thing. We do this without even thinking about it. Yet at AmigaDOS level, these operations require two entirely different commands: COPY and RENAME.

Problems start to arise when you want to use some of the more advanced features at your disposal. If you're an experienced Workbench user, with lots of software, it is likely you have already encountered the problem of dragging a program from one disk to another – only to find the copy won't work.

Workbench creates this problem because it cannot copy all the files (objects) belonging to each program. It just isn't smart enough; this facility could be implemented but would be too complex to use. When you drag (copy) a project or tool from one disk to another Workbench only copies two files. The first is the actual program (tool) or workfile (project) and the second is the icon itself.

This happens because Workbench only knows about what are called .info (dot-info) files – the icon images. When you copy a project icon, Workbench first copies the icon, then copies any file with the same name, minus the .info extension. Generally this is OK, but tools often have a lot of files associated with them. Depending on the complexity of the tool, some may need extra libraries, device drivers or just configuration files. Since Workbench does not know where these are it can't copy them – and so the problems start to appear.

This problem can be avoided by using AmigaDOS because, once you are well versed with the system and its foibles, you will soon know where to look for those extra files – and what to look for.

Indeed, a better understanding of AmigaDOS will help you get the best from the machine as a whole. For example, every time you "boot" a Workbench disk, the machine goes through a list of AmigaDOS instructions. The Startup-Sequence, as it's called, is responsible for this and is covered in later chapters.

Some applications work better with different configurations. Imagine you have a wordprocessor that can use Amiga fonts as many can. You may already be aware that there are some extra fonts on the AmigaFonts disk but how do you get at them? There simply is not enough room to move them onto the Workbench disk unless you either remove some other fonts. Using AmigaDOS allows you to access an entire disk's worth of fonts should you want to!

Amiga Scripts

Perhaps the most powerful feature of AmigaDOS is what programmers call its script or batch language. AmigaDOS has one of the most powerful batch languages around. Certainly, it is far more accessible than say the one found in MS-DOS. For this reason we have gone to great lengths explaining just what makes scripts tick; and how to get the best from them. Of course, you can run scripts from the Workbench once you have developed them.

Version 2

With the release of Workbench 2, access to AmigaDOS got somewhat easier as it became possible to enter AmigaDOS commands directly from Workbench. Workbench 2 also allows you to view all files as icons or text. Such features – probably borrowed from other systems such as Windows and GEM – after continued pressure from developers did a lot to enhance Workbench; but, they do not replace AmigaDOS and its power should never be taken lightly.

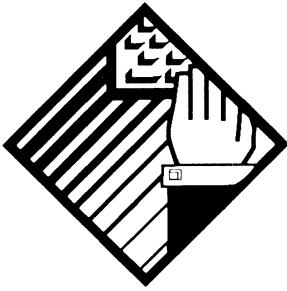
Version 2.1

Workbench 2.1 was the disk-based upgrade from version 2.04 supplied with the Amiga A500+ and is standard equipment on the A600. The most important change was the localisation of the software. This means that no matter where in the world you happen to be using the machine, it should be able to speak in your language or one you understand.

Version 3

Workbench 3 is very similar in most aspects to Workbench 2, although some of the menus have changed. For instance, Clean Up got its own shortcut – but this is minor detail.

More importantly the system got even more commands to handle some of the more powerful features found in the later systems – such as the 256 colour mode offered by the AGA chip set and the tuned disk filing system.



3: First Principals, First Words

Overview

- > Nursery English
- > Verbs and nouns
- > Simple Sentences

The Workbench is a program that AmigaDOS runs by default when you first start your Amiga. The computer term for starting a computer system is called 'Booting', ie, you 'Boot' your Amiga. It provides an easy way of managing your various computer tasks.

However, AmigaDOS itself is a much more powerful world when you skip the restraints of the Workbench, and more experienced programmers sometimes prefer never to see the Workbench. This doesn't mean that you have to discard it totally – far from it – and there is much to be said for having the Workbench available and accessing AmigaDOS with commands through it.

In its purest form AmigaDOS is operated by giving *commands*. These are special keywords which AmigaDOS is able to recognise, understand and act upon. This process is one of interpretation and for this reason AmigaDOS is often referred to as an interpreter.

Think of the way you learned English at nursery school – if you can think that far back! We tend to learn simple verbs (commands) and nouns (naming words) in order to make very simple sentences. For instance:

The cat sat on the mat

This describes an object (The cat); what it did (sat); and where it did it (on the mat).

Three principal objects here communicate the sentences meaning: "cat", "sat" and "mat". So we could abbreviate this phrase to:

Cat sat mat

and let our brains fill in the rest!

Undoing the Links

The linking words are really only necessary to fulfil the subtle nuances of the language and, more often than not, are redundant. So long as the key words are present, and in the correct order, the meaning of what is written remains clear. Speed readers take in information in such a way – by isolating *parts* of each sentence and letting their brains work the rest out.

Computer parsers (the proper name for phrase interpreters) are very simple things: much dafter than the human brain. Therefore, it is not necessary for you to think at some higher level: rather you must lower yourself to thinking the way a computer does.

AmigaDOS parses a sentence (called a command line from now on) by splitting it into smaller bits: words. Each word is analysed and acted upon separately. Let's take a look at a simple English phrase:

Vault

This word can be either a noun (a cellar or strong room) or a verb (to leap or jump over something) depending on the context in which it is used. We use this word – and thousands of others just like it – in our everyday lives without even thinking about it. In this context it is a verb – a command if you will. Verbs usually require some sort of object to work on: a noun. Therefore we could arrive at:

Vault the podium

or simply:

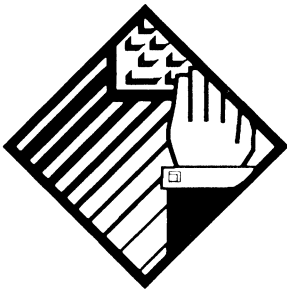
Vault podium

AmigaDOS commands work in just the same way without the subtlety of the English language. We always start a command line with a verb and proceed with the rest of the description. Whenever you encounter a command line you cannot understand try breaking it down into components and refer back to the base command. In the previous example of the feline getting comfortable by the fire, we would write:

Sat Cat Mat

The verb (command) is always written first and although this may look confusing at first you'll soon get used to it. AmigaDOS works like this: the verb always precedes the objects it acts upon. We'll see how this happens shortly.

A full list of AmigaDOS commands and descriptions can be found in the Command Reference Section of *Mastering AmigaDOS 3 – Reference*, while practical advice and tutorials can be found in the forthcoming chapters. Consult the index when seeking information on a particular command.



4: First AmigaDOS Command

Overview

- > Using Execute Command from Workbench
- > Entering a simple command
- > Making errors
- > More commands to try

"One picture is worth more than ten thousand words"

– Ancient Chinese proverb

To enter AmigaDOS commands from the Workbench you have two options: the first allows you to execute a single command, the second allows you to communicate via the keyboard to the underlying AmigaDOS. Some of the terms we'll be using here will be unfamiliar at this stage but don't worry; write them down and keep a list you can refer back to as you proceed through the rest of the book.

Select Execute Command (*Amiga-E*) and you will be presented with a little entry box – programmers call them string gadgets, but you can call it whatever you feel like. Make sure the gadget is active by clicking in it and type the following at the keyboard:

DIR OPT AD

Note that there is a space between the words DIR and OPT and also between OPT and A. The keys to press are as follows:

D
I
R
Space Bar
O
P
T
Space Bar
A
D

Do this very carefully and try not to make a mistake. At the end press the Return key (or Enter if you have one). If you have made a mistake then AmigaDOS will inform you by saying that something is not found. If this appears, close the window, check your line and try again. All being well Workbench will present an Output window containing a short list of directories currently on the Ram Disk. You can check this using opening Ram Disk and selecting Show...ALL from the Window menu.

What you have done is to ask AmigaDOS to list all of the directories held in the Ram Disk. The Command DIR is short for DIRectory. OPT is short for OPTion; A means 'All' and D, 'Directories'. RAM is listed because that is called the *default* or *current* directory.

In addition to the directories there are a number of files stored in the Ram Disk. Let's take a look at that. Close the Output window and choose Execute Command again. Now modify the line so it reads:

DIR ALL

and press the Return key again. The files are placed after the directory list and the directories themselves are clearly distinguished by the fact that they have the word '(dir)' written after them. Notice how the listing is indented slightly? Each indent shows one level lower of directory hierarchy.

Among other things, you will see your Preferences presets listed in there: so that's where they got to! You don't have to worry about what all this means at this stage – the important thing to learn is how the commands work.

When AmigaDOS commands are listed as in the examples above, it is not normal to keep saying 'press the Return key'. This is generally implied because it is the only way to end a typed line. Occasionally you will also see words inside angled brackets, for example:

DIR<Space>FONTS

In these cases you are not normally required to type the contents but issue the key press signified. The '<Space>' above would mean press the space bar. Similarly:

1>DIR

means type DIR and press the Return key – we'll keep to this convention from now on.

Letter Case

Most computer users are not proficient typists and so having to switch between character case can become a real pain. For this reason AmigaDOS commands are case independent. This means

that you can enter them either as lower case (small) characters or upper case (capital) characters, and even a mixture of both. The following are all examples of the DIR command:

DIR

dir

Dir

diR

DiR

They all work! To help distinguish commands from the main body of the book text we'll always show AmigaDOS commands in capital letters, but you can type them as you wish.

Unless you make your Output window large enough, text will scroll through it too quickly for you to read. You can cause a temporary halt simply by pressing a key on the keyboard. This is displayed on screen and effectively creates a pause. When you wish to continue press the <backspace> key to delete the character you typed. Note that you can type any character on the keyboard to stop the text scrolling except <RETURN>, <ENTER>, and <LEFT-ARROW>.

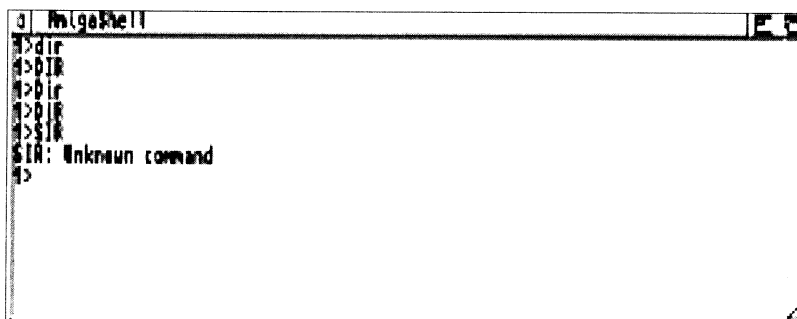


Figure 4.1. Case and spelling in AmigaDOS.

More Errors

If you make a mistake when you enter an AmigaDOS command and press Return without realising, then AmigaDOS recognises the fact and lets you know about it. Type the following at the keyboard:

1>SIR<Return>

and the following message is displayed:

Unknown command:sir

sir failed returncode 10

This is called an 'error message' and you should re-check the command you used and try again.

If you make a mistake and are aware of the fact before you 'issue' the command, then you can edit the mistake out. When you meet the Shell, a greatly enhanced version of what you have just used, you'll see how easy this really is.

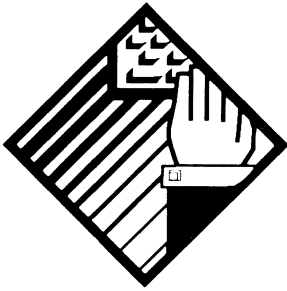
In the meantime there's no time like the present to start executing some commands for yourself. Every AmigaDOS command or Workbench tool is, without exception, called by name. Of course you don't know very many commands yet – but you are aware of the Preferences tools and Utilities. Try calling those up from the Execute Command box. Here are a few to get you started (these won't all work on a floppy-based system – see if you can work out why):

Format

Clock

ScreenMode

PrinterGFX



5: Formatting and Copying Disks

Overview

- > Format from Workbench
- > Disk storage capacity examined
- > Single drive backups
- > Multiple drive backups
- > Renaming disks
- > AmigaDOS formatting
- > AmigaDOS disk copying

"The golden rule is there are no golden rules."

– George Bernard Shaw

Disks are as important to your Amiga system as the screen is to show you what is going on. Because different computers access disks in slightly different ways they are supplied in a 'blank' format. So before you can use them you must perform a process called 'formatting'.

The formatting process lays down an electronic map AmigaDOS can use to save and retrieve your information quickly and efficiently. Because this process is so vital it is made relatively easy to perform, and whenever you purchase a box of disks it is always worth spending some time formatting them in one go. This way you'll always be able to place your hands on a disk ready to use whenever you need it.

We keep things simple for a while and show you how to do this from the Workbench and then from the Execute Command facility. This will help ease you in and again illustrate just how closely associated the Workbench and AmigaDOS are. But beware! Very soon we'll leave the Workbench pretty much for good and concentrate solely on use of the Amiga from AmigaDOS.

Formatting a Disk

For the time being we'll concentrate on formatting disks from the Workbench. To format a new disk remove the Workbench disk from the drive and insert a new blank

disk. The Amiga will try to read the contents of the disk. As the disk is unformatted it won't be able to so, after a few moments, it will display the disk icon on the Workbench with the name:

DF0:???

DF0: refers to the disk drive number (Disk Floppy drive 0) and ??? signifies that the Amiga cannot use the disk as it is bad or unformatted. Occasionally you might get this message when you use a disk that you know to be formatted and to contain files. This normally means that the disk has become corrupted or damaged in some way. Files on a corrupted disk *can* often be recovered but you need the right software to do so – it's surprising how much you can get back.

Now select the Icons Menu and located the item marked 'Format Disk'. This is used to format a new the disk, however it cannot be used yet as the option appears ghosted. To format (or initialise) the disk it must first be 'selected' by clicking the DF0:???? icon. Now go to Icons and select Format Disk.

A 'System Request' box appears on the screen. This 'Requester' is asking that you replace the Workbench disk back into the disk drive. This is because it needs to load in the various formatting routines it needs. If your Amiga has a second drive you can insert the Workbench disk here. The machine recognises the fact and gets on with the task in hand. If you have a single drive system you have to remove the 'Bad' disk and insert the Workbench disk. Again AmigaDOS will know about the change and continue.



When you format a disk it destroys all of the information on it. You can, for instance, reformat an already formatted disk. If you accidentally format a disk with information on, it is lost for ever.

This first requester, therefore, can be used as a safety net to enable you to double check before going any further. If you wish to abort the formatting click in the Cancel gadget (sometimes called a button).

If you are using Workbench 2.1 or higher, you will have other options such as International Mode and DCFS. Ignore those for now – we'll cover them later on.

If the disk has already been formatted, you have an extra gadget marked QUICK. This is a special option should never be used on a disk that has developed errors. This gadget does not appear when the disk is unreadable – ie, has never been formatted.

After a few moments the System Request reappears and asks you to insert the disk to be initialised (DF0:???) back in the drive. Do this and click on 'Continue'. A second safety net appears in the form of a requester box and you can abort at this point, should you so wish, by clicking on Cancel. Select 'Continue' to carry out the formatting in earnest.

The formatting now begins. As this happens a series of messages is displayed in the Initialise window. The formatting process requires AmigaDOS to mark 80 'cylinders' (160 tracks) on the surface of the disk. After formatting each track it verifies that it has done so correctly. When the count reaches 0 the formatting is complete and the Initialise window disappears.

The Bad disk icon is now replaced with a disk icon called 'Empty'. If you open its disk window you will see that it does have a Trashcan icon in position (as do all newly formatted disks unless you ask Workbench to omit it).

What the K?

When an AmigaDOS floppy disk is newly formatted it contains 880k of space free for use. The term 'k', or 'K', is shorthand for kilobytes and it is a measurement of storage. One kilobyte equates to 1024 bytes. A byte is a standard computer unit, and it may be easier to think of a byte as being equal to a single character. Thus the alphabet can be stored in 26 bytes. Therefore an 880k disk can store:

$$880 \times 1024 \text{ characters} = 901120 \text{ characters}$$

For very large storage measurements the Megabyte is used. One Megabyte (1Mb) is equal to 1,048,576 bytes or 1024k.

Strictly speaking the usable space on a standard AmigaDOS disk is somewhat less than 880k – a more realistic estimate is in the region of 837k – but we've left finer detail such as that for the Reference Volume.

Backing Up Disks

If there is one golden rule for computing it can be summed up in seven words:

Always keep backup copies of your disks

Disks are subject to corruption and occasionally you will not be able to recover the information on one. It's gone – lost forever. If you have a backup copy in a safe place you can use this (called the Master Copy) to make a new working copy. Of course you don't need to keep a backup of every single disk you have, just the ones containing important files. You can reduce the chances of disk corruption by looking after your disks and this subject is touched on below.

At the moment the most important disk in your collection is your Workbench disk. Without this you cannot use your Amiga. The disk supplied with your Amiga should be considered as a master copy and you should make a backup copy which is your working disk and the disk you use to start your Amiga.

Before you can make a backup copy of the Workbench disk you must first format a disk as already described. The disk that is to be duplicated is called the 'FROM' or 'source' disk and the disk onto which the files are to be copied is called the 'TO' or 'destination' disk. Secondly you should 'write protect' your master disk. To do this turn the disk so that the rear is facing you and with the tip of a nail or ball point pen slide the small black plastic tag in the left hand corner of the disk up so that it reveals a square hole. Whatever happens now you cannot write information to this disk ie, you cannot accidentally format it or overwrite it. This is not essential for backing up but it is a good habit to get into.

The exact process you use to backup your Workbench (or any other) disk depends on whether you are using a single or dual disk drive system. Both systems are examined below.

Single Drive Backup

Place the Workbench disk in the drive, select it by clicking on it once and then select the 'Copy' option from the Icons Menu on the Workbench Menu Bar. The disk copy utility now comes into play and a requester box is displayed asking you to insert the FROM disk into drive DF0. When this has been done click on 'Continue'. AmigaDOS now begins to read information from the FROM disk into memory. When it has read as much as it can, it asks you to replace the FROM disk (Workbench) with the TO disk (the 'Empty' disk). Do this and click on 'Continue'. The information read from the FROM disk is now written to the TO disk. This process repeats several times until the backup copying process is completed.

Note that the disk copy utility does not check to see that you insert the correct disks – you must therefore ensure you do so otherwise you are liable to corrupt your disks. AmigaDOS also 'creates' the 'TO' disk for you during the copying process and this is displayed on the Workbench as 'DF0:BUSY'.

When the disk copy window disappears the backed-up disk icon appears. This is named 'copy of xx' where 'xx' is the name of the disk copied.

Dual Drive Backup

If you have two disk drives then the process of backing up one disk onto a blank one becomes very much easier and cuts out the necessity to be on hand to make disk swaps. To make a copy of the Workbench insert it into one drive and an empty formatted disk into the other. Then drag the Workbench disk (the FROM disk) onto the top of the Empty (the TO disk) disk, so that it obscures it and then release the

mouse button. The Disk Copy utility is invoked and a requester is displayed. In addition the Empty disk icon on the Workbench is replaced by two new disk icons labelled 'DF0:BUSY' and 'DF1:BUSY'.

To proceed, click on the 'Continue' button. Each track is first read and then written in turn and this fact is indicated in the requester box. The process is quite quick and on completion the two busy disk icons are replaced with the backed up disk icon. This will be named 'copy of xx' where 'xx' is the name of the disk copied.

Renaming Disks

When the copy process is complete you will probably wish to rename the disk – at least to get rid of the 'Copy of ' portion of the name, though this is not required for any reason other than to make it more aesthetically pleasing. For example, you can rename:

Copy of Workbench 3

as plain:

Workbench 3

if you so wish. To do this, select the Copy disk and then select the 'Rename' option from the Icons menu in the Workbench Menu Bar. A small simple window will appear on screen with the disk name inside it. The small box is the cursor and this can be used to edit the name. To remove 'Copy of ' just press the Del key eight times until the cursor rests over the W of Workbench. Press the Return key to complete the renaming process.

The final task is to write out a disk label and stick it to the newly formed disk. Write the label before you affix it to the disk and be as concise as you can in naming it and listing its contents. Failure to label disks isn't a capital offense but a box full of unlabelled disks has the nasty habit of raising the blood pressure when it comes to locating a particular disk, as we've proved many times.

Formatting From AmigaDOS

When a disk is formatted from the Workbench using the Format option you are required to supply two items of information, even if you don't know it. Firstly you must indicate the disk to be formatted by selecting it and then you supply a name for the disk. The latter is actually done for you, and a name of 'Empty' is assumed.

When disks are formatted from AmigaDOS the same two items of information are needed. The disk is identified courtesy of the device name of the drive it is placed into. A name must be supplied can then be specified but you can use 'Empty' if you wish.

The basic format of the command is shown below. In computer terms this is normally referred to as its 'syntax'.

FORMAT DRIVE <disk> NAME <new name>

In AmigaDOS we talk of two variations on this theme: the command's synopsis and its template.

- A synopsis is an extended English description which demonstrates which 'arguments' are required and how to supply them. In a single sentence it describes all the possible variants a command can understand.
- The template demonstrates how the command expects to receive its arguments, and sometimes, their position.

Both of these are described in detail in *Mastering AmigaDOS 3 – Reference* but for this discussion we will simplify things somewhat.

This command breaks down into several parts.

FORMAT	This is the command itself.
DRIVE	This is a place holder known as a keyword and takes a single argument: the name of the disk drive to start formatting on. The keyword must be supplied along with the argument.
NAME	This is another place holder where you must specify the name of the disk. As before, the keyword must be supplied along with its argument.

The rules are exactly the same for formatting from AmigaDOS as they are from the Workbench. Formatting destroys anything that may be on the disk.

To format a disk in the internal drive and call it 'Empty' we could use the following command:

FORMAT DRIVE=DFO: NAME=Empty

based on what we've just shown though, the following would *not* work.

FORMAT DRIVE NAME

or

FORMAT DFO:Empty

How you proceed with the formatting again depends on whether you are using a single or dual disk drive system. Both flavours are outlined below. It is also possible to include some additional parameters which define the type of formatting to take place. These are described later (and in *Mastering AmigaDOS 3 – Reference*) and are not relevant at this point.

You will need to have access to a new blank disk and your Workbench disk. Remember to write protect your Workbench disk prior to formatting. Just in case.

Single Drive AmigaDOS Formatting

Enter the format command exactly as shown below:

```
FORMAT DRIVE DF0: NAME "Empty"
```

As with all AmigaDOS commands the case of the letters for the command is irrelevant, however, the combination used inside the double quotes to name the disk will be used exactly – including spaces if you want to. You are asked for the Workbench disk and within a few seconds a message will appear inside the Output window as follows:

```
Insert disk to be initialised in drive DF0: and press RETURN
```

Do this and the procedure is then much the same as described earlier. The 'DF0:BUSY' icon appears on the Workbench and then formatted and verified. On completion it takes its place on the Workbench as an icon.

Dual Drive AmigaDOS Formatting

This is much the same as for single drive formatting. This time, however, place the disk to be initialised in drive DF1: and the Workbench disk in DF0:. Then type the following into the Execute gadget:

```
FORMAT DRIVE DF1: NAME "Empty"
```

Formatting then continues as for a single drive system.

Renaming Disks

Should you wish to give a disk a new name directly from AmigaDOS then you can do so using the RELABEL command. However, this is difficult to use on a single drive system without the aid of the RAM Disk. As we haven't seen how this works yet, we'll put off our look at the RELABEL command until then. But you now know it exists.

Backing Up Disks

AmigaDOS provides the command DISKCOPY to totally copy the contents of one disk to another. The net effect is the same as that described in Chapter Four using the Workbench. The syntax of the command is:

```
DISKCOPY FROM <source> TO <destination> NAME <new name>
```

As you can see AmigaDOS can take three different parameters and provides you with the opportunity to rename the destination disk. This last part of the command is optional and if you don't wish to rename the disk then you can use a shortened version of the command as follows:

```
DISKCOPY FROM <source> TO <destination>
```

In this case the disk is given the exact name of the one it was backed up from, ie, it does not have 'copy of' prefixed to it. In both forms the relevant drive device names are substituted for <source> and <destination>.

The utility works by reading each of the tracks from the source disk and writing them to the destination disk and includes the information laid down on the source disk when it was originally formatted. This means that, when using DISKCOPY to backup a disk, you don't have to format it first, this is effectively done as the backup takes place. An obvious time and effort saver.

Single Drive DISKCOPY

To make a backup disk with a single drive system, calling it "Boot Disk" use the following syntax:

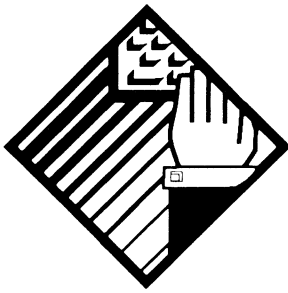
```
DISKCOPY FROM DF0: TO DF0: NAME "BOOT"
```

Note that the name appears in double quotes. You are asked to insert the source disk in drive DF0: and press RETURN. The process of reading information, swapping disk and writing information will take place as with a Workbench based backup.

Dual Drive DISKCOPY

This form is almost identical to the above process but does not require the swapping of disks. Ideally you should place the source disk in DF0: and the destination disk in DF1: and issue the correct form of the command. From the example given above this would be:

```
DISKCOPY FROM DF0: TO DF1: NAME "BOOT"
```



6: Drawers and Directories

Overview

- > Workbench drawers
- > AmigaDOS directories
- > Tree structures
- > Directory paths

"Stop yer grinnin', and drop your linen".

– from James Cameron's *Aliens*.

This is of course a book about AmigaDOS and as such it is not really our intention to delve into the workings of the Workbench in any great detail. However, much of the time the Workbench is acting as nothing more than a graphical interface to a variety of AmigaDOS commands so, as such, there is a lot of inter-related material.

For the purposes of this chapter we intend to look into how the concept of directories work by using the Workbench as an example as well. The concepts involved are all important and the graphical nature of the Workbench will help get these across.

Open the Workbench disk and study the window for a moment. You'll notice that several of the icons are shaped like drawers. There is no accident in the naming convention. Just as in life, we use drawers to store things in, and in a good filing system we are careful what goes into where.

The analogy falls apart down when you realise drawers can contain drawers, which in themselves contain drawers, which contain drawers and so on.

In much the same way as disks have disk windows, drawers also have their own windows. Open the Utilities drawer and a window opens displaying the drawer's contents. You might like to try this now and see what the end result is. In fact the drawer window is very much like the disk window in that it has all the same gadgets.

This is a part of an overall design philosophy built around what Commodore call *Intuition*. This is the Amiga's graphical user interface or GUI. Such systems are called WIMPS: Windows, Icons, Menus and Pointers. The word 'WIMP' is sometimes, quite unfairly, used to describe people who have never experienced anything else. The development of WIMPs – at Xerox Parc, not Apple by the way – was to make computers more accessible to plain users.

How much do you know about Intuition? Probably very little – but chances are you can use it without so much as even thinking about it. Every time you move the mouse, open a window or access a menu you are using Intuition. We're going to leave all that behind very shortly as we access another side of the machine.

Drawers and Other Bloomers

Why drawers? Why not? AmigaDOS supports what is called a hierarchical filing system meaning it can have multiple levels organised like a family tree. In fact, that's where the analogy was drawn from. A drawer may have zero or more siblings (brothers and sisters) and zero or more children. Remember that curious option from Workbench: 'Open Parent'? This opens the drawer window that spawned the one you are currently examining. Phew – let's draw an analogy.

Imagine your desk at home, office or school and that you have no access to files, or drawers or cupboards. You would have to keep every single book or sheet of paper on top of the desk. This would produce a cluttered area which would be difficult to work in and would raise your blood pressure when you tried to locate a particular item (*sounds like my office – M*). In an organised working environment you would arrange your books and papers into folders or drawers which would themselves be held in a desk or filing cabinet.

Drawers are designed to emulate this process. Every AmigaDOS disk has enough space on it to hold hundreds of thousands of characters of text. Few files (the work you create) ever reach this size and it is possible to store many files on a floppy disk (over 850 in fact).

If a disk had 50 files all stored in the main window it would make locating a particular file troublesome and involve a fair amount of scrolling around the window. Interestingly, on Workbench, a lot of files and drawers are normally hidden from view: because you don't need to see them.

Drawers are used to add some semblance of order to this nonsense. You can create a drawer, give it a name and then place the relevant files inside. For instance, you may wish to use your Amiga as a wordprocessor and as a means to program in ARexx. You could create two drawers on a disk and label them Wordprocessing and ARexx. You could then save each file you created in the relevant drawer. You might call the disk "Current Work".

You can go a step further: drawers may themselves contain drawers. Let's take the Wordprocessing drawer. You might find that you are producing several types of documents. For example, Home, Business and Club. You could create suitably named drawers inside the Wordprocessing drawer and then place the relevant files in the appropriate drawers.

We could take it a step further again: you might create two types of wordprocessor files for your Club – namely Letters and Memos. Once again these drawers could be created and used in similar fashion. In fact there is no limit to the number of drawers that can be 'nested' inside other drawers, subject to there being enough space on your disk to do so.

The term drawers is sometimes interchanged with the term directories. Strictly speaking, although the engine is the same thing, a directory only appears from AmigaDOS; a drawer can be viewed from AmigaDOS and Workbench.

Directories stored inside other directories are called 'sub-directories'. As a rule of thumb, drawers belong to Workbench, directories to AmigaDOS; even though they mean more or less the same thing. This allows you to illustrate clearly which one you are talking about. Figure 6.1 (overleaf) illustrates the scenario outlined above. Examine this carefully to see just how this hierarchy works.

The thing that should become immediately apparent from arranging your files in this way is that it makes it very easy to locate files when you next want them. If you want a Club memo file you'll know exactly where to look!

The Tree Structure

Because of its arrangement it is sometimes called a 'tree' or 'tree directory'. You can imagine the various sub-directories (the children) as being the roots of a tree or, if you turn the lot upside down, the branches of a tree.

The very top of the 'tree' structure, ie the top directory (the one that is displayed when you open a disk icon) is normally referred to as the 'root directory' for this reason. In Figure 6.1 'Current Work' is the root directory, and 'Wordprocessing' and 'ARexx' are sub-directories or children of the root and each other's siblings.

If you look down the root structure of the Word processing directory you will notice that several directories are on the same level but they are not connected directly. For instance, the directories 'Home', 'Business' and 'Club' are all sub-directories of 'Word processing' but there is no direct connection between them other than via the 'Word processing' directory. This is a very important concept and one that you should fully understand.

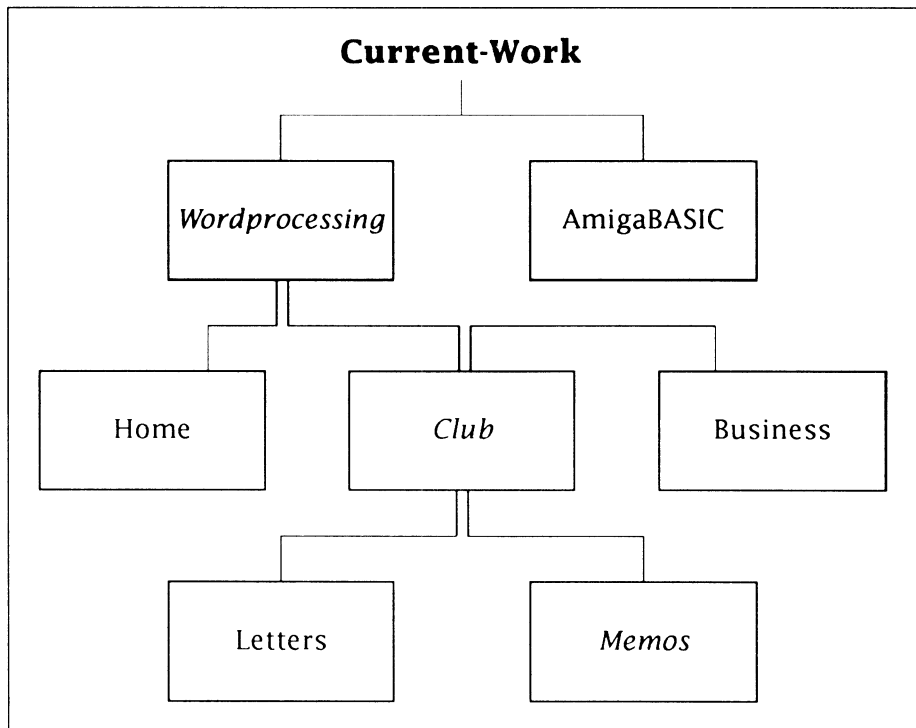


Figure 6.1. Directories and sub-directories.

Imagine you are working on a file stored in the Letters directory. This is called the 'current directory'. To go to the Memos directory you must first go 'up' to Club (ie up the directory tree) and then 'down' to Memos (down the directory tree).

Directory Paths

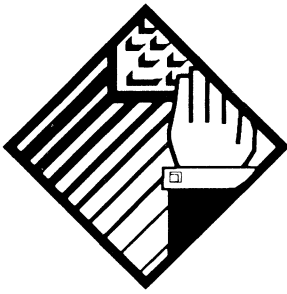
When you need to reference a particular file you can do so by giving what is called its 'path name'. The path name consists of the file name preceded by any directories and sub-directories which must be accessed to reach it. Each directory name is separated by a forward slash character (/). So to write the path name of a file called 'Report' in the Memo directory in Figure 6.1 we would use:

Current-Work:Wordprocessing/Club/Memos/Report

The last name is generally a file and not a drawer. Also it isn't always necessary for you to give the disk (volume) name in the root if the disk to be used is implied, in which case the path name becomes:

:Wordprocessing/Club/Memos/Report

Note the use of colon (':') which implies the root directory of the current disk. If you have more than one disk drive attached to your Amiga then you might wish to state the disk drive number and we shall cover this in due course.



7: Drives and Filenames

Overview

- > Drive naming conventions
- > Object naming conventions
- > Allowable characters

So far we have been rather blasé in our use of disk and drawer filenames. However they have been picked with care so they do not contain any illegal characters – characters that are not allowed as part of a filename. In this chapter we'll look at what you can and can't call files. But first, disk drives.

Disk Drive Names

A disk drive is a *device*. A base Amiga system will normally contain a single drive and extra drives may be added. Disk drives are not the only devices available for use on an Amiga, but we will concentrate on them for the time being.

To enable you to differentiate between drives each is assigned a different device name. This differentiation is not a problem on a single drive system but on a dual drive system it is very important. How would you tell AmigaDOS which one to use?

The default drive, the Amiga's internal floppy disk drive, is called 'DF0:', which can be thought of as 'Disk Floppy 0:'. Note that the end letter is a zero and not an Oh. These two characters are quite different on the Amiga, though they look much the same on screen, and you must ensure that you use the correct one. If you add a second drive, then this will be called 'DF1:' – 'DF2:' on larger Amiga models.

These device names can be used as part of a "file path". For instance, suppose we maintained the Current Work disk depicted earlier in DF1: and wished to get access to a file called 'Update' in the Memos directory. We could do so as follows:

DF1:Wordprocessing/Business/Memos/Update

Similarly the path to the same file if the disk was placed in DF0: would be:

DF0:Wordprocessing/Business/Memos/Update

Filenames

AmigaDOS filenames may be up to 30 characters (including spaces) in length. They can contain most of the characters available from the keyboard with a few exceptions:

- :** (colon)
- /** (slash)
- "** (double quotes)
- #** (hash)
- ?** (question mark)
- '** (tick)
- *** (star)

These characters are used *with* filenames on certain occasions but they cannot be included *in* filenames. Here are some examples of legal filenames:

Test File

One2Three

A-B-C

Add+Numbers

Join.The.Dots

You can see from the first example above that the space character can be included in a filename – this is *not* recommended!

It is a good to get into the habit of assigning filenames that mean something and to add an 'extension' to these to indicate the type of file it is. For instance, you might create some letter files to which you could add the extension LET. Use the full stop character to separate the filename from the extension thus:

SARAH.LET

MARC.LET

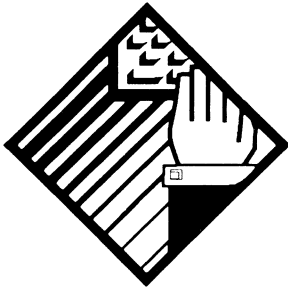
NATHAN.LET

These are basic extensions. But many applications will suggest their own and in addition a standard set of conventions have evolved over the years – some of which are summarised in Table 7.1

<i>Extension</i>	<i>Use</i>
.info	Amiga Workbench icon file. (dot-info).
.library	Amiga external function library file.
.device	Amiga external device handler function library.
.catalog	Amiga translation library for locale languages.
.ASC	Standard ASCII (American Standard Code for Information Interchange). A very simple – and universally accepted – text format.
.TXT	Usually (but not always) a formatted ASCII file.
.IFF	Amiga universal interchange file format. Graphics, text, sound, music.
.GIF	Compuserve's Graphic Interchange Format. Usually on the PC.
.TIF/.RIF	Graphics format used for scanned images on the PC and Mac.
.BAS	BASIC source code.
.AMOS	Source code developed in Mandarin's AMOS BASIC.
.C	Source code developed in the C language.
.ASM	Source code developed in assembly (machine) language.
.h	Header (include) file used with a C compiler.
.i	Include (header) file used with an assembler.
.REXX	Source code developed in the REXX or ARexx language.
.EXE/.COM	Executable code for the PC running MS-DOS.

Table 7.1 A selection of common filename extensions

It's probably fair to say, a lot of these are borrowed from the ancient operating system CP/M where a maximum of eight letters with a three letter extension were allowed. With modern computers we are nothing like as constrained but it can makes things easier when you have to use patterns – we'll examine this later.



8: Parsing and Synopsises

Overview

- > The basics of parsing
- > Introduction to synopsises
- > Verbs are commands
- > Adverbs applied to AmigaDOS
- > Using quotes
- > Multiple arguments
- > Final arguments
- > Templates vs. Synopsises

By now you should have a basic understanding of how to use AmigaDOS, but you've been entering commands parrot-fashion: without any idea of how they actually work. There's nothing overtly wrong in this because you have been learning – there's no way to avoid it.

In this chapter, we'd like to discuss some more theory. Nothing too fancy, just a few more foundation-building basics. Some of this chapter may seem scary at first – but *don't* let it get the better of you!

Parsing

Some way back we discussed the subject of parsing as it related to English, “the cat sat on the mat” and how it might translate into AmigaDOS as:

```
sat cat mat
```

“Sat” is a command, although in the strictest sense we would use the base verb, “Sit”. If this command were being given (to a person, for instance) what is more important is the use of objects or noun clauses, viz:

```
sit the cat on the mat
```

We are specifying an object (the cat) and telling someone where to place it (on the mat). We can simplify this into a more algebraic form by treating the verb as a command which can take one or more noun clauses.

Do what? Look at this:

```
SIT [object=<item>]  
[place=<where>]
```


This describes a possible synopsis for the verb SIT, offering several alternatives. The bits enclosed in square brackets [&] are optional keywords; the parts in angled brackets < & > must be supplied. Since the angled brackets are enclosed by the square brackets, these items are optional unless the keyword is supplied. If you think it looks tricky, consider that English is much more complex but you've mastered that! Try writing down all the uses you can think of for the verb "sit". This should become clear when you look at a few examples:

- a. **sit**
- b. **sit object=cat**
- c. **sit place=mat**
- d. **sit object=cat place=mat**

and their meanings:

- a. A direct command. Sit (down). (*Down* is implied)
- b. A command operating on an object. Sit (the) cat (down).
- c. A command giving a direction or target. Sit (on the) mat.
- d. This is *a* and *b* combined. Sit (the) cat (down on the) mat.

Modifiers

Moving on from this we also use something called an adverb to describe how something should be done: usually as a modifier to an entire sentence. For instance, gently, carefully, violently etc. Let's take one of these and add it to our synopsis of sit:

SIT [object=<item>] [place=<where>] [gently]

This in turn allows us to modify the whole sentence by describing how the action should be performed – here are three possible examples with the same meaning:

- a. **sit object=cat gently place=mat**
- b. **sit object=cat place=mat gently**
- c. **sit gently object=cat place=mat**

As trained English speakers, we would probably consider the first option as making the most sense but, as we shall see computer parsers can usually make some sense of it.

Purely for the sake of argument, let's add another adverb to this command synopsis. One that's close in meaning to "gently":

SIT [object=<item>] [place=<where>] [gently] [violently]

Also, out of deference to cat fanciers, we'll change the example slightly:

sit object=rat violently place=mat

You can see how these two “modifiers” (they’re called switches in AmigaDOS) are mutually exclusive. You can have either or neither in any sentence, but it does not make sense to use both like this:

```
sit object=rat violently carefully place=mat
```

Now, what about using another word in place of carefully? How about gently? Such alternatives are written in AmigaDOS synopses separated by a bar “|” (pronounced OR) like this:

```
SIT [object=<item>] [place=<where>] [gently|carefully] [violently]
```

This means you can use either (but not both) of these on a command line to achieve the same end. For instance, the first two lines here are valid: the third one is not.

```
sit object=rat carefully place=mat
```

```
sit object=rat gently place=mat
```

```
sit object=rat carefully place=mat gently
```

Still with us? Don’t worry if this all seems a bit alien – take a five minute break and re-read the first part of this chapter again. It’ll be second-nature before you know it.

Split Decisions

Now what would happen if you wanted to be just a little more descriptive about the source object: a cat or rat in this case? For instance, if there was a brown rat, black rat and piebald rat – how would you determine which one? Evidently you would construct a sentence such as this:

```
sit object=brown rat place=mat
```

but in practice such a command is not valid. This is because computer parsers break the line up by components separated by *white* space – space characters. The parser reads this line thus:

```
sit
```

```
object=brown
```

```
rat
```

```
place=mat
```

The word “rat” becomes disjointed in the parser’s titchy little brain and confuses it. It thinks the object is a “brown”. We can get around this in several ways, but the most common is to surround the offending text in quotes, thus:

```
sit object="brown rat" place=mat
```

AmigaDOS reads everything enclosed in quotes as a single word while the quotes themselves are absorbed. Therefore, it divides the line thus:

```
sit
object=brown rat
place=mat
```

Duality and Multiplicity

So far, so good. But what if you wanted to give the name of several items at the same time? Something like this for instance:

```
Sit rat, dog and cat on the mat
```

Alright, this sentence is becoming so contrived it is a little short of dreadful – but it'll suffice. We write the synopsis for these *multiple* arguments by surrounding the argument in curly brackets or braces { and }. Given this our command now looks like this:

```
SIT [object=<{item}>] [place=<where>] [gently=carefully |
violently]
```

so we could express a typical line like so:

```
sit object=rat cat dog place=mat
```

and AmigaDOS parses it thus:

```
sit
object=rat cat dog
place=mat
```

You might wonder how AmigaDOS knows which parts are parameters (user input) and which parts belong to the command. This is accomplished by a neat trick where AmigaDOS matches the input string (sentence) to something called a command *template*. Most AmigaDOS commands have some template – in fact the command's synopsis is usually built from the template: we'll cover the implications of this facility shortly.

Our *command's* template is defined by two positional keywords: Object and Place. When AmigaDOS encounters a keyword it reads everything following as a user-supplied *argument* until:

- a. It encounters the end of the line.
- b. It meets another template entry.

Following this premise we can see how the three animals are read into the keyword Object and "mat" is picked up by the keyword Place. Multiple arguments are separated by white space, just like any others: you can override this by the use of quotes as described previously – so here we go:

```
sit object="white cat" "black rat" "perky pink poodle"
```

Enough of that, you can tell your brain to put the white flag away now. There are a couple of things we still haven't considered. Up to now, we have assumed that every command must have a set of positional keywords, each of which optionally taking a user-supplied *argument*. The animals and objects in the these examples are arguments.

We may also define a situation where certain arguments are required for the command to work. Re-defining the above synopsis to accommodate this gives us:

SIT [OBJECT=]<object> [PLACE=]<place> [gently=carefully | violently]

In both such cases you can choose whether or not to supply the keyword. As a rule of thumb, in direct command lines this is not usually necessary. However, in AmigaDOS command programs (scripts) which we'll discuss later, it will usually be advantageous to specify the keywords. Besides, specifying the keyword allows you to position the argument anywhere you please.

One argument type we have not yet mentioned is the *final* argument. So last, but by no means least, here's what it does. A command shown as final must be the last thing on any command line. It does away with all the rules because everything from the start of the argument to the last character on the command line is entered: including any extra space. The final argument has some special uses which you'll see as you encounter it.

We'll leave this section with some summary information on how these synopses are built. Quite often however, the command's template is just as informative. You can get any AmigaDOS command to display its template simply by adding a '?' to the end of the command line. For instance:

DIR ?

DIR OPT/K, ALL/S, DIRS/S, FILES/S, INTER/S

Summary

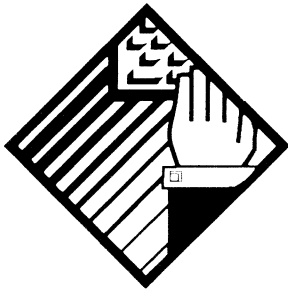
Template	Name	Function
/A	Required	Must have an argument.
/S	Switch	Applies if found.
/K	Keyword	Supplied with argument.
/N	Numeric	Arg <i>must</i> be a number.
/F	Final	Arg <i>must</i> be last on line.

Figure 8.1 Template types.

<i>Template</i>	<i>Synopsis</i>	<i>Alternative</i>
arg	[[Arg=] arg]	arg
arg/S	[Arg]	–
arg/A	[Arg=]<arg>	<arg>
arg/K	[Arg=<arg>]	[Arg=arg]
arg/K/A	<Arg=<arg>>	Arg=<arg>
arg/M	[Arg=]{arg>}	{arg}
arg/A/M	[Arg=]<{arg}>	{<arg>}
arg/N	[[Arg=]<nn>]	nn
arg/A/N	<[arg=]<nn>>	<nn>
arg/K/N	[Arg=<nn>]	[Arg=nn]
arg/F	–	–

Figure 8.2 Examples of command templates and synopses.

A more detailed discussion of templates and synopses can be found in *Mastering AmigaDOS 3 – Reference* where each command is defined in terms of a synopsis and template. And finally, don't let this chapter get you down – you can always come back to it later!



9: The Shell

Overview

- > Introduction to the Shell
- > Shell process numbers
- > Line history
- > Editing facilities
- > Keyboard shortcuts

The Shell was introduced with the release of AmigaDOS 1.3 and the facilities it provides are second to none on any micro. Shell is an enhanced version of the CLI (Command Line Interface) found in the original Amigas and provides a wide and useful range of editing tools making AmigaDOS a quicker and far more pleasant experience. In this chapter we'll look at some of the enhanced facilities.

Locate and open the icon marked 'Shell'. A window is displayed across the Workbench. You may like to select this icon and leave it out. If you haven't already done so, it's worth closing any open drawers at this point.

The Shell's window is properly called a *virtual terminal* or an *interactive console* but don't let that bit of jargon get the better of you – we'll come back to it later. It just means you can type things into it and it will spit things back at you. On conventional computers, like the PC for instance, you work in a single window which fills the whole screen.

This ropery old system should have been put out to grass years ago, but a surprising number of people are still constrained by it and cannot think in terms of anything else.

This system has certain advantages though, so for now practice your Workbench skills and make the Shell window fill the screen – this will enable you to concentrate on the job in hand.

The '1.SYS>' indicates that this is Shell process number 1. It is possible to have multiple Shell windows open and the Shell window number is displayed here.

For example, if you opened a second Shell window '2>' could be displayed as so on. The number is arbitrary and we'll cover it in more detail in the chapter on Multi-tasking. Within the text '1>' has been adopted as a convention. You'll find this is also true of Amiga Shopper's standard-setting AmigaDOS column.

The small square box immediately to the right of the '1>' is called the *cursor*. It is used to indicate the position at which text will appear as you type it at the keyboard.

The Arrow Keys

The four arrow or cursor keys which sit below the Del and Help keys are the key to using the Shell. First enter the following commands one after the other:

```
1>DIR
```

```
1>LIST
```

```
1>DIR DIRS
```

Gosh – what a lot of stuff! When the prompt reappears press the <up arrow> key. The DIR DIRS command appears at the prompt. Press <up arrow> again and the LIST command appears. Press the <down arrow> key and DIR DIRS appears. Press RETURN and the command DIR DIRS is executed.

Each time you enter a line of text and complete it by pressing the RETURN key the Shell records the fact. This in effect gives you a complete command history and you can scroll backwards and forwards through it as you wish. The commands are stored in a special 2k 'history' buffer. When this limit is reached, the commands at the start of the buffer are 'forgotten' to make way for the new ones. This isn't as bad as it might seem as a lot of commands can be held in 2k – 2560 LIST commands for instance!

Unless you are a proficient typist (and few computer users are) then you will invariably make mistakes. On the old systems you had only two choices, either hit the RETURN key and re-type the line after the ensuing error message, or delete backwards to remove the erroneous section and re-type. Yak, blar – phooey.

Shell makes this a thing of the past. Table 9.1 lists the various line editing commands provided.

<i>Key</i>	<i>Action</i>
<up>	Go back one line of history.
<down>	Go forward one line of history.
<left>	Move cursor left one position.
<right>	Move cursor right one position.
<SHIFT-up>	Search back through history (as <Ctrl-R>)
<SHIFT-down>	Delete current line (as <Ctrl-X>)
<SHIFT-left>	Move to start of line.
<SHIFT-right>	Move to end of line.
<BACKSPACE>	Delete character left of cursor.
	Delete character under the cursor.
<A-C>	Copy marked text to the clipboard.
<A-V>	Paste text from the clipboard.
<Ctrl-A>	Move cursor to start of line.
<Ctrl-B>	Go to the last command in the buffer.
<Ctrl-C>	Abort current command.*
<Ctrl-D>	Abort current script.*
<Ctrl-E>	Send break-E to current command.*
<Ctrl-F>	Send break-F to current command.*
<Ctrl-H>	Same as <Backspace> key
<Ctrl-i>	Same as <Tab>
<Ctrl-J>	Linefeed. Not the same as <Return>.
<Ctrl-K>	Delete from the cursor to end of line.
<Ctrl-L>	Clears current Shell window.
<Ctrl-M>	Perform a <RETURN>
<Ctrl-Q>	Pause a scrolling display.*
<Ctrl-R>	Finds a line based on current line.
<Ctrl-S>	Restart a scrolling display.*
<Ctrl-U>	Delete from cursor to start.
<Ctrl-W>	Move cursor to next tab stop.
<Ctrl-X>	Delete whole of line.
<Ctrl-Z>	Move cursor to end of line.
<Ctrl-\>	Marks end of file (or closes Shell window)
* Running commands only	

Table 9.1. The Shell's keyboard Commands.

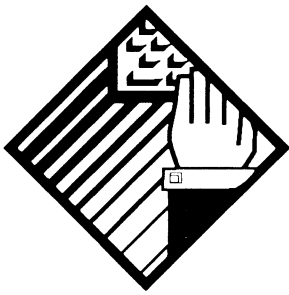
The best way to become familiar with these is simply to use them. A few minutes using each is all it takes and from that point on you will find a significant difference in your use of AmigaDOS.

The Shell also provides a useful search facility in that it allows you to seek out the last occurrence of a particular command, thus avoiding the need to scroll through a potentially long command history list. To use this simply type the start of the command name and then press <Ctrl-R>. For example:

DIR<Ctrl-R>

will locate the last time the DIR command was used. Typing <Ctrl-R> again will seek out the next previous occurrence and so on.

- You can get the same effect by holding down either Shift key and pressing the up arrow. This facility is case sensitive and so it is important that you use the same case.
- See Chapter 29 – *The Console Device* for details on console copy and paste.



10: Shell Commands

Overview

- > External or Internal?
- > Introduction to RESIDENT
- > Default paths
- > Making commands resident
- > Errors from RESIDENT
- > Using RESIDENT commands
- > What went wrong?

We have already said the Shell is a command-line interpreter or parser. Strictly speaking this isn't quite true – it's a convention which has been adopted since the Amiga's earliest beginnings.

The actual parsing (dividing of a line into its component parts) is done not by the Shell, but by the command itself. All the Shell does is separate the command from its command line and hand the whole lot over to the AmigaDOS library's EXECUTE call.

We'll cover the strange subject of libraries further on; suffice to say for now – it just works that way. Don't get confused between the EXECUTE *library call* and the EXECUTE *AmigaDOS command* we'll meet later!

The Problem

Commands come in two distinct flavours: internal and external. As a Workbench wizard you don't need to know that, but now you're becoming familiar with AmigaDOS proper you do. Internal commands are built into the machine's ROM operating system whereas external commands have to be loaded from disk *first*.

Internal commands are never loaded into memory – because they are already available in ROM. Technically, an internal command is very slightly slower than one which has been placed in RAM because of the way computers work, but the difference is so marginal it makes no odds. Besides, an external command has to be loaded from disk first and it is this which takes the time – measurable in seconds for really large ones!

External commands are also known as *transient* or *diskloaded* because they are only placed temporarily in the computer's memory. Once the command has completed, it is purged from memory and any *resources* it used are returned to the system. (Resource is just an all-encompassing name for things such as memory, disk drives, printer ports and the like.) However, they do have the advantage of being easily replaceable. If a bug is discovered in an internal command, it may be difficult or impossible to fix – thankfully this is rare.

The Solution

Clearly the whole process of command execution could be speeded up if the command was retained in memory, thereby obviating the need to load it into memory each time it is needed. In days gone by, we *used* to do this by loading commands into the RAM disk which is a very fast disk simulation – but things have come a long way since then!

The Internal command, RESIDENT manages this aspect. Its basic synopsis is as follows:

```
RESIDENT [[FILE=]<command name>]
```

We're limiting ourselves to the parts of the command we'll be using. You'll find a complete description, plus information on what makes up a resident command, in *Mastering AmigaDOS 3 – Reference*.

Let's take a look at the Internal commands first. Enter this:

```
1>RESIDENT
```

AmigaDOS responds by displaying a huge list of the commands currently residing in the system. Typically, these will be shown as illustrated in Table 10.1 below.

Name	Use count
Alias	INTERNAL
Ask	INTERNAL
Cd	INTERNAL
...	
.bra	INTERNAL
.ket	INTERNAL
.key	INTERNAL

Figure 10.1 Typical internal commands

All these commands are stored in the ROM operating system – Kickstart – and are instantly accessible from any Shell at any time. Each only uses a small amount of memory (about 4k) which is automatically allocated to the Shell! Apart from that, you won't notice anything strange about them – every one works just like any other AmigaDOS

command: only faster. There's a huge advantage to this approach which will become clear when you program in AmigaDOS – but we'll leave that topic for a later chapter.

- You'll notice at the top of this list a heading "Use count". This has to do with the multi-tasking operating system and will be covered later.

Internal commands are all well and good, but what about getting at those external (diskloaded) commands? Most, but not all, of the external commands can be made resident. Let's look at some commands that can: typically these are commands that you'll want to use all the time, but for technical reasons have not been included in the ROM. DIR is a good example in the current release. Remember how DIR is used to list the contents of a disk? But which disk?

Re-boot your machine from scratch to make sure everything is back to normal. Now grab your Extras disk and pop it in the internal drive, eject all other disks and enter this:

1>DIR DF0:

A system requester pops up asking you to insert Workbench – do that now. (Make sure you put Workbench in the internal drive if you have more than one). What did you get – a listing of the Workbench disk or one of the Extras disk?

You got Workbench, right? Here's why.

1. The disk in df0: is Extras – the one you want to examine.
2. When AmigaDOS parses DIR from the command line it looks for the command first in the resident list and then on the current disk (Extras) and fails to find it.
3. AmigaDOS realises the command is not there so it decides to look elsewhere for it by searching something called the "path". The path tells AmigaDOS to search the current boot disk – Workbench in this case.
4. AmigaDOS checks the current disks, can't find Workbench and asks for it back: "Please insert volume..."
5. You replace Workbench. AmigaDOS locates, loads and executes DIR.
6. DIR now has control and examines its arguments – which read DF0:. Aha, says DIR, and dutifully gets a directory of the disk currently *mounted* in the internal drive: Workbench.

Oh dear, oh dear. There are several ways around this problem, but we'll use the resident approach because it is simpler and illustrates the point.

We mentioned AmigaDOS uses something called the “path” to locate its commands. The path is actually a list of disks and directories AmigaDOS searches every time you enter a command. AmigaDOS commands proper live in a special directory on the Workbench disk called “C” – for Commands no less!

The C directory on the boot disk is automatically made a member of a special set of directories determined by the system at startup. It’s proper AmigaDOS name is actually, “C:” pronounced “C-colon.” Nifty idea that – because it means you can access any AmigaDOS command without every having to know where it is. In addition, this also gives us a very simple formula for making most commands resident: just enter “RESIDENT C:” followed by the command’s name and hey presto – you’ve got it!

Let’s try that now:

```
1>RESIDENT C:DIR
```

Nothing *seems* to happen apart from the chunk of a disk drive. But there’s more to that than meets the eye. Repeat the earlier example by getting a directory of the disk currently in the internal drive.

```
1>DIR DFO:
```

Try repeating that command with a number of different disks. Notice how you don’t constantly get asked for the Workbench and you always get a listing of the current disk? Smart, eh!

Resident Commands

To resident or not to resident – that is the question. Try this:

```
1>RESIDENT C:BindDrivers
```

object is not of required type

AmigaDOS has generated an error condition saying “Object is not of required type” and, as is typical with such things, the error message is about as clear as mud. What on earth is an object? Surely this command deals with commands, so why doesn’t the error say “Command not of required type” or “BindDrivers cannot be made resident because...”?

Serious questions indeed. For the answer we must remember that AmigaDOS is a part of an overall philosophy; and commands are only a small, albeit important, part of that. An “Object” can be anything loaded from disk – and RESIDENT can deal with more than just commands, although we have to concede the error could be more informative.

The command “BindDrivers” is special in that is fire-and-forget. It is only used once when the machine is started never again after that. Making it resident would be futile – so the practice is discouraged with an error.

Other command are noteworthy in that they cannot be made resident because they simply do not come up to scratch. Very large applications cannot be made resident either, but why would you want to? The implications of all this are fully discussed in the companion work, *Mastering AmigaDOS 3 – Reference*.

Where's Wally Now?

Enough of all that – try this:

```
1>RESIDENT C:Format
object not found
```

Oh dear, here we go again – another command that can't be made resident... or can it? Look at the error message closely. "Object not found". This suggests that AmigaDOS – strictly speaking RESIDENT – could not find the FORMAT command. Where is it then?

It's a matter of historical convention that commands such as FORMAT are not placed in the AmigaDOS command directory, but in another place altogether. Such commands are those used directly by none other than: Workbench. We'll let you into a secret, FORMAT lives in the system drawer: and AmigaDOS already knows that! It even supplies a command to determine where any executable command is. It's called WHICH and here's an example of its basic use:

```
1>WHICH FORMAT
Workbench3.0:System/Format
```

This tells us the FORMAT is in the system drawer of the Workbench3.0 disk. The result will look slightly different on your monitor because the disk's (volume) name is also given. This also tells us the command is in the current *path* because WHICH uses the path to locate a command in exactly the same way the parser does. This gives rise to the following command:

```
1>RESIDENT Workbench3.0:System/Format
```

It is a point of fact we can refer our boot disk universally in such circumstances as "SYS:" so the command is more simply written:

```
1>RESIDENT SYS:System/Format
```

You'll find an interesting script (AmigaDOS program) called AutoResident in *Mastering AmigaDOS 3 – Reference*, under WHICH that combines these functions into one automatic procedure.

Whose Fault?

Blame it on the bellboy – that's what I say. The principle of the error message should be reasonably familiar to you by now. However, occasionally AmigaDOS will simply throw up an error number with no text. For instance:

Last command failed because of Error code 205

The FAULT command can often provide the answer to this, or to remind you of a specific error message which you cannot remember. For example, you have an error message number of 120. Information on this can be sought using:

1>FAULT 120

Fault 120: argument line invalid or too long

Sometimes error numbers will have no text associated with them and simply reply with the word 'Error' and the number. In addition, you can include multiple error numbers on a line separated by spaces, should you wish to seek what might lie behind several error numbers. For example:

1>FAULT 221 98

Fault 221: disk full

Fault 98: Error 98

You can use WHY to provide information about the failure of the last command – especially in cases where the command has failed and not generated an error message. Here's a working example (although the >NIL: technique is too advanced to discuss at this stage).

1>WHICH >NIL:

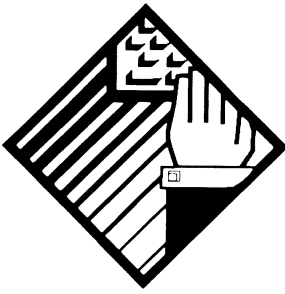
1>WHY

Last command failed because : required argument missing

In some instances (usually because the command worked in a way you did not expect) WHY will not have anything to work with and shows the following:

The last command did not set a return code

Return codes are covered in the chapter on environmental variables.



11:

Shell's Special Features

Overview

- > ESC key effects
- > Introducing ECHO
- > Prompts explained
- > Introducing PROMPT
- > Special prompt characters
- > Introducing ALIAS

As we've already discussed the Shell is the AmigaDOS command line interpreter. It takes your input and hands it to the required AmigaDOS command; then communicates any response back.

In addition to this basic operation, Shell has other facilities to make your life more interesting and simpler too. Two AmigaDOS commands are used exclusively by the Shell, and others allow us to experiment with some of the features afforded by the console device, CON:.

Introducing ECHO

AmigaDOS's ECHO command is primarily used with the script language to display messages on the screen, however it is worth examining some facilities this simple command provides. By consulting *Mastering AmigaDOS 3 - Reference* you will find that ECHO actually provides some natty facilities for string handling such as you might find in a high-level language such as BASIC; but we won't cover them here.

Echoing Linefeeds

The option NOLINE can be used in conjunction with ECHO to suppress a linefeed. ECHO also provides a mechanism whereby linefeeds can be issued from within the string of text. This is done by inserting the special character string '*n' (NEWLINE) in the text. For example:

```
1>ECHO "This is line one  
*nThis is line two"
```


will print on screen as two lines as follows:

```
This is line one
```

```
This is line two
```

And just in case you ever wish to use the '*n' combination in a line of text you can suppress the action just by inserting an extra asterisk, '**n', for instance:

```
1>ECHO "Use **n to issue linefeeds inside ECHO"
```

```
Use *n to issue linefeeds inside ECHO
```

Starry, Starry Nights

We saw in earlier, that the ESC key can be used from the command line to induce special effects on the screen. These effects range from changing the next style to italics to changing its colour. Because these effects are part of the console they can also be used from within your own commands by embedding them in ECHO strings, simulating the ESC key with the special combination '*e' (pronounced star-E or Starry – get it?).

```
1>ECHO "This is normal text"
```

```
This is normal text
```

```
1>ECHO "*e[3mThis text is in italics*e[0m"
```

```
This text is in italics
```

```
1>ECHO "*e[1mThis text is in bold*e[0m"
```

```
This text is in bold
```

```
1>ECHO "*e[4mThis text is underlined*e[0m"
```

```
This text is underlined
```

```
1>ECHO "*e[31mWBLACK *e[32mWHITE *[33mBLUE *e[0m"
```

```
BLACK          BLUE
```

```
1>ECHO "*e[1m*e[3me[4mVincent would have been proud!*e[0m"
```

```
Vincent would have been proud!
```

You will see what sort of things are possible. The full command set and effects possible from within ECHO and EVAL strings appears in Chapter 29 – *the Console Device*.



Quotes must be used around any ECHOed string that requires the use of a '*e' escape code.

Other commands that use the console display can also use these same ESC sequences to achieve visual effects. The combinations are endless and these ESC codes provide you with the means to brighten up your screen displays with very little extra effort.

Cue -> Shell

The first of Shell's 'Shell only' commands allows us to customise the prompt string, which, by default displays the current path and process number (see Chapter 25 – *Multi-tasking*).

The command's synopsis is as follows:

PROMPT [String]

PROMPT can be used at any time during a Shell session, but only affects the current Shell window – such operation is defined as being *local*.

This command is best viewed by example. We'll start these examples with a standard prompt:

```
1>PROMPT Hello>
Hello>
Hello>PROMPT Mark>
Mark>
Mark>PROMPT :
:
:PROMPT
1.SYS>
```

As you can see from this last example, when you give the command without an argument it automatically returns to the system's default prompt.

In these examples, we have used plain text – but the system applies other things such as the process number and current directory. You can add these to your own prompts by including a special *variable* where they will appear. These are as follows:

```
%N   The process number
%S   The current directory
%R   The RC (returncode) from the last command.
```

You can add these strings to any part of the prompt, but it is up to you to ensure they make sense! Each one should be separated with a *delimiter* such as a full stop (period) as in these examples. You should also included a terminating character to show where the prompt ends. In AmigaDOS this is usually a "greater than" symbol or right angled bracket: >; alternatively you might prefer a Unix-like percentage symbol. The choice is yours, but if you want to include spaces, the prompt sting *must* be enclosed in quotes. Also, if you want to include the %, it must be *escaped* with a second % – see the examples below.

Here's an example session where the current directory is WB3: and the current process is number 6:

```
1>PROMPT 1>
1>CD System
1>PROMPT %S>
WB3:System>PROMPT %N>
6>PROMPT %S.%N>PROMPT %%
%>PROMPT
6.WB3:System>
```

Alas, Poor AmigaDOS...

As you become more proficient with AmigaDOS you will discover yourself using more and more commands with the same (or very similar) syntax. Some of these can get very long and quite involved – FORMAT and DISKCOPY are two examples we've met already but there are a number of others such as CHANGETASKPRI!

The Shell provides a facility to assign an alias to a command configuration you use a lot. The command is called ALIAS and it has the following synopsis:

```
ALIAS <alias> <command>
```

ALIAS is not a difficult command to use once you grasp what it actually does. When the command line is scanned, an alias in the first position is expanded. The first argument from the command line is placed at the defined position within the expansion and any remaining arguments follow on immediately after.

Okay, the last part is a bit chewy, but you'll soon get the hang of it. Let's show some simple examples. Imagine you used FORMAT and DISKCOPY a lot. Entering the names is a bind, so we can define aliases for these thus:

```
1>ALIAS DC DISKCOPY
1>ALIAS FT FORMAT
```

Now we can replace the following lines:

```
1>FORMAT DRIVE DF0: NAME EMPTY
1>DISKCOPY FROM DF0: TO DF1:
```

with:

```
1>FT DRIVE DF0: NAME EMPTY
1>DC FROM DF0: TO DF1:
```

All right, they're a little shorter but we can improve on those. By including one pair of square brackets in the alias command, we can pass a single argument to any position. Look at the FT alias above. It would be much better if we could just pass the name of the disk drive to work on – here's how:

```
1>ALIAS FT FORMAT DRIVE [] NAME EMPTY
```

and we use it like this:

```
1>FT DF0:
```

Now let's get really clever! We'll assume you just want to format floppy disks DF0:...DF3: and include everything but the disk drive number. This gives us:

```
1>ALIAS FT FORMAT DRIVE DF[: ] NAME EMPTY
```

which is used thus:

```
1>FT 0
```

As you can see, aliases allow us to compact commonly used commands into next to nothing – provided we can identify what actually *needs* to be entered. It is interesting to note here, you can add other parameters after the main alias. So, to format a disk without the Trashcan we can add the NOICONS switch directly on the command line thus:

```
1>FT 0 NOICONS
```

ALIAS Effects

The special screen effects outlined above can be transformed into useful commands by using the Shell's ALIAS command. Type this directly into a Shell window:

```
ALIAS ITALICS ECHO "*e[3m"
```

And now type:

```
ITALICS
```

The net result is that the typeface in use from within the Shell changes to italics. Create a new command to reset this:

```
ALIAS NORMAL ECHO "*e[0m"
```

and type:

```
NORMAL
```

to return things to the way they were.

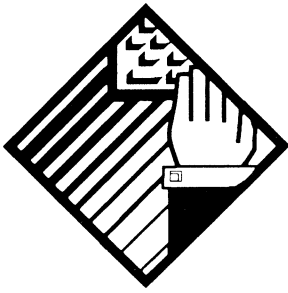
The potential here is quite good for all sorts of reasons, not least is that you can define a whole series of new script commands for use from within the Shell. These could be part of your Startup-Sequence file or better still a separate file that is invoked from the Startup-Sequence or by you whenever you feel that you are going to need it.

The listing below provides some of the definitions you could save under a new file in the S: directory under a suitable filename, EFFECTS might be appropriate. More control sequences are shown in Chapter 29 – *The Console Device*.

```
; File Name EFFECTS
ALIAS CLS ECHO "**ec"
ALIAS NORMAL ECHO "**e[0m"
ALIAS BOLD ECHO "**e[1m"
ALIAS WHITE ECHO "**e[2m"
ALIAS ITALICS ECHO "**e[3m"
ALIAS BLUE ECHO "**e[30m"
```

And of course the possibilities do not end here. The ESC commands can be used in the prompt sequence to give this effect:

```
PROMPT "**e[33m%n.%s *e[0m"
```



12: The Ram Disk

Overview

- > Introduction to the Ram Disk
- > Advantages of using RAM:
- > Sizing up the Ram Disk
- > RAM: after a soft reset
- > System directories in RAM:

To say that the Amiga's Ram Disk is useful is like saying that Rasputin was a little odd! The Ram Disk comes in two flavours which act rather like floppy disk drives to the extent that they are given device names.

Whereas DF0: and DF1: are used for the floppy drives, the names RAM: and RAD: are assigned to the Ram Disk drives. We will examine the use of RAM: in this chapter and come back to the use of RAD: later on.

RAM is an acronym for Random Access Memory. This is the memory into which your programs and applications are loaded from disk. By partitioning off an area of this memory, you can use it to save and load programs and data.

The partitioned area is known as the Ram Disk, and AmigaDOS automatically installs it for you when you boot the system.

The Pros and Cons

If you are using a single drive system you will already have encountered a number of hindrances when going about the examples we have worked through. Not least amongst these is the need to have to continually swap over disks not least between the Workbench disk and your own 'Empty' disk. Similarly, many of the commands provided by AmigaDOS are very difficult to use because they are designed to act on a disk in the disk drive but also need the Workbench disk — a conflict of interests. Even on a dual drive system, there can be a

considerable amount of disk juggling especially as some of the larger programs require a dual drive system to work. The Ram Disk can be used to solve these problems.

The biggest drawback with a Ram Disk is that it is *volatile*. Being a software simulation of a hardware device it is only active while the computer is switched on. The moment you remove the power or 're-boot' its contents are lost forever. If you wish to preserve them you must save the Ram Disk contents onto a floppy disk first. However this is generally a small price to pay for the benefits.

Being a mechanical device there is always going to be a physical delay when software loads. Every time you access a file or data on a traditional disk it takes time: time for the command to be sent, time for the disk drive motor to start up, time for the data to be read from the surface of the disk, and time to transfer it into the computer's memory – it's a wonder we don't fall asleep waiting! By using a Ram Disk these mechanical restrictions no longer apply. Access is virtually instantaneous.

Size and Use

Portioning a section of RAM off for use as a Ram Disk has another drawback. The memory reserved for the Ram Disk is now no longer available for programs. This isn't generally a problem, and is made less so by the fact that AmigaDOS always adjusts the size of the Ram Disk so that it only needs enough room to store the file it holds. Up until now we haven't used the Ram Disk at all and as such it is empty, and so it doesn't require any RAM at all, therefore AmigaDOS doesn't give it any! This isn't strictly true: the Ram Disk needs the minimum of 1k to operate when it's empty. However it's such a small amount – 0.05% of the total on a 2Mb A1200 – it can almost be forgotten about.

In practice you treat the Ram Disk as a physical drive for all purposes other than saving anything important before you switch off. You create directories and move up and down through them in a similar fashion. AmigaDOS commands can be issued to access files in the Ram Disk simply by substituting the RAM: device name as appropriate.

Workbench Ram Disk

For our first look at the Ram Disk we'll return to the Workbench. If you open the Ram Disk window you'll see that it's much like any other disk window. Notice that the title bar shows that the Ram Disk is full, even though it's empty! The reason is that the Ram Disk isn't occupying any space at the moment so it has none to spare (actually there is a small overhead of RAM in use).

Open the Workbench disk and copy the Format and FixFonts icons into the Ram Disk by dragging them into the Ram Disk window. This can be done by moving each across separately or by selecting them both together and moving them in a single step.

It is possible to select all the files in an active window by dragging a box around them, or by using Select Contents from the Window menu.

Box dragging is a technique borrowed from the Macintosh. All you have to do is to hold the left mouse button down and move the mouse. As if by magic, a rectangular box appears ghosted around the icons. When you release the left mouse button, any icons covered (even partly) by this area are selected. To move the icons, simply hold the Shift key and drag the icons as explained above. You can deselect single icons by clicking on them while the Shift key is depressed.

Remove the Workbench disk from the disk drive and double click on the Format icon. Note that it starts up without need of the Workbench disk and that it happens much more quickly.

Run FixFonts by double clicking on its icon in the Ram Disk. Failure. Despite copying the program across you are asked for the Workbench disk. This is because Fixfonts needs access to more information that it cannot find in the Ram Disk, so it asks for the Workbench disk. This shows that simply copying a program onto another disk isn't always good enough. For this reason you must be careful when you are copying files to ensure that you copy everything you need. This knowledge only comes with experience and that will come as you delve deeper into the wonders of AmigaDOS. Read on.

Amiga Reset

Occasionally you will want to restart your Amiga. There are two ways to do this. Firstly the most obvious, switch the power off and wait *at least* 30 seconds before switching on again. A second reset is also available directly from the keyboard. Either side of the space bar you will see the Amiga keys signified by the letter 'A'. Keep these pressed down and push the CTRL key. You will need two hands to do this for good reason – it prevents it happening by accident. In books and magazines you will often see this key combination referred to thus:

Ctrl-Amiga-Amiga

or:

<Ctrl><Amiga><Amiga>

(Amiga fanatics call this the Vulcan Nerve-death grip – from the original Star Trek’s Mr Spock. Fans of Star Trek TNG can liken this to a trigger-happy Worf in a bad mood.)

Try it for yourself and reopen the Ram Disk. The files you copied across have gone.

Ram Disk and Shell

From the Shell the Ram Disk is treated in much the same way as a physical drive. It remains volatile so you must remember to backup any information held in it to a floppy disk and it uses the device name RAM:.

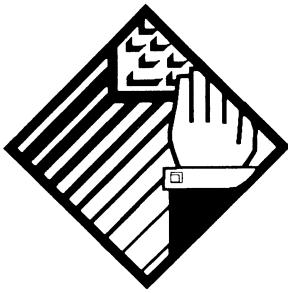
You can make the Ram Disk the active selected drive like this:

```
1>RAM:
```

If you now directory the Ram Disk using DIR you will notice that it contains several directories which AmigaDOS creates during the boot sequence. Each has a special function and is employed by AmigaDOS transparently. The use of each directory is described briefly in Table 12.1 for the inquisitive and their exact role will become apparent as we learn about the relative subjects in due course.

<i>Directory</i>	<i>Function</i>
clipboards	Clips from the clipboard.
env	Global environmental variables.
env/sys	User preferences files
env/multiview	Private MultiView files
t	Used to store script temporary files.

Table 12.1. The Ram Disk Directories.



13: Wildcards and Pattern Matching

Overview

- > The theory behind patterns
- > Pattern matching characters
- > List of example patterns

If you have ever played any card games you may have occasionally used the rule where 'Jokers are wild'. The Jokers are left in the pack of cards and if you are dealt one it can be used to represent any card you wish.

Wildcards work in much the same way and this is where the name derives from. Certain characters can be used within filenames to represent any character or set of characters you may wish. The most common combination is:

#? Represents any number
of any characters

Pattern matching stops when any non-wildcard character string is encountered – just about anything in other words – but can be re-started at any point. This means that you can specify patterns of almost any length and complexity.

Patterns can be used with a large number of AmigaDOS commands – those that act on objects: whether they be directories, files or any combination of the two. It is not possible to apply a pattern to a device: unless the device is acting upon an object.

Don't worry about this – it'll all become second nature soon and although you'll forget the theory, you'll be able to put it into practice.

Table 13.1 lists the wildcard characters and their actions. These things do tend to look a little off-putting at first, but you'll learn from the examples that they're really quite simple. – keep them that way!

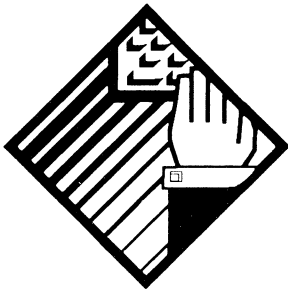
<i>Wildcard</i>	<i>Action</i>
#c	Matches any repeating sequence of character "c"
?	Matches any single character – must find at least one
%	Matches zero or one occurrence of any character
[...]	Matches one or more specified character(s) at any position. It is not case-sensitive and therefore, [ABCF] is the same as [abcf]. The characters do not have to be in any order.
#[...]	Matches zero or more specified characters at any position. Not case sensitive – useful when used with numbered files.
[x-y]	Matches characters in the ASCII range x-y. The range is not case sensitive, so [A-Z] would match all letters in the alphabet, but ignore numbers. Note: the range must be ascending in ASCII value from left-to-right.
#[x-y]	Matches zero or more characters in the range x-y. The range is not case sensitive, so [0-9] would match numbers only. Note: the range must be ascending in ASCII from right-to-left.
Modifier	
~	Reverses the action of a basic pattern or group
(	Opens a group of wildcard characters – must be closed.
	Separates alternate wildcard patterns in a group.
)	Closes a group of wildcard characters – must be opened.
'	Escapes the following wildcard character.

Table 13.1. The wildcard and pattern matching characters

It is commonly thought these patterns can only be used with file-based commands such as DELETE and COPY. You can use these characters anywhere a pattern is called for – including in a SEARCH string. Table 13.2, over the page, shows some typical examples: looks 'orrid doesn't it? Just be thankful you've got this power – you don't have to use it unless you want to!

This Matches	These
#?	– Everything!
~#?	– Nothing!
A#?	– Strings starting with A
#?.info	– Strings ending in .info
(A B)#?	– Strings starting with A or B
#?(info pat)	– Strings ending with info or pat
~(#?.info)	– Everything but dot-info files
~(.info .pat)	– Everything except dot-info and dot-pat files.
[ADG]#?	– Strings starting with a,d or g
#[A]B#?	– Strings starting with 0 or more letters "a" and continuing B...
[A-G]#?	– Strings starting with: a,b,c,d,e,f or g.
[A-C X-Y]#?	– Strings starting with: a,b,c, x, or y
A?B	– Literal question mark: A?B
A#B	– Literal hash mark: A#B
A'?'B	– Literal hash-question mark: A#?B
A"'"B	– Literal tick: A'B
AB#?D	– ABCD, ABCCD, ABfoobarD, etc.
Master#?.1.#?	– Mastering DOS2.1.DOC
A??B#?D	– AeeBxxxxD, AxzBzD, A12B3456D
A(B C)D	– ABD, acd
A(#B C)D	– ABD, ABBD, ABBBBBD, AbbbbbbD, ACD
A(B C?)D	– ABD, ACD, Acld, ACaD
A%C	– AC, ABC
Test[123]	– String "test" ending with 1, 2 or 3.
Test#[123]	– String "test" ending with any combination of 1, 2 and 3.
#?[0-9]	– Anything ending in any single digit between 0 and 9
#?#[0-9]	– Ending with zero or more digits between 0 and 9.
Test#[0-9]	– String "test" with zero or more digits between 0 and 9.
#?[0-9]#?	– Any string containing at least one digit somewhere.

Table 13.2. Examples of wildcard usage – try to keep yours simple.



14:

Managing Directories

Overview

- > The default prompt
- > Changing directories from Shell
- > Using CD with patterns
- > Basic use of MAKEDIR
- > Creating sub-directories
- > Using MAKEDIR with multiple arguments

"I shall shoot the next person to call me an expert... Hey you, hand me that water pistol!"

– Mark Smiddy, Confessions of a Computer Journalist

Open the Shell and make a mental note of the prompt. Usually, when we want you to enter something, it will be shown preceded by a standard prompt '1>'. For instance:

1>DF0:

means type 'DF0:' into the Shell window. However, for these examples we are going to show the Shell prompt. Make sure your Workbench disk is in the internal drive and enter:

1>DF0:

1.Workbench3:>

The prompt changes and now delivers the name of the disk drive in DF0:.. Now let's flip through some other directories. Notice in these examples, how a disk drive is always followed by a colon.

1.Workbench3.0:>System

1.Workbench3.0:System>RAM:

1.RAM DISK:>t

1.RAM DISK:T>DF0:

1.Workbench3.0>DF0:

Important: in these examples, we've "dropped" the usual CD command part which is correct and faster in AmigaDOS 2 and higher. You must use CD if you find yourself confronted by an earlier version!

Using CD

The CD command has a number of uses but most of them are hangovers from the previous systems. For instance if you wanted to make RAM: the current directory, then you would have to enter:

```
1>CD RAM:
```

and to get back to the system disk you would enter:

```
1>CD SYS:
```

In light of the new system it is fair to say this looks a little clumsy – but it served a purpose at the time.

CD also had the effect of displaying the current directory (which as we have seen is now part of the Shell prompt). This facility is still available – and with good reason, because you can define a prompt which is no more than a simple arrow! CD now has the following synopsis:

```
CD [DIR=<dir|pattern>]
```

Therefore it can be used like this:

```
1>CD
```

```
Workbench3.0:
```

```
1>CD System
```

```
1>CD
```

```
Workbench3.0:System
```

“Boring”, we hear you cry – “what’s the point of all this?” Well, imagine you had to enter a directory somewhere deep down in the hierarchy. Let’s say it had a path like this:

```
df0:Wordprocessing/Wordworth/Documents/Letters/Memos
```

That would mean you could enter the directory “Memos” like this:

```
1>df0:Wordprocessing/Wordworth/Documents/Letters/Memos
```

or like even like this:

```
1>df0:
```

```
1>Wordprocessing
```

```
1>Wordworth
```

```
1>Documents
```

```
1>Letters
```

```
1>Memos
```

any takers? No! We thought not – and there wouldn’t be a point to all this unless AmigaDOS had a solution. It does – because you can apply a pattern to directory pathnames: provided you use the CD command. Therefore we can change from wherever we are to this directory thus:

```
1>CD df0:W#?/W#?/D#?/L#?/M#?
```

Much more sensible, don't you agree? There is a snag to this method though. You may remember from the chapter on pattern matching that `#?` matches anything. By adding a letter at the start of this we can force AmigaDOS to look for words *beginning* with that letter.

That means for example, the `W#?` in part of the path could match: Wordprocessing, wibbly-wobbly, wally, wallbanger and thousands of other possible combinations. AmigaDOS isn't daft though, it simply reports an error – here's one you can try:

```
1>CD SYS:L#?
```

More than one directory matches

and you can see exactly which ones are causing the problem using `DIR` like this (use Shell's editing facilities to insert the new command):

```
1>DIR SYS:L#?
```

```
Workbench3.0:Locale (dir)
```

```
Workbench3.0:Libs (dir)
```

```
Workbench3.0:L (dir)
```

Fixing this is simply a matter of deciding which directory you actually want and finding a pattern to match. In this example the following does the trick for `Locale`:

```
1>CD SYS:L#?e
```

and this does it for `Libs`:

```
1>CD SYS:L#?s
```

New Directions

The process of creating directories using AmigaDOS commands is straightforward. Remember that a directory is a storage area and although you don't have to use them, they are incredibly useful. Moreover, judicious use of directories will keep your disks easier to manage later. The command for creating an AmigaDOS directory goes like this:

```
MAKEDIR <{[path] new directory}>
```

where `<name>` is the name you wish to call the directory including the disk name or the drive name too. Let's add some directories to the Ram Disk.

```
1>RAM:
```

```
1>MAKEDIR Directory
```

```
1>MAKEDIR A.n.other
```

Now, lets take a look at that:


```
1>DIR Test:
A.N.Other (dir)
Clips (dir)
Directory (dir)
Env (dir)
T (dir)
```

As you can see there are already three other directories residing in the Ram Disk. These are created (by MAKEDIR) when you boot the machine with Workbench and are essential for the machine's operation. When examining the Ram Disk from Workbench reveals these directories are not shown – unless you switch to Window...Show...All Files. That is because MAKEDIR does not automatically create the icon's .info file. These .info files contain information which is only used by Workbench and have no relevance to AmigaDOS.

Directory Structure

The main directory on any disk is called its root directory and is signified by the colon in the device or volume name. For instance "RAM:" is the root directory of the RAM: disk device and "Workbench:" is the root directory of a disk (volume) labelled Workbench. You may also create directories inside directories. To see this in action it is helpful to work on a blank disk.

Get a new disk and format it with the name Test. You can do this from Workbench if you wish or use AmigaDOS if you prefer. We've already covered the command but here is the command line to save you looking it up:

```
1>FORMAT DRIVE DFO: NAME Test NOICONS
```

We can now refer to the new disk as the volume "Test" and access its root (base) directory as "Test:". See Figure 14.1.

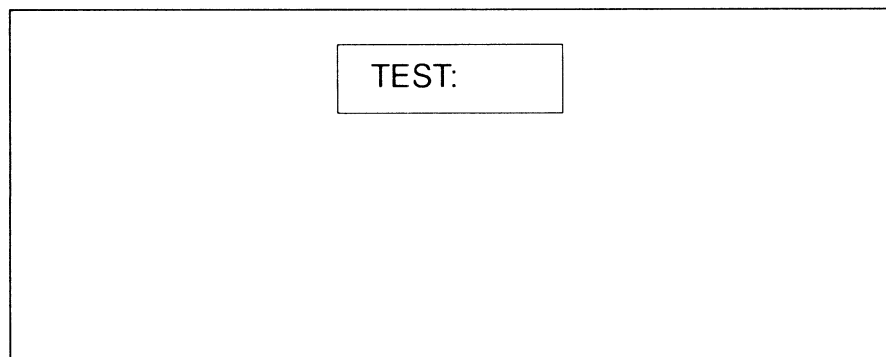


Figure 14.1. A blank disk has no structure.

The NOICONS option tells AmigaDOS not to place the Trashcan on the new disk; it'll just get in the way and we'll not show it in the examples here. You may switch it off from Workbench by clearing the "Put Trashcan" checkbox. Keep this disk handy – we'll use it for the examples in this chapter.

Now enter the following:

```
1>RESIDENT C:MAKEDIR
```

That just makes sure you'll have the command ready and will save fiddling around on a single drive system.

Now let's create some directories. Pop the "Test" disk in the internal drive and enter this:

```
1>MAKEDIR DF0:One
```

```
1>MAKEDIR DF0:Two
```

```
1>MAKEDIR DF0:Three
```

This gives us a disk layout as shown in Figure 14.2. You can see this for yourself by getting a directory thus:

```
1>DIR Test:
```

```
One (dir)
```

```
Two (dir)
```

```
Three (dir)
```

Notice how in this example DIR's argument is the name of the working disk – Test: and not the name of the disk device df0:. On a single drive system you'll be asked to change disks a couple of times – but you'll get the correct list because you asked for the disk by its volume name.

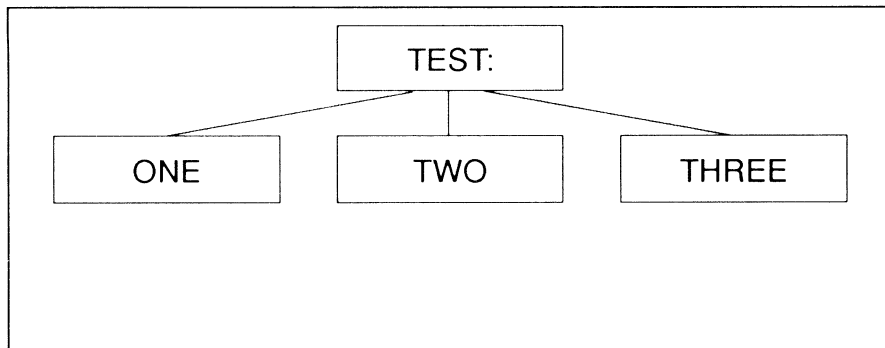


Figure 14.2. This disk has three sub-directories in the root.

Subdirectories

Now let's make the current directory one of the directories on the new disk:

```
1>Test:One
```

and add some more directories in the current directory like this:

```
1>MAKEDIR One
1>MAKEDIR Two
1>MAKEDIR Three
```

(You might like to use Shell's history and editing facilities to do this).

Our disk now has a structure like the one shown in Figure 14.3. You can examine the whole disk like this:

```
1>DIR Test: ALL
```

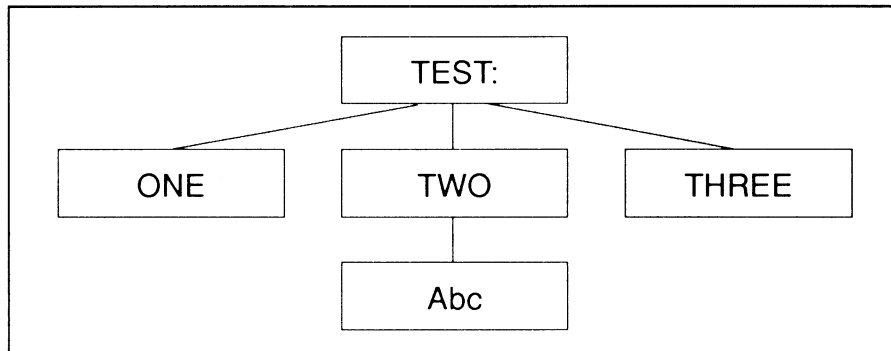


Figure 14.3. This disk has subdirectories of subdirectories.

There are a couple of things to consider here: not least of which is the path. Paths are a fairly tricky subject which you will only master through experience, so don't let them get you down. As we saw much earlier, AmigaDOS uses special characters to separate paths: the colon (:) is used to signify the root directory or a disk and the forward slash (/) is used to separate directories in the hierarchy. We can use this to our advantage when using directory-based commands such as MAKEDIR.

For example, we can create a subdirectory called "Abc" in the directory "Two" like this:

```
1>Test:Two
1>MAKEDIR Abc
```

or

```
1>MAKEDIR Test:Two/Abc
```

This second may look more complex until you consider it is just the previous two-step command combined into one! We have explicitly told MAKEDIR where to create the new directory.

Moreover, since MAKEDIR will accept multiple parameters we can take things such as this:

```
1>MAKEDIR DFO:One
1>MAKEDIR DFO:Two
```

```
1>MAKEDIR DF0:Three
```

and write them like this:

```
1>MAKEDIR DF0:One DF0:Two DF0:Three
```

Relative Paths

Most DOS users would agree paths are tricky concepts to get your brain around – especially when things start to get more esoteric. As you have seen it is possible to change to a disk or directory by typing its name, so:

```
1>Test:
```

```
1>Two
```

and we can execute that example in one step thus:

```
1>Test:Two
```

Now let's make some new directories in that directory at progressively lower levels:

```
1>MAKEDIR Test:Two/A
```

```
1>MAKEDIR Test:Two/A/B
```

```
1>MAKEDIR Test:Two/A/B/C
```

```
1>MAKEDIR Test:Two/A/B/C/D
```

Now change to the "D" directory like this:

```
1>Test:Two/A/B/C/D
```

So what would the following do?

```
1>/
```

Try it! You'll find yourself back at the next directory up the tree! Try this then:

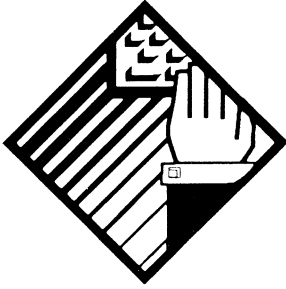
```
1>../
```

and you back up two more directories! Before we leave this topic, just try this:

```
1>MAKEDIR /E
```

That may look a little odd – but it actually creates a new sub-directory in the directory currently above the one we're in – which is Two.

- Take a break now and we'll come back to this later.



15: Managing Objects

Overview

- > What is an Object?
- > Object related commands
- > Duplicating an object
- > Changing an object's name
- > Moving an object
- > Deleting objects

The word "object" is AmigaDOS terminology to describe any physical entity on a mass-storage device: that is a file or directory.

A directory can be a user sub-directory, directory assignment or logical link. A file can be a user's text file, a program or command.

Certain commands only work on certain types of object – we have already met **RESIDENT** which acts on specially designed AmigaDOS *executable* code.

In this chapter we're going to look at commands which manipulate objects. This may seem a little grand – but it encompasses all those little housekeeping tasks we do almost daily. Copying files, changing their names and moving them around.

Backups and Cock-ups

If there's one thing we can't stress strongly enough it's to keep backup copies of your files. Backups are more correctly called *redundant* copies although this word is more usually associated with a person being *made redundant* and put "on the dole".

However this is not a book on semantics. The backup is redundant inasmuch as you have another identical copy – but if something happens to it then the redundant backup suddenly becomes very useful indeed!

Backups aside, you will often find it is necessary to copy files from one disk to another, or between directories, for a variety of different reasons. We won't concern ourselves with the whys here, just the how.

Duplicating Objects (COPY)

The AmigaDOS command to make a duplicate of any object is summarised below. We've only included the relevant options here, as this command is capable of far more.

COPY <[FROM=]{source}> <[TO=]destination> [ALL]

Where –

source A file, group of files, pattern, directory or device.

destination A file, directory or device.

If that sounds dreadful, it isn't. There's just more to this command than meets the eye! For instance let's copy the **FORMAT** command from the System directory on your Workbench disk to the Ram Disk:

1>COPY SYS:System/FORMAT to RAM:

There, that was painless wasn't it? Look at the Ram Disk and you'll see that **FORMAT** has been copied. In fact, you will now be able to execute the command without having to wait for a *s l o o w* floppy disk drive.

Now look in the Ram Disk from Workbench. No cheating, now – make sure that Window...Show...All Files is switched off. You can't see it! That's because each Workbench object must have a dot-info file to go with it. These items contain the picture of an icon plus any Tooltypes information. Okay then, we'll copy the dot-info file too:

1>COPY SYS:System/FORMAT.info to RAM:

Is that right? Well, it'll do; but it would have been much simpler to use a pattern in the first place: and make sure the whole thing went over in one go. We'll do that now:

1>COPY SYS:System/FOR#? to RAM:

FORMAT.info..copied

FORMAT..copied

Simultaneous COPY & RENAME (at Destination)

We'll meet the **RENAME** command proper in just a moment but there are times when it might be handy to give a file a new name while you copy it. Why? Trust me – it's too early to explain. Here's how you do it using **FORMAT** as a source file:

1>COPY SYS:System/FORMAT to RAM:F

1>COPY SYS:System/FORMAT.INFO to RAM:F

"Cor blimey guvnor, they didn't use a wildcard!" Ah, you spotted that one? Well there's a good reason for that which you'll see in just a moment. For now you might like to try using the **"F"** command now present in your Ram Disk: and available from Workbench.

Multiple Copies, No Patterns

It's worth mentioning here that, in some circumstances, you can use COPY with multiple *source* arguments. That means you can copy more than one thing at a time to the same destination and use a pattern to boot. You'll see how this works when you use it. For the moment we'll leave you with a working example:

```
1>COPY S:?PAT C:DIR S:S#? to RAM:
```

Simultaneous COPY & MAKEDIR

Why would you want to use COPY to create a new directory? Why not! Why spend two pounds, when you can get the same thing for a quid? Why buy Radium Automatic when Sea Foam does the same job?

In practice, this feature can be amazingly useful in all sorts of situations and best of all you don't even have to remember to add any esoteric switches: just the name of the directory as a destination! Here's an example using the entire System directory:

```
1>COPY SYS:System to RAM:STm
```

```
RAM:STm      [created]
```

```
DiskCopy.info..copied
```

```
DiskCopy..copied
```

```
...etc.
```

It may not be obvious at this point, but you cannot create more than one directory on the command line in this fashion. You'll have to use MAKEDIR to do that. It's also interesting to note that patterns are not *required* although you can use one. AmigaDOS creates a directory under the following conditions:

- The source object is a multiple selection, directory or pattern.

and

- An object of the same name does not exist on the destination.

Copying Directories

In the last example we used copy to create a directory on the destination. But what if you just wanted to copy some or all of a directory? Actually, the command is almost identical – because in essence, AmigaDOS is doing the same job. As before you need to isolate what's going where. For the sake of this example, we'll copy the Utilities directory to the Ram: disk.

```
1>COPY SYS:Utilities to RAM:
```

```
Clock.info..copied
```

```
Clock..copied
```

```
...etc
```


This is exactly the same as:

```
1>COPY SYS:Utilities/#? to RAM:
```

But you may already have a suitable directory. Let's create a dummy one in the Ram Disk just for the sake of example:

```
1>MAKEDIR RAM:Utils RAM:Utils/Time
```

You'll notice this creates a directory "Time" which is a sub-directory of Ram:Utils. This is for example only – but it demonstrates something you might want to do. Here's the command which takes Clock and its icon and places it in Ram:Utils/Time.

```
1>COPY SYS:Utils/Clock#? to RAM:Utils/Time
```

Copying Directories with Subdirectories

There's more to COPY than meets the eye: as you can see (*arrrg – Ed.*). One of the more important features is its ability to duplicate a complete directory structure. That means a directory, its siblings and children, their children and so on. The facility is the same as DIR ALL – which as you may remember lists all directories and subdirectories. At this stage, we'll leave you with an example to experiment with.

```
1>COPY DEVS: to RAM: ALL
```

Copying From and To the Current Directory

Just when you thought it was safe to go back to your keyboard and relax, AmigaDOS throws you another loop – double-quotes (""). These have the effect of referring to the current directory when an argument is required. Do what?

Okay, think of it this way. If you use DIR in the following fashion:

```
1>DIR
```

you get a sorted listing of the current directory. If you supply an argument, say "SYS:" like this:

```
1>DIR
```

you get a listing of that directory instead.

COPY does not work that way though. You must supply something for the source and destination arguments – in other words, it doesn't make sense to enter:

```
COPY from to RAM:
```

or

```
COPY from RAM: to
```

Let's be honest, it looks pretty stupid too! AmigaDOS will trap errors such as these and complain – so you can try these examples if you wish. However, you *can* refer to the current directory by supplying double-quotes to *stuff* the required argument, viz:

```
1>COPY from "" to RAM:
```

or

```
1>COPY from RAM: to ""
```

You can add any other options to the command line, since AmigaDOS treats the quotes as if they were a real argument.

Changing an Object's Name (RENAME)

You can change the name of any object, be it a file or sub-directory, with the RENAME command. You cannot change the name of a disk with this; use RELABEL to do that – or better still do it from Workbench! The basic synopsis for this command is as follows:

```
RENAME [FROM=]<object>> [TO|AS=]<object> [QUIET]
```

This means you must specify the name of the object you want to rename and its new name. Sound easy enough, eh? To see this in action we'll take you step-by-step through a sequence which you could use:

```
1>COPY S:SPAT to RAM:
```

```
1>RAM:
```

```
1>RENAME SPAT as SPIT
```

Note in this example, which you can try for yourself, we're very careful to specify the working (current) directory. There's a very good reason for this – as you can see from the next example. Don't enter this, just take the error as read:

```
1>RAM:
```

```
1>RENAME S:SPAT as SPIT
```

```
Can't rename S:SPAT as SPIT because rename across devices
attempted
```

The error says it all – but it does not say why you cannot rename across devices. There is a good reason for this as we'll see in the next section.

Moving an Object (RENAME)

RENAME can be used to move an object around the directory structure on any disk. For this reason, you should be very careful when attempting to *rename* something that includes a path. If the source and destination paths are different and on the same drive, the object is moved; otherwise an error is generated. If you want to

move an object between drives you should use a combination of COPY and DELETE. A script to make the decision for you is described in *Mastering AmigaDOS 3 – Reference* under RENAME.

For illustration, we'll copy the contents of the S directory into Ram and manipulate them from there. You may follow this example at home quite safely:

```
1>COPY S: RAM:S  
RAM:S      [created]  
1>RAM:
```

Now we have the entire contents of S in Ram (they're just little text files) we can create a new directory...

```
1>MAKEDIR Test
```

and move one of the files into it...

```
1>RENAME S/Spat TO Test  
Renaming RAM Disk:S/Spat as RAM Disk:Test/Spat  
1>DIR Test  
Spat
```

Sure enough – there it is. There's nothing magic in this, it's a simple matter of changing a "link" in the filing system structure – a couple of bytes! You can do the same thing with multiple objects too... In this example, we'll move the remainder of the files to "Test" and back again. You'll notice use of the QUIET switch here which stops all those annoying progress messages:

```
1>RENAME S/#? TO Test QUIET  
1>RENAME Test/#? TO S QUIET
```

Neat that. You may wonder what will happen if an object already exists in the source directory. Let's find out:

```
1>COPY S/Spat to Test  
1>RENAME Test/Spat to S  
Can't rename RAM Disk:Test/Spat as RAM Disk:S/Spat because  
object already exists
```

Tut. (This error will stop a multiple RENAME – watch out for that!) What are you going to do now? Simple, just change the name of the destination object, like this:

```
1>RENAME Test/Spat to S/Splot
```

Renaming Directories

Directories are no different to files as far as most AmigaDOS commands are concerned – that is why they are called “objects”. However, when you rename/move a directory you affect its entire contents: subdirectories too! This is rather handy, but it seems like a strange concept until you get used to it.

Important: if you want to move a Workbench-based directory, do it from Workbench or make sure you move the dot-info file to the same location!

- It's interesting to note, if you try renaming using a filename as a directory, AmigaDOS reports the error: “no more entries in directory”.

Removing Objects (DELETE)

Delete is a very dangerous command – more so since Commodore included an option to delete files that had been explicitly protected against deletion! This option is covered in the companion reference work, and we will not discuss it further here. The command's basic synopsis is as follows:

```
DELETE <{object|pattern}> [ALL] [QUIET]
```

Removing a File

Basic file removal with delete is even simpler than using a copy. In this example, we'll copy a text file from the Workbench disk and delete the copy.

```
1>COPY S:spat to RAM:
```

```
1>DELETE RAM:spat
```

Removing a Group of Files

Deleting a group of files can be done either by patterns, multiple files or a mixture of both. In this example, we'll copy a group of files from Workbench – those poor scripts again – and delete them.

```
1>COPY S:?PAT to RAM:
```

```
1>DELETE RAM:DPAT RAM:SPAT
```

or using a pattern:

```
1>DELETE RAM:?PAT
```



You must be very careful using patterns with DELETE in release 2 and higher. In old versions, the following line would generate an error:

```
1>DELETE RAM: #? ; DO NOT ENTER THIS!!!
```

in later versions it attempts to delete the Ram Disk – which it can't and then proceeds to delete the entire contents of the current directory. *Ouch!*

Removing a Directory

Since directories are storage vessels, it would be unwise to remove them while they still have contents. In fact, the structure which allows us to move entire directories or files in an instant, is the structure which prevents us from removing a directory! Files are attached to a directory like leaves on the branch of a tree. If you remove the branch, the leaves naturally go with it!

If you try to delete a directory that has one or more entries, AmigaDOS responds thus:

```
1>DELETE S
```

```
S    Not deleted: directory not empty
```

There are several solutions to this problem, and which one you use is decided by what action you actually want to take. For example, did you want to remove the directory or just its contents?

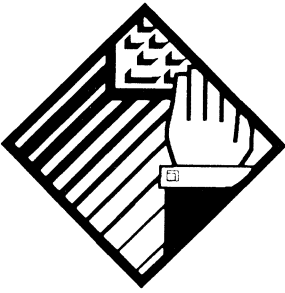
Here's the way to get rid of an entire directory – there's no warning and no way back from this action: use it with care! This example deletes a temporary directory we created on the Ram Disk earlier in the chapter:

```
1>DELETE RAM:Test ALL QUIET
```

A much safer bet is to RENAME any files/directories you don't want to the Trash and empty the Trashcan from Workbench. Typically, you might use a line such as this:

```
RENAME :Foo/Baa/#? TO :Trashcan
```

You should note from this example though, you can only use the Trashcan on the *same* disk as the objects you are trying to remove.



16: Advanced Use Of DIR

Overview

- > Using DIR with paths
- > DIR with pattern matching
- > DIR's switched options
- > DIR in interactive mode

The back slash (/) can be tagged onto the end of the DIR command, to allow you to catalogue the directories above you. For instance, if you were in the Wordprocessing directory and typed DIR you would get no listing unless you had already created some files there. However:

```
1>DIR /
```

would list the directory above which contains the directories and files we have already mentioned and created, but will not change your directory level.

In addition it is also possible to use full file path names in association with the DIR command should you so wish. For instance, on a dual drive system you may wish to catalogue the disk in drive DF1: which can be done using:

```
1>DIR DF1:
```

Commands such as:

```
1>DIR  
DF1:Wordprocessing/Club
```

and

```
1>DIR DF0:ARexx
```

are also legitimate.

However, this basic use is not always suitable for a particular purpose. For instance, you had 200 items in one directory and wanted to list just specific types, this could be rather troublesome.

In an earlier chapter we discussed the use of filename *extensions* which act as "aide memoires" to the contents of a file. For example, we suggested that letters were followed by the extension ".LET". Similarly, extensions such as .ASC

and .TXT suggest readable ASCII files; whereas .IFF suggests something to be viewed with a suitable application such as MultiView (AmigaDOS 3.0 and higher).

We can apply patterns to DIR to ensure it picks out the relevant objects. For instance, if we wanted to list the items in the current directory which had the extension "IFF" we could use this:

```
1>DIR #?.IFF
```

Similarly, we can include a disk drive name like this:

```
1>DIR df0: #?.IFF
```

and a directory path too:

```
1>DIR df0:Pictures/ #?.IFF
```

Combined Patterns

This is all well and good, but what if you wanted to include all text files in a listing? Say for instance a directory contained both .ASC and .TXT files. Here is how to do it.

```
1>DIR #?(.TXT|.ASC)
```

in fact, you don't even need the dot – which means you could enter:

```
1>DIR #?(TXT|ASC)
```

You can also select the files which *do not* match a given pattern by negating the whole group with a tilde (~) like this.

```
1>DIR ~(#?(TXT|ASC))
```

When using patterns with DIR you might encounter the following errors:

No more entries in directory

Object not found

bad template

Of these, only the last one is of any real consequence: it just means some part of your pattern is wrong – you may have missed a bracket out of a group for instance. *Always try to keep patterns simple.*

The first error (no more entries...) usually only shows up when you ask for it with WHY. It's an internal warning used by AmigaDOS to tell DIR it has completed searching the directory. "Object not found", on the other hand, probably means you misspelt the name of a directory.

DIR Options

You may also use DIR in a more advanced fashion by the use of several options tagged onto the end of the command. These are summarised in Table 16.1.

<i>Switch</i>	<i>OPT</i>	<i>Effect</i>
ALL	A	Lists all objects from this level and sub-levels.
FILES	n/a	Lists files only.
DIRS	D	Lists directories only.
INTER	I	Interactive mode. (See text).

Table 16.1. The DIR Options.

The options were extended from single letters to words to make their meaning clearer and this is the preferred method. The D (DIRS) option was introduced is used to list directories only. The FILES option works like DIRS but only lists the files found in the current directory, any directories found are ignored. The A or ALL option will list all files in the current directory along with the files and sub-directories from that point on. For example, if you use the A or ALL option on the Workbench disk then every file it contains will be listed.

Note the word OPT is not required when the full version of the option is given. Thus the following both list all the directories on a disk:

```
1>DIR SYS: OPT A
```

```
1>DIR SYS: ALL
```

Try this for yourself (making the Shell window as big as you can before doing so) and notice how the files inside directories are indented within the directories. This makes it much easier to distinguish files and directories from one another.

The A (ALL) option is all encompassing and is generally not that useful for anything other than familiarising yourself with the general contents and hierarchy of the files on a particular disk. Patterns can be used with this facility but they can have some queer effects, see *Mastering AmigaDOS 3 – Reference* for more information.

Interactive DIR

The I (INTER) option is much more useful as it allows you to selectively investigate directories and the files they contain. The usefulness of this probably won't be that apparent to you at this point in time but it will come into its own as you increase your understanding of AmigaDOS. The best way to see how the I (INTER) option works is to try it with the Workbench disk.

```
1>DIR DF0: INTER
```

```
Trashcan (dir) ?
```


AmigaDOS is now waiting for some input from you, in this case it is effectively saying, "The first file I have found is a directory called Trashcan. What should I now do?" How you respond will determine what happens next. Table 16.2 lists the various keypress responses you can make and their effect.

Let's examine each one in turn.

Pressing RETURN simply moves the DIR command onto the next directory or file skipping over the current one displayed.

Pressing 'E' when the ? prompt is displayed after a directory distinguished by the '(dirs)' after its name, moves you down into the directory allowing you to examine its contents.

<i>Command</i>	<i>Action</i>
Return	Move to next item.
E	Enter the displayed directory.
B	Leave the current directory and move back up one level.
Del	Delete the current file.
C	Execute any AmigaDOS Command.
T	Type a file (see TYPE).
?	List the available commands.

Table 16.2. Interactive Options.

Pressing 'E' effectively makes DIR do a CD <directory> although you can't use a pattern of course.

Pressing 'B' works in the opposite direction to 'E' and moves you back up the directory tree assuming you have gone down into it. It's the equivalent of CD /.

Typing DEL (*not* the DEL key) deletes the file. Don't try this yet until you have learned just how dangerous this can be!

Pressing 'C' allows you to issue an AmigaDOS command within the DIR command. Once the command has been completed you will be returned back into DIR at the point you left off. When you type 'C' AmigaDOS will respond with the following prompt:

Command ?

Enter your command here. A safe one to try is DIR itself, for example:

DIR DIRS

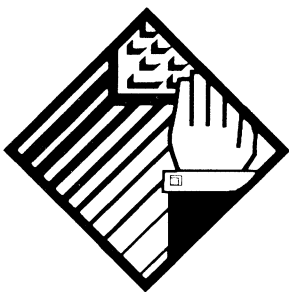
The directories in the current directory (if any) will be listed and return you back to interactive mode. Never ask for another Interactive DIR like this – the output gets very confusing. It's like someone setting fire to a newspaper while you are reading it.

Alternatively you can type 'C' and follow it with the command held within double quotes on the same line, eg:

C "DIR DIRS"

The overall effect is the same.

Pressing 'T' will type the file contents onto the screen. This is only useful if the file is known to contain text. It effectively gives the TYPE command to AmigaDOS which is discussed later and more fully in *Mastering AmigaDOS 3 – Reference*. '?' gives you on-line help at any time.



17: An Introduction to LIST

Overview

- > LIST compared to DIR
- > Using LIST
- > Limiting Output
- > Date and Time windows
- > Pattern matching
- > Sub-strings

LIST is a very large, complex and important command. It has been pivotal in the development of AmigaDOS as far back as version 1.3 when it was first used to automatically create script files. We'll come back to scripts in a later chapter, for now we're going to concern ourselves with the mechanics of the command.

DIR and LIST

The command shares a lot in common with DIR in that it is designed to display the contents of any directory. "Why," you may ask, "would I need two commands to do the same job?" The answer is there is more to LIST than meets the eye.

Here are the main differences:

- DIR collects the requested object names and displays a sorted list of them. LIST on the other hand displays all the requested objects as it finds them.
- DIR displays just the names of any objects in two columns; LIST displays the object's name and a lot of extra information in a single column.
- LIST can display objects by date.
- By default, the command shows the directory information, date and time.
- LIST does not have an interactive mode.

Using LIST

The synopsis for this command is huge and we won't bother with it here. For now, let's just worry about what the command does. Figure 17.1 shows a typical instance – cut down slightly for the sake of illustration.

```
1>LIST sys:
Directory "sys:" on Tuesday 13-Jul-93
C                               Dir ----rwed 07-May-93 18:09:52
Prefs                           Dir ----rwed 07-May-93 18:07:22
Prefs.info                      724 ----rw-d 07-May-93 18:07:22
Devs                             Dir ----rwed 07-May-93 18:07:34
Devs.info                      632 ----rw-d 07-May-93 18:07:34
Utilities                       Dir ----rwed 01-Jul-93 13:38:48
Utilities.info                 632 ----rw-d 07-May-93 18:07:38
System                         Dir ----rwed Thursday 12:59:10
System.info                   632 ----rw-d 07-May-93 18:07:41
This "filenote" is attached to System.info
Wbstartup                      Dir ----rwed 07-May-93 18:07:41
Wbstartup.info                 632 ----rw-d 07-May-93 18:07:41
Expansion                      Dir ----rwed 07-May-93 18:07:48
Expansion.info                 632 ----rw-d 07-May-93 18:07:48
S                               Dir -----d Today      19:25:23
Rexxc                          Dir ----rwed 07-May-93 18:08:07
Locale                         Dir ----rwed 07-May-93 18:08:21
Libs                           Dir ----rwed 07-May-93 18:08:31
L                               Dir ----rwed 07-May-93 18:08:32
Fonts                          Dir ----rwed 07-May-93 18:09:38
6 files - 13 directories - 44 blocks used
```

Figure 17.1: A typical example of LIST's output!

There's a lot of information there (told you so) let's take piece by piece.

The Header

Directory "sys:" on Tuesday 13-Jul-93

This information is displayed at the start of every directory listing provided the directory has been explicitly asked for. If the command line had just read:

1>LIST

this information would not have been printed. It also appears at the start of each directory when the ALL switch is specified. Apart from the directory information, the remainder of this line should be pretty obvious. The time and date shown are when the LIST command was given: if your machine doesn't have a real-time clock, this will probably be wrong!

The Body

Name	Size	Flags	Date/Day	Time
		hsparwed	dd-mm-yy	hh:mm:ss
C	Dir	----rwed	07-May-93	18:09:52
Prefs	Dir	----rwed	07-May-93	18:07:22
Prefs.info	724	----rw-d	07-May-93	18:07:22
Devs	Dir	----rwed	07-May-93	18:07:34
...				
Libs	Dir	----rwed	07-May-93	18:08:31
L	Dir	----rwed	07-May-93	18:08:32
Fonts	Dir	----rwed	07-May-93	18:09:38

This is the most interesting bit (never); so this illustration puts all the bits into perspective. The "Name" field is where the object's name will appear; "Size" is shown in bytes or "Dir" for sub-directories; the "Flags" are shown here for completeness – they are discussed under Advanced Object Management. Date and Time show when the file was created or last modified or, if a directory, when something inside it was created, modified or deleted. Datestamps are described later.

The Tail

6 files - 13 directories - 44 blocks used

The "Tail" is always shown (unless the NOHEAD option is used). It appears at the end of the listing and summarises the number of files, directories and blocks used. Blocks are mapped sectors: a file takes two or more; a sub-directory two. This is in fact a bug, as directories actually only use one block.

Limiting Output

We can limit what LIST displays in a number of ways for example by including patterns. However, the command provides two useful options, SINCE and UPTO, not provided by DIR which allow objects to be selected by *day* or *date*.

SINCE is a keyword which takes a date/time to list objects from. Objects created or modified on or after the given date are shown:

```
1>LIST sys: SINCE Thursday
Directory "sys:" on Tuesday 13-Jul-93
System          Dir ----rwed Thursday 12:59:10
t               Dir ----rwed Today   19:25:23
shots           Dir ----rwed Yesterday 10:33:25
S               Dir -----d Today   19:25:23
4 directories - 8 blocks used
```

UPTO on the other hand has the opposite affect – in this example the command's output has been edited to save space:

```
1>LIST sys: UPTO 7-May-93
Directory "sys:" on Tuesday 13-Jul-93
Classes          Dir ----rwed 07-May-93 18:09:40
C                Dir ----rwed 07-May-93 18:09:52
.backdrop        14 ----rw-d 07-May-93 18:09:55
...
L                Dir ----rwed 07-May-93 18:08:32
Fonts            Dir ----rwed 07-May-93 18:09:38
27 files - 19 directories - 618 blocks used
```

You may also use UPTO and SINCE with times – right down to the second would you believe – although this only works with objects created or modified on that particular day. Here are some real examples taken straight from an A3000. The time these objects (directories) were created was the time the machine was last booted. The default output looks like this:

```
1>LIST
ENV              Dir ----rwed Today      10:32:10
Clipboards       Dir ----rwed Today      10:31:59
T                Dir ----rwed Today      10:32:09
3 directories - 6 blocks used
```

Now we'll add a filter to display objects created before 10:32am:

```
1>LIST UPTO 10:32
Clipboards       Dir ----rwed Today      10:31:59
1 directory - 2 blocks used
```

and the same again, only this time list objects created at or after 10:32am.

```
1>LIST SINCE 10:32
ENV              Dir ----rwed Today      10:32:10
T                Dir ----rwed Today      10:32:09
2 directories - 4 blocks used
```

Date Windowing

This topic might sound rather grand, but it only means defining a period of time – a window – within which things can be displayed. We can use combination of SINCE and UPTO to accomplish this and, as they're *keywords*, it doesn't matter a monkey's wossisname which way round you put 'em.

```
1>LIST SINCE 10:32 UPTO 10:32:10
T                Dir ----rwed Today      10:32:09
1 directory - 2 blocks used
```

It is important to note in this example that you can include the seconds in one argument, but totally forget about them in the other. We've created a ten second window here purely for the sake of example. You determine how large or small the window is.

Here is the same sort of thing applied to days:

```
1>LIST sys: SINCE Thursday UPTO Yesterday
Directory "sys:" on Tuesday 13-Jul-93
System          Dir ----rwd Thursday 12:59:10
shots          Dir ----rwd Yesterday 10:33:25
2 directories - 4 blocks used
```

Patterns In LIST

LIST has an impressive set of features – and a very useful one is also one of it's least used: patterns. You may have noticed, when you used the ALL switch with DIR you didn't get quite what you expected. Here's an example listing the Ram Disk:

```
1>LIST E#? ALL
ENV          Dir ----rwd Today      14:56:47
1 directory - 2 blocks used
```

```
Directory "ENV" on Wednesday 14-Jul-93
Kickstart    6 ----rwd Today      14:55:38
Workbench     5 ----rwd Today      14:55:38
MultiView    Dir ----rwd Today      14:55:37
Sys          Dir ----rwd Today      14:55:37
2 files - 2 directories - 4 blocks used
```

```
Directory "ENV/MultiView" on Wednesday 14-Jul-93
Workbench Screen 28 ----rwd Today      14:55:37
MultiView        28 ----rwd Today      14:55:37
2 files - 4 blocks used
```

```
Directory "ENV/Sys" on Wednesday 14-Jul-93
WBPattern.prefs 250 ----rw-d Today      14:55:37
serial.prefs     68 ----rw-d Today      14:55:37
icontrol.prefs   62 ----rw-d Today      14:55:37
sound.prefs      318 ----rw-d Today      14:55:37
printerps.prefs  158 ----rw-d Today      14:55:37
Pointer.prefs    318 ----rw-d Today      14:55:37
palette.prefs    434 ----rw-d Today      14:55:37
wbconfig.prefs   54 ----rw-d Today      14:55:37
printergfx.prefs 72 ----rw-d Today      14:55:37
printer.prefs    162 ----rw-d Today      14:55:37
locale.prefs     894 ----rw-d Today      14:55:37
input.prefs      78 ----rw-d Today      14:55:37
font.prefs       518 ----rw-d Today      14:55:37
13 files - 26 blocks used
```

All very nice, but that does generate a lot of output, besides which we only asked for objects starting with "E" didn't we?

Is it a bug? Actually no. LIST searches the current directory for any directories starting with "E" and lists the contents of those (and any subdirectories they may have). DIR, DELETE, COPY and so on all work in exactly the same way; but LIST gives us more power! Look at this:


```
1>LIST E#? pat=pr#? ALL
Directory "ENV" on Wednesday 14-Jul-93

Directory "ENV/MultiView" on Wednesday 14-Jul-93

Directory "ENV/Sys" on Wednesday 14-Jul-93
printerps.prefs      158 ----rw-d Today      14:55:37
printergfx.prefs     72 ----rw-d Today      14:55:37
printer.prefs        162 ----rw-d Today      14:55:37
3 files - 6 blocks used

TOTAL: 3 files - 6 blocks used
```

The PAT keyword allows us to specify an optional pattern. LIST searches the same directories and sub-directories, but applies the pattern to every object it finds! In this case we asked for any files starting with "pr" to be listed.

There's another option which allows us to show object names containing something: SUB. Here's an example of it in use.

```
1>LIST E#? SUB=pr ALL
Directory "ENV" on Wednesday 14-Jul-93

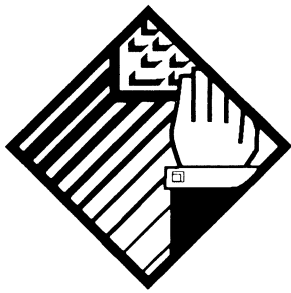
Directory "ENV/MultiView" on Wednesday 14-Jul-93

Directory "ENV/Sys" on Wednesday 14-Jul-93
WBPattern.prefs      250 ----rw-d Today      14:55:37
serial.prefs         68 ----rw-d Today      14:55:37
icontrol.prefs       62 ----rw-d Today      14:55:37
sound.prefs          318 ----rw-d Today      14:55:37
printerps.prefs      158 ----rw-d Today      14:55:37
Pointer.prefs        318 ----rw-d Today      14:55:37
palette.prefs        434 ----rw-d Today      14:55:37
wbconfig.prefs       54 ----rw-d Today      14:55:37
printergfx.prefs     72 ----rw-d Today      14:55:37
printer.prefs        162 ----rw-d Today      14:55:37
locale.prefs         894 ----rw-d Today      14:55:37
input.prefs          78 ----rw-d Today      14:55:37
font.prefs           518 ----rw-d Today      14:55:37
15 files - 30 blocks used

TOTAL: 15 files - 30 blocks used
```

All these files have one thing in common – they all contain the string "pr". It isn't hard to spot – it crops up in "Prefs" and all these end in just that.

It's also important to note that a pattern is not necessary with SUB: This example is the same as "PAT=#?pr#?" of course, it's a lot easier to type "SUB=PR". Just to confuse matters even further – if that were possible – you can even use a pattern in the sub-string. *Arrrrghh!*



18: Basic System Control

Overview

- > Setting the Time and Date
- > Setting the clock
- > Adding disk buffers
- > Getting a version number

The time and date held and set by DATE are maintained when the Amiga is powered and running. Like the main memory of the Amiga the bytes that hold the information are lost when the power is taken away. However, a 'battery-backed' clock can be fitted to the Amiga and takes the form of a small amount of memory which is constantly powered by a battery. This means its contents are maintained while the power is switched off. The nicad battery re-charges when the machine is switched on, but it will run down after a short while – typically around two weeks.

The battery-backed clock can be altered by use of the SETCLOCK command which has the following syntax:

SETCLOCK OPT LOAD|SAVE|RESET

The value of the battery-backed clock can be transferred into the system clock using the command:

SETCLOCK OPT LOAD

The system's date and time, which can be altered with DATE, can be moved to the battery-backed clock using:

SETCLOCK OPT SAVE

If you are not sure whether or not you have a battery-backed clock fitted or not try entering one of the commands. If the upgrade is not fitted a message similar to the one following will be displayed:

Internal clock not functioning

RESET will reset the clock after a serious software failure.

Adding Buffers

Normally AmigaDOS does not hold much information about the contents of your disks in its memory. This can make the use of disks a bit annoying. For instance, consider how long it takes to directory the contents of the Workbench disk. Several seconds even if you have first made the DIR command resident with:

RESIDENT C:DIR

The DIR command goes and reads the directory information from the disk every time you request it.

The area which AmigaDOS reserves to hold disk information is called a 'buffer', and the size of this buffer area can be increased using the ADDBUFFERS command. The syntax of the command is:

ADDBUFFERS DRIVE=[<Drive>] [[Buffers=]<Size>]

Each drive has its own buffer associated with it so it is therefore possible to adjust disk drive buffers individually. The <Size> of the buffer is given in blocks of 512 bytes (0.5k) and in general a setting of about 25 achieves the best results.

To see the difference the command can make to the way in which you work, and more particularly how it can be made to speed things up, try this example.

1. Insert the Workbench disk into drive DF0: and make DIR resident thus:

1>RESIDENT C:DIR

2. Now call the command:

1>DIR

The process of reading and displaying the directories and files takes several seconds or so.

3. Now increase the buffers allocated to DF0: with ADDBUFFERS and check the directory again as follows:

1>ADDBUFFERS DF0: 25

1>DIR

4. The process still takes a few seconds, but now do it once again:

1>DIR

The process is almost instant. This is because the display information has been read into the buffer space recently allocated to DF0:. AmigaDOS remembers that it is there and simply gets the information from it without having to access the disk itself.

The memory allocated as disk drive buffer space by ADDBUFFERS is taken directly from the Amiga system memory, so unless you have plenty of Ram fitted it is best to keep buffer allocation down to a minimum. Another point to bear in mind when using the command is that it does not accept logical assignments in place of disk names. This command only affects the floppy drives DF0:...DF3:

Device Information

The INFO command can be used to tell you the state of play with regards to your various drives and disks. Typing INFO displays a few lines on screen which look a bit like what follows for a single drive system with the Workbench disk in the internal drive, and a few files in the Ram Disk:

```
1>INFO
```

```
Mounted disks:
```

Unit	Size	Used	Free	Full	Errs	Status	Name
RAM:	78K	157	0	100%	0	Read/Write	Ram DISK
DF0:	879K	1599	159	91%	0	Read/Write	Workbench 3.0

```
Volumes available:
```

```
Ram DISK [Mounted]
```

```
Workbench 3.0 [Mounted]
```

The headings may need a few words of explanation:

Unit	This is the device name.
Size	This is the total available capacity of the disk. On a normal floppy formatted disk it is 880k. On a Ram Disk it is volatile and will change to suit the user's requirements.
Used	This displays the number of 'blocks' that have been used to store the information. A block is 0.5k in length.
Free	This is the number of free blocks remaining on the disk.
Full	This is the amount of space used on the disk expressed as a percentage.
Errs	This is the amount of space used on the disk expressed as a percentage.
Status	This lists the Read/Write status of the disk.
Name	This lists the name assigned to the disk.

The second section lists the names of the disks in the drives so that you can ascertain them without the physical need to remove them from the disk drive.

If you require information about one particular device then you can specify the device name as a parameter. For example, to list info on the Ram Disk only you could use:

INFO RAM:

Which Version?

With the continuous development of AmigaDOS and Kickstart and its components such as the Workbench you may occasionally wish to ascertain, from within a program, which version of the software you are running on, to enable you to then select a section of program to operate most efficiently under a particular software version. The VERSION command can be used for this purpose and it has the following syntax:

VERSION

Used on its own the command will return the current version. This will typically look like this:

1>VERSION

Kickstart 39.106. Workbench 39.29

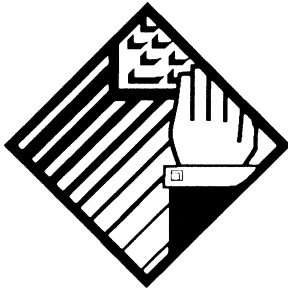
This command may also be used to discover the version numbers of almost everything from scripts to AmigaDOS commands and libraries provided they include a specific version string. For example:

1>VERSION S:Startup-sequence

1>VERSION C:VERSION

1>VERSION DEVS:printer.device

- Internal commands are shown by the Shell number unless they have been replaced by other code.



19:

Advanced Directories

Overview

- > The search path
- > New paths
- > Overriding the path
- > Locating commands
- > Logical directories
- > Basic assignments
- > Assigning volumes

Every time you type in a command such as DIR or FORMAT AmigaDOS knows exactly where to find it. But these commands are stored all over the place – and AmigaDOS has to be told where to look.

The PATH command looks after this side of things and is used to define the default *search* path that AmigaDOS will follow every time a command is entered. We touched upon this briefly when we discussed the RESIDENT command.

PATH is a fairly simple command to use, although it can have some powerful and wide reaching effects. By default it shows a listing of the current search path, like this:

```
1>PATH
Current Directory
RAM Disk
Workbench:Utilities
Workbench:System
Workbench:S
Workbench:Prefs
C:
```

The exact makeup of the path is determined by the machine it's running on – so the listing on your machine may well look different to the one shown here.

OK, so what's happening? Every time you enter a command AmigaDOS looks for it in every directory listed under the path. It looks first in the current directory – if the name is not found it looks in the next directory on the list and so on until either:

1. The filename (command) is found. AmigaDOS then tries to execute the command. However, if the filename matching the command is *not* a command, AmigaDOS complains.
2. The path has been searched entirely. If the command is not found the AmigaDOS complains that the command could not be found and generates an ERROR condition (RC=10). We'll discuss such things in a later chapter.

Point 1 has important consequences if you are not careful with the path. Let's see what happens. First, we'll make sure we're somewhere that can't do any harm by switching to the Ram Disk:

```
1>RAM:
```

Now we'll create a dummy file using re-direction, a technique which will be discussed later.

```
1>ECHO >DIR "Rubbish"
```

and try to use the command:

```
1>DIR
```

```
DIR: File is not executable
```

Oh dear! What happens if we swap back to somewhere else and try to execute DIR?

```
1>SYS:
```

```
1>DIR
```

```
DIR: File is not executable
```

Yikes! Now what? This can be very worrying if it comes out of the blue – indeed, your first thought may be that something had overwritten the DIR command. AmigaDOS has tried to execute the first file call 'DIR' that it found in the current path – and that is our dummy file in the Ram Disk.

Path Override

In a case such as this, you can make sure you get the command you want by specifying the full path as part of the filename. AmigaDOS will recognise that and go looking where you tell it to. Interestingly, this also makes things faster in script files if you can be bothered to enter all the extra stuff.

For example, we know that all AmigaDOS commands are stored in the C directory of the current startup (boot) disk. Assuming you have not deleted the dummy file, you can call DIR like this:

```
1>SYS:C/DIR
```

This directory is also *assigned* a name by the system – "C:" so you can access commands like this too:

```
1>C:DIR
```

Not all AmigaDOS executable commands are stored in the C directory though. FORMAT for instance, is located in the "System" directory so it can be called from the boot disk like this:

1>SYS:System/FORMAT

SYS: is also an assignment which allows AmigaDOS to locate the current boot disk, but you can access any command from any disk just by specifying the drive number. Let's say you wanted to access FORMAT from a Workbench disk (but not the boot disk) in the internal drive you could enter:

1>DF0:System/FORMAT

Locating Commands

You can find the absolute location of any command in the current path using WHICH. This simple little command can help to diagnose those problems outlined above. WHICH is used like this:

1>WHICH FORMAT

Workbench:System/FORMAT

There's nothing to it really. WHICH's also has some switched options that can help:

ALL Forces WHICH to display all the occurrences of the file in the search path. For instance:

1>WHICH DIR

RES DIR

Workbench:C/DIR

Workbench:C/DIR

You will notice here that DIR is displayed twice under the C directory – this is not a mistake, it means that the C directory is being scanned twice. Once as an assignment (C:) and once as an absolute path (Workbench:C). The mention of RES means that DIR is also present in the resident list – ie, it was stored in memory using RESIDENT.

RES WHICH displays the file if it is in the resident list – the rest of the path is not searched.

NORES WHICH displays any named files that are *not* located in the resident list.

Assigned Directories

You will probably now already have realised the tedium that can come with having to type and re-type long directory path roots. While the use of the Shell's editing facilities alleviate this to some degree, long pathnames can become a pain in the proverbial. The command ASSIGN can be used to simplify access to various locations.

The ALIAS command, described earlier, could be used to assign a short name to a long path name, however ALIAS forms part of the Shell and can only be used from within it. It is no good unless you are running the application or program from that Shell.

ASSIGN solves this problem in one fell swoop and has the added advantage in that it may be included in your startup and script files as required. Moreover, some applications will expect to find particular directory paths assigned for them.

In fact, AmigaDOS makes extensive use of ASSIGN to locate specific directories no matter where they happen to be. This technique, known as a logical device assigning, finds use for the Fonts and Locale directories: plus a number of others.

Some directories are allocated automatically when you switch on – others have to be configured specially. One you have already used on a number of occasions is SYS: – that's a logical directory assignment which, by default, "points to" the root directory of the boot disk.

The basic synopsis of the command is as follows:

ASSIGN [<name>]: [<path>]

Figure 19.1 shows what happens if ASSIGN is used without any arguments. It's fair to say all this looks a little oppressive at first – but don't worry – it's all very straightforward; or at least the tricky bits aren't worth worrying over at this stage.

All you need to know is that everything you see here works like a disk device. With a few exceptions, you can write to them, get directories of them and even create more directories there.

The Rules



Assignments work like devices: *Every name on the left-hand side of the list must be followed by a full colon (:) in a command that uses it.*

This may seem quirky, but it becomes second-nature very quickly. For example, you can get a list of all the AmigaDOS commands simply by typing:

1>DIR C:

```

1>ASSIGN
Volumes:
Ram Disk [Mounted]
DH2 [Mounted]
Apps [Mounted]
37MB [Mounted]
WB3.0 [Mounted]
Empty

Directories:
HELP                <LOCALE:Help>
LOCALE              WB3.0:Locale
KEYMAPS             WB3.0:Devs/Keymaps
PRINTERS            WB3.0:Devs/Printers
REXX                WB3.0:S
CLIPS               Ram Disk:Clipboards
T                  Ram Disk:T
ENV                 Ram Disk:ENV
ENVARC              WB3.0:Prefs/Env-Archive
SYS                 WB3.0:
C                   WB3.0:C
S                   WB3.0:S
LIBS                WB3.0:Libs
                   + WB3.0:Classes
DEVS                WB3.0:Devs
FONTS               WB3.0:Fonts
L                   WB3.0:L

Devices:
PIPE RAM CON RAW SER
PAR PRT WB_2.x DFO
DF1 DH0 DH1 DH2

```

Figure 19.1 ASSIGN and the directories – this example is from an Amiga A3000 running AmigaDOS 3.

Similarly, you can copy any ARexx scripts with a command such as this:

```
1>COPY REXX:#?.REXX to RAM:Rexx
```

You can even select a directory directly by entering the name of its assignment, viz:

```
1>DEVS:
```

Like real devices, assignments can have directories you can get at them in the same way:

```
1>DEVS:Keymaps
```

or:

```
1>CD DEVS:Ke#?
```

Name	Default path	Status:†	Contents
SYS:	?	Auto	Root directory of boot disk.
C:	SYS:C	Auto	AmigaDOS Commands.
L:	SYS:L	Auto	Handler libraries.
S:	SYS:S	Auto	Scripts.
LIBS:	SYS:Libs	Auto	External function libraries.
	+ SYS:Classes	Added	New class libraries
DEVS:	SYS:Devs	Auto	Device drivers.
FONTS:	SYS:Fonts	Auto	Typefaces.
REXX:	S:	Manual	ARexx scripts
KEYMAPS:	DEVS:Keymaps	Manual	Old keymap files.
PRINTERS:	DEVS:Printers	Manual	Printer drivers.
ENV:	RAM:Env	Manual	Global environmental store.
ENVARC:	SYS:Prefs/Env-Archive	Manual	Current user preferences
T:	RAM:T	Manual	Temporary files
CLIPS:	RAM:Clipboards	Manual	Clip files.
LOCALE:	Locale:	Late	Country localisation info
HELP:	Locale:Help	Late	Country help files.
† Auto – assignment made during bootup.			
Manual – assignment made by startup-sequence.			
Added – manual assignment added to an existing one.			
Late – manual assignment called when required.			

Table 19.1 The directory assignments in AmigaDOS 2/3.

or even:

1>KEYMAPS:



It is important to note that AmigaDOS commands are not aware of additional manual assignments (such as Classes) and you must access these directories separately. Therefore while this would work:

1>DIR DEVS:

it will *not* display the contents of the Classes directory...

Long Directory Names

Ask yourself – which of the following command lines would you rather type?

1>DIR SYS:Prefs/Env-Archive

or

1>DIR ENVARC:

Consult Table 19.1 containing the default assignments and you will notice that ENVARC: is an assignment of "SYS:Prefs/Env-Archive". So, in other words, these command lines do exactly the same thing. A logical directory is a simple way to assign short names to long paths. The assignment, ENVARC: is constructed thus:

ASSIGN ENVARC: SYS:Prefs/Env-Archive

You can use this method to assign any directory on any disk device to any convenient name. For instance, if you program in AMOS and keep all your work in a directory with the path:

DH1:Programming/AMOS/Source

You can assign the directory a simple name like this:

ASSIGN AP: DH1:Programming/AMOS/Source

Once done, the directory may be accessed from any file requester just by entering "AP:".

Volume Names

ASSIGN can be used to get around the everyday hazards of software loaded onto hard disks because it can make a directory appear as if it is a disk! This method can be used at any time – but correctly written software should handle all this for you.

Such problems can be easily identified when a piece of newly installed software suddenly presents a requester like the one shown in Figure 19.2.

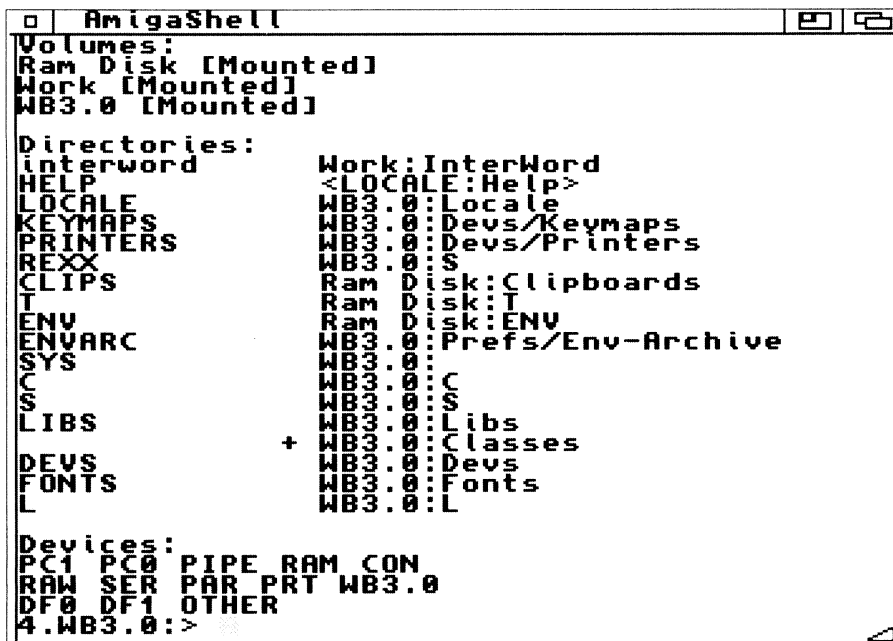


Figure 19.2. A volume requester calls for ASSIGN.

AmigaDOS does this whenever an unknown volume, device or assignment is accessed. In fact, when the list of names (like the one in Figure 19.3) is searched, AmigaDOS does not distinguish between types: it simply looks for a match. If a one is found, AmigaDOS attempts to perform the requested action and passes any errors back; if a match cannot be found, this requester appears.

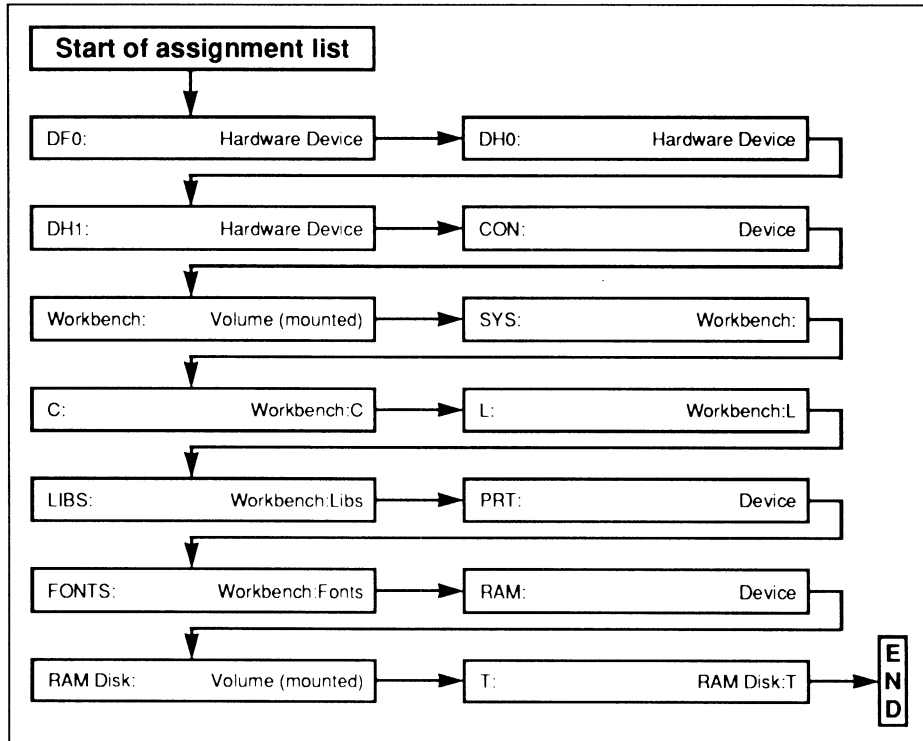


Figure 19.3 The list of assignments is never sorted in memory!

New assignments are added to the end of the list and “point” to specific directories or devices. For instance, let’s say we’ve created a directory on the hard disk to house the new software and copied the relevant files across. We can create the new assignment like this:

```
1>ASSIGN "MEGA ED:" DH1:Editor
```

Alternatively, we can create an assignment automatically to the current directory like this:

```
1>CD DH1:Editor
```

```
1>ASSIGN "MEGA ED:" ""
```

The new assignment can be seen added to the end of the current list in Figure 19.4. Note that in both these examples, the name of the assigned directory (a volume name in this case) must be surrounded in quotes because of the space in the name – watch out for this.

It is worth noting that, you cannot assign a volume name when the same volume is already mounted in some drive. If such an attempt is made AmigaDOS complains:

Cannot cancel nnnn

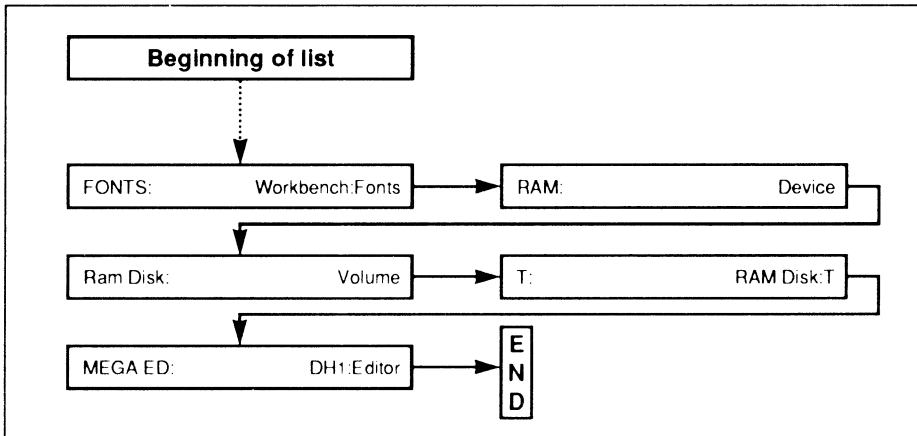


Figure 19.4. A new assignment is added to the end of the existing list.

ASSIGN With Drives

ASSIGN can also be used to give special names to disk drives. For instance, if you are used to working on a PC and referring to the internal drives as A: and B: then the following would do the trick:

1>ASSIGN A: DF0: PATH

1>ASSIGN B: DF1: PATH



It is not advisable to assign "C:" to the hard disk because that name is reserved for the command directory.

In these examples you will notice the use of the special switch, PATH. This has the following affects:

1. The assignment is not called until it is accessed by some AmigaDOS command. This is important because a disk drive device is not valid until a disk is loaded in the drive.
2. The assignment is valid for any inserted disk. A normal assignment will lock (bind) to the first disk it sees.

You can use these drive assignments just as you would access any normal Amiga drives. For instance, the following are all valid:

```
1>FORMAT DRIVE A: NAME EMPTY
```

```
1>B:
```

```
1>DIR A:
```

```
1>MAKEDIR A:Directory
```

```
1>LIST B:
```

and so on.

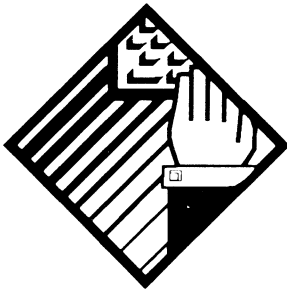
PC Drives

This technique may be extended to the CrossDOS utility (Workbench 2.1+) which labels the two PC devices PC0: and PC1: respectively. Using these assignments instead, you can access Amiga drives by their names and PC drives by theirs:

```
1>ASSIGN A: PC0: PATH
```

```
1>ASSIGN B: PC1: PATH
```

It is interesting to note that these drives may be formatted with the normal Amiga FORMAT command to 720K PC format.



20: Advanced Object Management

Overview

- > Protection
- > Changing object flags
- > Linking disk objects.
- > Date Stamping
- > Changing dates and times

As we have already discussed, an *object* is something stored on an AmigaDOS disk – it's a collective term for a file or sub-directory. In this chapter we'll discuss three commands which handle objects in a more esoteric fashion.

Protection Racket

Every single disk object, no matter what it is, has various bits of information stored with it. We have already seen that this information can be displayed with the aid of LIST (it appears between the object length and date).

The information supplied in this third column is the protection status and each individual character is called a *flag* – one of H, S, P, A, R, W, E and D.

Every object has eight flags associated with it and the conditions of these determine how AmigaDOS will treat it. By default when a object is created it has four flags set to the 'on' condition. When a flag is 'on' then the action it represents can be carried out. For example, the delete flag is on by default and so the object can be deleted.

When a flag is on then it is displayed in the LIST list by the use of a unique letter. The eight flags and their purposes are listed in Table 20.1.

<i>Flag</i>	<i>Name</i>	<i>Set</i>	<i>UnSet</i>
R	Readable	Can be read	Cannot be read
W	Writable	Can be written to	Cannot be written to
D	Deleteable	Can be deleted	Cannot be deleted normally
E	Executable	Is executable code	Is not executable code
S	Script	Is a script file	Is not a script file
P	Pure	Can be made resident	Cannot be made resident
A†	Archive	Is archived	Is not archived
H‡	Hidden	Is hidden	Can be seen

† Cleared when a file is modified. Not used otherwise.
‡ Not used by AmigaDOS.

Table 20.1. The object protection flags.

By default the 'rwed' flags are set and this means that the object can be read, written, executed and deleted as a program. The other flags are 'off' and therefore the object is not a script, isn't pure, isn't archived and isn't hidden.

A Practical Example

Disks can be protected by opening the plastic shutter in the corner of the disk. Information can be read from the disk, thereby allowing files to be run and accessed, but they cannot be written to or, more correctly, the disk itself cannot be written to.

A simple example would be that you wouldn't want a program object to be deleted but you would want to be able to save the information the program produces and to have this on the same disk. This is where the AmigaDOS command PROTECT comes in.

To see how the flags interact let's create a dummy object to work with. Use ED and save the object in the Ram Disk. The ED object can contain any information you wish, gibberish if you prefer, or you can copy across one of the startup files used earlier and rename it. Call the object TEST and ensure the Ram Disk is the current drive.

The syntax of the PROTECT command is:

```
PROTECT <Filename> +|-<[r|w|e|d|s|p|a|h]>
```

The switched options ADD and SUB can be used in preference to '+' and '-'. To see how these works, we'll make a dummy file in the Ram Disk:

```
1>ECHO >RAM:TEST "Lies, and damned lies"
```

D – The Delete Flag

To make the TEST object not deleteable it is necessary to remove or turn off the delete flag – 'D'. This is done as follows:

```
1>PROTECT RAM:TEST -D
```

or alternatively:

```
1>PROTECT RAM:TEST D SUB
```

Now LIST the Ram Disk and notice that the 'd' flag is no longer displayed instead you get:

```
----rwe-
```

The object can not now be deleted. Try it:

```
1>DELETE RAM:TEST
```

```
RAM:Test Not Deleted: object is protected from deletion
```

To re-enable deletion type:

```
1>PROTECT RAM:TEST +D
```

or

```
1>PROTECT RAM:TEST D ADD
```

The 'd' flag will now be displayed in any list and the object can be deleted.

W – The Write Flag

This determines whether a object can be written to or not. The flag is supported by directory objects, but has no effect:

```
1>PROTECT RAM:TEST -W
```

In other words you can read what it contains but you cannot add to it. Note however that some programs do not try to add information to the current object but instead re-save it. In this case the old object information is overwritten and the default flag status is restored. ED (discussed later) is a prime example. If you update your TEST object with a new line of text and save the changes, LIST will show that the 'w' has indeed been reset to on. Otherwise you re-enable a object to be writable with:

```
1>PROTECT TEST +W
```

E – The Execute Flag

This flag allows you to define whether or not a object may be executed as an AmigaDOS transient command program. The flag is supported by directory objects, but has no effect.

Turn DIR's 'e' flag off :

```
1>PROTECT C:DIR -E
```

and now try to run it...

```
1>DIR
```

Execute permission not set

Just to save embarrassment, it may be wise to put the 'E' flag back into effect like this:

```
1>PROTECT C:DIR E ADD
```

R – The Read Flag

This one is not used by AmigaDOS prior to release 2, except in FFS – even then not all programs support it. In theory at least, by clearing this flag applications programs and AmigaDOS will ignore it or print an error.

```
1>PROTECT S:SPAT -R
```

```
1>COPY S:SPAT TO RAM:
```

Error in reading – file is read protected

Destination file RAM:SPAT removed

```
1>TYPE S:SPAT
```

Hummm... In the last case, No output (and no error either) but don't forget to set read permission back on again just in case you want to edit this script!

```
1>PROTECT S:SPAT +R
```

A – The Archive Flag

The 'a' flag is for use by backup programs. It is not read or written by AmigaDOS, but is cleared when a file is written to or modified. The idea is to set the archive flag on every file when it is backed-up. When the time comes to do the next backup, only those files whose archive bit is *clear* are copied.

S – The Script Flag

Under the Shell the name 'script files' is given to collections of AmigaDOS commands. The 's' flag allows you to inform AmigaDOS that the object is a script file and can be executed.

Setting the flag forces AmigaDOS to launch EXECUTE with the command line you supplied – just like a real program! By setting the 's' flag you also allow AmigaDOS to ignore the setting of the 'e' flag. So even if the 'e' flag is off, if the 's' flag is set the file will be executed.

```
PROTECT TEST S ADD
```

P – The Pure Flag

The RESIDENT command can be used to make named objects stay in memory and be executed from memory as and when they are called. This avoids the need to have to continually load in

commonly used programs from disk thereby speeding up the whole process. However not all programs can be made resident. For instance:

```
1>RESIDENT C:BINDRIVERS
```

Cannot load C:BindDrivers: Object not of required type

If you list the object information for LOADWB and a object that can be made resident, such as DIR you will find that their flags are set as follows:

```
BindDrivers      ----rwed
DIR              --p-rwed
```

For a object to be able to be made resident it must have its Pure bit set and the 'p' flag allows this to be defined.

You should not normally change this flag on supplied programs unless you are fully aware of the consequences. However, you can force a program to become resident by using the command's PURE option. The conditions for the use of PURE are outlined under RESIDENT in *Mastering AmigaDOS 3 – Reference*. In any case it should be used with extreme caution.

H – The Hidden Flag

The 'h' flag is a red-herring! It doesn't work with any current release of AmigaDOS or Kickstart come to that.

Linking Disk Objects

The principal idea of links is to make objects available in any part of the hierarchy. They work like assignments with a few important differences:

1. A link can be made to either a file or a directory, whereas an assignment can only be made to a directory.
2. A link is a permanent part of the disk structure and does not have to be created every time the machine is booted.
3. In the current release, links cannot be made between volumes – assignments can access other volumes.

A Worked Example

The topic of linking is rather advanced and is better illustrated with some examples. Generally speaking it is intended for hard disks (where the hierarchy may span ten or more levels) but for the purposes of illustration, we'll work on the Ram Disk.

1. First of all we'll create some temporary directories:

```
1>RAM:
```

```
1>MAKEDIR ONE TWO
```

2. Now we'll copy the 'S' directory from the boot disk to the Ram Disk – this will give us some files to work with:

```
1>COPY S: TO S
```

3. When that task is complete we can set about making the links. First we'll make a simple file link:

```
1>MAKELINK SPAT TO S/SPAT
```

4. Taking a directory of the current disk (RAM:) should confirm you have a file marked 'SPAT <hl>'. Try that now. More interestingly, you will also note that when the Ram Disk is LISTed the file appears with a size of 288 bytes. However, the link itself is only two blocks long – no matter how big the original file is!

5. Now let's change directory and make another link:

```
1>One
```

```
1>MAKELINK SPAT-LINK TO /SPAT
```

Showing the directory once more, confirms the link is present. Now we can access SPAT from here too. You can prove this by entering this:

```
1>MORE SPAT-LINK
```

Link Directories

Directory links are constructed in more or less the same way as any other. Here's a simple example using the directories created above:

```
1>MAKELINK RAM:TWO/ONE TO RAM:ONE FORCE
```

This creates a link in the directory 'RAM:TWO' which links to the directory 'RAM:ONE'. The FORCE switch must be used. You can get a listing of RAM:ONE from RAM:TWO in either of two ways:

```
1>DIR RAM:ONE ; conventional
```

```
1>DIR ONE ; much shorter!
```

Is there a point to all this? Well in this example we're constructing very simple links between files and directories – but those objects could be anywhere on the disk; and there is no limit to the number of links you can have. You can link any object any number of times too – you can even link to a link.

Time and Dates

Whenever AmigaDOS creates or modifies an object, the operation's time and date is written to the object's header block. This technique is called date-stamping and can be used with some AmigaDOS commands to locate and sort objects.

As we have already shown in Chapter 17, LIST can provide *date windows* to select objects by date and/or time.

The AmigaDOS command SETDATE allows you to change the date stamp of any AmigaDOS object – it isn't clear why anyone would want to do this, but there you go. SETDATE has the following basic synopsis:

SETDATE <object> [date|time]

Objects are affected in the following manner:

Files

Are stamped with the current date and time when:

1. the file is created.
2. the file is modified.
3. the file is copied. (The destination gets stamped *unless* COPY's CLONE switch is used.)

Directories

Are stamped with the current date and time when:

1. the directory is created.
2. an object is created inside the directory. (It is not stamped when an object is modified inside the directory).
3. an object is removed from the directory.

Changing Dates

Dates can be used to view disk and file activity over a specified period of time with LIST, but SETDATE can be used to fix or modify the timestamps. Here's a worked example.

1. First, create a new directory in the Ram Disk:

```
1>MAKEDIR RAM:test
```

2. Now check the date stamp using LIST:

```
1>LIST RAM:te#?
Directory "RAM Disk" on Tuesday 13-Jul-93
test          Dir   ----rwed   Today   12:09:38
1 directory - 2 blocks used
```

3. We can change the date or time of this directory using standard notation (like LIST). For instance, we can change the day:

```
1>SETDATE RAM:Test 12-Jul-93
```

4. and check that:

```
1>LIST RAM:te#?
Directory "RAM Disk" on Tuesday 13-Jul-93
test          Dir    ----rwed    Monday    12:09:38
1 directory - 2 blocks used
```

5. Or, we can change the time thus:

```
1>SETDATE RAM:Test 20:00:00
```

6. and check that:

```
1>LIST RAM:te#?
Directory "RAM Disk" on Tuesday 13-Jul-93
test          Dir    ----rwed    Monday    20:00:00
1 directory - 2 blocks used
```

7. Remember though, if something is moved to this directory, it will be re-stamped with the current time and date.

```
1>COPY S:SPAT RAM:Test
```

```
1>LIST RAM:te#?
Directory "RAM Disk" on Tuesday 13-Jul-93
test          Dir    ----rwed    Today     12:19:21
1 directory - 2 blocks used
```

As you can see, the datestamp has been changed to indicate the contents of this directory have been modified.

Since SETDATE is an *object-oriented* command it can be used to change the time and date stamping of files too. You may like to try this with the contents of the "TEST" directory contained in this example.



SETDATE has an ALL switch. This can be used to send the same information to all files in the current directory and any subdirectories below it.

File Date

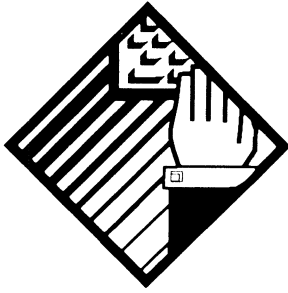
The DATE command has an additional option, TO, which allows it to write the current time and date information to a file. The syntax is:

```
DATE TO <filename>
```

For instance:

```
DATE TO Now
```

Will write the current date and time to a file called 'Now'. As always if the file does not exist it is created. The information is written as a text file and can then be displayed with TYPE or by loading it into ED, for instance.



21: Advanced File Management

Overview

- > Viewing text files.
- > Examining text files.
- > Examining binary files.
- > Searching for files
- > Searching in files
- > Filenotes

More is a program that you may have already used several times without knowing, especially if you have been reading ReadMe files supplied with some third party applications. MORE is an ASCII text file display program – it allows text files (such as those created in ED) to be displayed on-screen in a window.

From the Shell MORE is launched using the following syntax:

```
MORE <filename>
```

where <filename> is the name of the file you wish to view.

If you do not supply a filename then MORE prompts you for one. A file dialogue box is displayed and you can use this to locate the file you wish to view. An alternative and preferable method is to use the RUN command thus:

```
RUN MORE
```

at which point MORE opens its own window and a full file requester interface to enable you to select the required file.

When the file is located MORE displays it from the beginning in it's own window. (Note: is you can't lay your hands on a suitable file you can always use the Startup-Sequence.) At the base of this window MORE displays a message along these lines:

```
- - - More (86%) - - -
```

This informs you that there is more text to see and that the program has so far displayed 86% of the text. Note that the number will almost certainly vary between systems depending on the alterations that you may have

made to your startup file. To display the rest of the text press the <SPACE> key. More text will be displayed. When there is no more text to display the prompt at the base of the screen will change to:

- - - End of File - - -

More Commands

More has a number of key commands that it recognises which become useful when you are using it in conjunction with long text files.

<i>Command</i>	<i>Effect</i>
<SPACE>	Display next page of text.
<BACKSPACE>	Display previous page of text.
<RETURN>	Displays next line of text.
<	Displays first page of text.
>	Display last page of text.
%n	Displays from a point approx n% into the file
<CTRL-L>	Refresh display
/<string>	Search for <string> - case sensitive
.<string>	Search for <string> - non-case sensitive
n	Find next occurrence of the string
h	Display on-line help.
q	Quit
<CTRL-C>	Quit
E	Edit using editor set in ENV:EDITOR.

Table 21.1. The MORE commands.

Most of these commands should be selfexplanatory. The two search options can be useful – if your search string is located then the line in which it occurs is moved to the top of the MORE window. You can then edit the file – should you so wish – by entering your active Editor by pressing 'E'. Note that this is only possible if you have defined it in the environmental variable EDITOR as described in Chapter 27.

More Piping

The More program can also be used in conjunction with PIPE to display incoming or re-routed information. Try this simple example to see how it works.

In a Shell window create the More pipeline by typing:

RUN MORE PIPE:A

More invokes a large window – you may wish to re-size this to a more suitable size and to reveal your Shell if it has been hidden. From the Shell type:

COPY s:startup-sequence TO PIPE:A

The file will be copied into the window just as if it had come from a file. The main difference with a pipe is that it can generate output until the cows come home, without ever using more than 2K of buffer space.

Examining Text Files

Every file on your disk is a binary file. This might seem strange, but a text file is just a special case of a binary file. In other words, it only contains bytes within the range 32-127. The American ASCII standard defines this range as 7-bit values containing only printable characters. The remaining 32 (non-printable) values have special meanings as control characters.

The upper 127 characters are undefined in ASCII and left to specific computer architecture (and fonts). On the Amiga, for instance, the character 0x9B is a control sequence introducer (CSI) and is interpreted by the Shell window driver as being the start of a special string of characters to change settings such as colour or font style. Generally speaking therefore, it is unwise to attempt to display any file which does not only contain pure ASCII characters – MORE warns you if it encounters anything out of the ordinary.

In AmigaDOS Shell windows it is more usual to use TYPE, which prints an unfiltered file to the screen. If that file contains odd binary characters or special “CSI” sequences they are interpreted by the console device. For this reason, this command should be used with caution if the file is unknown. Although the CSI sequences are not fatal they can cause confusing results.

Typically, you use the command thus:

1>TYPE S:Startup-Sequence

and you can add line numbers to the output like this:

1>TYPE S:Startup-Sequence NUM

```
1 ; $VER: startup-sequence 39.9 (9.8.92)
2
3 C:SetPatch QUIET
4 C:Version >NIL:
5 C:AddBuffers >NIL: DFO: 15
6 FailAt 21
```

A line number is added to the display every time a carriage return is encountered – no restriction is placed on the length of individual lines and no attempt is made to break lines at spaces. The listing can be paused at any point by pressing the spacebar and started by pressing <Backspace>.

Examining Binary Files

Binary files can be anything from plain text to executable commands, sounds and pictures. Quite literally anything. You can use TYPE to view such files as a Hex/ASCII dump like this:

```
1>TYPE C:TYPE HEX
```

0000:	000003F3	00000000	00000001	00000000	...Û.....
0010:	00000000	0000016D	000003E9	0000016D	m...È...m
0020:	4E55FFB0	48E73732	307C0004	26507E14	NU.∞HÁ720 ..&P~.
0030:	43FA031A	70242C4B	4EAEFDD8	2440204A	C'..p\$,KNÆ~ÿ\$@ J
0040:	20086700	02EE7013	720041ED	FFD410C1	.g..Óp.r.AÌ.'.i
0050:	51C8FFFC	41FA0302	22084DED	FFD4240E	Q».A'..".MÌ.'\$.
0060:	76002C4A	4EAEFCE2	2B40FFB4	4A806610	v.,JNÆ,+,@.¥J.f.
0070:	4EAEFF7C	22002403	4EAEFE26	600002AA	NÆ. ".\$.NÆ.&'..™
0080:	4EAEFFC4	2B40FFC4	203C0000	021A7201	NÆ.f+@.f <....r.

A typical hex dump – the first few lines of C:TYPE V39

This probably looks like gobbledygook unless you happen to program in C or assembly language. The listing splits like this:

- Column 1. One word offset from start of file in hex.
- Column 2-5. Four longwords of binary data.
- Column 6. 16 bytes of ASCII. Non-printing characters are "."

We can learn a lot from a quick glance at a file such as this. For instance, the first longword (4 bytes) is a hex digit 000003F3 which just happens to tell AmigaDOS this is a "load file" which can be scatter loaded by machine and relocated if necessary. Do what? You don't have to worry about that as the Amiga takes care of it for you. However, other file types such as IFF files always start with "FORM" or "CAT" as the first four characters of the ASCII dump – which is a dead giveaway. Skip one longword and you'll find the four letter IFF FORM type – usually something like ILBM (graphics) SMUS (music) or 8SVX (sampled sound). Take a look at some of your own files like this – it's surprising what you can discover!

Searching For Files

It is not unusual to lose a file from time to time – especially on a hard disk. Moreover, if you program in an advanced high-level language such as C or Pascal or use an assembler then you will probably have a lot of “include” files containing the Commodore-supplied macros and operating system definitions. Since these are hidden away in directories it is sometimes difficult to remember the exact path in a hurry. SEARCH can do this for you quickly and easily – the basic syntax for this usage is as follows:

```
SEARCH FILE <path> <filename|pattern> [ALL]
```

Imagine you wanted to find the C include file “dos.h” somewhere on the hard disk partition “DH0:”. The command would look something like this:

```
1>SEARCH FILE DH0: DOS.h ALL  
DH0:1c/Include/dos/dos.h
```

Search pattern

File searches may also include a pattern – as in the following example which searches a hard disk partition for any file starting with “di”:

```
1>SEARCH FILE SYS: DI#? ALL  
WB3.0:C/Dir  
WB3.0:C/DiskChange  
WB3.0:Disk.info  
WB3.0:System/DiskCopy  
WB3.0:Libs/diskfont.library  
WB3.0:Fonts/diamond.font
```

Searching in Files

Another useful facility of SEARCH is its ability to look inside text files for strings of text. SEARCH looks inside binary files too, but the output is rarely of any use. If SEARCH is looking inside a binary file it typically complains with a warning message “line too long”. Although it is possible for this to occur in a text file, it is much less likely. Used in this way, SEARCH has the following basic syntax:

```
SEARCH <path|pattern> <text> [ALL] [QUICK] [PATTERN] [NONUM]
```

Programmers can use this facility to locate text and definitions within include files. For example, you might want to find the include file which has the memory type definitions for AllocMem (MEMF_PUBLIC, MEMF_CLEAR etc.) and unless you happen to know these are in Include:Exec/Memory it could take you some time to

find them. Here's an example everyone can try – locating the RESIDENT command in the Startup-Sequences:

```
1>SEARCH S:Start#? RESIDENT
11 Resident >NIL: C:Assign PURE
12 Resident >NIL: C:Execute PURE
56 Resident Execute REMOVE
57 Resident Assign REMOVE
```

You will notice here that SEARCH locates and displays the line number of each occurrence. You can include a pattern in the path by default (as in this example) and the search string too. If you want to include a patterned search string the PATTERN keyword must be used, viz:

```
1>SEARCH DH1:TextFile B(a|en)#? PATTERN
200 and the fruit bats are easily distinguished by their
221 long-eared bat can be easily identified by its dist-
1229 but this is a major constituent of benzenoic acid.
1439 and also typical of the ring-shaped Benzene molecule.
```

Finally, if you use SEARCH reasonably often you will find that the message:

```
Line x truncated
```

or alternatively:

```
Warning: line x too long
```

appears quite often. This is normal, and just means that the line being searched is too long for a normal script file. This happens because SEARCH looks into every file in the directory or disk and cannot differentiate between script files and any other type of files!

Attaching Comments

File comments may be useful in some circumstances when you need to know more about a particular file or group of files. Comments are a string of up to 79 characters attached by the FILENOTE command thus:

```
1>FILENOTE S:Startup-sequence "Like PC's AUTOEXEC.BAT"
```

The command is very easy to use as this example demonstrates, but you must usually supply quotes around the string or the Shell will become confused. You can include ANSI escape sequences for colour but this should be avoided in general. The quote character may be included by escaping it with *.

Multiple files can be qualified with a pattern and an ALL switch is supported, although this is of limited use. Comments are only

shown by the LIST command and in the Workbench's Icon>>Information window.

- Filenotes do not take up extra room on the disk because space is already allocated in the file's special "header" or "fib" block. It is not possible to attach a filenote to a user-directory because of the different block type.

Sorting To Order

The SORT command is a vastly underrated command. The crux of this command is it takes a text file and sorts each line into alphabetical order. It also allows you to specify at which point in the line the command should start to sort on; a most useful feature as shown in the examples below. But first the essential command syntax:

```
SORT FROM <filename> TO <filename> [COLSTART <num>]
```

Let's look at a practical example.

First create a file made up of a list of files. The best example to try is to work with the command files on the Workbench disk as follows:

```
1>RAM:
```

```
1>LIST TO RAM:FILES SYS:C NOHEAD
```

This will list all of the files in the C directory of the SYStem disk to the Ram Disk as a file called 'Files'. If you get a directory of the Ram Disk you will see it is in place. The directory list can be examined by loading it into ED thus:

```
1>ED FILES
```

What you see will indeed be a list of the files with the normal information detailed by the LIST command, namely size, flags, plus date and time of creation.

Now we can sort the files into alphabetical order:

```
1>SORT FROM FILES TO ALPHAFILES
```

The sort will take but a few seconds. If you load the resultant 'AlphaFiles' file into ED you will see that the files have indeed been sorted into alphabetical order.

Another useful sort would be to arrange the files in order of size. This can be performed by using the COLSTART option. AmigaDOS files are not normally bigger than five numbers and ED is itself the largest file. The first column is number '1' and moving the cursor across and counting we find that the first number in the file's size is at column 28. So to create a new file sorted according to size we use:

1>SORT FROM ALPHAFILES TO SIZEFILES COLSTART 28

This file can be read into ED and you should see that the files are listed according to size, starting with the smallest first and ending with ED, the largest.

Joining Files

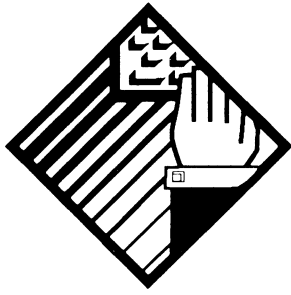
The JOIN command can be used to link several files together to form one new file. Any number of files can be appended. The basic syntax is:

JOIN <file1> <file2> [<more files>] AS <new file>

One large new file is created and the original files remain intact and unaffected. For example, if you have created the three files based on LIST in the SORT examples, you could combine this to make one large file called 'BIGFILES' with:

JOIN Files ALPHAFILES SIZEFILES AS BIGFILES

The files are added to the new file in the order in which they are specified.



22: Advanced System Control

Overview

- > Changing the system fonts
- > Changing the keyboard
- > Controlling the Printer
- > Checking the CPU
- > Switching fast memory

Commands detailed in this chapter are not so much advanced as rarely used or only used in special circumstances. For more detailed information consult the appropriate listings in *Mastering AmigaDOS 3 - Reference*.

Font Control

The following two commands are directly responsible for controlling Amiga fonts:

Setting Fonts

The default font (typeface) for the system windows is usually determined from the Workbench preferences. However, you can change the font used by the Shell window with SETFONT. It should be noted that the console.device (the software driving the Shell windows) cannot cope correctly with proportional fonts – in other words most of those supplied. Also, setting the size too large may produce some very odd results. For example:

```
1>SETFONT pearl 8
```

This command may be useful in scripts for changing the font of a console window temporarily – however, it must be set back to something sensible on exit.

Note also that it is not possible to determine the font in current use without special software. If in doubt use TOPAZ 8 which is available in ROM or better still, use a private console window.

Noting Font Changes

If you add or delete font sizes from a bitmapped font directory,

the corresponding "dot-font" font description must be updated using FIXFONTS. The command works identically from AmigaDOS and Workbench as the following example demonstrates.

1>FIXFONTS

This command updates and re-writes the dot-font files for every font it finds so it can take some time to execute and should only be used when necessary. If in doubt, it does no harm to use it.

Keyboard Control

The Amiga's default keyboard layout is determined by a translation table that usually resides in ROM. This is OK for the American manufactured machines but causes mysterious results in other countries which may have extra characters (such as ç and ß) marked on the keycap or have keys such as @ and # in different places: the UK for instance.

During startup, the default keyboard is set by IPREFS in AmigaDOS 2.1+, or by SETMAP prior to that version. For this reason it is unusual to set the keyboard to another layout after the machine has booted. The keyboard layouts are always stored in the DEVS:Keymaps directory, available as KEYMAPS: from 2.1. Applications occasionally use custom made layouts for special purposes – a keypress does not necessarily have to generate a single ASCII character.

Two functionally similar commands are provided in different versions: SETMAP reads the keyboard layout directly from DEVS:Keymaps/?? and SETKEYBOARD reads the layout from a defined path or DEVS:Keymaps/?? by default. SETMAP is used by AmigaDOS during startup. Later versions rely on IPREFS and the settings determined by the INPUT preferences.

Examples:

1>SETMAP GB ; usually done during startup

1>SETKEYBOARD KEYMAPS:GB

One other point worth noting here is that SETMAP is stored in the System drawer (with an icon) and is only available from AmigaDOS.

Printer Control

With the exception of the Preferences tools (not covered in this volume) AmigaDOS has little to do with the printer as a specific, hardware device, although it is supported by the non-specific device interface. You can, for instance, send a text file directly to the printer like this:

1>COPY DFO:SOMETEXT TO PRT:

However, the CMD (Command) utility can be used to gain some control over what the printer receives. In essence, CMD grabs information sent to the printer port and copies it to a file (or files) instead. This command can also be used to provide rudimentary print spooling, although this facility is of limited use. CMD is typically started as a process so it can wait around until needed, viz:

```
1>RUN CMD PARALLEL.DEVICE RAM:PRINT
```

```
[CLI 2]
```

CMD expects as an argument the full name of a device driver, "parallel.device" in this case, and a destination file to send output to. In order to test CMD, we can simulate something using the printer by copying a text file to the PRT: device:

```
1>COPY S:Startup-sequence PRT:
```

```
Redirected 629 bytes from parallel.device to RAM:print
```

```
Cmd re-direction of parallel.device removed
```

You can examine the file "RAM:Print" using TYPE as explained earlier, although in a simple case such as this one it will be a simple text file. Provided there is enough room in RAM you can "buffer" dumps from applications such as Deluxe Paint and print them later.

This gives rise to an interesting possibility whereby you can save a printer dump to file and print it on another machine. Let's say a friend has a 24-pin colour printer and you only have a basic machine. You could set your Preferences printer driver comply with their machine and save a "spool" file to disk like this:

```
1>CMD parallel.device DF0:print
```

Now you could start printing from DPaint until CMD responds with something like this:

```
Redirected 420621 bytes from parallel.device to DF0:print
```

```
Cmd re-direction of parallel.device removed
```

Now take that disk to your friend's machine and print it like this:

```
1>COPY DF0:PRINT TO PAR:
```

It is important to copy the final file to PAR: and *not* PRT: because the spool file has already been filtered through the PRT: device.

- If your machine has a serial printer then you must specify "serial.device" instead of parallel. Similarly, if your friend's machine has a serial printer, you should copy the print file to SER: instead. Note also that CMD does not conform to the usual AmigaDOS template guidelines – preferring a Unix-like command line instead. See *Mastering AmigaDOS 3 – Reference* for more information.

Processor Control

All Amiga models are based on one of the Motorola 680x0 series central processing units. Even the most basic 68000 model features highly advanced 16/32 bit architecture and, while no longer leading edge technology, is still a powerful and capable workhorse. Later machines are fitted with a 68020 (A1200), 68030 (A3000 and A4000/030) or 68040 (A4000) plus a variety of specialist custom chips.

Some control of the processor is capable from AmigaDOS via the CPU command although this is of limited use to the vast majority of users. In default mode CPU operates as:

```
1>CPU
```

```
System: 68030 68881 FastROM (INST: Cache Burst) (DATA: Cache  
NoBurst)
```

This information reveals the current processor configuration and, on my machine, the presence of a floating point unit (FPU). In this case the machine in question is an Amiga 3000 with a soft-kickstart (RAM loaded). Instruction and data caching are switched on. This means the vast majority of code will fly along. Other machines give different results. Note that CPU incorrectly identifies the 68882 FPU as a 68881.

Note: on machines running Kickstart 3, it is usually better to use the startup options to disable the processor caches rather than relying on the CPU command.

- With AmigaDOS 2.1 and higher, the machines configuration is better displayed with SHOWCONFIG.

Memory Control

The Amiga's memory system is extremely clever and the user has very little control over what happens to it. Two AmigaDOS commands manage the basics:

Available Memory

AVAIL takes a *snapshot* and shows the current memory usage. The snapshot is only valid for the exact instant the command runs and each invocation may generate a different result. All figures are returned in bytes. Example:

```
1>AVAIL TOTAL
```

```
412730
```

No More FAST Memory

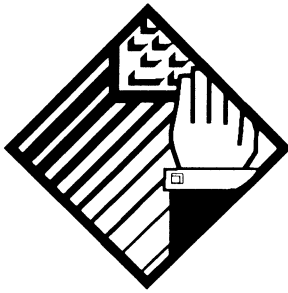
In very dire circumstances or when you want to simulate a machine with CHIP memory only, you can use NOFASTMEM. This switches off

all FAST memory attached to the system but only affects future allocations of FAST memory not current applications.

It is supposed to provide compatibility with very old software, in practice such software is usually badly written and often, does not provide any output and *must* be run as a second process. Example:

```
1>RUN NOFASTMEM
```

```
[CLI 4]
```

23: Introducing Re-direction

Overview

- > Handles and Streams
- > Working with re-direction
- > Output re-direction
- > Append mode
- > Input re-direction

Re-direction is one of the most powerful features of AmigaDOS and also one of the trickiest to get to grips with. Re-direction is a technique whereby either or both of the console's stream handles are changed to a user-defined destination before a command is executed.

Confused? Don't worry, put as laconically as that, it doesn't make much sense at all. Analogies are the keys to understanding here and the phrase causing the real confusion is *stream handles*.

Before we proceed any further therefore, ask yourself this: "What's a *handle*?". A doorknob or even a nickname perhaps? However you define it, a handle is something you grasp hold of. The console device gives AmigaDOS (actually the Shell) two handles: the output handle is where a command sends (messages) information to and the input handle is where it gets information (what you enter) from.

Now let's give *stream* the same treatment. A stream is a flow of something – usually water, but it doesn't have to be. If the stream is a flow of data, then the input stream is the information entered at the keyboard. Similarly, the output stream is the information displayed in the Shell window. Under most circumstances therefore, we can view the input stream as being a drip and the output stream as being a flood.

Taking this one step further we arrive at the stream handles. These are the places where the Shell expects to send and receive data: and this is what re-direction acts on. There, that was painless.

When you enter a command in the Shell window and press Return the command reads the command line from the current input handle. More correctly called the Console Input Stream or CIS. When the command wants to tell you something, it sends it back to the Console Output Stream – COS. (There is another little beastie hanging around called CES – Console Error Stream, but this can usually be assumed to be the same as COS.)

It's no accident that COS and CIS look exactly like file handles. The Shell window could just as easily be a file stored in the Ram Disk for all the current command could care. This is the essence of re-direction and is probably better illustrated with some examples.

Output Re-direction

The '>name' sequence has a special meaning to the Shell, it says "Open a file called 'name' and send its file handle as COS. 'Name', is usually, but not always, a file for example:

```
1>CD ; default mode for CD
```

Workbench3.0:

```
1>CD >RAM:DIR ; now send it to a file with re-direction
```

```
1>TYPE RAM:DIR ; and observe what happened
```

Workbench3.0:

As we have said, the output file does not have to be a file at all. For the next example we'll send a sorted directory listing to the printer:

```
1>DIR >PRT: ALL
```

As you can see, there really is nothing to it. Re-direction of this type can be avoided in some cases if the command supports a TO argument: which has the same effect. Examples include EVAL, ECHO and LIST.

It is important to note with both of these methods, any existing file at the destination is overwritten. If a file exists (or is in use by some other process) you will get a Shell error "cannot open re-direction file". This should not be a problem in command mode, but calls for very careful choice of names in script files.

Hush Now...

It is interesting to note since any device can be used for output, it is also possible to tell a command to shut up completely by channelling its output to the dummy sponge device NIL:. Consider the following example:

```
1>LIST >NIL:
```

Such fiddles do have a purpose though. For instance, you might want to suppress output from some command while it works. While many commands support a QUIET mode, a few do not. Moreover, when the QUIET switch is used, errors are still reported to CONSOLE: (*). Re-direction to NIL: fixes this one permanently.

```
1>DELETE RAM:FUMANCHU QUIET
RAM:FuManChu not deleted: object not found
1>DELETE >NIL: RAM:FUMANCHU QUIET
1>; nothing, zippo, quiet, hush... peace
```

Output Re-direction (Append Mode)

When output is being sent to a real file, that is not a terminal or some other non-filing system (disk) device, then you can make use of the append mode. This is similar to normal output, but every time a new command line is entered, the output is tagged on to the result from the last. This is similar to the JOIN command. Append mode output re-direction is achieved using ">>name". Here is a tediously clichéd example, for want of a better one.

```
1>ECHO >>T:HELLO "Hello" NOLINE
1>ECHO >>T:HELLO "world"
1>TYPE T:HELLO
Hello world
```

If the file does not currently exist it is created. Be extra careful to remove such temporary files created in scripts – they will cause havoc if the script is executed more than once and the bug can be a beggar to find.

Input Re-direction

Re-direction of the input handle is much less common than that of the output handle and is most often sent to NIL: when an application is being run in the background, for example:

```
1>RUN <NIL: >NIL: BLANKER SECONDS=60
```

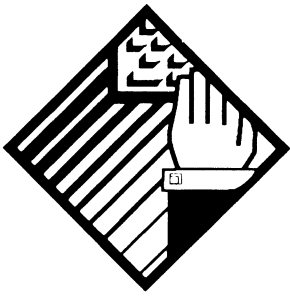
This also gives rise to a useful alias, RUNBACK:

```
ALIAS RUNBACK RUN <NIL: >NIL:
```

which is used in the following way:

```
1>RUNBACK BLANKER seconds=60
```

Input re-direction can also come from a file – we'll see how this works in Chapter 34 on EVAL.



24: Advanced LIST

LIST can be used to create output to any device supplying basic information about files or directories. However, we have only just touched on the surface of what it can do. There is very much more which make it perhaps the most versatile of AmigaDOS commands. Table 24.1 lists the various parameters and/or options that can be used in conjunction with this amazing command.

The LIST Options

Table 24.1 describes only briefly the options provided by LIST. The options that we have not so far encountered are detailed here.

KEYS

The KEYS option is of use if you should ever need to read data directly off a disk using a suitable program tool. It effectively lists the number of the block where a particular file starts. This can then be used to locate the information on the disk directly. For example:

```
1>LIST C:ED KEYS
```

```
Directory "C:" on Wednesday 02-Sep-92
```

```
ed    [1269] 25044 --p-rwed Today 11:51:33
```

```
1 file - 50 blocks used
```

Overview

- > LIST options
- > Using LFORMAT
- > LFORMAT qualifiers

The number held within the square brackets, 1269 in this case, is the block number.

Note: You should not attempt to access data directly from a disk unless you are quite sure you know what you are doing! In any case direct disk access should always be performed on a backup copy of the disk – just in case.

<i>Option</i>	<i>Action</i>
KEYS	Position of header blocks on disk
DATES	Dates and time of creation is shown
NODATES	Dates and times are suppressed
PAT	Files listed to pattern supplied (redundant)
TO	Outputs TO a file
SINCE	Displays objects created since a date (and time)
SUB	Displays objects containing sub-string
UPTO	Dated UPTO the date/time given
QUICK	Shows names only
BLOCK	Gives object size in disk blocks
DIRS	Shows directories only
FILES	Shows files only
LFORMAT	Format using a special string
NOHEAD	Suppress 'header' information
ALL	List all files in all subdirectories

Table 24.1. The LIST options.

When this keyword is supplied, every object is shown with a number in square brackets corresponding to the key position of its FIB on disk. The key number is only useful for disk hackers – AmigaDOS commands do not use them directly. (The maximum number of keys on any disk is given by doubling its storage capacity in K – so an 880K floppy has 1660 key blocks.) For example:

```
1>LIST C:LIST KEYS NOHEAD
```

```
List      [ 302]  5904 --p-rwed Today 10:11:05
```

BLOCK

The size of files and directories is shown as the number of 512 byte blocks it occupies. LIST incorrectly shows a directory as being two blocks long. Directories are not shown as "dir" in this mode. For example:

```
1>LIST C:LIST BLOCKS NOHEAD
```

```
List      10 --p-rwed Today 10:11:05
```

DATES

By default, LIST attempts to substitute recent dates by a day name and future dates by the string 'Future'. For example:

```
1>LIST C:LIST NOHEAD
```

```
List          5904 --p-rwed Today 10:11:05
```

```
1>LIST C:LIST DATES NOHEAD
```

```
List          5904 --p-rwed 24-Sep-93 10:11:05
```

QUICK

As the name implies this option forces AmigaDOS to produce a quick display of the files, by listing only the filenames. NOHEAD is assumed. For example:

```
1>LIST C:E#? QUICK
```

```
ED
```

```
EXECUTE
```

```
EDIT
```

ALL

This option works in much the same way as DIR's ALL switch. It causes LIST to search through the complete directory hierarchy. Provided the NOHEAD option has not been supplied, LIST provides two totals. One appears at the end of each listed directory and the other at the end of the listing showing the total number of files, directories and blocks listed. This summary total is shown in capital letters.

LFORMAT

A case of saving the best to last. In terms of its ability to save you an awful lot of work and typing at the keyboard, the LFORMAT option truly comes into its own. In its simplest form LFORMAT allows you to insert a string of text within the LIST output stream. The best way to visualise this is to actually see it, so try this example for size:

```
LIST C:RUN LFORMAT "Command is %s*n"
```

LFORMAT strings do not send a newline character, so you must specify one and the "*n" in the above example does this.

A typical line of output will be:

```
Command is Run
```

The string 'Command is ' has been printed and then the name of the file as it was encountered. The '%s' is a parameter used by LFORMAT which is taken to mean 'insert filename at this point'.

The power of the LFORMAT becomes apparent now – because it can write your script files for you! For example, suppose you wished to copy every file that contained '.info' into the Ram Disk. This can be

done by using a suitable LIST LFORMAT combination to create a text file that can then be executed. to do this first create the script file, called 'INFcopy', using LIST:

```
LIST TO RAM:INFcopy SYS:#?.info LFORMAT "COPY %s RAM:*n"
```

This looks a bit complicated, so let's examine each section of it in turn:

```
LIST TO RAM:INFcopy
```

This directs the LIST output to the Ram Disk and places it in a file called 'INFcopy'.

```
SYS:#?.INFO
```

This makes LIST select all files on the root directory of the system disk that end with '.info'.

```
LFORMAT "COPY %s RAM:*n"
```

This is written to the 'INFcopy' file with the full filename inserted at the position of '%s'. Thus if the first file that LIST encountered on the disk was 'Trashcan.info' then LFORMAT would write the following line to the INFcopy file:

```
COPY Trashcan.info RAM:
```

When the LIST command has completed it will have written all the relevant lines to the RAM Disk. The file is, like any script file, all plain text, so you can load it into ED or MORE and examine it further. Alternatively you can get it to out the task it was created for by typing:

```
EXECUTE RAM:INFcopy
```

Open the RAM Disk window you should see various icons displayed. Of course, this example is nothing new and is rather a long-winded way to perform a task that could be completed much more efficiently using the COPY command directly itself. However, it does give you an insight into what is possible.

More Ss

The %s parameter can be included in the LFORMAT string to signify that the filename should be inserted at that particular point. In fact you can use %s up to four times within a command line and the meaning that each of these takes is given in Table 24.2.

<i>Parameter</i>	<i>Meaning</i>
%s	Filename only
%s%s	Path and filename
%s%s %s	Path and filename, filename
%s%s %s%s	Path and filename, Path and filename

Table 24.2. The s% parameters.

The dual uses of %s effectively allow you to be able to specify source and destination path and filenames as required.

Imagine that you have a directory containing a large number of files which are all delete enabled and which you wish to protect against accidental deletion: by clearing the 'd' flag. The obvious answer is to use the PROTECT command on each one in turn. But this is very long-winded to say the least, better to get LIST to do it for you. For example, copy all the files from the "System" directory on the Workbench disk into the Ram Disk:

COPY FROM DFO:SYSTEM/#? TO RAM:SYSTEM

and then switch to the destination directory:

RAM:SYSTEM

If you list the files you will see that the 'd' protection flag is set (ie, it is present). To enable us to protect them from deletion it must be unset with '-d'. To create the file to do this we can use:

LIST TO RAM:unD RAM:System/#? LFORMAT "PROTECT *"%s%s*" -D"

This will create a file of command lines, which might look like this:

PROTECT "RAM Disk:SetMap" -D

PROTECT "RAM Disk:DiskCopy" -D

The whole process has been made that much simpler, and you can try it out on the RAM Disk copy of the System directory by typing:

EXECUTE unD

If you re-LIST you will see that the delete flag has been unset in each. In fact, PROTECT has got its own pattern matching but this is a simple way of showing how the LFORMAT string works.

The LFORMAT keyword can be used in conjunction with the date windowing and PAT, SUB and ALL keywords, the others have no effect. A simple example of LFORMAT usage might look like this:

1>LIST SYS: LFORMAT "Directory: %S Object: %S*n"

Advanced LFORMAT

The LFORMAT string is passed through a ROM function called RawDoFormat. C and assembly language programmers may be familiar with this function because it provides simple C-style output formatting. Table 24.3 lists the possible qualifiers.

A formatting parameter can appear between the % sign and the qualifier, taking the following format: -nnn.nnn.

- A minus sign, which indicates the string is to be left justified within its field.
- nnn Width. A number indicating the width of the field in which the qualifier is to be printed. If this number is omitted, the field width is determined by the length of the qualified string. May be omitted.
- . Separator. A period separating the field width from the precision. Must be present if the precision digit is being used, should be omitted otherwise.
- nnn Precision. A number indicating the number of characters to display starting from the right end of the string.

<i>Qualifier</i>	<i>Meaning</i>
%A:	The object's attributes (protection flags).
%B:	File size in blocks.
%C:	Comment.
%D:	Creation (or last modified) date.
%K:	Physical position on disk.
%L:	Length in blocks.
%N:	Name.
%P:	Complete path (without an object name).
%T:	The time associated with the object.
%S:	Pathname only.
%S%S:	The pathname and object name.
%S%S%S:	Pathname and object name, pathname.
%S%S%S%S:	Pathname, object name, pathname, object name.

Table 24.3: LIST's LFORMAT qualifiers.

This may all seem a little complex at this stage (unless you have programmed in C) but it is simple and very convenient. Here are some examples:

1. We can simulate use of the QUICK switch like this:

```
1>LIST C:LIST LFORMAT "%n*n"
```

```
List
```

2. Or improve on it slightly like this:

```
1>LIST C:LIST LFORMAT "%p%n*n"
Workbench3.0:C/List
```
3. AmigaDOS filenames are limited to a maximum of 30 characters, but few people use more than eight plus a three letter extension – so we can set a maximum field width to 13 – 8+3 plus 1 for the "." like this (brackets are shown for clarity):

```
1>LIST C:LIST LFORMAT "(%13n)*n"
(      List)
```

4. If a name is too long, it is trimmed back to fit inside the field. Most will be flushed right – although you can override this by including a minus symbol like this:

```
1>LIST C:LIST LFORMAT "(%-13n)*n"
(List      )
```

5. The length of filenames can vary between 1 and 30 characters. By using 30 as a field width, you can ensure the listing lines up correctly like this:

```
1>LIST C:(LIST|DIR) LFORMAT "%30n@%t*n"
List                               @12:31:32
Dir                                @12:31:34
```

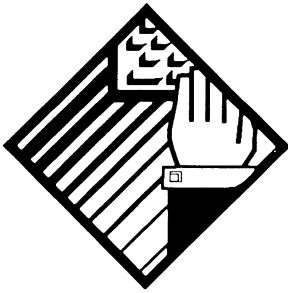
6. AmigaDOS always prints the seconds as part of the time. It has a fixed field width of eight digits – two for each of the three numbers and two extra for the colon separators. Therefore, we can use the precision digit to trim off the unwanted seconds and second colon like this (note the period in front of the digit):

```
1>LIST C:LIST LFORMAT "%n modified at %.5t *n"
List modified at 12:31
```

There is no limit to the length of the LFORMAT string apart from the maximum length of the command line – 254 characters – so this gives a lot of scope. All new code should prefer the use of the new C-style qualifiers instead of the old "%s" formatting.

<i>Field specifier</i>	<i>Meaning</i>
-	Left justify within field.
nnn	Width. Determines the maximum width of the qualified field. Optional.
.	Dot.. Separates the field width from the precision. Must be present if precision is being used. Not required otherwise.
nnn	Precision. Number of characters to display from the left of the string.

Table 24.4: Qualifier extensions



25: Multi- tasking Amiga

Overview

- > Multi-tasking Examined
- > Output Insertion
- > A New Shell
- > RUNning commands

One of the major features of the Amiga is its multi-tasking capabilities. Multi-tasking is the ability to run several programs at once – where each program is seen by the Amiga as a task to do – hence the terminology.

A task can be any process that can be carried out by the Amiga: printing a file, formatting a disk, running several Shells; even all of these at once.

Tasks started by AmigaDOS are called “processes”. This distinction may seem trivial, but it is vital that you understand at least the naming conventions at this point. Advanced texts on AmigaDOS and associated topics will expect you to understand this – it’s better, therefore, to think of the Amiga as having a multi-tasking operating system and AmigaDOS as being a multi-process Device Operating System.

Multi-processing is not the same as multi-processor architecture. You don’t have to know what the differences are, so long as you are aware that they exist. The Amiga is a single processor machine with pre-emptive multi-tasking.

Pre-emptive What?

There are several definitions of multi-tasking and the three main ones are task switching, co-operative multi-tasking and pre-emptive multi-tasking.

The ability of your Amiga to multi-task effectively depends, to some extent, on the amount of memory installed. Each process will require a certain amount of

memory and other resources (ports and so on), therefore the more processes are launched (Amiga buzz-talk meaning "set in motion") the more memory is required. For instance, a 1Mb A500+ is capable of running several 100k programs, but clearly a 2Mb A1200 will be able to double the amount of similar sized processes it can run.

Who Does What?

Although it may seem as though Exec – the multi-tasking executive – is running several programs at once, this is not strictly true. The CPU is only capable of executing one instruction at a time and the illusion of running many programs at once is created by sheer speed and some timing trickery.

In a simplistic example, if there are three processes running, the Amiga runs a bit of process one, then a bit of process two and then a bit of process three. Having serviced all three processes it starts the loop again by running a bit of process one. All this happens at staggering speed to provide a working multi-tasking environment that is invisible to the user and programs alike.

When a large number of processes are running, the time it takes for the operating system to get back to the first process becomes perceptible, and the computer seems to slow down. High priority processes can grab the processor for their own use whenever Exec feels the need however, thus avoiding bottlenecks found in round-robin co-operative multitasking.

Some processes require to be continuously "run" by the operating system – in particular, games and graphic demonstrations. Some programs are more docile in operation – they go to sleep, and hang around until something wakes them up. For example, memory permitting, running two or more wordprocessors should not have any apparent drag on the system. The usefulness of multi-tasking becomes more apparent the more you use it. Being able to print a long file and to continue working at the computer is something many systems – including some of the best personal computers – are unable to do.

Types of Multi-tasking

Task switching is the most basic form. In this scheme, the processor is shared equally between two or more loaded tasks. Only one task can be running at any instant – the others are held until invoked by the user. Therefore if a spreadsheet was in the middle of a recalculation and the user switched to a wordprocessor, the wordprocessor gets control of the CPU and the spreadsheet enters a sleep state. Specially written software (such as clocks) can give the appearance of multi-tasking by using hardware interrupts.

Co-operative multi-tasking is similar to task switching, but under this scheme more thought is given to the task controlling the processor. If a task requires immediate attention from the user, further task switching is disabled until the request has been satisfied.

Pre-emptive multi-tasking is the most advanced form. The Amiga was the first, and is still the only home micro design to feature it as standard – strange when you consider Motorola designed the 680x0 series with multi-tasking in mind!

In a pre-emptive environment, one task has control of the processor while others are in a wait state. Each task is allocated a certain amount of time and if it doesn't enter a waiting state in that time, loses control of the processor.

However, if a background task requires immediate attention, it can halt the current task and take control of the processor. This is totally invisible to the task – it just stops dead in its tracks until the executive gives it the processor back again. Other tasks can live and die in the time it takes to get round all the waiting tasks just once.

Generally speaking all this is completely invisible when you use the machine. For example, did you know that every disk drive is controlled by a separate task? Imagine you have two drives attached to the machine and inserted a blank disk in each. You could stop whatever you are doing, start formatting both disks, and then carry on where you left off.

While the disks are being formatted, the CPU is being switched between two format processes and two controlling "trackdisk.device" tasks while also servicing anything you happen to be doing with the mouse and keyboard.

Co-operative Multi-tasking

Although not strictly a multi-tasking feature, we thought this would be a good time to introduce a little used feature of the Shell – *command insertion*. Essentially, this facility allows you to insert the first line of output from any command into the input stream of another. Replacement is triggered using the open single quote (ALT+' on most keyboards). Typically you might use this with DATE and ECHO thus:

```
1>ECHO "The date and time is 'DATE'"
```

```
The date and time is Saturday 7-Jul-93 12:30:23
```

This facility can be used in more advanced scripts with GETENV to load the current contents of a variable. (Environmental substitution is usually performed when a command line is interpreted):

```
1>SETENV Time 'DATE'
```

```
1>ECHO "The time is 'GETENV TIME'"
```

A New Shell

As you learn more about AmigaDOS you will invariably find the need to open additional Shell windows. Without the Workbench there is no access to the Shell icon so the NEWSHELL command is provided for this purpose.

The basic synopsis of this command can be expressed thus:

NEWSHELL [WINDOW=<window>] [FROM=<from file>]

The 'window' and 'from file' are optional. From files are scripts and we'll not cover them just yet. By default, NEWSHELL attempts to execute the default script, "S:Shell-Startup". Typically you can open a Shell using the default parameters like this:

1>NEWSHELL

The new Shell window opens with a message like this:

New Shell process 2

Workbench_3.0:>

This Shell is automatically active and any keyboard input will be directed to its window until another is selected. Such a Shell could be used to allow direct access to the Ram Disk like this:

1>RAM:

You now effectively have a window on the Ram Disk and one on the system boot drive. Close the second window and open a new Shell with a specified window like this.

1>NEWSHELL CON:100/10/500/100/Shell2

A second Shell window appears covering most of the top of the screen, leaving enough space to the left to allow the Workbench icons to be displayed. The title "Shell2" is displayed in the window. Any interactive device may be used for the window description – see Chapter 28 on Devices.

Running Single Window Multi-tasking

You may have already discovered that it is possible to start several programs at once from Workbench. If you haven't already tried this, now would be a good time!

In a programming environment it is normally better to run the programs directly from the Shell window itself. This can be done simply by typing the name of the program to be run as an AmigaDOS command.

CD Utilities

Clock

The program runs but the Shell window does not respond to any further commands typed. The command RUN overcomes this obstacle, allowing programs to be set into motion and then returning the Shell prompt allowing the command line interface to be used as per normal. The syntax of the command is:

RUN <program>

Try running the clock program once again using the RUN command as follows:

RUN Clock

[CLI 4]

The Clock program is set in motion and the prompt returns to the Shell window. You will notice the message "[CLI XX]" to indicate that another Shell process has been started. RUN is very similar to NEWSHELL, however processes started in this way share the current console rather than running in their own. This has some important implications which we will discuss shortly. Like NEWSHELL, a process started by RUN inherits the current task stack size and is normally able to access the local environmental variables although this is unlikely to be necessary.

You have to re-activate the Shell console window (by clicking in it) before you can do anything. Commodities exchange 'AutoPoint' provides an answer (see Appendix E on Commodities), but it can cause as many problems as it solves. At this stage, therefore, don't use AutoPoint just yet. Now enter this:

1>RUN Calculator

[CLI 6]

Note this Shell has been started as CLI 6 (CLI is retained for compatibility with older system. Prior to release 2, you could only start 20 processes in this way before getting an error. That doesn't mean you can only use RUN 20 times, but that you can only have 20 processes running at once. From AmigaDOS 2 onwards, processes are handled by a list and the only restriction on the number running is the amount of available memory.

RUN Like a Shell

RUN is only usually used to start a single process, however, it may be desirable for RUN to chain a series of commands as if they had been started from a Shell. A series of commands can be linked in this way by adding a '+' to the end of each line – the plus means "more follows". For example, the following program copies the programs above to the Ram Disk, run each one in turn and then delete them from the Ram Disk:

1>RUN ECHO "Copying Files to RAM:" +

CD SYS:Utilities +

```
COPY FROM (clock|calculator) TO RAM: +  
CD RAM: +  
RUN Calculator +  
RUN Clock +  
ECHO "Deleting Files" +  
DELETE RAM:(clock|calculator) +  
CD SYS: +  
ECHO "All Done!"
```

Note that when you press <Return> after typing the first line the command is not executed because the AmigaDOS interpreter has recognised the '+' character and realises that there is more to come. Nothing runs until a line is entered that does not terminate with the '+'. For this reason you should be careful not to add any extra characters, such as a space, after the '+' character.

Closing the Shell

So far, you have probably closed all your Shell windows using the Close box. There is nothing wrong with that, but AmigaDOS provides two other methods: firstly you can send the End-of-File message to the console by pressing:

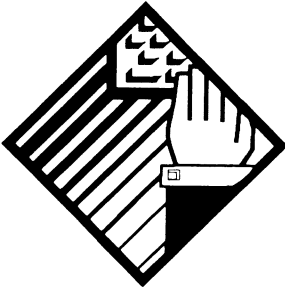
```
<Ctrl-\\>
```

or you can use the ENDSHELL command. This command attempts to stop the Shell process, close the window and free any resources it used.

In any case, it is possible some program you have RUN-launched has grabbed hold of the console I/O handles and is unwilling to let go. In this case the Shell will not close until that process is complete – pulling the rug from under a process is not polite and would lead to a system-wide crash in any case.

If this becomes a problem (and it may do when you start adding bits to the user-startup) a solution will have to be found. In fact, the answer is simpler than you might imagine. When a process is started, you can give it a pair of "dummy" input and output handles. This means it cannot send information to the console, but this should never be a problem. This method is explained in Chapter 23 on re-direction, but for completeness, here's how to start a Clock and prevent it from locking the Shell.

```
1>RUN <NIL: >NIL: SYS:Utilities/Clock
```



26: Advanced Multi- Tasking

Overview

- > Using WAIT
- > Closing the Shell
- > Changing Stack
- > Changing priority

Waiting for something to happen is boring – like watching paint dry or waiting for Mark Smiddy to get copy in. In a computer system you can wait for an event by constantly polling a variable and waiting for it to change or you can execute a loop x number of times. Both these schemes use valuable processor time that could be used by other processes and is generally regarded as a quite rude: one should respect other tasks.

If your process needs to hang around for some time, AmigaDOS provides an excellent command to do just that: WAIT. This command can wait for a specified period from a second to several hours or until a specified time. Moreover, it does so without logging a massive overhead on the multi-tasking sub-system. Typically you can wait for about a second like this:

```
1>WAIT
```

after approximately a second you get the prompt back. Not very interesting really. Similarly, you can wait for several minutes like this:

```
1>WAIT 60 MINS ; wait for  
about an hour.
```

WAITing for some time is a specialised technique used as a central part of the "Pest 3" alarm system. We can use a simple example here with RUN +.

```
1>RUN WAIT 30 SECS +  
ECHO "Hello again!"  
[CLI 9]  
1>
```


after about 30 seconds (the timings are not very accurate) the message pops up on your Shell. You can also wait until a certain time – although this is useless unless your machine is fitted with battery backed-up clock. Example:

```
1>RUN WAIT UNTIL 17:40 +  
ECHO "It's 20 to 6! Shouldn't you be at home now..."  
[CLI 10]  
1>
```

The Pest, a much more sophisticated example, is included on the companion disk.

Process Status

So how do you know what processes are running and find out more information about them? AmigaDOS provides for this with the STATUS command. Try this:

```
1>STATUS  
  
Process 1: Loaded as command: Status  
Process 2: Loaded as command: conclip  
Process 3: Loaded as command: Workbench  
Process 4: Loaded as command: Clock
```

This lists the various processes that are running under AmigaDOS at that moment – STATUS does not display the underlying tasks and there can be many more of them. The 'process' number is the number allocated to the process when it was launched. Transient commands executing within the Shell, in this case STATUS, are allocated – Shell number 1 in this instance – otherwise the Shell is said to be "Empty". The other process numbers are the ones assigned by the AmigaDOS when they were initially run.

More detailed information may be supplied by STATUS, for example:

```
1>STATUS 4 FULL  
  
Process 4: stk 4096 gv 150 pri 0 loaded as command: Clock
```

lists full information for Process number 4 and:

```
1>STATUS 4 TCB  
  
Process 4: stk 4096 gv 150 pri 0
```

supplies much the same information as with FULL except that the command name is not given. The COMMAND option allows you to locate the process number assigned to a particular command process. For example:

```
1>STATUS COMMAND=CLOCK  
  
4
```

returns the process number currently running CLOCK. If more than one process is running the command, all process numbers are returned – one to a line:

```
1>STATUS COMMAND=WAIT
```

```
9
```

```
10
```

This tells us that process numbers 9 and 10 are currently running a WAIT command.

Stack and Priority

Every process has a certain amount of stack allocated to it and a priority that Exec uses when deciding how important it is in the scheme of things. The stack and priority are displayed by STATUS described above. Stack is very important to AmigaDOS commands because they use it to store such things as local variables and return addresses from subroutine calls. Complicated stuff this and something that shouldn't be fiddled with lightly.

Most AmigaDOS commands run just fine with the system's default stack of 4000 bytes, however a few such as SORT and DIR work better with or require larger stacks to work. You can therefore, increase the size of the stack, but do not set it below this level – unless you are sure of what you are doing. The stack for the current Shell can be set to 5000 bytes thus:

```
1>STACK 5000
```

This stack will be used by all commands launched afterward and by any Shells you start with NEWSHELL.

Priority

A high-priority process always gets precedence over a low priority one – although it will not necessarily run any faster. The range of process priorities is set from -128 to +127 although the system tends to stick in the range between -20 and +20. User tasks must NOT set a priority higher than this or the system might get wrapped in knots. For safety's sake, stick in the safe range between -5 and +5 or, better still, don't fiddle with this parameter at all. You can set the priority of your own Shell (and any processes launched from it with CHANGETASKPRI like this:

```
1>CHANGETASKPRI 2 ; make me high priority
```

```
1>CHANGETASKPRI -4 ; I don't need much time
```

Jobs with a low processor overhead, such as something being copied to the parallel or serial ports can generally be set to a low priority. Normal AmigaDOS commands do not need high-priority.

Quitting Processes

To be able to quit a process rather relies on the software running the process recognising the fact that you want to. This is not as daft as it may first seem as many programs are written in such a way, which means that they cannot be interrupted at all. If a program is running in its own Shell window, and is written to allow interruption, then selecting the window and typing one of the following will halt the program:

<Ctrl-C>

<Ctrl-D>

<Ctrl-E>

<Ctrl-F>

Most AmigaDOS programs respond to the <Ctrl-C> break code. Running scripts (including the Startup-Sequence) can be stopped using <Ctrl-D>. Programs started with RUN do not have their own input console, so BREAK is used instead. The syntax for BREAK is:

BREAK <process> <option>

For example to attempt to stop process number three use:

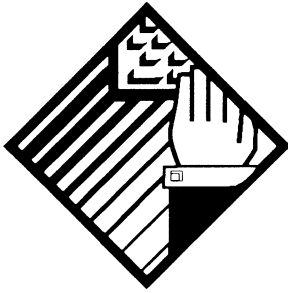
1>BREAK 3 ALL

Which sends all the possible break codes to the process three's message port. See *Mastering AmigaDOS 3 - Reference* for more details about the BREAK command and its use in conjunction with STATUS to break a specific command.

Caution!

Running several programs at once is fine, so long as the majority of them can get on with the job without requiring continual input from you. Remember that your Amiga has just the one keyboard and all processes have to share this. Having to supply keyboard input continually to various programs can become a pain. If you need to, remember to select the window running the process first.

It is dangerous to use multi-tasking programs which require continual access to disks, and in single drive systems it can be suicidal. A situation will invariably arise where a process requests a disk, which is supplied, only to have another process request a second disk. Even without the wrist pain created the disks will invariably become corrupted. On dual drive systems two processes can be made to use respective drives but the process of reading the information from the disk surfaces can, in certain circumstances, become slow and laborious.



27: Environmental Variables

Overview

- > Introduction to variables
- > Using \$
- > The ENV: directory
- > Writing to ENV:
- > Using variables in calculations
- > Indirection

The term 'environmental variables' sounds rather grand and in many ways it is. They are one of the underlying reasons behind the power of AmigaDOS! The term simply means DOS variables. Classically such variables are used to control the environment – the prompt, default path and so on.

This chapter explores environmental variables and the environment handler in detail. However, some of the concepts involved are mutually inclusive. This is something of a "chicken and egg". You must understand "the chicken" before you can fully understand the "the egg"; but some things with eggs require knowledge of chickens. For this unavoidable reason you may have to read this chapter several times before you will grasp the concepts. Don't let this put you off – we all have to go through it.

Environmental variables were introduced with release 1.3. and it was not until release 2 that the full power of AmigaDOS was harnessed. In AmigaDOS 2 and onwards, you can read the values of RC and Result2. These are private environmental variables – returned by programs – used mainly by IF and WHY.

This way of passing information between programs is very useful, especially when it might rely on information typed at the keyboard or the result of an AmigaDOS operation. Clearly these may not be known at the time of writing the program.

There are two commands associated with the creation and reading of environmental variables. They are listed below with the syntax of each:

```
SET <name> <string>
GET <name>
SETENV <name> <string>
GETENV <name>
```

SET creates the variable and calls it '<name>'. It then assigns the string to it. GETENV reads and displays the contents assigned to the variable <name>. SETENV and GETENV work in much the same way. Let's see this in action with a very simple example:

```
1>SETENV A_name Bruce
1>GETENV A_name
Bruce
```

Cute, but not a lot of use. It is important to notice here you cannot use the string "NAME" for the name of the variable. Although this doesn't cause an error, the variable itself remains unset!

MORE has an environmental variable to tell it which editor to use. For instance, say you wanted to use ED as the environmental editor then you can use SETENV to assign it to the variable 'Editor' by supplying the *full* path not just the editor's name thus:

```
1>SETENV editor C:ED
```

Similarly, you could use an external editor – on a complete different disk.

```
1>SETENV editor Prottext-program-disk:Prottext
```

If MORE needs to access the editor, it asks you "Insert volume Prottext-program-disk in any drive". It is important to realise that MORE will only ask you for that disk *if* it needs to access the editor. This can save a lot of disk swapping every time you use MORE to read a file.

A Useful Variable

If the system can use environmental variables then so can you. For instance, you might want to define a different prompt for each Shell window. It would be a nuisance to have to call PROMPT every time you start a new Shell – so why not use a variable. For this example, we've called our variable "prompt" but the name is arbitrary. Since AmigaDOS doesn't read this variable directly, we will have to tell it about it first. In the file S:Shell-Startup: remove the line PROMPT %N.%S and add the following:

```
PROMPT $prompt
```

For this example we have shown the prompts as they should appear on your screen:

```
1.SYS:>SETENV prompt %N:-
```

```
1.SYS:>NEWSHELL
```

and in the new Shell window:

```
2:-SETENV prompt %S>>
```

```
2:-NEWSHELL
```

and the new Shell window appears:

```
SYS:>
```

You can go on altering the prompt as much as you like!

A Fistful Of Dollars

For this example we have been treating variables more or less as constants – but there is no reason why you cannot use them in the script language too. To read a variable in a script you just prefix it with a dollar symbol, viz:

```
IF $name EQ "Mark"
```

where "name" is the name of an environmental variable. Similarly, you could test for a number like this:

```
IF VAL $count NOT EQ 10
```

Numeric values can be calculated using EVAL – but this involves writing to the variable directly. A subject discussed later in this chapter.

For A Few Dollars More

AmigaDOS pre-parses the command line and expands any environmental variables before passing it to the main code. In effect, this means you can specify a variable just about anywhere – and this means serious power!

In order to harness this power, all you have to do is tell AmigaDOS which parts of the command line are variables and which bits are normal parameters. As we have already shown with the prompt example above, this just means prefixing any variables with a dollar! Here are some trivial examples:

```
1>SETENV Author1 Mark
```

```
1>ECHO "Hello $Author1"
```

```
Hello Mark
```

```
1>SETENV Author2 Bruce
```

```
1>ECHO "Hello $Author2"
```

```
Hello Bruce
1>
1>SETENV here DF1:devs/keymaps
1>SETENV there DEVS:keymaps
1>COPY FROM $here TO $there
1>ECHO "$here and $there"
DF1:devs/keymaps and DEVS:keymaps
```

ENV:

Environmental variables created with SETENV are stored as text files on the Ram Disk. Actually, they are stored in the ENV: assignment – pathname 'RAM DISK:env'. AmigaDOS lets you see where ENV is pointing to at any time using the ASSIGN command. Try this:

```
1>ASSIGN ENV: EXISTS
```

```
ENV Ram Disk:ENV
```

Listing the contents of this directory displays the names of any environmental variables that have been created since you switched on:

```
1>DIR ENV:
```

```
prompt
```

```
Editor
```

You will have noticed there are far more files than those we have just created. Some of these are private to the IPREFS daemon and should be left well alone!

Since these files are stored on a real device, they can be written using ECHO, however, you should use SETENV and GETENV wherever possible. The environment handler also includes "private" environment variables. These are stored internally and accessed slightly differently. Now try this:

```
SETENV test XXXX
```

```
1>TYPE ENV:test
```

```
XXXX
```

```
1>
```

On a Workbench 2 machine the result is like this:

```
1>TYPE ENV:test
```

```
XXXX1>
```

This seems a little strange – the prompt seems to be stuck on the end of the variable contents! To see why, type the contents of the variable as a hex dump, like this:

```
1>TYPE ENV:test HEX
0000: 5858585858      XXXX
```

If you know your ASCII table, you will immediately notice the absence of 0A at the end of the dump – this is also called a line feed (LF). Environmental variables are text files – but don't have an LF at the end. Try this example to see what this would look like with a real text file:

```
1>ECHO >ENV:test "XXXX"
1>TYPE ENV:test
XXXX
1>TYPE ENV:test HEX
0000: 5858585858 0A  XXXX.
1>GETENV test
XXXX
1>
```

As you can see, GETENV blithely prints the contents of the variable including the terminating line feed – so you end up with a blank line. You can avoid this when writing environmental variables directly by using ECHO's NOLINE switch. In practice, it doesn't make much difference.

If you are reading them back – like the PROMPT example above – this method is better; because re-direction makes AmigaDOS swallow the last character in the variable (file). Let's take another look at that example and see how it works. It is worth going over again, since it's a tricky concept to grasp – yet an important one too.

Consulting *Mastering AmigaDOS 3 – Reference*, you will find that PROMPT does not have a command template – but it takes a new prompt string as an argument. What we need to do therefore, is supply the argument from a variable. This is simply:

```
1>PROMPT $variable
```

Neat, clean and easy – but what about applications that must run under AmigaDOS 1.3 too? The command string looks like this:

```
1>PROMPT <ENV:variable >NIL: ?
```

We know ENV: normally lives in the Ram Disk but, in theory, it could be anywhere. Using the ENV: logical assignment ensures we access the right directory. The filename is the name of the variable.

The clever bit is triggered by the "<" re-direction operator. This forces the command to read its line input from a file. This can be any file – but in this case it is an environmental variable. However, the command only reads its command line "interactively" if the "?" operator is specified. This also has the effect of triggering the command to display its help template – so we sink this to NIL: using ">NIL:".

- Backward compatibility is a serious issue – so be careful where you use AmigaDOS 2+ specifics. Consider the next example using ALIAS.

Writing Variables Directly

Remember the example above where we had to specify an extra character at the end of the line? Ignoring the fancy prompt, it read something like this:

```
1>SETENV prompt %S>>
```

The extra ">" was just absorbed when PROMPT executed. Writing this variable directly removes the need to add padding characters such as these. The simplest way to do this is with ECHO:

```
1>ECHO >ENV:prompt "%S>"
```

It would be even better to if you could use an ALIAS to do most of this for you. This would seem feasible:

```
1>ALIAS SETVAR ECHO >ENV:[]
```

Unfortunately, ALIAS does not work like this in release 1.3 – it writes a file called ENV:[]! If you intend using the prompt example above you might like to try this:

```
1>ALIAS SETPROMPT ECHO >ENV:prompt
```

and use it:

```
1>SETPROMPT %S.%N>
```

This does work – but does not accept spaces because ECHO treats them as extra options. This can work to your advantage if you want to suppress the extra character for some reason:

```
1>SETPROMPT %S.%N> NOLINE
```

The same idea can be incorporated into any number of directly set variables. For instance, you might want to change MORE's editor from time to time.

```
1>ALIAS SETEDITOR ECHO >ENV:editor
```

the use it like this:

```
1>SETEDITOR C:ED
```

Using Environmental Variables In Calculations

EVAL is the AmigaDOS calculator. On its own, this is not a lot of use – but the output can be written to a file; and that implies to an environmental variable! First create a variable containing any number. The value and name are arbitrary at this point so we'll use good old "fred".

```
1>SETENV FRED 1
```

Now let's imagine "fred" is an incrementing counter, that is we add one to him every time we execute the program. BASIC maths tells us to use:

```
PRINT FRED + 1
```

Or, take whatever is already in fred, add one to it and display the result. EVAL defaults to print results of calculations so we can do this:

```
1>EVAL $fred + 1
```

```
2
```

Cute but not a lot of use if you need to use fred as a counter. Going back to BASIC, we would use:

```
FRED=FRED + 1
```

Take whatever is already in fred, add one to that and put the result back in fred. Fred is a file in the ENV: assignment, so we have to read that variable, add one and write the FILE back. We can read environmental variables directly using \$var and EVAL has the TO keyword which makes it write directly to a file, so:

```
1>EVAL $fred + 1 TO ENV:fred
```

does the trick. You can add as many variables as you need to, thus:

```
FRED=FRED + MARK * SHARON + BARRY / BRUCE
```

could be expressed:

```
EVAL $fred + $mark * $sharon + $barry / $bruce TO ENV:fred
```

In AmigaDOS 1.3 though, EVAL cannot read variables in that way – nor can it write to a variable it is currently reading. This causes a few problems, but we can get round those. Remember the example above using PROMPT's interactive mode to read a variable? The same technique can be extended further to apply to EVAL. Lets try an example which just echoes the result to the screen. First the BASIC listing:

```
FRED=1
```

```
PRINT FRED+1
```

and now, AmigaDOS...

```
1>SETENV FRED 1
```

```
1>EVAL <ENV:fred >NIL: OP=+ VALUE2=1 TO=* ?
```

2

This might not seem very simple – but it is the only way to get AmigaDOS 1.3 to use an environmental variable in a calculation. Basically, AmigaDOS reads the file ENV:fred and adds one to it. Then it sends its output to the "*" file; the current console window!

In most cases it's better to send the result to a *proper* file and then deal with it from there. However, because of the way EVAL works in this release, it is not possible to write to the file whilst you are reading it. In other words, on the same command line. For this reason we must store the result in a temporary work file and write it back from there, viz:

```
1>EVAL <ENV:fred >NIL: OP=+ VALUE2=1 TO=ENV:tmp ?
```

```
1>COPY ENV:tmp TO ENV:fred
```

Sadly, you can only redirect CIS (with <) once in each command line so you can only read one variable at any one time. In practice, this is rarely a major problem.

Resetting Environmental Variables

In the unlikely event you want to remove an environmental variable completely, you can either give it a null definition or delete it from the ENV: assignment. You should use the special command UNSETENV. Here are some quickie examples:

```
SETENV FRED 1 ; give fred a value
```

```
UNSETENV FRED ; clear fred
```

or:

```
ENV:FRED ; obliterate fred
```

Whereas in AmigaDOS 1.3 this approach is used:

```
SETENV FRED 1 ; give fred a value
```

```
SETENV FRED ; clear fred
```

or:

```
ENV:FRED ; kill fred
```

Pro Bono Publico

This is where it gets complicated – we said this was a chicken and egg chapter; and here's another large screwdriver: AmigaDOS 2 and higher have two different types of environmental variable – LOCAL and GLOBAL. At first glance they look the same – but LOCAL variables are stored in memory and cannot be accessed through ENV:. Also, LOCAL variables can only be SET or READ by the process that created them.

<i>GLOBAL</i>	
EDITOR	Path and filename of MORE's editor
BRUTAB	Path and filename of BRU's configuration file
Kickstart	Kickstart version number
Workbench	Workbench version number
<i>LOCAL</i>	
Process	The number of this process
RC	Result code – used by IF
Result2	Secondary result code – used by WHY

Table 27.1. The LOCAL and GLOBAL environmental variables.

Up till now, we have been using the GLOBAL (system wide) definition command SETENV. This writes the variable to a file in ENV: as we have already shown. It can also be used to list all the current definitions. Typical values (with their owners) are shown in Table 27.1:

```
1>SETENV
EDITOR C:ED
PROMPT %N>
BRUTAB S:BRU-SETUP
Kickstart
Workbench
```

By using the alternative command, SET, we can define a LOCAL environmental variable that can ONLY be accessed by the process which created it. First open another Shell and type the following in either:

```
1>SET local 123
1>SETENV global 456
1>EVAL $local + $global
579
```

Now nip to the second Shell and do the EVAL again.

```
2>EVAL $local + $global
579
```

This works perfectly as local environmental variables are passed to child processes provided the child Shell is launched from the Shell holding the local variable in the first place and is *not* launched directly from the Workbench or any other Shell.

This is all true for version 2.04 and above but not for releases prior to this. If you try the example as detailed above on version 2.0 through to 2.03 for instance the result returned from the second Shell is 0. The result is incorrect and yet no warning is reported. Nevertheless the variable LOCAL has not been carried over to the second process. You can prove this by viewing them with GETENV (globals) and GET (locals):

```
2>GETENV global
```

```
456
```

```
2>GET local
```

```
Can't get local
```

AmigaDOS is reporting that it cannot find the variable you just defined because it is LOCAL, ie, belongs to the other process (Shell).

This can get rather confusing – especially because the \$ operator described above can read either type! In fact, \$var searches the LOCAL list *first*. If the variable was not there it searches the GLOBAL List. Note: If the variable is undefined, the text is displayed as typed.

The only way to discover more about environmental variables is to use them. However, as a parting gesture for this chapter, here is something to really put your brain through the wringer – indirection. Try this one for yourself:

```
1>SETENV indirect fred
```

```
1>SETENV fred Freeoww!
```

```
1>GETENV $indirect
```

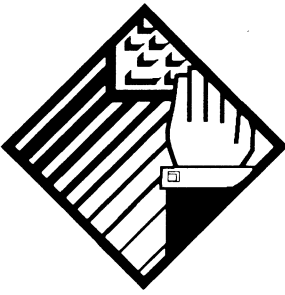
```
Freeoww!
```

```
1>SETENV $indirect Amazing!
```

```
1>GETENV fred
```

```
Amazing!
```

This may seem a strange idea but it can be very useful. For example, you can pass values between processes using a shared indirection variable, which points to the variable you want to access. Commodore use this technique to “pick up” the values of Workbench and Kickstart from the process locals written by VERSION in Workbench’s Startup-Sequence.



28: Devices

Overview

- > About devices
- > CONSOLE:
- > NIL:
- > PIPE:
- > SPEAK:
- > PRT: & PAR:
- > RAW:
- > AUX: & SER:

The Amiga supports numerous other devices which can be used under AmigaDOS and a complete list of these is given in Table 28.1.

Any of the devices listed can be used in much the same way as the ones we have already examined. For example, in file pathnames and in COPY commands.

An example that we haven't seen yet relates to the use of COPY with CON: so that it can be used as a means of displaying a text file on the screen in its own window. Try this example, which lists the Workbench Startup-Sequence file on screen in its very own window:

COPY S:Startup-Sequence to *

It all flashes past a bit quick but shows that it does indeed work. By the same token, you can reference the current window using '*' which is a pseudonym for the current Shell window. Strictly speaking all new code should use the pseudo device "CONSOLE:" but old (and easier) habits die hard.

Many of the devices mentioned here require to be MOUNTed before use and this is generally done as part of the Startup-Sequence file. The MOUNT command relies on information about how the device should be used and the handler to be used by the device. The handler which is specified is located in the L: assignment.

Specific knowledge of each device handler is not required to enable you to use the device in question. However, each of the devices and

the rules for MountList entries are discussed in *Mastering AmigaDOS 3 – Reference*.

<i>Name</i>	<i>Function</i>
*	Current console window
CONSOLE:	As * from 2.0.
AUX:	Un-buffered serial I/O
CON:	The console device
Dfn:	Floppy disk drive (n is drive number)
NIL:	The null device
PAR:	The parallel port
PIPE:	A pipe handler
PRT:	The printer (routed to SER: or PAR:)
RAD:	Recoverable RAM disk
RAM:	The RAM disk
RAW:	The raw console device
SER:	The serial device
SPEAK:	A speech handler. Not available in 3.0

Table 28.1. Amiga Devices.

CONSOLE:

This pseudo device can be used at any time you want to get something from the current console window or send something to it. Typically this option is used in scripts, where it is clearer than '*'. Input from CONSOLE: is terminated with <CTRL-\\>. See Chapter 29 for specific information on the console device, CON:

RAM: and RAD:

RAM: and RAD: are the Amiga's Ram Disks. See Appendices A and B for more information on these.

NIL:

This device is widely used when something has to be disposed of quietly. In fact NIL: is a dummy device in that it is used in places where you want data discarded. For example, it can be used with the '>' operator to stop information being displayed on screen.

In the Startup-Sequence file it has the form:

```
ENDCLI >NIL:
```

and this ensures that the Amiga's close down message does not appear. Try the following example for yourself:

LIST >NIL:

The LIST command actually takes place but nothing appears on-screen as it is sent to the NIL: device. Similarly, you can use this device with CMD in special cases to quietly disable the printer and prevent software getting hung up waiting for a non-existent printer to respond.

NIL: can also work as an input device in special cases. This can allow programs to be run in the background without allowing them to get a "lock" on the current window. This can stop RUN launched processes from preventing the Shell window from closing. This only works on programs which do not require interactive user input however. For example, to run a program called Sleepy completely in the background you would use the following:

RUN <NIL: >NIL: Sleepy

Note: although this prevents Sleepy from grabbing the console handles (more correctly called CIS – Console Input Stream, and COS – Console Output Stream) you can still pass command line options.

PIPE:

It is possible to channel or 'pipe' program output in various directions using the '>' operator. The example of NIL: above shows how the output from LIST can be piped to the NIL: device. The same operator could have been used to channel the screen output to a file. For example the following command:

LIST >ProgInfo

would send the output from LIST into a file called 'ProgInfo'. This could then be examined at your leisure with ED or EMACS.

The PIPE: device allows program output to be channelled more efficiently and allows output from one program to be used as the input for another. Unnamed pipes are not yet supported so it is not possible to use pipes directly as may be used in MS-DOS for instance. If you have never met this before, then don't let it worry you. The pipe handler was written supports a 2K asynchronous FIFO (First In First Out) buffer. It sounds a lot more complex than it is.

Before the PIPE: device can be used it must first be mounted with:

MOUNT PIPE:

This is done by the Workbench disk as part of its Startup-Sequence (provided the device has been moved to DEVS). The PIPE: device is an extremely useful one. For instance it can be used to channel the output of a program running in one Shell window to be displayed in another Shell window. To see this in action open two Shell windows. Into the first Shell window type:

MORE PIPE:Output

This effectively clears the Shell window for display and re-title it 'PIPE:Output'. Into the second Shell window type:

LIST >PIPE:Output

The LISTed output is displayed in the PIPE:Output window. Note: in release 1.3, the pipe name was case-sensitive. This bug was fixed for release 2 bringing it into line with the rest of AmigaDOS. It's worth bearing in mind however.

If the information being listed is too long to fit in the PIPE:Output window the prompt 'MORE' is displayed. Select the PIPE:Output window and then press the space bar to continue listing. Once the listing is completed the PIPE:Output window reverts to its former Shell guise.

The following command could be used to display the contents of the Startup-sequence file in the same named window: In the first Shell window type:

MORE PIPE:Output

In the second Shell window type:

COPY FROM S:Startup-Sequence TO PIPE:Output

This example also illustrates how PIPE: can be used to transfer information from one place to another. This can be particularly useful if you want to re-route large amounts of data using a temporary file held in the Amiga's memory and avoid the need to use either a slow floppy disk or the Ram Disk. You simply specify the PIPE: filename, ie, 'PIPE:tempfile' and it appears as a normal file to the applications. (Note: this trick assumes that the application does not attempt a Seek() which is not possible with a pipe. Seek() is only during random access filing applications like databases etc.).

PIPE: can also be used to route information through various devices and an interesting example of this is given later in this chapter under the description of SPEAK:

Finally you should note, PIPE: can get 'jammed' causing Shell to report "Shell error: unable to open re-direction file". If this happens, you should 'flush the pipe'. The fastest and safest way to do this is by using TYPE. Let's assume you have a PIPE:A full of useless information:

TYPE >NIL: PIPE:A

The idea is to channel all the unwanted information down the sink – the sink being NIL:. COPY does not work for this example in early releases of AmigaDOS 2, so we used TYPE instead!

A better way to use pipes is to multi-task them, several examples in *Mastering AmigaDOS 3 – Reference* use this feature. For now though,

we'll leave pipes with one last example – where the output from LIST is sorted using a pipe to speed and streamline the operation:

```
RUN LIST "" TO PIPE:A +
```

```
SORT PIPE:A CONSOLE:
```

SPEAK:

The SPEAK: device allows you to use the in-built ability of the Amiga to speak, using the speaker on the display unit (monitor/TV). However, this is only applicable to versions of Workbench prior to Workbench 3.0. The device can be mounted using:

MOUNT SPEAK:

In addition to its MountList entry, SPEAK: also requires the presence of the 'narrator.device' and 'translator.library' in the DEVS: and LIBS: directories respectively. These are all included on the Workbench 2.0x disks as standard.

From AmigaDOS the command SAY can make direct use of the SPEAK device. For example, ensure that the volume is fully set on the monitor and try the following:

```
SAY "Hello there I am your Amiga"
```

The full syntax of the SAY command is:

```
SAY [<options>] <text>|<filename>
```

A text file can also be redirected to SPEAK: by specifying a filename. For instance, try creating a few lines of text in ED and save it as "Speech" in the Ram Disk. This can be heard by using:

```
SAY -x RAM:Speech
```

Note that the option '-x' must be used to inform SAY that what follows is a filename and not actual text. A full list of SAY options can be found in Table 28.2.

<i>Option</i>	<i>Function</i>
-m	Uses male voice
-f	Uses female voice
-r	Uses robot (monotone) style voice
-p<num>	Set pitch (<num>=65 to 320)
-s<num>	Set speed (<num>=30-400)
-x	Filename follows not text

Table 28.2. The SAY and SPEAK Options.

These are best explored by trying them out. Where number parameters are required for pitch and speed, the higher the value supplied the higher the pitch or speed. Occasionally you will find

that what you have typed doesn't always sound as it should. In such cases you will need to invent a spelling that achieves a more phonetically correct pronunciation. For example try the following example:

SAY I am deaf

It will sound more like "I am deef". To make 'deaf' sound like deaf you will need to use:

SAY I am deff

An awful lot of fun can be had so beware! Here are a couple more examples:

SAY -s65 - r I am robert the rowbot

The SAY command provides an easy to use command line interface with SPEAK: and it may postfixed with a number of options which are detailed in Table 28.2.

These options are appended to the SPEAK: command using the OPT keyword and separating each option with a slash character. For example, you can listen to the contents of the S:Shell-Startup file by typing:

COPY FROM S:Shell-Startup TO SPEAK:

Using options a faster female voice could be employed with:

COPY FROM S:Shell-Startup TO SPEAK:OPT/f/s160

SPEAK: can be used in combination with other devices, for example PIPE:. In the following example, a Shell window is made into a PIPE: input window which outputs any incoming information to SPEAK:. A second Shell window is used to copy text information into the first Shell window. So open two Shell windows and into the first type:

COPY PIPE: talking TO SPEAK:

Into the second Shell window type:

COPY FROM S:Shell-Startup TO PIPE: talking

PRT:

This is a device that has been specially assigned for the printer no matter whether it is connected to the serial or parallel port. You should not normally send data directly to either the parallel or serial port as PRT: performs a number of tasks in addition to just redirecting the file to the correct port. It filters the correct printer codes for a start!

Printers are one of the biggest source of confusion. While they are relatively simple to operate, getting them to perform and use the various features can be problematic. The most important point is to select the right printer from Preferences and ensure the driver is in

DEVS:Printers. Once you've done this, the rest is, easy(ish). Printing text is the simplest job. This can be achieved in two main ways:

COPY FROM S:Shell-Startup to PRT:

or:

TYPE >PRT: S:Shell-Startup

PAR:

This device is linked directly with the socket (port) labelled with a printer on the rear of your Amiga. It is normally referred to as the parallel port. Note that you should not generally send data directly to PAR: if a printer is attached there. Only information that is specifically not intended for a printer should go to PAR:, for example to a piece of hardware attached to the parallel port which is not a printer.

SER:

This device communicates with the socket marked with the modem icon on the rear of your Amiga. A modem is a device that is normally associated with Comms – the transfer of data from one computer to another – or perhaps an attached MIDI interface. A text file could be copied to the serial port using:

COPY FROM <filename> TO SER:

Alternatively keyboard input could be sent directly to the serial port from the current window by using:

COPY FROM * TO SER:

Pressing <Ctrl-\> (Control backslash) aborts this operation.

Note, the SER: device has a RAM buffer which must be filled before either information is sent or received. This can give rise to some interesting problems which are beyond the scope of this discussion. For most purposes the AUX: device (below) is far more useful.

RAW:

This is a special type of window device that is used, for example, by ED. RAW: is primarily intended for developers who need to get raw keyboard and mouse information before AmigaDOS starts massaging it. It would be out of context to give a thorough discussion of RAW: at this stage, because it is so inherently complex.

Put simply what happens is this: When you open a RAW: window, all the line editing functions, normally handled by the console processor, are handed to you as a stream of characters – actually,

ANSI escape sequences. By sending information to RAW: it can even start sending information back! This is outside the realms of what can reasonably be handled by AmigaDOS at command level and therefore outside the coverage of Mastering AmigaDOS. Note however, the RAW: window swallows <Ctrl-\> (EOF) characters and cannot normally be closed down without re-booting – you have been warned. For the interested, here's one example of RAW: usage.

```
RUN TYPE RAW:0/0/400/50/RAW! OPT H
```

As you can see, TYPE must receive at least 16 characters before it generates any output whatsoever – this is an effect of buffering, incidentally. While this demonstration can be useful to discover the ASCII values of keystrokes – it can achieve little else at this level.

ENV:

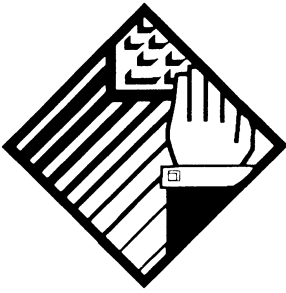
This pseudo-device is the 'environmental variable' assignment. It lives in the Ram Disk by default. Environmental variables are covered in Chapter 27.

AUX:

This device provides an un-buffered serial input and output using the serial port like a console in preference to the Amiga screen and keyboard. It could, for example, be used to allow input from another terminal connected to the serial port. This input could be captured and transferred to a file called 'InputAux' in the Ram Disk with:

```
COPY FROM AUX: TO RAM:InputAux
```

It turns out there is more to AUX: than meets the eye however. It can be used to form the basis of a simple multi-user terminal – where the Amiga plays host to another machine! Chapter 41 is dedicated to this.



29: The Console Device

Overview

- > Introducing CON:
- > CON: parameters
- > CON options
- > Console Copy & Paste
- > Using ANSI with CON:

Console windows are very widely used – every Shell for instance runs in a console window. You can use them in situations where you want to send the output from a command, say TYPE or LIST, to a separate window.

The console device takes arguments like a directory path, although the meaning is completely different. The basic synopsis for using this device is as follows:

```
CON:X/Y/Width/Height/Title/  
[Option1]/[Option2...]
```

Where:

X: is the number of pixels offset from the left of the screen. Must be a positive value between 0 and the maximum width of the screen (minus the width of the window).

Y: is the number of pixels offset from the top of the current screen. Must be a positive value between 0 and the maximum height of the screen (minus the height of the window).

Width: is the width of the window in pixels. (Must be more than 10 and, but does not have to be wide enough to fit the title. A value of -1 forces the window to open to the maximum width of the current screen. If very low values are used the window's output will be covered by the borders (unless the NOBORDER option is in effect).

Height: is the height of the window in pixels. (Must be more than 60. Remember that in most instances, 10 pixels are used for the title bar although this depends on the font used for window titles.)

Title: is a series of ASCII characters. You may not include ANSI escape sequences in this string. A title must be supplied if you intend using one or more of the option strings detailed below.

One of more of the following options may be applied:

SIMPLE: Refresh is handled by AmigaDOS rather than Intuition. This is the default when no options are specified. SIMPLE is mutually exclusive with SMART. For example:

```
1>NEWSHELL WINDOW=CON:0/0/-1/-1/Simple/SIMPLE
```

SMART: Console Copy & Paste is disabled and refresh events are handled by Intuition. This only affects the window when it's resized, so should be used with care. Such windows are about 20 per cent faster than a SIMPLE windows, but they hog extra CHIP memory. SMART is mutually exclusive with SIMPLE. For example:

```
1>NEWSHELL WINDOW=CON:0/0/-1/-1/NoCopy&Paste/SMART
```

CLOSE: Include a CLOSE box in the available window gadgets. Although this function is not necessary, it should be used with the WAIT option to allow users to back out without having to use <CTRL-\.>. For example:

```
1>COPY S:SPAT CON:0/0/-1/200/Typed/CLOSE/WAIT
```

WAIT: The window remains open until the user clicks the CLOSE box. Under normal circumstances, the console window closes when it receives EOF from the sending command. For example:

```
1>COPY CONSOLE: TO CON:0/0/-1/200/Typing/WAIT
```

AUTO: The window is not opened until required. This can be very useful if your application will hang around for some time until the window is required. For example:

```
1>RUN COPY PIPE:a TO CON:0/0/-1/-1/PIPE_Out/AUTO
```

NODRAG: Window dragging is disabled (the window cannot be moved by its title bar).

NOSIZE: The sizing gadget is not supplied. This option is useful if you have to display a fixed amount of pre-justified information and don't want the user fiddling with the window size.

NOBORDER: The window is rendered without borders. This option is most useful when used with **NOSIZE**. For example:

1>COPY CON:0/0/-1/200/Typing/NOBORDER TO CONSOLE:

BACKDROP: The window is rendered behind all others. This option is very useful when used on a blank custom screen (such as a public screen) where it gives a very professional appearance. It can also be used on a Workbench screen provided that no other windows are open – the disk icons are covered. For example:

1>COPY CONSOLE: TO CON:0/0/-1/-1/Backdrop/BACKDROP

WINDOW 0xAddress:

The window is opened on the screen pointed to by address. Officially documented but of little use to user programs which would be unable to determine the address of any screen in a given environment.

SCREEN "Name":

The window is opened on the named public screen. It is important to note that a public screen cannot close until all windows open on it are closed. At the time of writing very little software supports public screens.

Copy and Paste

The idea of Console Copy and paste is to perform simple editing operation such as might be found in modern wordprocessors. Operating the facility is a simple.

1. Select an area of text within a window by dragging the mouse over it. (Selections are displayed in reverse video.)
2. Press (Amiga-C) to copy the text to the internal clipboard.
3. Place the cursor where you want the text to go (in a Shell window this can be anywhere on the current command line) and press (Amiga-V).

A couple of caveats go with this facility. If a carriage return is copied, it causes a Shell line to execute immediately which may be undesirable. Also, you cannot COPY from a SMART console window. By default, ED and NEWSHELL open windows with SIMPLE refresh so this should not be a problem.

See *Mastering AmigaDOS 3 – Reference* on CONCLIP for more information.

The Console and ANSI

The console device supports a number of extended commands defined by the American National Standards Institute. ANSI controls are special sequences additional to the early standards defined by ASCII. Most of the commands start with a control sequence introducer (CSI) followed by one or more letters or numbers defining the effect. The CSI is actually 1B in hexadecimal but we can send the console a special sequence `"*e["` to represent this.

A list of the most important sequences appears in overleaf. Many of these codes can be sent using ECHO, incorporated into LFORMAT strings from EVAL and LIST, or even included in filenotes. It should be noted however, that some of these sequences cannot be entered directly using ECHO from the console because they cause some very strange effects. Most importantly, case is important in these sequences – the ASCII character is just a convenient carrier for what is really numeric data. A few control sequences are only available as numbers and should be sent by a variable constructed with EVAL. Such sequences may be useful in specially designed scripts which take control of the current Shell window however. Examples:

```
1>ECHO "*ec" ; clear the screen and reset modes
1>ECHO "Cursor OFF: *e[0 p" ; switch cursor off.
```

Mode switches for style, foreground and background modes can be mixed in a single string in any order, separated with semi-colons.

```
1>ECHO "*e[1;32;43m Mixed blessings..."
1>ECHO "*e[3;7m Arhum...*e[0m"
```

The following examples use EVAL to work with pure ASCII control codes:

```
1>EVAL >ENV:Beep 7 LFORMAT "%c" ; set up BELL
1>EVAL >ENV:Cls 12 LFORMAT "%c" ; set up Form feed
1>EVAL >ENV:Shift 14 LFORMAT "%c" ; alt characters

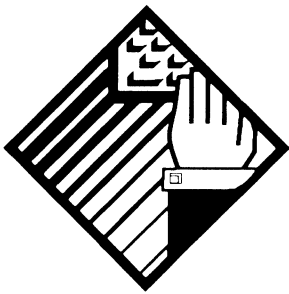
1>ECHO $Cls ; clear the screen
1>ECHO $Beep ; it's The f-f-Flash!
1>ECHO "Beep me $Beep now!"
1>ECHO "abcdefg $Shift abcdefg" ; worthless or useful?
```

You can even use the printer. Most models support the second example, although this is not required:

```
1>ECHO TO PRT: "$Cls" ; the printer ejects a page...
1>ECHO TO PRT: "$Beep" ; and will even beep at you!
```

07	Bell – flash the screen and sound the speaker.
08	Backspace – move cursor back one position.
09	Vertical tab – move cursor up one line.
10	Line feed. Move cursor down one line.
12	Form feed – clear screen.
13	Carriage return – move cursor to start of line
14	Set MSB of each character. (Print extended chars)
15	Clear MSB of each character.
*e[c	Clear window and turn all modes off.
*e[0m	All modes turned off.
*e[1m	Bold text enabled.
*e[3m	Italic text enabled.
*e[4m	Underlined text enabled.
*e[7m	Inverted (reverse video) text enabled.
*e[8m	Text becomes invisible (grey on grey).
*e[3xm	Text colour becomes colour x (0-7).
*e[4xm	Background colour becomes colour x (0-7).
*e[n@	Insert n spaces at cursor position.
*e[nA	Cursor up n lines. Default 1
*e[nB	Cursor down n lines. Default 1
*e[nC	Cursor forward n characters. Default 1
*e[nD	Cursor back n characters. Default 1
*e[nE	Cursor next n lines (to column 1). Default 1
*e[nD	Cursor back n characters. Default 1
*e[nH	Cursor to row n (Y position).
*e[;nH	Cursor to column n (X position).
*e[y;xH	Cursor to position X,Y.
*e[J	Erase from cursor to end of display.
*e[K	Erase from cursor to end of line.
*e[I	Insert line above line at cursor.
*e[M	Delete line at cursor.
*e[nS	Scroll up n lines.
*e[nT	Scroll down n lines.
*e[nt	Set page length to n lines (in current font).
*e[nu	Set line length in characters of current font.
*e[nx	Set left offset in characters.
*e[ny	Set top offset in lines.
*e[0 p	Disable cursor: note space between zero and p!
*e[p	Enable cursor. Note space!

Table 29.1. The ASCII/ANSI and Amiga console codes



30: ED The Screen Editor

ED is a text editor which provides some simple but effective word processor-like features. It is available on all releases of AmigaDOS. An alternative, more powerful, editor is available: this is called MicroEMACS and described later.

Both can be used to create text files and files of AmigaDOS commands called "scripts". This chapter examines the use of ED and shows how simple batch files can be created using it.

Creating a File

ED's basic syntax is:

ED <filename>

This means ED must have the name of a file to work with. The file does not have to exist, however.

For speed, make the ED command resident for ease of use and log onto the Ram Disk:

RESIDENT DFO:C/ED

CD RAM

Try the following from the Shell

1>ED TestED

ED loads and opens a large window on screen. In the bottom left hand corner of the screen the following message appears:

Creating new file

This creates a file called TestED on the Ram Disk.

By default the space allocated for this file is enough for 40,000 characters. However, should you wish to create a larger file then the use of the SIZE option can enable this. For example:

```
1>ED Test SIZE 60000
```

allocates space for 60,000 characters in the file.

The window for ED is much the same as those allocated to Shell windows. The program is effectively totally keyboard controlled, although you can make limited use of point and click and console Cut & Paste. Note also that, when an ED file is running, you can not return to the Shell window that launched it.

A Simple Script File

A script file can consist of any number of AmigaDOS commands arranged line after line. Type the following into the keyboard:

```
CD DF0:
```

```
DIR DIRS
```

ED uses CTRL and ESC key combinations to invoke the various commands at your disposal. To save this small file to the Ram Disk and then quit ED use:

```
<ESC-X>
```

When you press <ESC-X> the following appears in the bottom left hand corner:

```
*x
```

This is the *command display area* and '*x' represents <ESC-X>. While you're using ED, it's best to think of ESC as being the command key. To execute the command you now need to press <Return>.

Executing Script Files

EXECUTE carries out the task of executing the contents of a script file. It takes each line it encounters and sends them to AmigaDOS as though each line had been typed in at the keyboard. As AmigaDOS receives each line it interprets it and acts accordingly. The syntax is:

```
EXECUTE <filename>
```

Execute the TestED file by typing:

```
EXECUTE RAM:TestED
```

Both commands will be executed, providing you with a list of the directories stored on the disk in drive DF0: It also leaves you set with DF0: as the currently selected drive.

Editing ED Files

We can now make a simple addition to the TestED script file to ensure it returns to the Ram Disk after it has completed executing. Make the Ram Disk the current directory and then re-enter the command that was initially used to create the file TestED:

```
1>ED TestED
```

Whenever ED is asked to create a file it looks in the current directory (or the directory specified) to see if the named file exists. If it does it opens it and displays its contents. If no such file exists it creates one and signifies the fact by displaying the 'Creating new file' message. The cursor can be moved to the end of the file with the aid of the <down-arrow> key, at which point the 'CD RAM:' command can be added. The script file should now read:

```
CD DF0:
```

```
DIR DIRS
```

```
CD RAM:
```

This can be saved with <ESC-X> and then re-executed with:

```
1>EXECUTE TestED
```

When executed this switches to DF0: display the directories in the root directory of the disk and returns to the Ram Disk.

Echoing Status Messages

When a script file is executing, it is common practice for the file to display status messages to inform the user what is happening. This is what happens when you boot your Amiga which displays various copyright and status messages as the start-up process is carried out. These can be inserted into batch files quite simply by using the ECHO command. The syntax of the command is:

```
ECHO "<text to be printed>"
```

Here are a couple of examples that can be inserted into the TestED file after the first and second commands:

```
ECHO "Changed to DF0:"
```

```
ECHO "Returning to Ram Disk"
```

By default the ECHO command prints a <RETURN> character after it has echoed the message within the double quotes to the screen. There will be occasions when you don't wish this to happen. The NOLINE option stops this <RETURN> being printed. For example, to display a message followed by the date and time:

```
ECHO "Today's date and time is: " NOLINE
```

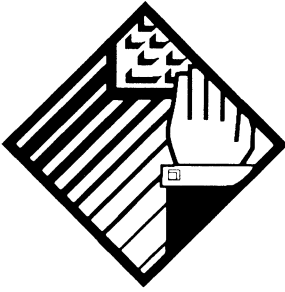
```
DATE
```

This could be added to the end of the TestED file to finish it off looking like this:

```
CD DF0:
ECHO "Changed to DF0:"
DIR DIRS
ECHO "Returning to RAM Disk"
CD RAM:
ECHO "Today's date and time is: " NOLINE
DATE
```

Save this with <ESC-X> and re-execute.

ED has many more features and these are described and listed in Appendix D.



31: Introduction to Scripts

Overview

- > What are scripts
- > Running scripts
- > Simple scripts
- > Script parameters

Script files have been mentioned occasionally in this volume – but we have not fully defined what they are yet. This is yet another one of those chicken and egg situations which is all unavoidable with most languages. AmigaDOS is not particularly small as languages go – with something like 140 command verbs to learn most with numerous confusing variations.

A script is just a collection of one or more AmigaDOS commands. The analogy is like a script for a play or film: a set of pre-determined instructions the actors play out. In AmigaDOS, scripts provide a method to customise functions, automate other functions and even add new commands – without fiddling with C or assembly language.

The simplest form of a script is the ALIAS command – a one line script capable of taking a single parameter. Scripts proper are much more powerful. There is no practical limit to the number of commands you can have in a script file, although the speed of AmigaDOS itself sets some restrictions.

Executing Scripts

Simple scripts can be created automatically using LIST with its LFORMAT parameter, but real scripts are typically more involved than this. By convention, scripts are stored in S: (the S directory on the boot disk) but this is not a requirement. Scripts themselves

are usually constructed using ED or MEMACS (see Appendices C and D) and the finished item has the "S" protection flag set thus avoiding the need for EXECUTE. For example, let's imagine you had just created a script called S:LOOPS, then it could be made into a command like this:

```
1>PROTECT S:LOOPS +S
```

and executed thus:

```
1>LOOPS
```

If the script is edited again however, you will need to set the 'S' flag again. For speed you may wish to add the following ALIAS to S:Shell-Startup:

```
ALIAS X EXECUTE
```

This conveniently allows the invocation of EXECUTE by entering a single letter, so the following two commands have the same effect:

```
1>EXECUTE LOOPS
```

```
1>X LOOPS
```

If the script has been programmed to accept parameters (discussed later) then these can be added as normal.

Some Simple Examples

As we have said, a script is a series of AmigaDOS commands. That suggests anything you can enter from the Shell prompt can be inserted in a script. The major advantage is once the script has been written and debugged, you'll always have it at hand. For the sake of these examples, we'll be using the Ram Disk, so enter this:

```
1>RAM:
```

Now we'll create a simple script with ED:

```
1>ED S:Hello
```

Enter the script like this:

```
ECHO "Hello" NOLINE
```

```
ECHO "World"
```

Save the file as described in Appendix C and execute your script:

```
1>EXECUTE Hello
```

```
Hello World
```

Nothing flash, but it works. Most scripts are a lot more involved than this. Here is a more useful example: one to provide a sorted listing of the current directory. Create the following script file as "RAM:SLIST":

```

ECHO "SortList... working..." NOLINE
LIST TO T:Temp
ECHO "  sorting"
SORT T:Temp TO T:Sorted
MORE T:Sorted
DELETE T:(Temp|Sorted) QUIET

```

Once saved, this can be executed thus:

```

1>EXECUTE SLIST
SortList... working... sorting

```

Such an example shows that many operations can be carried out in one easy lesson. Clearly, it would be much better if we could also send in a path and even better if we could multi-task such a script in several different Shells at the same time. In its current state, if two different invocations of this script were to run it would be possible for the temporary files to become corrupt.

Script File Parameters

AmigaDOS provides you with the ability to pass parameters to script files when they are executed from the command line. The parameters are included on the command line and obviously after the command name. For example, a script file called 'Test' might expect two parameters to be passed to it and could therefore be executed with:

```
EXECUTE TEST Today Tonight
```

or if you have set the script flag of the command:

```
TEST Today Tonight
```

Extending this capability to your script files is quite straightforward and requires the inclusion of 'key variables' at the start of the file. Key variables take the form:

```
.key <varONE>[,<varTWO>...]
```

The start of the script file must start with a '.key' – this is very important and there must be no space before it. This must then be followed with the names of the variables you wish to hold the information that is to be passed into it.

The term 'variable' is applied to special containers which can be used to hold information and you can have as many of these as you can fit on the line, each separated by a comma. The only rule to remember here is that the variable names must be positioned either side of the comma – there should not be any spaces, otherwise AmigaDOS will assume that there are no more variables to come, regardless of whether there are or not!

When you use variables in scripts, AmigaDOS copies a pre-processed file to T: in which all the variable substitutions have been made. This typically takes the form T:Command-00-01 where the numbers show the process calling EXECUTE and the nesting level if the script has been called recursively. You don't need any knowledge of this file, but you should know it exists and takes time to create. Therefore, if your script doesn't need variables, don't include dot commands.

In the example of TEST given above, the first variable in the list (typically varONE) is used to hold 'Today' and the second defined variable name (typically VarTWO) to hold 'Tonight'. Note that the variable names you use are arbitrary – use what you wish. These names do not change, hence what does change is the information that can be placed into them, thus the term variables. Try this following example:

Use ED and the Ram Disk to create a simple text file containing the following four lines:

```
.key name1, name2
ECHO "The first name was <name1>"
ECHO "The second name was <name2>"
ECHO "Hello <name1> and <name2>!"
```

The variable names are inserted into the program at the desired position and are identified as variables rather than commands or test by being enclosed within the chevron brackets '<' and '>'. As the example shows they can be used in the program more than once, in fact as many times as needed. Set the files script flag to allow AmigaDOS to execute it correctly:

```
PROTECT Test S ADD
```

and then run the program passing a couple of names to it en route separating each by a single space:

```
TEST Marc Sarah
```

The reply will come back as follows:

```
The first name was Marc
The second name was Sarah
Hello Marc and Sarah!
```

This principle can be applied to the SLIST script described earlier in order to supply a parameter for the path:

```
.KEY Path
ECHO "SortList... working..." NOLINE
LIST <Path> TO T:Temp
```

```

ECHO " sorting"
SORT T:Temp TO T:Sorted
MORE T:Sorted
DELETE T:(Temp|Sorted) QUIET

```

This modification might seem simple enough, but it gives the script a whole new aspect with the following synopsis:

```
SLIST [path|pattern]
```

These examples show a few possibilities:

```

1>EXECUTE SLIST
1>EXECUTE SLIST DFO:
1>EXECUTE SLIST RAM:T
1>EXECUTE SLIST C:E#?

```

The Unbalanced Key

The above examples are fine, but what if you run the program without passing any parameters? Well that depends on the program, but it will still run. Any information that is left out 'sets' the associated variables with a 'null string', which means just that – nothing!

An alternative way around this is to end a predetermined default using the '\$' character. For instance, in this example you might put default values in the following way:

```

.KEY Path
ECHO "SortList... working..." NOLINE
LIST <Path$SYS:> TO T:Temp
ECHO " sorting"
SORT T:Temp TO T:Sorted
MORE T:Sorted
DELETE T:(Temp|Sorted) QUIET

```

If the Path variable is not supplied, the text after the '\$' and up until the closing angular bracket will be used instead.

The other scenario that might arise is where too many items are passed as parameters for a command. In this case AmigaDOS generates an error message. For example using:

```
1>EXECUTE SLIST SYS: DFO:
```

In this example returns:

```
EXECUTE: Parameters unsuitable for key "Path"
```

The Key Options

Four options are available for use within a key definition with the same effects as those found in AmigaDOS:

<i>Option</i>	<i>Effect</i>
/A	Required argument (compulsory)
/K	Keyword (optional)
/F	Final. Must be the last argument on the line
/S	Switch. (See <i>Mastering AmigaDOS 3 Reference</i>)

Table 31.1. Key options available in scripts.

These options can be embedded into the key definition just like any normal command template, for example:

```
.key Path/A,Pattern/K
```

In the above example 'Path' is compulsory and must be supplied, but 'Pattern' is optional and may be ignored. To see how this works, we'll use a modified version of the SLIST script:

```
.key Path/A,Pattern/K
ECHO "SortList... working..." NOLINE
LIST <Path> TO T:Temp SUB=<Pattern|#?>
ECHO " sorting"
SORT T:Temp TO T:Sorted
MORE T:Sorted
DELETE T:(Temp|Sorted) QUIET
```

Executing this script thus:

```
1>EXECUTE SLIST
EXECUTE: Parameters unsuitable for key "Path/A, Pattern/K"
```

Create an error message because the compulsory parameter has not been included – the following do work:

```
1>EXECUTE SLIST SYS:#?
1>EXECUTE SLIST C:#? Pattern=BUF
```

You are now coming very close to creating a whole new AmigaDOS command – using nothing more complex than other commands.

Defining Key Variables

We saw above that if a value is not passed on the command line to any particular variable, then it can have a default used with the aid of the dollar operator, '\$'. This is fine but it has drawbacks. If you

wish to use the variable several times in the body of a program then you must also insert the default message at each point as appropriate.

Also, we have not discovered any mechanism for allowing variables to be defined within the body of the script file. The '.DEF' function overcomes this. The following short script (called MESSAGE) illustrates how it can be incorporated into the body of a program:

```
.KEY message
; define default message
.DEF message = "non existent!"
; now echo it to screen
ECHO "Message was <message>"
```

Now execute this:

```
EXECUTE MESSAGE
```

and the following will be printed on screen:

```
Message was non existent!
```

But if you supply a parameter on the command line as follows:

```
EXECUTE MESSAGE alright!
```

Then the following will be printed:

```
Message was alright!
```

The DEF function is intelligent, it tests to see if the .KEY variable is empty and if so assigns the new value. If the variable is not empty, ie, it had some information passed to it from the command line, it leaves it well alone.

Variables can be defined in programs using this method, however the variable must first be defined with .KEY whether you intend to pass it information or not. The following program, called CONSTANT, demonstrates how this would work:

```
.KEY one,two,three
; Now define constants
.DEF one="ONE (1)"
.DEF two="TWO (2)"
.DEF three="THREE (3)"
ECHO "<one>, <two>, <three>"
```

When executed this will produce:

```
ONE (1), TWO (2), THREE (3)
```

Changing the Notation

The use of the characters '<', '>', '\$' and '.' allows you to indicate special actions to AmigaDOS or to indicate embedded script commands. However, there are invariably times when you want to use these characters in your program text for whatever reason. This fact is recognised and it is possible for you to determine which characters are used for what. Table 31.2 lists the commands associated with each:

<i>Command</i>	<i>Changes</i>
.DOLLAR	Redefine '\$' character
.DOT	Redefine '.' character
.BRA	Redefine '<' character
.KET	Redefine '>' character

Table 31.2. Commands for redefining file parameters.

As an example AmigaDOS uses the the \$ character to indicate environmental variables and alternative choices. Alternatives – programmers call them OR choices – are sometimes indicated using the bar (|) character. We could use this in our programs thus:

```
.key name ; change $ to |
.DOLLAR |
; now try it out
ECHO "Hello there <name|no named person>!"
```

The character which is to become the new one is simply placed after the command keyword, though there should be a space between each. As a rule it isn't worth altering the '.' character as very few lines start with this! Besides it can create problems as it only comes into effect after the command has been executed, therefore any commands before this that might need it, for example, '.key' must have it! This can be a bit confusing. Remember also that new definitions only apply to the current program. If you then execute a second program the default will be used unless you redefine them.

Multi-Tasking in Scripts

The ability of the Amiga to multi-task means that scripts must also be able to multi-task. In other words your scripts should be "clean" and must not interfere with anything else. Any variables they create for their own use must be removed on exit, and any global variables or temporary files must be have unique names: this last one is a problem.

Or, at least it would be if AmigaDOS left it up to us to decide if any other scripts of the same name were already running. Every script you had would have to post a global flag to say it was running and others would have to either wait for it complete, work around it or rudely crash the system.

AmigaDOS provides an elegant solution. If a script is being pre-parsed (includes a dot-command such as .KEY) the process number is available in a special variable <\$\$>. It is also available as the local variable "\$Process" but the latter method is shorter and backwardly compatible too. You can tack this onto the names of any global variables (created with SETENV) or filenames. It doesn't matter what the eventual filename is, provided you append the <\$\$> to all occurrences. Here is SLIST modified so it will work correctly no matter how many processes are running it simultaneously:

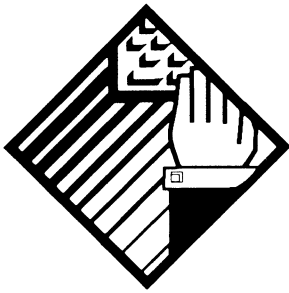
```
.key Path/A,Pattern/K
ECHO "SortList... working..." NOLINE
LIST <Path> TO T:Temp<$$> SUB=<Pattern|#?>
ECHO " sorting"
SORT T:Temp<$$> TO T:Sorted<$$>
MORE T:Sorted<$$>
DELETE T:(Temp<$$>|Sorted<$$>) QUIET
```

If this command were executed from Process number 1 with the following command line:

```
1>SLIST PATH=C:#?
```

The pre-parsed file (T:Command-00-01) would look like this:

```
.key Path/A,Pattern/K
ECHO "SortList... working..." NOLINE
LIST C:#? TO T:Temp1 SUB=#?
ECHO " sorting"
SORT T:Temp1 TO T:Sorted1
MORE T:Sorted1
DELETE T:(Temp1|Sorted1) QUIET
```

32: Flow Control in Scripts

So far, the AmigaDOS commands we have examined have only been capable of operating in a linear fashion. In other words, one after the other. However, AmigaDOS provides a number of commands which allow decisions to be taken, and to act upon these to move to a different point in a program. The commands that give it a 'decide and go' capability are detailed briefly in Table 32.1 overleaf.

SKIP and LAB

The SKIP and LAB commands are used in combination to allow you to move from point to point in a program. The syntax of the commands is:

```
SKIP <labelname>
```

```
LAB <labelname>
```

Where <labelname> is the name of the label you assign. Use ED to create this 'JUMP' file to see how it works:

```
ECHO "Amiga" NOLINE
```

```
SKIP todos
```

```
ECHO "Will this print?"
```

```
LAB todos
```

```
ECHO "DOS"
```

When this is executed the first line prints 'Amiga' on screen and NOLINE suppresses the carriage return (linefeed). At line two the SKIP command is encountered and this seeks out the corresponding LAB command which matches with the same label name. Control is transferred to this point and any commands between the SKIP and matching LAB are ignored. The command

after this then prints 'DOS' on screen to complete the word that we all love best.

<i>Command</i>	<i>Action</i>
IF	Test <condition> and do if true.
ELSE	Do this IF <condition> is false.
ENDIF	Finish test.
SKIP	Jump to labelled point.
LAB	Create Labelled point.

Table 32.1. AmigaDOS structure commands.

The BACK switch can be used to transfer control back towards the start of a program. The extended version of JUMP shows how this is implemented:

```
ECHO "Amiga" NOLINE
SKIP todos
;
LAB goback
ECHO "I wonder if this will print"
SKIP toend
;
LAB todos
ECHO "DOS"
SKIP goback BACK
;
LAB toend
```

It helps to insert blank comment lines before LAB commands – this breaks the program up into more manageable segments and should help you to locate and sort out any bugs.

Now every line in the program gets executed. After 'DOS' is printed, the program control SKIPS BACK to 'goback' and the following ECHO command is executed which last time was ignored. Finally the program SKIPS to 'toend' where there are no more commands to be executed. The inclusion of BACK is vital as it informs AmigaDOS about which direction to search for the corresponding LAB command. Note also the use of reasonably sensible label names – these should give some indication as to what is going to happen when program control gets there. This isn't to give AmigaDOS any clues or advanced warning but to help you and any subsequent onlookers to understand the program flow.

IF What?

The IF command allows you to check out whether a condition is true or false and then take action accordingly. For example, enter and save the following (use TESTIF as the filename):

```
.key CommandName
IF "<CommandName>" EQ "DIR"
    DIR
ENDIF
```

Now execute the program with:

```
EXECUTE TESTIF DIR
```

When the program starts the value 'DIR' will be assigned to the variable 'CommandName'. The IF statement is then encountered. EQ is one of several operators that can be used in conjunction with IF and it stands for Equals. Thus the second line of the program reads 'IF the value held in the variable CommandName Equals DIR execute all the commands that follow up until the command ENDIF is encountered'. In this case it's a simple DIR command. And this is exactly what happens – the Equals comparison is not case dependent, so it won't matter if you enter 'DIR' or 'dir' or any other combination, the effect is the same. The list of conditions that can be used within IF...ENDIF restraints are listed in Table 32.2.

<i>Option</i>	<i>Action</i>
NOT	Negates the condition under test
WARN	True if RC >=5
ERROR	True if RC >=10
FAIL	True if RC >=20
<x> EQ <y>	True if <x>=<y>
EXISTS <file>	True if named file exists
<x> GT <y>	True if <x> is greater than <y>
<x> GE <y>	True if <x> is greater than or equal to <y>
<x> NOT GT <y>	True if <x> is less than or equal to <y>
<x> NOT GE <y>	True if <x> is less than <y>
VAL	Test parameters as numbers

Table 32.2. IF optional tests.

The EXISTS option can be particularly useful, as you can use it to test for the presence of directories or files. The following example does just that, it tests to see if a file is present in the TO position and informs you of the fact:

```
.key in,out
IF "<out>" EXISTS
    ECHO "Overwriting old copy of <out>"
ENDIF
COPY FROM <in> TO <out>
```

For this to work correctly you must use full filename specifications for the destination which will include the filename, for example:

```
EXECUTE TESTIF2 FROM DFO:C/ED TO RAM:ED
```

This example illustrates the need in good programming practice to use full filenames whenever there may be a chance of ambiguity. Few of us do, as we all tend to get keyboard lazy.

ELSE What?

The ELSE statement is optional and can be included within the bounds of an IF...ENDIF statement to allow an alternative operation to that should the condition under test prove false. For example:

```
.key Name1
IF <Name1> EQ "AmigaDOS"
    ECHO "AmigaDOS was entered"
ELSE
    ECHO "You didn't type AmigaDOS." NOLINE
    ECHO "You used <Name1>."
ENDIF
```

If the parameter 'AmigaDOS' is passed to the file then the condition following IF will be true and therefore any commands following the condition up until the ELSE is encountered will be executed. If, on the other hand, the parameter passed is not 'AmigaDOS' then control is passed to the commands immediately following the ELSE statement and any more that may be included up to the ending ENDIF statement. Note that the keywords IF, ELSE and ENDIF must begin on new lines, although there is no limit to the number of IF...ENDIF structures that may be embedded within a file – subject to the normal memory limitations.

Error Numbers and WARN, ERROR and FAIL

You will have by now encountered several error messages. This isn't a slur on your programming ability but simply a fact of life – we all get things wrong from time to time. The most common reason for the generation of an error message is that the command has been typed in incorrectly or the command expected some parameters that weren't given. For example, if you simply type in ED without giving a filename as should be the case, it will return with the error message:

Last command failed because argument line invalid or too long

If you ever get an error message and it seems rather obscure then you can often get some additional information with the aid of WHY. The syntax of this command is simply:

WHY

Occasionally you don't get an error message at all, instead you get an error number – this is simply a number, normally three digits. Again WHY can often throw more light on exactly what caused the error. When in doubt, ask WHY.

When you set about writing your own AmigaDOS programs you need to bear in mind that they will, from time to time, exit because they have encountered an error. This is particularly true when they are expecting information that simply isn't available. As an example, enter this simple program using a suitable Filename, eg, 'ErrorTest':

```
.key var1
```

```
COPY FROM <var1> TO RAM:
```

```
ECHO "File <var1> copied to Ram Disk"
```

As you can see the file expects to have the name of the file to be copied passed to it as part of the command line used to execute the file. But what happens if no filename is passed to the program. If you try it by using:

```
EXECUTE ErrorTest
```

The following error message is displayed:

```
Can't open T0 for input - object not found
```

```
copy failed returncode 20
```

The same message is generated if you specify a command name which doesn't exist. Clearly, this is a situation that could happen and it would be one worth catering for. AmigaDOS allows this.

The key to it all is the 'returncode' specified by AmigaDOS. There are four returncodes and the number of the code indicates the level of the error. Table 32.3 lists the four returncodes.

<i>Returncode</i>	<i>Function</i>
0	Normal exit
5	Warnings
10	Normal errors
20	Fatal errors

Table 32.3. The AmigaDOS return codes.

The returncode issued by AmigaDOS on execution of the ErrorTest program was 20 – a fatal warning. The term fatal means that it has stopped the program from going any further by aborting it there and then.

Glance back to Table 32.2 and you will see that IF caters for these three error types in its options. This lets us test to see if they have occurred and act accordingly. However, there is one more fly in the ointment. We have to set the FAILAT level accordingly. FAILAT is an AmigaDOS command that allows you to determine what level returncode will stop AmigaDOS going any further. Normally the FAILAT level is 10 where normal errors as well as severe ones will stop the program from going any further. If we wished this limited to only severe, fatal errors, this threshold would have to be raised to 20 using:

FAILAT 20

in your program. However, in this example the returncode is 20 so, to 'trap' fatal errors, FAILAT must be raised beyond this:

FAILAT 21

Using a combination of SKIP and LAB we could re-write the 'ErrorTest' program to trap the error and act accordingly. Here is the revised program:

```
.key var1
;set FAILAT to trap bad filenames
FAILAT 21
COPY FROM <var1> TO RAM:
; now test if warning and jump to doerror if true
IF FAIL
    SKIP doerror
ENDIF
; otherwise carry on
ECHO "File <var1> copied to the RAM Disk"
; skip over error to end
```

```

SKIP end
; print error message
LAB doerror
ECHO "Aborted copy. " NOLINE
; determine rest of message
IF <var1> EQ ""
    ECHO "No Filename supplied."
ELSE
    ECHO "File <var1> not found."
ENDIF
;
LAB end

```

There a couple of things to note in the improved program. First, each new section of the program is preceded with a comment explaining what the aim of the section is. Try to get into the habit of doing this now as it will help you enormously when you come to debug or read your programs. Secondly, we have included a second IF...ENDIF structure in the 'doerror' routine. This tests to see if a filename was supplied, if not the message 'No filename supplied' is printed otherwise a message saying the named file was not found is given. This isn't vital to the operation of the program but it does tidy the program operation up and is good programming practice. If you try executing the program with:

```
EXECUTE ErrorTest
```

the following message is displayed:

```

Can't open T0 for input - object not found
Aborted copy. No filename supplied.

```

Alternatively if you specify a non-existent filename:

```
EXECUTE ErrorTest DF0:Silly
```

this message is given:

```

Can't open DF0:Silly for input - object not found
Aborted copy. File DF0:Silly not found.

```

A Better Example

SEARCH returns a 0 if the object sought is found, and a 5 (=WARN) otherwise. This can be used from within a script to test for a specific string, as the following example shows:

```

.key name/A,string/A,opt1,opt2
search "<name>" "<string>" quiet <opt1> <opt2>

```



```
if warn
    echo "Locate error: <string> not found in <name>"
endif
```

If you save this script as LOCATE then its usage is:

```
LOCATE NAME/A,STRING/A,FILE,ALL
```

The FILE switch is required when looking for a filename. NAME can be a directory, file or "" – something must be supplied here.

```
1>LOCATE S: .KEY ; to locate a string in a file
```

```
1>LOCATE "" MyFile FILE ; to locate a file in a directory
```

Asking Questions

The IF WARN set can be used in combination with the AmigaDOS command ASK to get Yes/No answers from the user. The syntax of the ASK command is:

```
ASK "<a question string>"
```

When ASK has displayed your question string on screen it waits until you press either the 'Y' or 'N' key (followed by <Return>). If the 'Y' key is pressed, then a warning condition is set (returncode 5) which can then be tested with IF. Now try this following short program, called 'AskTest', to see for yourself:

```
; ASK demo >>ASKTest
ASK "Do you like me (Y or N)?"
IF WARN
    ECHO "And I like you as well!"
ELSE
    ECHO "No! You have upset me now."
ENDIF
```

Pressing 'Y' or 'N' and then <RETURN> (or 'y' or 'n' as the command is not case sensitive) responds with the appropriate message. Pressing any other key will simply make the ASK prompt be repeated.

Numbers With IF

So far the use of IF has been to work on information assuming that it was a string. (The term 'string' is used to signify a 'string of characters', in other words anything typed in at the keyboard and ended by a <Return>.) However, you will occasionally want to use IF to manipulate numeric values. For instance typing in '55' at the keyboard will pass it into your program as the two alphanumeric characters '5' and '5'. It will be as though you had typed in 'XY' – characters not numbers. There is nothing we can do about this at

the input stage but the VAL option can be used to convert a string into a number for testing. Try this following short program which is called 'TestVAL':

```
.key num1
;test to see if num1>10
IF VAL <num1> GT 10
    ECHO "Number was greater than 10"
ELSE
    ECHO "Number was less than or equal to 10"
ENDIF
```

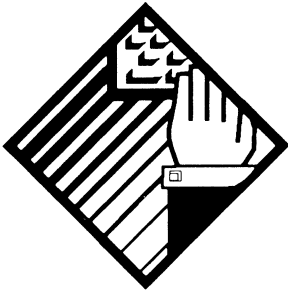
The VAL operator comes immediately after the IF and there must be a space prior to the variable name. Execute this with:

```
EXECUTE TestVAL 5
```

and the response is:

```
Number was less than or equal to 10
```

The GT operator was used to test for Greater Than, the various other 'numeric operators' detailed in Table 32.1 can also be used in combination with VAL, you might like to try them out for yourself.



33: Workbench and IconX

Overview

- > Workbench and Scripts
- > How IconX helps
- > IconX windows

Workbench and Scripts

Being able to run a program from the Workbench requires that it has an icon to represent it. As should be apparent from your understanding of the Workbench disk there are a lot more directories and files on it than are displayed in the Workbench disk window when it is opened.

The key to an icon display are the files labelled '.info'. To see this, use the Shell to duplicate and rename an .info file. For example, the one associated with the System directory on the Workbench disk.

```
1>COPY SYS:System.info TO RAM:
```

```
1>RAM:
```

```
1>RENAME System.info New.info
```

Opening the Ram Disk window will reveal that the new drawer is indeed present. Trying to open the new drawer window has no effect, however, as it has nothing to open.

ICONX

Although it is located in the C directory ICONX isn't an AmigaDOS command as such, it is a Tool which can be used to assist in the running and managing of scripts. In this instance ICONX provides the means to allow your own script files to be run and executed from the Workbench by clicking on the icon that you have chosen to associate them with.

The icon .info file that you use in this case is important, as it must allow you to define the default tool used by the program it is to

run. This information can be found in the Info box associated with the icon. But don't bother looking too much – the handiest one to use is the Shell icon.

First create a simple script file that can be used as the worked example. As always, whenever trying out a new technique or idea keep everything simple to ensure that you don't make any errors. Then once you have mastered the particular technique you can go for it! Use ED to create the following file as Tester:

```
ECHO "This is Tester - the DIY Workbench Script File"
```

```
DIR DIRS
```

```
ECHO "The End - Returning to the Workbench"
```

Copy the Shell.info file across from the Workbench disk to the RAM Disk and rename this Tester.info:

```
COPY FROM DFO:Shell.info TO RAM:Tester.info
```

Open the Ram Disk window and you will find that the standard Shell icon is in position but is called Tester. Select this by clicking on it once only and open the Info box available from the Workbench menu. The Default Tool is set to:

```
SYS:System/CLI
```

Click in this box to insert the cursor, delete the line and type this:

```
C:ICONX
```

Select Save and then run the Tester script file by double-clicking on the newly created icon. The ICONX window is opened on screen and the script file will be executed within it.

The script file has run its course inside a standard Iconx window. However, it is generally much neater to specify your own window as this can then be made of the correct size to suit your requirements, be positioned on the screen at the point you wish and have the title of your choice. This can be done, but you will first need to call up the icons Info box as described above.

The key is the TOOL TYPES gadget near the base of the info box. Click on the ADD gadget and a cursor appears. Now type the following line into it:

```
WINDOW=CON:100/100/425/100/Tester
```

Note that it is important that the first word 'WINDOW=' is entered in capital letters otherwise it doesn't work. What you have done here is to inform the Iconx handler that you wish to use your own window details. Save this and then double click on the Tester icon. Now Iconx runs the Tester script file inside the Shell just defined, roughly central on the screen.

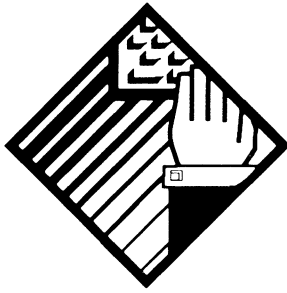
You can add a second TOOLTYPE command should you so wish. The command DELAY allows you to determine how long the window stays open on screen. For example, to set a delay of ten seconds from completing the script file to closing the window, add the following to the TOOLTYPES:

DELAY=600

DELAY works in units of 1/50th of a second, so this causes a delay of ten seconds. If you prefer you can enter:

DELAY=0

and then the window remains open until you press <CTRL-C>. It is not possible to pass parameters directly to scripts started by ICONX. There is a way around this – but this involves using recursion which is not covered here.



34: Evaluating and Manipulating

Overview

- > What is EVAL?
- > Simple EVAL usage
- > EVAL operators
- > EVAL I/O conversions
- > EVAL in scripts

We have seen that the VAL option of IF can be used to test for a numeric value rather than a string. However this has rather severe limitations in that it does not, for instance, allow a numeric value to be added to or subtracted from, to name but two operations.

In a nutshell EVAL allows you to calculate simple mathematical expressions using a combination of parameters or environmental variables. The basic syntax of the command is:

```
EVAL <expression> [TO <file>]
```

In its simplest form EVAL just takes two values supplied to it and performs the operation specified by the expression operation. For example:

```
EVAL 2 * 5
```

multiplies 2 by 5, printing the result on screen. In a similar fashion:

```
EVAL 4*5
```

multiplies 4 by 5. The multiplication sign is represented by the asterisk and this is called the 'operator'. We can use this operator to create a simple program called 'MULTIPLY':

```
.key num1, num2 ; read two  
numbers and evaluate  
ECHO "<num1> times <num2> is  
" NOLINE  
EVAL <num1> * <num2>
```

Executing this with:

```
EXECUTE MULTIPLY 2 5
```

gives:

```
2 times 5 is 10
```


Note that in AmigaDOS 1.3. there is a bug which makes it imperative that you insert a single space between the numbers and operators to get the right result. If you don't, the bug creates havoc – the results are very unpredictable.

This type of program can be used with a variety of EVAL operators which are listed in Table 34.1.

<i>Operation</i>	<i>Function</i>
+	Addition
-	Subtraction
*	Multiplication
/	Division integer
mod	Division remainder
&	Logical AND
	Logical OR
~	Logical NOT
<<	Shift left bitwise
XOR	Exclusive OR
EQV	Bitwise equivalence
(and)	Parenthesis - 1.3.2 upwards

Table 34.1. The EVAL operators.

Several of these will need some explanation. Arithmetic with EVAL is performed in integers, ie, answers are not given as fractions or as decimal points. Therefore:

EVAL 7 / 2

returns 3 with the remainder of 1 being 'lost'. The 'mod' (modulo) operator can be used to pick this up and display it:

EVAL 7 mod 2

returns 1.

The &, | and ~ characters act on the numbers at their binary or bit level. If you are unsure as to what binary numbers are and how they can be manipulated you may want to skip this next bit as you probably won't be very interested in these operations at the moment – come back to it later.

For example:

1>EVAL 128 & 129

128

At bit level this would be:

$$\begin{array}{r}
 10000000 \\
 10000001 \text{ AND} \\
 \hline
 = 10000000
 \end{array}$$

1>EVAL 128 | 3

131

At bit level this would be:

$$\begin{array}{r}
 10000000 \\
 00000011 \text{ OR} \\
 \hline
 = 10000011
 \end{array}$$

The << operator is used to shift bits left a specified number of times. For example:

1>EVAL 2 << 10

2048

Note that each move left effectively performs a multiplication by two, ie, $2*2*2*2*2*2*2*2*2*2=2048$. Similarly:

1>EVAL 5 << 6

320

Moves the bits of the number 5 left six times as though it had multiplied 5 by 2 six times:

$$5*2*2*2*2*2*2=320$$

FORMATED Input and Output

There is more to EVAL than meets the casual eye – as you will learn by consulting *Mastering AmigaDOS 3 – Reference*. One of the most important functions is its optional LFORMAT string which works in a similar fashion to that supplied by LIST.

The major difference is where LIST inserts substrings of object names and so on, EVAL inserts the result from a calculation (zero or more times according to a formatting string). This might not seem very spectacular, but some useful conversions are possible.

First though, input formatting. By default EVAL sees numbers as decimal integers – but it can read octal, hexadecimal and even single characters. Although number conversions can be supplied in calculations, they are described here in isolation for clarity.

Prefix	Indicates
'	Character
0 or #	Octal (base 8)
0x or #x	Hexadecimal (Base 16)

Table 34.2. EVAL's Input qualifiers.

Using these we can show:

```
1>EVAL '@ ; value of character "@"
```

```
64
```

```
1>EVAL 0xFF ; convert FF (hexadecimal) to decimal
```

```
255
```

```
1>EVAL #77 ; convert 77(octal) to decimal
```

```
63
```

Output formatting is available to perform the conversions in reverse – especially useful if you need a strange character code to control the console device. The LFORMAT keyword is used with the following conversions:

<i>Qualifier</i>	<i>Indicates</i>
%C	Character
%N	Decimal
%On	Octal with n=1-9 digits of precision
%Xn	Hexadecimal with n=1-9 digits of precision

Table 34.3. EVAL's output (LFORMAT) qualifiers.

The 'postfix' precision qualifiers are a little confusing if you are used to a more conventional 'infix' approach adopted by the C language or even LIST, but there you have it. It is also important to note the precision *must* be supplied in hex and octal conversions or EVAL will assume the default value of 1.

The LFORMAT string does not automatically supply a newline. This may seem odd, but it is to maintain compatibility with the global environmental variables. You can include normal escape sequences for the console device though. Here are some examples:

```
1>EVAL 255 LFORMAT "%N in hex is %X2*n"
```

```
255 in hex is FF
```

```
1>EVAL 64 LFORMAT "The ASCII character %N is *"%C**"n"
```

```
The ASCII character 64 is "@"
```

```
1>EVAL 63 LFORMAT "%N in octal is %O2*n"
```

```
63 in octal 77
```

```
1>EVAL 7 LFORMAT "%C" TO ENV:Bell ; init variable
```

```
1>ECHO "Ring my $Bell <- Flash, huh?" ; and use it!
```

```
Ring my <- Flash, huh?
```

EVAL in Scripts

A useful by-product of EVAL is that it allows us to combine its use with environmental variables to perform loops. A loop is simply that. Consider the following short program:

```
ECHO "Loop #1"
ECHO "Loop #2"
ECHO "Loop #3"
ECHO "Loop #4"
ECHO "Loop #5"
```

When run it should be obvious that it will print out 'Loop #' with the appropriate number from 1 to 5. This is okay, but is rather wasteful, consider the length of the program that would be required to print the numbers 1 to 20 on the screen. A loop allows us to program this much more efficiently. In English, and with a little bit of algebra, we can write a pseudo-program to do this along these lines:

```
startnumber=1
ECHO startnumber
add one to startnumber
is startnumber greater than 20?
if not then go back to the line starting with ECHO
otherwise all done
```

This can be translated into the following AmigaDOS program called 'UpCount' which does much the same thing. The program looks rather complicated but is relatively straightforward. Type it in and try it for yourself, then we'll examine it. The program is not commented at this point and, for speed, only counts to 5.

```
ECHO >env:count 1
LAB start
ECHO "Loop #" NOLINE
TYPE env:count
EVAL <env:count >NIL: T0=t:qwe<$$> value2=1 op=+ ?
TYPE >env:count t:qwe<$$>
IF VAL $count NOT GT 5
    SKIP start BACK
ENDIF
ECHO "All done"
```

All being well, when this is run it should display on the screen:

Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
All done

The program is rather trickier than those listed before so if you get any untoward error message check the appropriate point. In particular please note that on the fifth line of text there is a space between the '+' and '?' characters right at the end. Let's examine each line one by one.

ECHO >env:count 1

The loop value starts at 1 and this line prints the value '1' to an environmental variable called 'count'. Being an environmental variable it should be stored in ENV:.

LAB start

This line marks the re-entry point for a forthcoming SKIP command.

ECHO "Loop #" NOLINE

Print 'Loop #' on the screen and suppress the linefeed.

TYPE env:count

Print the current value held in the environmental variable called 'count'. This is printed after 'Loop #' and moves the cursor to the next line.

EVAL <env:count >NIL: T0=t:qwe<\$\$> value2=1 op=+ ?

This line adds one to the value of 'count'. It reads in, and EVALs the value of count and then writes it to a temporary pipe file called 'qwe<\$\$>'. The <\$\$> is used to define the current Shell number and this is important as it ties the file to the Shell from which it was generated. For example, if this program is being run from Shell process number 1, then the qwe file created will be called 'qwe1'. 'value2' is assigned the value of one and this is the increment value. Finally, the 'op+?' defines the operation to be performed, + (addition) in this case. So the contents of 'value2' are added to the contents of the temporary variable.

TYPE >env:count t:qwe<\$\$>

This line writes the newly incremented value held in 'qwe<\$\$>' and writes in to 'count' thereby updating its value for future reference.

IF VAL \$count NOT GT 5

This line tests to see if the VALue of 'count' is NOT Greater Than 5. Note that the \$ prefix on count is a shorthand way of specifying an environmental variable.

SKIP start BACK

If the IF test is true then the program control skips back to the label called 'start' defined earlier and the whole process repeats.

ENDIF

ECHO "All done"

Otherwise the program has run its course and ends with an 'All done' message.

You may need to read through the above example a few times, and experiment a bit for yourself. Listed below are a few more example loop programs that are documented and should give you a further insight into performing loops within AmigaDOS.

The above example works under all versions of AmigaDOS. However, incrementing is a much simpler proposition from AmigaDOS 2 onwards. As such the previous script can be written more eloquently thus:

```
; AmigaDOS 2/3 Example  
SETENV Loop 1  
LAB start  
ECHO "Loop #" NOLINE  
GETENV Loop  
EVAL $Loop+1 TO ENV:LOOP  
IF VAL $Loop NOT GT 5  
    SKIP start BACK  
ENDIF  
ECHO "All Done"
```

The following example scripts show how variations on loops both incrementing and decrementing can be written for AmigaDOS 3 and AmigaDOS 2. In addition the AmigaDOS 1.3 version is given for backwards compatibility.

Program: LOOPER

Function: Perform a simple decrementing loop to 0

Input: Must specify loop start count number >0

Use: EXECUTE LOOPER 5

Notes: Here you specify the start for the loop counter. The operator value is '-' as this performs a negative count, ie, it subtracts the value of 'value2'.

AmigaDOS 3/2 Version:

```
.key loop/a
.bra {
.ket }
SETENV Loop {loop}
ECHO "This program will loop $loop times"
LAB start
ECHO "Loop #" NOLINE
GETENV Loop
EVAL $Loop-1 T0=env:Loop
;
IF VAL $Loop GT 0
    SKIP start BACK
ENDIF
```

AmigaDOS 1.3 Version:

```
.key loop/a
;demo using eval to perform a simple loop
;reset < and > to use { and }
.bra {
.ket }
;setup loop count and inform
ECHO >env:loop {loop}
ECHO "This program will loop {loop} times"

;mark start of main loop structure
LAB start
;echo current loop count
ECHO "Loop #" NOLINE
TYPE env:loop
```

```

;evaluate and decrement loop count
EVAL <env:loop >NIL: T0=t:qwe{$$} value2=1 op=- ?
TYPE >env:loop t:qwe{$$}
;Evaluate and if greater than 0 do again
IF VAL $loop GT 0
SKIP start BACK
ENDIF
;else inform all done
ECHO "All done"

```

Program: DUAL

Function: Perform a dual (two) decrementing loop to 0

Input: Must specify loop start count number >0

Use: EXECUTE DUAL 6

Notes: Here you specify the start for the loop counter. The operator value is - as this performs a negative count, ie, it subtracts the value of 'value2' which is 2. Given an input value of 6, the program will output loop numbers of 6, 4, 2 and 0. Use the previous script listings changing the EVAL lines as follows:

AmigaDOS 3/2 Version:

```
EVAL $Loop-2 T0=env:Loop
```

AmigaDOS 1.3 Version:

```
EVAL <env:loop >NIL: T0=t:qwe{$$} value2=2 op=- ?
```

Program: MULT

Function: Perform a multiply by two incrementing loop to the value specified.

Input: Must specify loop start count number >0 and a final end value

Use: EXECUTE MULT 2 64

Notes: Here you specify the start for the loop counter and also the end value.. The operator value is * as this performs a multiplication by 2 (the value of value2). Given the above input the program displays loop values of 2, 4, 8, 16, 32 and 64.

AmigaDOS 3/2 Version:

```

.key start/a,end/a
.bra {
.ket }

```

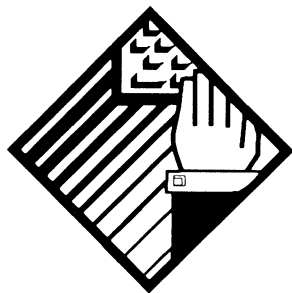


```
SETENV Loop {start}
LAB start
ECHO "Loop #" NOLINE
GETENV Loop
EVAL $Loop*2 T0=env:Loop
;

IF VAL $Loop NOT GT {end}
    SKIP start BACK
ENDIF
```

AmigaDOS 1.3 Version:

```
.key start/a,end/a
;demo using eval to perform a single
;multiplication by 2 loop based on two variables
;start is start value, end is final count
;reset < and > to use { and }
.bra {
.ket }
;setup loop count and inform
ECHO >env:start {start}
;mark start of main loop structure
LAB start
;echo current loop count
ECHO "Loop#" NOLINE
TYPE env:start
;evaluate and multiple current value by 2
EVAL <env:start >NIL: T0=t:qwe{$$} value2=2 op=* ?
TYPE >env:start t:qwe{$$}
;Evaluate and if not greater than {end} loop back
IF VAL $start NOT GT {end}
    SKIP start BACK
ENDIF
;else inform all done
ECHO "All done"
```



35: Startup- Sequences: An Introduction

As you progress deeper into AmigaDOS, you will inevitably start to wonder precisely what goes on in the Startup-Sequence. New users are often taken aback at the length of time the Amiga takes to get the Workbench ready. You switch on, insert a disk and wait... and you wait, and wait, and wait.

The disk finally stops grinding away and the Workbench screen appears. But what has been going on during the intervening period? That, of course, is the subject of this chapter. In subsequent chapters we'll examine the Startup-Sequences for AmigaDOS 2.04, 2.1 and 3 in depth.

When you first switch on (or reset) the Amiga polls drive DF0: to check for a disk. If it doesn't find one it displays its startup message asking you to insert a disk. The latest Kickstart displays an attractive animation of a disk inserting itself – very quaint.

When you insert a disk the Amiga attempts to boot from it. This requires a special boot routine located at track zero, sectors zero and one. Making a boot disk requires the use of the INSTALL command which is covered in detail in *Mastering AmigaDOS 3 Reference*.

If the boot program is missing, the Amiga complains by throwing up the "insert a boot disk" animation. If a boot sector program is found and verified, the machine loads and blithely executes the program. That, incidentally, is how 'Limpet'

viruses get into the machine. This causes the machine to start its internal boot procedure and assign the default directories to the boot disk.

Next, it looks for a file in DEVS: called System-Configuration, loads that and sets up the screen colours, pointer style and so on. These are the settings set by the Workbench "preferences" tool(s). If the file does not exist, the machine uses its default ROM settings.

More or less at this point the AmigaDOS screen appears. However, even though a fair amount has happened, the poor user has very little control over the chain of events. At this point the machine is technically ready for use, but it continues by looking for a file in the boot disk's S directory (S:) called Startup-Sequence. Then it loads and executes that too! This is what takes the time – leading many people to think it's a slow machine. This is far from the case, as we shall soon see .

What follows is a more prosaic way of describing some of the technicalities of booting, serious users should skip to the next subsection. Right, so you switch the machine on and insert a disk.

Amiga: "Yo, DF0:!! You got a disk?"

DF0: "Sure have dude!"

Amiga: "Ok go to track zero and let's see what yer got..."

FX: Grind

DF0: "Eat sectors sucker"

Amiga: [checks boot sector checksum, starts executing boot program] "Chocks away. DF0: Go to track 39."

FX: Scrunch

Amiga: [Reads directory and locates devs/system-configuration at track 76]. "DF0: Go to track 76."

FX: Grind

Amiga: [Reads system-configuration and sets preferences. Screen becomes visible.] "DF0: go to track 39"

FX: Scrunch

Amiga: [Reads Startup-Sequence and *may* attempt to write temporary workfile T:Command-00-01.] The boot we all know and love starts now.

FX: Skwarrck, buzz, grind etc.

Now you know a little more about what the Amiga is up to when the initial boot sequence starts. This is simplified, but most of the AmigaDOS technicalities have been outlined, if not explained. The part that concerns you is the Startup-Sequence file – the sequence of commands executed to customise the machine. In Amiga terms, booting without a Startup-Sequence is akin to putting on your shoes without putting your trousers on first; or perhaps, taking a kick at a

football which isn't there. The effect is much the same. It can be done, but it doesn't achieve much. An old Chinese proverb states, "Man wearing shoes and no trousers, is looking to get arrested."

Starting The Engine

The Startup-Sequence is a script, so beginners who have bypassed the three chapters on script files should give at least the first a cursory glance to familiarise themselves with some of the concepts.

Booting itself occurs at several levels, but we won't be concerning ourselves with the hardware side of things here. The term "boot", comes from the expression "pulling yourself up by your own bootlaces [bootstraps]" and applies more to electronic feedback loops than it does to computers; maybe that's what gave rise to the term Kickstart. Kick after all, is synonymous with boot [slang].

Quite simply the Startup-Sequence is just like any other AmigaDOS script, with one important difference: it is automatically executed without any need for user intervention. Masochists could, if they desired, rename the startup-Sequence to something else and execute it manually every time the machine started. This may seem daft, but it's worth knowing because, even if you destroy a Startup-Sequence, the Amiga needn't be totally crippled.

More to the point, Startup-Sequences help to make the Amiga into the machine it is – powerful and adaptable. For this reason we have crammed as many examples into this chapter as we could – but the list of potential possibilities could fill a book. (Now there's an idea!)

Guru's Meditation

Before progressing any further with this chapter, here's how to save yourself some agony and embarrassment. We urge you to do this now and keep the example in mind before tinkering with the proper Startup-Sequence. Always, always work on a copy of the Workbench disk – never the original. This applies to hard disk users too.

- Open a Shell window if you haven't already done so.
- At the prompt type the following:

COPY S:Startup-Sequence SYS:HELP

- and press <Return>

This creates a duplicate startup script in the root directory of your Workbench disk. The name "HELP" was chosen because it's easy to remember. You can call the file anything you want: SUS, START, FRED – anything, provided you remember what it's called. The reasoning behind this may not seem obvious yet, but when you start tinkering with startup files you'll make mistakes. When you do (and believe us, you will folks) it may prove difficult, or even

impossible, to get the machine to make much sense. Imagine if the first command in the script causes a Guru! This way, when something goes wrong – don't reach for the Valium – just refer to the following instructions:

- Reset the machine
- Press and HOLD the key marked CTRL
- Press and HOLD the key marked D
- Wait until the Amiga displays the message:

***** Break**

- Release both keys. Relax.
- At the prompt type:

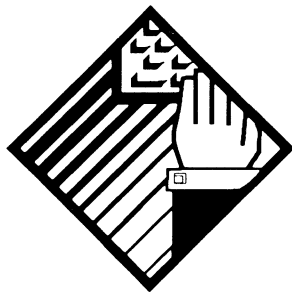
EXECUTE HELP

and press <Return>. Now the machine should boot as normal.

You may experience some curious “requesters” when you try this, but anything is better than no Workbench at all!

Ketchup on your Startup

Now we've dispensed with the rudiments of kitchen safety, we can discuss the ingredients of a Startup-Sequence. This analogy may seem a little detached, but a startup is just like a good curry – more or less a few essential ingredients with the rest being personal taste. This analogy can be extended further, because even the quantities of those essentials can be tailored for individual needs. **ADDBUFFERS**, for instance, is the command used to allocate much needed sector buffers to the floppy drives. This command usually appears early on in the Startup-Sequence but the actual number of buffers varies. This depends on the number of drives in use – and the applications using them.



36: AmigaDOS 3.0 Startup- Sequence

If you're lucky enough to own an Amiga 1200 or one of the chosen few to have an A4000, your Startup-Sequence will look something like the one shown below. (The version number and release date – line 1 – will give you a better clue.) This Startup-Sequence is similar regardless of whether your machine is hard or floppy disk based – the changes are noted in the text. Here's how it all works:

1. This is the script's VERSION string. The ';' at the start of the line tells the AmigaDOS script interpreter to ignore the line: programmers would call this a remark or comment line. In AmigaDOS 3 however, there is more to this than meets the eye: the VERSION command has been updated (again) and can now read a version string directly from a script! If you haven't tried the command now is a good time to start. Here are a couple of examples to get you going:

VERSION S:Startup-Sequence FULL

SPAT VERSION S:#?

The command looks for a special four-part string in the first line of the text file like this:

**\$VER: ScriptName Version
Date**

The first three parts must be present for the version string to be valid, the date is only printed when the FULL switch is present on the command line.

2. Patches late-breaking bugs in the system software. (Since I was using a beta-release ROM, it is not possible to say exactly what the latest fixes are.) The QUIET switch is a new option and replaces redirection to NIL: as found in earlier versions. It is important to note here Commodore has realised AmigaDOS reads commands faster if the path is specified rather than implied – so all external commands like SETPATCH in this case, begin with C:. In earlier releases of the software the Startup-Sequence relied on the default path setting and searched the current directory first. The speed increase, while not vast, is apparent.
3. This reads the current system variables for Workbench and Kickstart and stores them in two local environmental variables: WORKBENCH and KICKSTART. This command does not have a QUIET switch so re-direction to NIL: is used to suppress the output. Screen display at any point during Startup forces Intuition to open a default screen and window which is undesirable, since this should be handled by IPREFS .
4. Allocates about 7K of cache buffer to the internal drive. Re-direction to NIL: prevents the command generating output. If you have sufficient memory, this value can be increased to between 20 and 25. Also, second external drives can also be buffered here.
5. Raises the failure level to 21, above the fatal error stage. This prevents commands such as COPY and MAKEDIR stopping the script before the screen becomes visible at Step 36.
6. Creates the following directories using MAKEDIR's new multiple argument parser (present from 2.04):

RAM:T	(temporary files)
RAM:Clipboards	(clipboard unit files)
RAM:ENV	global environmental variables
RAM:ENV/Sys	(System-wide environmental (Prefs) settings)
7. Copies the contents of the environmental variable archive (ENVARC:) to the Ram Disk. ENVARC: is an automatic assignment created when the machine is booted.
- 8-9. Makes ASSIGN and EXECUTE resident. The PURE switch is used in case the flag has been cleared for some reason. Redirection to NIL: is used to suppress error output such as "PURE bit not set" if this does occur.
10. Creates the ENV: assignment (logical directory). This directory is automatically temporary since it's held in RAM and is used for a variety of user and system variables.
11. Re-assigns the temporary directory to the Ram Disk (T directory) for speed.

The AmigaDOS 3 Startup-Sequence

```
1.  ; $VER: Startup-sequence 39.9 (9.8.92)
2.  C:SetPatch QUIET
3.  C:Version >NIL:
4.  C:AddBuffers >NIL: DFO: 15
5.  FailAt 21
6.  C:MakeDir RAM:T RAM:Clipboards RAM:ENV RAM:ENV/Sys
7.  C:Copy >NIL: ENVARC: RAM:ENV ALL NOREQ
8.  Resident >NIL: C:Assign PURE
9.  Resident >NIL: C:Execute PURE
10. Assign >NIL: ENV: RAM:ENV
11. Assign >NIL: T: RAM:T
12. Assign >NIL: CLIPS: RAM:Clipboards
13. Assign >NIL: REXX: S:
14. Assign >NIL: PRINTERS: DEVS:Printers
15. Assign >NIL: KEYMAPS: DEVS:Keymaps
16. Assign >NIL: LOCALE: SYS:Locale
17. Assign >NIL: LIBS: SYS:Classes ADD
18. Assign >NIL: HELP: LOCALE:Help DEFER
19. IF NOT EXISTS SYS:Fonts
20.     Assign FONTS:
21. EndIF
22. BindDrivers
23. C:Mount >NIL: DEVS:DOSDrivers/~(#?.info)
24. IF EXISTS DEVS:Monitors
25.     IF EXISTS DEVS:Monitors/VGAOnly
26.         DEVS:Monitors/VGAOnly
27.     EndIF
28. C:List >NIL: DEVS:Monitors/~(#?.info|VGAOnly) TO T:M
    LFORMAT "DEVS:Monitors/%s"
29.     Execute T:M
30. C>Delete >NIL: T:M
31. EndIF
32. SetEnv Workbench $Workbench
33. SetEnv Kickstart $Kickstart
34. UnSet Workbench
35. UnSet Kickstart
36. C:IPrefs
37. C:ConClip
38. Path >NIL: RAM: C: SYS:Utilities SYS:Rexxc SYS:System
    S: SYS:Prefs SYS:WBStartup SYS:Tools
    SYS:Tools/Commodities
39. IF EXISTS S:User-Startup
40.     Execute S:User-Startup
41. EndIF
42. Resident Execute REMOVE
43. Resident Assign REMOVE
44. C:LoadWB
45. EndCLI >NIL:
```


12. Creates the CLIPS: assignment and points it to the Ram Disk's Clipboards directory. Large clipboards such as IFF files are stored here.
13. Creates the REXX: program assignment and points it to the scripts directory, S:. The S: assignment is made automatically when the machine is booted with a Workbench disk. REXX: can be assigned anywhere (you may even wish to create a separate directory for ARexx programs).
14. Makes the long awaited PRINTERS: assignment. This logical directory, which for downward compatibility should remain where it is, makes access to printer drivers much simpler. For instance, you can check which printers are installed by entering:

1>DIR PRINTERS:

as opposed to:

1>DIR DEVS:Printers

15. Does for keymap drivers what PRINTERS: did for printer drivers. Once again, for downward compatibility this assignment must be left alone.
16. Locale (the country localisation drivers and translators) are housed on a separate volume for floppy systems and in a directory on hard disk systems. This tells the system software where to look.
17. This ADDs classes to the available libraries – Classes are translations for a variety of software interpreters and were discussed in more detail last month.
18. Creates a late-binding assignment (HELP:) to LOCALE:Help. This assignment is hidden in the system assignment list until something attempts to access it, for instance:

1>DIR HELP:

At that point, AmigaDOS searches the LOCALE: assignment (a separate disk on a floppy-based installation) and look help to the help sub directory it finds there. This is primarily used by developers and the system and is of little consequence to end users.

19. Tests if a Fonts directory exists on the current system (boot) disk. If the directory is found control jumps to Step 21, otherwise it continues at Step 20.
20. Removes the automatic FONTS: assignment. This prevents the machine asking for a volume called FONTS whenever fonts are accessed.
21. Closes the IF...ENDIF construct opened at Step 19.

22. Attaches any software expansion drivers to the system. This, I suspect, has been retained for ancient compatibility.
23. This one is a real surprise... MOUNT has been given pattern matching! This means there is no longer any need to explicitly mount devices such as RAD and PIPE – instead these are copied from the Storage disk into the Devs/DOSDrivers drawer (using Workbench if you prefer) and mounted automatically during Startup. Each device driver has an accompanying dot-info file which explains the need for ~(#?.info) pattern – everything but dot-info. One side-effect of this means there is no longer a single mountlist for all devices, but in the event that is probably a good thing. Third-party mountlists can be placed in this DosDrivers drawer/directory to make the device active on Startup; alternatively you may alter the User-Startup file.
24. Tests for the existence of the Monitors directory in DEVS:. If one is found execution continues at Step 25, otherwise it skips to Step 31.
25. Checks for the file: VGAOnly in the DEVS:Monitors directory. If found, execution continues at Step 26, otherwise it jumps to Step 27.
26. Runs the VGAOnly monitor driver – there seems to be little rhyme or reason to this strategy and I have been unable to test for its purpose.
27. Terminates the IF...ENDIF construct opened at 26.
28. Creates a script file in RAM T which contains a list of available monitors like this:

DEVS:2024

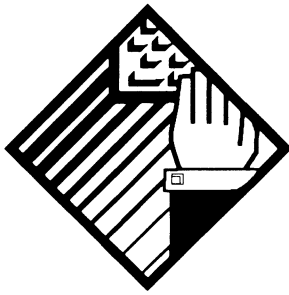
DEVS:Multiscan

The actual contents of the file vary depending on the contents of the DEVS:Monitors directory. You will find extra monitor drivers on your storage disk. A3000 owners with Multiscan monitors (for instance the CBM 1950) can use this feature to access the Productivity mode ore easily.

29. Runs the script file created in Step 28 and makes the monitors available. The main change here is each monitor file is an executable module in its own right thus rendering the 2.04 system commands ADDMONITOR and BINDMONITOR redundant.
30. Removes the temporary script created in Step 28 and cleans up the T: assignment. Strictly speaking this is correct, but it makes little difference.

31. Closes the IF...ENDIF construct opened at 24.
32. Sets the global environmental variable Workbench to the contents of the variable Workbench – the version number. These version numbers are generated by VERSION in Step 3.
33. As 32, but for Kickstart.
34. Clears the local environmental variable Workbench...
35. ...and the same again for Kickstart.
36. Runs IPREFS – which in turn sets the user preferences for the machine. The screen appears at this point: and not before unless something has reported an error to screen. This will not normally happen since all commands capable of reporting something have their output sunk to the NIL: device.
37. Makes the console clipboard active.
38. Sets the paths for Workbench. These allow you to access any part of the Workbench from Shell without needing to know where the actual command is: FORMAT is a good example.
39. Tests for the presence of a User-Startup script. If one is found execution continues at Step 40, otherwise it resumes at Step 41.
40. Runs the User-Startup sequence – the only bit you should touch until you are totally “au fait” with the machine.
41. Closes the IF...ENDIF construct opened at 39.
- 42-43. Remove EXECUTE and ASSIGN from the resident list and free some extra memory.
44. Activates the Workbench. Remember, this command only launches Workbench it is *not* the complete Workbench program.
45. Closes the initial Shell window and allows you to get started with the machine.

In spite of the apparent length of this Startup-Sequence, the whole thing executes in a few seconds: well, at least in far less time than the original 1.3 Startup took!



37: AmigaDOS 2.1 Startup- Sequences

Following quickly in the wake of AmigaDOS 2.04/2.05 came AmigaDOS 2.1. This first saw the light of day with the launch of the A600 and later versions of the Amiga A500 Plus. The Startup-Sequence is listed below and a blow-by-blow description is given in the pages that follow.

1. This is a special version string which can be used with the `VERSION` command to discover the version and release date of any script. The version number here is 38.22 with a release date of 24th April 1992. You can check this with the `VERSION` command like this:

```
1>VERSION S:Startup-sequence
```

```
Startup-sequence 38.22
```

```
1>VERSION S:Startup-sequence FULL
```

```
Startup-sequence 38.22 24.4.92
```

You can check for a specific Startup-Sequence like this in a script:

```
VERSION >NIL: S:Startup-sequence  
version=38
```

```
IF WARN
```

```
echo "This requires Startup  
version 2.1!"
```

```
ENDIF
```

or to be more exact about which one you want (to fix a bug for instance).

```
VERSION >NIL: S:Startup-sequence  
version=38 revision=22
```

```
IF WARN
```

```
echo "This requires Startup  
version 2.1 (revision 22)"
```

```
ENDIF
```

2. Patches known bugs in the operating system. Note the full path is given here because that speeds things up very slightly. The QUIET switch tells SETPATCH to work quietly in the background.
3. Reads the system version numbers and stores them in two local environmental variables: Workbench and Kickstart. Re-direction to NIL: is used to prevent the command from generating output and confusing the IPrefs daemon which does not start until much later in the script.
4. Adds 15 cache buffers to the internal drive (you can increase this number slightly for extra access speed).
5. Raises the failure level to 21 to prevent any following commands from stopping the Startup-script in its tracks.
6. Creates the following system directories in the Ram Disk:
 - T**
 - Clipboards**
 - ENV**
 - ENV/Sys**
7. Copies the user preferences settings to the Ram Disk. This affords very fast access to the preferences and improves the system's performance. The NOREQ option stops any requesters appearing if the ENVARC: assignment has not been made. This assignment is automatic from AmigaDOS 2, as are FONTS:, C: and DEVS: in previous releases.
8. Creates the ENV: assignment. This should not fail, but if it does no warning will be given.
9. Assigns T: to the Ram Disk's T directory for fast access to temporary files.
10. Assigns CLIPS: to the Clipboards directory in RAM.
11. Forces the default placement for ARexx scripts to the existing S: (scripts) directory on the system disk.
12. Creates an assignment for PRINTERS:. This is used by the new Preferences tools.
13. Creates the KEYMAPS: assignment for the Preferences tools.
14. Creates the LOCALE: assignment. Note: if you have installed Workbench 2.1 on floppy disk, this assignment may not exist in this format, since the Locale (country specific) information resides on a separate floppy.
15. Tests if a "Fonts" directory is present on the current boot disk. If it is, execution skips to Step 16; if not it continues.
16. This is where the existing FONTS: assignment is cleared.

The AmigaDOS 2.1 Startup-Sequence

1. ; \$VER: startup-sequence 38.22 (24.4.92)
2. C:SetPatch QUIET
3. Version >NIL:
4. AddBuffers >NIL: DFO: 15
5. FailAt 21
6. MakeDir RAM:T RAM:Clipboards RAM:ENV RAM:ENV/Sys
7. Copy >NIL: ENVARC: RAM:ENV ALL NOREQ
8. Assign >NIL: ENV: RAM:ENV
9. Assign >NIL: T: RAM:T
10. Assign >NIL: CLIPS: RAM:Clipboards
11. Assign >NIL: REXX: S:
12. Assign >NIL: PRINTERS: DEVS:Printers
13. Assign >NIL: KEYMAPS: DEVS:Keymaps
14. Assign >NIL: LOCALE: SYS:Locale
15. IF NOT EXISTS SYS:Fonts
16. Assign FONTS:
17. EndIF
18. BindDrivers
19. Mount >NIL: DEVS:DOSDrivers/~(#?.info)
20. Resident >NIL: C:Execute PURE
21. IF EXISTS DEVS:Monitors
22. List >NIL: DEVS:Monitors/~(#?.info) TO T:M
 LFORMAT 'DEVS:Monitors/%s'
23. Execute T:M
24. Delete >NIL: T:M
25. EndIF
26. SetEnv Workbench \$Workbench
27. SetEnv Kickstart \$Kickstart
28. UnSet Workbench
29. UnSet Kickstart
30. IPrefs
31. Echo "Amiga Release 2.1. Kickstart \$Kickstart,
 Workbench \$Workbench"
32. ConClip

```
33. Path >NIL: RAM: C: SYS:Utilities SYS:Rexxc SYS:System
    S: SYS:Prefs SYS:WBStartup SYS:Tools
    SYS:Tools/Commodities
34. IF EXISTS S:User-Startup
35. Execute S:User-Startup
36. EndIF
37. Resident Execute REMOVE
38. LoadWB
39. EndCLI >NIL:
```

17. Closes the IF...ENDIF construct opened at Step 15.
18. This command has been retained for ancient compatibility. It looks for any hardware drivers in the Expansion drawer and mounts them. Few users will ever require this.
19. Mounts all the device drivers in the DEVS:DOSDrivers drawer. This innovation is a major departure from the old system where all devices were mounted, one at a time, from a separate mountlist. The wildcard pattern "~(#?.info)" ensures MOUNT only attempts to mount real drivers, but it means a user can add or remove devices simply by dragging an icon from the DEVS/DOSDrivers drawer. Extra drivers are located on the storage disk with this release.
20. Makes EXECUTE resident in the system list. If EXECUTE is not present in the C: directory (as it should be) no error will be reported, but the monitor and User-startups will fail.
21. Checks if the Monitors drawer is present in DEVS. If it is, execution continues at Step 22; otherwise it jumps to Step 25. Note you do not need this drawer unless you are using a special monitor - such as a multi-scan.

```
DEVS:Monitors/Multi-scan
DEVS:Monitors/A2024
DEVS:Monitors/Super72
```
22. Creates a script (T:M) based on the contents of the Monitors drawer. Note the pattern matching which prevents the icon files from being included. The script's exact content depends entirely on the contents of the Monitors drawer. The new monitor tools replace the old system commands ADDMONITOR and BINDMONITOR. Typically, this script might take the following format:
23. Executes the script created at Step 22 and binds the monitor hardware to the system. When first used, this only makes the monitors available to the ScreenMode Preferences.

24. Removes the temporary script created at Step 22 and frees up memory.
25. Terminates the IF...ENDIF construct opened at Step 21.
26. Sets the global environmental variable "Workbench" to the contents of the local environmental variable "Workbench". This just makes it available from any Shell. (Local variables are removed when the Shell closes.)
27. As 25 but for Kickstart.
28. Clears the local variable "Workbench" – but why?
29. As 28 for Kickstart.
30. Starts the IPrefs Daemon which sets the Preferences and makes the user-defined Workbench screen.
31. Displays the initial startup message with the version numbers of Workbench and Kickstart substituted at the appropriate positions.
32. Activates the console clipboard software. When this command is running, text copied from the console is placed in the CLIPS: assignment for use by other software. Basic console clipping is supported from the Shell so this command can be omitted if you wish.
33. Sets the following paths. Re-direction to NIL: is used to prevent error messages when a path does not exists.

RAM:

C:

SYS:Utilities

SYS:Rexxc

SYS:System

S:

SYS:Prefs

SYS:WBStartup

SYS:Tools

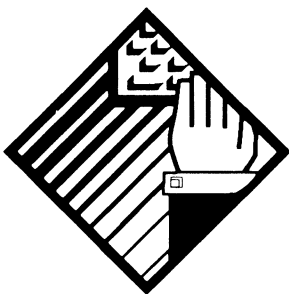
SYS:Tools/Commodities

34. Tests for the existence of a User-Startup-Sequence and...
35. Executes a User-Startup-Sequence if necessary. Your own startup preferences should be placed in this file and not in the main Startup-Sequence. For instance to add disk buffering to drive 1, User-startup would read like this:

ADDBUFFERS DF1: 22

36. Terminates the IF...ENDIF construct opened at step 34.

- 37. Removes EXECUTE from the Resident list.
- 38. Activates the Workbench.
- 39. Closes the initial Shell window. Re-direction to NIL: is used to prevent the default "Shell process 1 ending" message from appearing.



38: AmigaDOS 2.04 Startup- Sequences

The startup featured here was taken directly from the 2.04 distribution disks. Minor revisions (ie, 2.05 *et al*) are very similar and what follows will be pertinent. The listing itself can be found overleaf.

Details of the Startup-Sequence for the original AmigaDOS 2 as supplied with early machines such as the A3000 can be found in Appendix B.

Explanation

1. Calls SETPATCH and fixes certain bugs in the V2 ROM. The current patch list (as of 2.04) is noted in *Mastering AmigaDOS 3 Tutorial*, Reference. Output re-direction to NIL: stops anything being printed at this stage.
2. Sets these two local environmental variables – Workbench and Kickstart – with values retrieved from the ROM. At 2.04 these are:
Kickstart = 37.175
Workbench = 34.28
Output re-direction to NIL: prevents the command displaying the version numbers.
3. Allocates 15 extra cache buffers to the internal drive – 5 buffers have already been automatically added during ROM-bootstrap. This command is redirected to NIL: to prevent the message *DF0: has 20 buffers*.

4. Raises the failure level to 21 – so nothing can stop the script in its tracks! Since the screen remains invisible for at least the first half of this script, if something did stop it, you might never get to find out why!
5. Adds the EXECUTE command to the resident list. Note the two extra parameters PURE and ADD are not usually required here. Once again, output is redirected to NIL: to stop any messages appearing. Typically this would only occur if something went wrong: you deleted EXECUTE for instance.
6. Creates the following directories in RAM:

T	The temporary directory.
Clipboards	For clipboards (clips).
ENV	The environmental handler directory.
ENV/SYS	A system directory used by the Prefs tools.
7. This looks a lot worse than it is. Basically, this copies the permanent Preferences settings (all 16 of them) from disk to RAM. ENVARC: is a late-binding assignment made automatically during the initial bootstrap (to SYS:Prefs/Env-Archive) and bound here. The ENV: assignment is fixed at Step 8. The extra switches are as follows:

ALL	Copy files and sub-directories too.
QUIET	Suppresses normal progress messages but does not affect error reports.
NOREQ	Stops requesters like <i>please insert volume</i> etc. from being generated. (It probably isn't required here.)
- 8-10. Create the ENV:, T: and CLIPS: logical assignments.
11. Sets the REXX: assignment to point to the scripts directory. This allows you to place your ARexx scripts in S:.
12. Tests for the existence of a *Monitors* directory on the boot disk. If one is found execution continues at Step 13, otherwise it branches to Step 16.
13. The Monitors directory is usually empty. If it is required, any files in it are actually mini-scripts! You'll find these in the Monitorstore directory on the Extras2 disk. This line joins the scripts together (note the pattern excludes dot-info files) as one big script and saves them in the T: assignment for use...
14. ...here. This executes the script created at Step 13. Typically it might look like this:

```
sys:System/Addmonitor NUM=4 NAME=A2024.Monitor
sys:System/Addmonitor NUM=2 NAME=pal.Monitor
```

The AmigaDOS 2.04 Startup-Sequence

```
1.  c:setpatch >NIL:
2.  c:version >NIL:
3.  addbuffers >NIL: df0: 15
4.  Failat 21
5.  Resident >NIL: C:Execute PURE ADD
6.  mkdir ram:T ram:Clipboards ram:env
   ram:env/sys
7.  copy >NIL: ENVARC: ram:env all quiet noreq
8.  assign ENV: ram:env
9.  assign T: ram:t ;set up T: directory for
   scripts
10. assign CLIPS: ram:clipboards
11. assign REXX: s:
12. if exists sys:Monitors
13.   join >NIL: sys:monitors/~(#?.info) as
      t:mon-start
14.   execute t:mon-start
15.   delete >NIL: t:mon-start
16. endif
17. BindDrivers
18. setenv Workbench $Workbench
19. setenv Kickstart $Kickstart
20. IPrefs
21. echo "Amiga Release 2. Kickstart $Kickstart,
      Workbench $Workbench"
22. conclip
23. mount speak:
24. mount aux:
25. mount pipe:
26. path ram: c: sys:utilities sys:rexxc
   sys:system s: sys:prefs sys:wbstartup add
27. if exists sys:tools
28.   path sys:tools add
29.   if exists sys:tools/commodities
30.   path sys:tools/commodities add
31.   endif
```

```

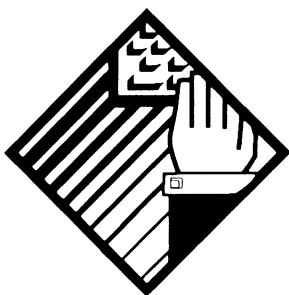
32. endif
33. if ${sys/keyboard} NOT EQ "${sys/keyboard}"
34.   setmap ${sys/keyboard}
35.   else
36.   PickMap sys: initial
37. endif
38. if exists s:user-startup
39.   execute s:user-startup
40. endif
41. LoadWB
42. endcli >NIL:

```

15. Tidies up by removing the monitor script.
16. Closes the IF...ENDIF construct opened at Step 12.
17. Executes BINDDRIVERS and adds any special hardware to the system list. This command is only required for older expansions and is retained for compatibility.
18. Creates a global environmental variable *Workbench* using the value stored in the process local variable, *Workbench*. Local variables like this will disappear when the Shell closes, this preserves them for later use.
19. Like Step 18, this creates a global environmental variable *Kickstart* using the value stored in the local variable, *Kickstart*.
20. Launches IPrefs – Intuition Preferences. IPrefs is a *daemon* – a background process which manages all the new Preferences settings (the ones stored in ENV:Sys). The initial Shell window does not appear until this command is running. If a command wants to print something before IPrefs runs, Intuition automatically restores the default parameters. In this case, IPrefs complains bitterly: *close all windows except drawers*. For this reason, you should *not* remove any of the >NIL: output re-direction from any commands prior to this one.
21. Displays the current release and version numbers. Note these are taken from the automatic local variables generated by VERSION at Step 2 and are not entered directly.
22. Starts CONCLIP – the console clipboard process.
- 23-25. Mount SPEAK: (speech), AUX: (un-buffered serial), and PIPE: (interprocess FIFO) devices.

26. Adds the following directories to the default PATH settings: RAM:, C, Utilities, RexxC, System, S, Prefs, WBStartup. The automatic path includes C: and the current directory only.
27. Looks for a Tools directory on the current boot disk (this usually only applies to hard disks). If one is found, execution continues at Step 28, otherwise it branches to Step 32.
28. Adds the Tools directory to the current path.
29. Looks for the Commodities directory in the Tools directory of the current boot disk – we already know there is a Tools directory thanks to Step 27. If one is found, execution continues at Step 30, otherwise it branches to Step 31.
30. The Commodities directory is added to the search path...
31. ...the IF...ENDIF construct opened at 29 is closed...
32. ...as is the one opened at 27.
33. This line looks a bit horrifying at first; in essence it merely checks to see if the keyboard variable has been set. There are two possibilities:
`${sys/keyboard}` returns `${sys/keyboard}` and control jumps to Step 35.
 or :
`${sys/keyboard}` returns the name of the current keymap and control continues at Step 34.
 The test value:
`${sys/keyboard}`
 always returns:
`${sys/keyboard}`
 The dollar symbol is *escaped* by the asterisk.
34. If the keymap has been chosen, this calls SETMAP with the appropriate argument which, in the UK, is *gb*.
35. If execution reaches here from Step 34, it branches to Step 37, otherwise it continues at Step 36.
36. Executes PICKMAP (it's a script incidentally) and allows you to select a keymap which is appropriate for your country.
37. Closes the IF...ELSE...ENDIF construct opened at Step 33.
38. Tests for the presence of a User-Startup...
39. ...and executes it if necessary.
40. Closes the IF...ENDIF construct opened at Step 37.

41. Activates the Workbench at last.
42. Finally closes the initial Shell window. The re-direction to NIL: here prevents the command generating a *Shell Process 1 ending* message.



39: Customising the Startup

The Startup-Sequence is quite complex and there is little beginners can do to change it. However, as we have mentioned, Commodore have thoughtfully added a patch to allow anyone to extend the startup with some of their own commands – the User-Startup script. With a little thought, this is all most users will ever need to modify – and that's what these first scripts will do.

Early versions of Workbench 2 were distributed with a default American keyboard setting. Getting this to work just about anywhere else in the world requires adding your own keymap. This is the first thing you should do and you can create it in the usual way using ED and it should be saved into the S: directory. Insert this line to the first line of your User-Startup:

SYS:System/SETMAP gb

Note: This assumes that (a) you are using a UK keyboard and (b) you have copied the appropriate keyboard driver into the DEVS:Keymaps directory.

Workbench 3 of course does not suffer from this problem and provided the correct keymap is available, AmigaDOS does it automatically with IPREFS.

Assigning Extra Paths

The ASSIGN command is one of the most powerful in the whole of AmigaDOS and it has received a considerable revamp in the latest release. Here are some suggested

assignments which you may find useful – they don't take much RAM and since ASSIGN is resident anyway, they don't slow down the Startup-Sequence much. Note that these modifications apply to Workbench 3 too.

- 1 ASSIGN PRINTERS: DEVS:Printers PATH**
ASSIGN PRINTX: Storage3.0:Devs/Printers PATH
- 2 ASSIGN KEYMAPS: DEVS:Keymaps**
ASSIGN KEYS: Storage3.0:Devs/Keymaps PATH
- 3 ASSIGN UTILS: SYS:Utilities**
- 4 ASSIGN FONTS: SYS:Fonts DH1:Fonts ADD**
- 5 ASSIGN FONTS: MoreFonts: PATH ADD**

1. These assignments are useful when you need to copy a couple of printer drivers. This example assumes you are going to copy from a disk in drive df0: and that disk has already got the correct directories. This saves having to rummage around for a disk labelled Storage3.0 or some such thing! Note, the PATH switch stops ASSIGN asking for the disk until the assignment is referenced.

1>COPY DF0:devs/Printers/Xerox_4020 to DEVS:Printers

becomes:

1>COPY PRINTX:Xerox_4020 to PRINTERS:

2. These assignments work much like the printers example above except in this case AmigaDOS asks for the correct disk. Note: you must ensure the disk name is Storage3.0 (or Extras2 in Workbench 2) or this example will not work.
3. This is a little goody for people like us who have to type with just a couple of fingers and find UTILITIES a dog to spell correctly. It can be used at any time you need to access the Utilities drawer on the system disk, for instance:

1>DIR UTILS:

is equivalent to:

1>DIR SYS:Utilities

4. This uses another special feature of the AmigaDOS ASSIGN command in that it can assign more than one logical path to each logical device name. For this example, we'll assume you have a hard disk (SYS:) and some extra fonts on a further partition (DH1:). The system software can now read and use fonts from both of these. Note however, this feature might not be supported by older software.

5. Finally, another example using FONTS:. After a while you may find you have accumulated more FONTS: than can comfortably fit on a your boot disk. Using this line, you could have a disk (labelled MoreFonts) which contained nothing but fonts. The Amiga only asks you for the MoreFonts disk when it needs to access the fonts. You can clear unnecessary fonts from your boot disk freeing up much needed space.

The Auto-Startup Directory

This is a new directory we have created to supplement the Workbench WBStartup drawer. If you have already tried using this you may have discovered it can only run files which execute then return after less than a few seconds. But this is also a great place to pop things like a clock or one of the Commodities.

Auto-Startup uses an idea borrowed from the Startup-Sequence to launch almost any program (or programs) you care to drop in a startup drawer. We've called the directory "Auto-Startup" for this example. You should create this directory on your Workbench disk before trying to use this.

Now add the following lines to the User-Startup file:

```
IF EXISTS SYS:Auto-Startup  
LIST >T:run-start SYS:Auto-Startup/-(#?.info) LFORMAT "RUN  
>NIL: %S%S"  
EXECUTE T:run-start  
ENDIF
```

For example, if you wanted a clock every time you started the machine, you could just drag the clock icon from its resting place in Utilities and drop it into the Auto-Startup drawer. This technique is not perfect, but it works for most run-launchable programs; typically any Workbench icons which are shown as TOOLS on their information windows will work. This means you cannot start a Shell from this drawer because Workbench sees it as a Project! Also, if you want to use this for, say REXXMAST, you must remove the call to REXXMAST from the Startup-Sequence. You can launch scripts from here by first setting their "S" protection bit and moving them when shown as text. You will not be able to pass any arguments however.

A Complete Example

This is a typical User-Startup script using the examples we have just discussed. This one sets up a UK keymap, runs some user programs, assigns some extra directories, moves REXX: to an external disk and starts a new Shell. The contents can vary according to your preferences.

```
SYS:System/SETKEYBOARD gb
IF EXISTS SYS:Auto-Startup
LIST >T:run-start SYS:Auto-Startup/~(#?.info) LFORMAT "RUN
>NIL:%s%s"
EXECUTE T:run-start
ENDIF
ASSIGN PRINTERS: DEVS:PRINTERS
ASSIGN KEYS: DEVS:KEYMAPS
ASSIGN UTILS: SYS:Utilities
ASSIGN REXX: AREXX: PATH
ASSIGN TOOLS: Extras3.0:Tools PATH
NEWSHELL
```

Modifying the Workbench Startup-Sequence

On close examination there is little that can be done to improve on what Commodore have done with Workbench's Startup-Sequence because it is almost completely self-contained. Any tweaks or changes should be performed in User-Startup.

Before starting to tinker with it, you should place a copy of a system-configuration file taken from a Workbench disk into the SYS:DEVS directory. Although preferences have changed dramatically for this release, necessitating the addition of the IPREFS daemon, the old structure is still supported. When this file is present you do not have to wait until IPREFS starts. If you neglect to do this, and the script crashes before IPREFS sets the preferences colours, you won't be able to see the display.

The only commands which can be safely removed or altered are ADDBUFFERS, SETCLOCK and REXXMAST. The latter only needs to be removed if you are running REXXMAST from the Auto-Startup drawer as described above. If in doubt – do not touch.

ED-Startup File

The subject of the ED-Startup file was touched on briefly in an earlier chapter. Here we'll look at how they are constructed and provide a few simple examples that you can try out for yourself.

AmigaDOS is provided with an Ed-Startup file and you can examine this with the following command:

1>RUN ED S:ED-Startup

This might give you some indicators as to how it is structured especially if you have already used the ED menus available. However, it is very simple to construct your own personalised menus should you so wish. This following example does just that and in doing so explains how the menu system works. At the end of it you can return to the original menus as you wish simply by restoring the original ED-Startup file.

The first thing to do before you proceed is to rename the supplied ED-startup file for future use thus:

1>RENAME S:ED-Startup S:ED-Backup

Create a new ED file called 'ED-Startup' thus:

1>ED S:ED-Startup

To define a series of menus and actions you must stick to a rather strict set of rules. We say strict because even a simple error can confuse ED and the result is that when you re-run ED there will be no menus present. The syntax is as follows:

si <Object number> <Identifier> <"Menu string"> ["<ED command>"]

Let's take each in turn and we'll follow them with a simple example to show how they work in theory:

si This 'command' informs ED that a menu definition follows and is followed by a space.

<Object number>

This is the sequence number of the menu or menu option. The first item on the menu bar is always number 0. The first option in the first menu will be number 1. The second option in the first menu number 2, and so on. If a second menu is used then the number of the name to appear on the menu bar is the next in sequence.

<Identifier>

This is one of four numbers which determines what the definition is. They are:

- 0 End. This is always the last definition.
- 1 This informs ED that this item is to appear on the Menu bar.
- 2 This informs ED that this is a menu option.
- 3 This draws a grey bar across the menu.

ED Menu Examples

The following example provides ED with two menus:

```
si 0 1 "Movement"
si 1 2 "Top of File   <ESC>t" "t"
si 2 2 "Bottom of File <ESC>b" "b"
si 3 1 "About & Out"
si 4 2 "About ED <ESC>sh" "sh"
si 5 2 "QUIT      <ESC>q" "q"
si 6 0
em
```

Note the use of 'em' at the end of the menu definition – this is essential. If you save this file (as S:ED-Startup) and re-run ED you should find two menus installed under the titles 'Movement' and 'About & Out'.

Examine the program above and you will see how the 'object number' starts at 0 and increments to 6 – thereby giving the sequence order of the menus starting from top left (ie, the first menu title) to the bottom right (last option on last menu).

You should also be able to see how the identifiers work. '1' indicates a menu name, '2' a menu option and '0' the end of the menu definition.

The first string gives the details that are to appear on the menu and this also by tradition includes the key sequence. This is followed by a space and then the ESC key sequence inside quotes. Note that it is not necessary to use ESC simply the letter or letters that could be used in conjunction with the ESC key.

It would have been possible to include all four items in a single menu and separate the two major option groups by a greyed bar. The following script shows how this could be done.

```
si 0 1 "Menus"
si 1 2 "Top of File   <ESC>t" "t"
si 2 2 "Bottom of File <ESC>b" "b"
si 3 4
si 4 2 "About ED <ESC>sh" "sh"
si 5 2 "QUIT      <ESC>q" "q"
si 6 0
em
```

A slight modification has been made to the menu title and then option 3 has been edited so that the identifier reflects a '4' which produces a grey line to separate the two groups. Again, save this as ED-Startup and see for yourself.

Menu Information

Many of ED's extended commands require you to provide items of information. These can be implemented with relative ease by including the normal response of the extended command in the command string. For example, the following short ED-Startup script implements a "Save as" option in a menu:

```
si 0 1 "Project"
si 1 2 "Save as... <ESC>sa" "sa ? /Filename: /"
si 2 0
em
```

With this installed as the ED-Startup script next time you run ED just the single menu option appears. Should you select it then a file dialogue box appears in the screen which can be used in the normal fashion.

In this example the '?' implements a 'get details' response which in this context is a filename. The text between the two '/' characters is echoed to the screen and here 'Filename: ' appears at the top of the file dialogue box.

A find string facility could be added thus:

```
si 0 1 "Project"
si 1 2 "Save as... <ESC>sa" "sa ? /Filename: /"
si 2 2 "Find... <ESC>f" "f ? /Find: /"
si 2 0
em
```

If the find option is used then the 'Find: ' text is displayed at the base of the ED window. Enter the text being sought and press RETURN in the normal fashion.

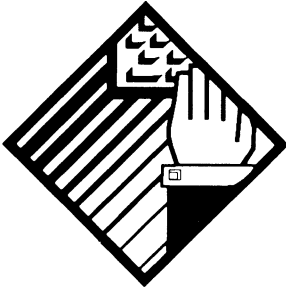
Of course some ED extended commands require two items of information – the search and replace command for instance. These can be catered for by repeating the 'query-text' text a second time in the command definition. The following example adds a Query-Replace facility to the earlier script by implementing <ESC>eq:

```
si 0 1 "Project"
si 1 2 "Save as... <ESC>sa" "sa ? /Filename: /"
si 2 2 "Find... <ESC>f" "f ? /Find: /"
si 3 2 "?Replace <ESC>eq" "eq ? /Replace this: / ? /With this: /"
si 2 0
em
```

If the '?Replace' option is selected the 'Replace this:' text is displayed at the base of the ED window. Enter the search string and then the replacement string as then prompted by the 'With this:' text.

Designer Menus

That then is all there is to implementing your own ED menus. If you are planning to produce your own fully customised set then you will find it beneficial to first sketch out exactly what you want on a sheet of paper before creating the ED-Startup file itself. If not you can re-instate the original menu system supplied by deleting the newly creating one (or renaming it for future use) and renaming the original back to Ed-Startup.



40: Customising Disks

So far we have concerned ourselves with using working copies (ie, backups) of the Workbench disk. However, the disk is just about full to the brim and there is little or no room for any additional information to be stored.

Despite supplying much of the information that you need to operate your Amiga, most of it remains redundant. For instance, if you plan to work exclusively from the Workbench, the files associated exclusively with the Shell aren't needed. You need to be careful here though, because some files are used by the Workbench – it may need them even if you don't. Likewise, if you plan to use the Shell exclusively, then some of the files associated with the Workbench become superfluous.

You may be wondering why this is. Just why did the Amiga's designers supply you with all this apparently useless stuff? The simple answer is that you are one user in tens of thousands – and the Amiga is a system which can be tailored to suit anyone's needs.

Unlike some systems, Commodore do not expect you to put up with what you are supplied with – they are giving you the option to design a system that suits you and you alone. This far-sightedness comes at a cost, of course. Not in terms of money – the Amiga offers unequalled power for the money, Amiga also offers you power without the price. The cost to you is time and effort. Commodore have done their part in supplying the tools and it's up to you to get the best

from them. It's up to us to show you how – and that's what this chapter is about.

Files that are essentially demos are obviously not important as are files used to support hardware that you do not have. For example, if you are using a parallel printer, then the file that handles the output to a parallel printer is required. However, files associated with other printers, such as a serial printer, are not likely to be required.

The final requirement for each of your disks will be determined by you. Computing is so much a personal task that every individual has their own likes and dislikes. Within a few weeks of using your Amiga seriously, you will become aware of what files you wish to use and those that you never use. The final flavour of your customised disks will also, to a large extent, depend upon the hardware you have. On a single drive system you may wish to use the RAD Disk, on a dual drive system you may wish to use DF1: exclusively for saving programs and files. At this point it is worth customising copies of your Workbench disks.

When you reach this stage, the term Workbench disk becomes redundant and it is better to refer to it as your own 'Startup' or 'boot' disk – the disk you use when you first switch on your Amiga. This comes from the cliché about pulling yourself up by your own BOOTstraps (we call them laces in the UK). A slightly cynical colleague of ours prefers to think of the term more literally: "To start the machine you have to boot it, right. Kick it up the backside and wake it up." To save embarrassment, we've used slightly more euphemistic language in this quote – but you get the gist.

There is no rule that says you can only have one startup disk, in fact you will soon find that you are using several different startup disks. For instance, we have a Workbench startup disk, a Shell startup disk, an AmigaBASIC startup disk, a machine code startup disk, a wordprocessing startup disk – and so the list goes on. In fact, it would probably be true to say that you should have a specialist startup disk for each of the main tasks that you employ your Amiga for.

To customise a startup disk the best place to start, in most cases, is from a copy of the original Workbench disk. It is much easier to delete files from a backup disk than it is to remember which files to copy to a new disk. For this purpose, a brief description of the various directories on a standard Workbench3.0 disk is given at the end of this chapter.

Creating System (Bootable) Disks

Whereas the above examples of startup disks are customised by cutting down existing copies of Workbench disks, there will be occasions when you want to create even more specialised disks. These will be created from newly formatted disks. However, it isn't possible to use a newly formatted disk as a startup disk as it does not contain the information required by the Amiga for the booting procedure. This information must be installed onto it and, once in place, the disk becomes a 'System disk' which can be used to startup the Amiga.

The command used to install a system disk is `INSTALL`, which has the following format:

```
1>INSTALL <drive>
```

The command writes information to the disk in a special area called a 'boot block'. This is the area in which the Amiga looks for the information it needs from the system disk. The disk can now be used to boot your Amiga but that's all. It is rather useless as it stands.

The Startup-Sequence File

When the Amiga is booted from the standard Workbench disk it goes through a special series of actions. These actions, and their sequence of operation do not just happen – they are determined by a section of "bootstrap" code in Kickstart and customised by a special script file stored in the `S` directory called 'Startup-Sequence'. Like any other script file it consists of a series of AmigaDOS commands linked together in a predetermined way, and which ultimately leave you with a Workbench.

If you want to examine your Workbench disk Startup-Sequence file then do so: *only* experiment on a copy. The best way to do this is to copy the file into the Ram Disk and then use `ED` to examine it. To do this type the following command lines:

```
1>COPY FROM DF0:S/Startup-Sequence TO RAM:
```

```
1>ED RAM:Startup-Sequence
```

This contains many commands that we have not examined, but we will do so in due course.

An Example Startup Disk

The rest of this chapter is given over to the development of a simple startup disk. The aim is to produce a disk that can be used exclusively for MEMacs. While this may be of limited use for your particular needs, it does clearly outline the various considerations that you must bear in mind when planning a disk of your own. The use of a simple task should not cloud the underlying issues

which are of great importance.

The first job is to format, name and install the disk. This example assumes a single drive system, dual drive users can speed the process up by using both drives – keep Workbench in one external drive and the disk being worked on in the internal drive.

First format a new disk as outlined in Chapter Five – using the command:

```
1>FORMAT DRIVE DFO: NAME "ED"
```

You could use any name. We've used 'ED' because MEmacs is an editor. Besides, it means less typing later on! Once formatting is completed, install the disk like this:

```
1>INSTALL DFO:
```

Required Directories

The first thing to decide is the directories that must be created on the disk. A subset of the AmigaDOS command set is required and these are always located in directory 'C'. The Startup-Sequence must also be loaded into a directory called 'S'. A directory called 'TextFiles' could also be created to hold the files you create, thereby keeping them out of the root directory which might otherwise become cluttered. Lastly, we'll need some extra directories for system files.

Primarily, this operation requires extensive use of two commands COPY and MAKEDIR. In all current versions of AmigaDOS these commands are transient (disk loaded). Therefore to speed things up we'll make a copy of them in RAM.

```
1>RESIDENT C:COPY
```

```
1>RESIDENT C:MAKEDIR
```

The directories can be created thus – first for our use:

```
1>MAKEDIR ED:TextFiles
```

Now for system use:

```
1>MAKEDIR ED:C
```

```
1>MAKEDIR ED:DEVS
```

```
1>MAKEDIR ED:DEVS/Keymaps
```

```
1>MAKEDIR ED:DEVS/Printers
```

```
1>MAKEDIR ED:DEVS/Clipboards
```

```
1>MAKEDIR ED:FONTS
```

```
1>MAKEDIR ED:L ED:LIBS
```

```
1>MAKEDIR ED:Prefs ED:Prefs/env-archive
```

```
1>MAKEDIR ED:S
```

```
1>MAKEDIR ED:System
```

Be sure to use the volume name 'ED:' as shown, otherwise AmigaDOS will assume that you wish to carry out the task on the Workbench disk and ask for it back. You could reference the disk using the drive's device name, say DF0:, but it's much safer to use the volume name – that way AmigaDOS knows which disk you are talking about!

Cataloguing the disk should return the following:

```
1>DIR ED: DIRS ALL
Textfiles (dir)
C (dir)
DEVS (dir)
  Clipboards (dir)
  Keymaps (dir)
  Printers (dir)
FONTS (dir)
L (dir)
LIBS (dir)
Prefs (dir)
  env-archive (dir)
S (dir)
System (dir)
```

The Root Directory

For the purposes of this demonstration, we'll put just the editor we're planning to use in the root directory. For this example we've stripped the disk to its bare minimum for one of these applications. If you intend using a commercial program you'll probably find it requires additional files. If in doubt, copy the entire contents of each required System directory listed above.

The System, L and Libs directories are only required if the application offers disk formatting and so on.

MEMACS can be found on the Extras disk in the Tools directory. This command moves it to the root directory of our ED disk:

```
1>COPY Extras3.0:Tools/MEMACS to ED:
```

The Command Directory

The next decision to be made is which command files will need to be copied across into the C directory of the ED disk. This task is made a little easier as some files are required by default and the need for others should become self-evident. For this application we'll only need CD and DIR plus a few others to handle the startup

“housekeeping”. Table 40.1 lists the commands to be copied across, with a brief description of each. Several of these commands are new to this stage of the book, but they are covered in *Mastering AmigaDOS 3 Reference*.

<i>Command</i>	<i>Action</i>
ADDBUFFERS	Speed disk access
ASSIGN	Create a logical assignment
COPY	Copy file from one point to another
IPREFS	Manages preference settings
MAKEDIR	Make new directory
SETPATCH	Cure bugs

Table 40.1. Commands to copy.

On a single drive system the transient command files required could be copied to the Ram Disk first, however, since we’re constructing a boot disk which will require quite a bit of copying, it’s better to make COPY resident (or copy it to RAM:) so you won’t have to fiddle with a lot of disk swapping. Transient commands are any executable programs which are loaded at run time then purged from the system when they complete. Many of the transient commands are internal to the Shell and are not in the C directory. Don’t worry yourself unduly about this, it’s meant to make life easier.

Using the following commands we can copy some AmigaDOS commands to the ED disk. We’re going to use the default C assignment, C:, to locate these and secrete them on the ED volume by using the volume name. This is a short-cut which you should become familiar with – especially if you only have a single drive. You’ll still have to swap disks, of course, but there’s less room for mistakes.

1>C:

1>COPY (ADDBUFFERS|ASSIGN) to ED:C

1>COPY (MAKEDIR|SETPATCH) to ED:C

1>COPY (IPREFS|COPY) to ED:C

The Devices Directory

Next we’re going to need the System-Configuration file from the DEVS directory of your Workbench disk. Just in case you decide to print your finished text (why shouldn’t you) we’ll also need the printer device and a printer driver.

If you don't have a printer, you can ignore the lines pertaining to printers, but it's good practice to try them out anyway so you know what to do in future. Here's how to copy the relevant files – you could use wildcards, but for the purposes of example we've listed the complete filename:

```
1>COPY DEVS:System-configuration to Ed:Devs
```

```
1>COPY DEVS:Printer.device to Ed:Devs
```

That sorts out the hardware side of things, now what about your keyboard layout? We're assuming that you have already moved the appropriate keymap file to your boot disk. If not, now would be a good time! Keymaps are stored on the Storage disk in:

Storage3.0:Keymaps

The next two lines will do the trick in either case. You only need to use the first line to move the correct "keymap" to your boot disk, if you haven't already done so use:

```
1>COPY Storage3.0:Keymaps/GB to DEVS:Keymaps
```

```
1>COPY DEVS:Keymaps/GB to ED:Devs/Keymaps
```

Since we're going to prepare a disk that's capable of doing some printing; we'll need the right printer driver too. Our setup will assume you have changed to one of the most common printers, an Epson (or Epson clone). Once again these two lines show how you can move the correct driver to your boot disk before copying it to the ED disk.

```
1>COPY Storage3.0:Printers/EpsonX TO DEVS:Printers
```

```
1>COPY DEVS:Printers/EpsonX TO ED:Devs/Printers
```

Workbench 2.04 users will find the Keymaps and Printers subdirectories on their Extras disk in the Devs directory.

The Libraries Directory

This contains any libraries required by the system software and any software running. This varies quite dramatically from release to release and is entirely dependent on what commands will be executed. Some libraries (like the dos.library) are built into Kickstart, others are loaded from disk and it's these which concern us. For instance: Designer Mouseware's 'Sleepy', Steve Tibbet's 'DiskX' and many more require a third-party library called the arp.library. The VERSION command needs the version.library. BINDDRIVERS requires the info.library. ED, ICONED and MORE need the ASL.Library (which is a subset of the arp.library). The list goes on and the only safe solution at this stage is to copy the lot:

```
1>COPY LIBS:#? to ED:Libs
```

The Fonts Directory

The Fonts directory contains very little (in fact nothing!) as the majority of fonts are held on the Fonts disk for reasons of space. The simplest way would be to copy all the Fonts across using a command such as:

```
1>COPY Fonts: ED:Fonts ALL
```

But that would require more space than is available. The best method is to decide on what fonts you wish to install and then copy them across. For example if you wished to install Sapphire on your ED disk you could use:

```
1>COPY Fonts:Sapphire.font to ED:Fonts
```

```
1>COPY Fonts:Sapphire/#? to ED:Fonts/Sapphire
```

However, we're not off the hook just yet. Remember those .font files? They contain information about the contents of the font sub-directories. Since we've been fiddling with the contents of those directories, the .font files have to be updated – otherwise applications will not know which font style and sizes are available. This can be accomplished in several ways – but the easiest requires a little trick.

The command to update the .font files is called, not surprisingly, FIXFONTS and it always works on the FONTS: logical assignment. By changing this (temporarily) we can update the .font files:

```
1>ASSIGN FONTS: ED:FONTS
```

```
1>FIXFONTS
```

Optionally, you can put everything back as it was before. It isn't really necessary in this case – but it helps to know how.

```
1>ASSIGN FONTS: SYS:FONTS
```

This may create a:

```
Can't find SYS:FONTS
```

message if you are solely using the default font supplied on your Workbench disk. Not to worry.

If you have selected a non-standard font for windows and so on you should ensure that you copy all the fonts selected to this directory. This is the safest way:

```
1>COPY FONTS: ED:Fonts ALL
```

The Handlers Directory (L)

Files in this directory are not required by this application, however some applications will need certain handlers. CrossDOS, for instance, required the CrossDOSFileSystem and the FileSysTrans directory; the serial device makes use of the aux handler and so on. Better be safe than sorry and copy the lot at this stage:

```
1>COPY L:#? to ED:L ALL
```

The Env-Archive Directory

The files in ENVARC: belong to the system configuration and are beyond the scope of this book.

```
1>COPY ENVARC: ED:Prefs/env-archive ALL
```

The Sequences Or Scripts Directory

Finally we can create the Startup-Sequence file.

```
1>ED ED:Startup-Sequence
```

Type the following lines into ED, checking each carefully as you go:

```
ADDBUFFERS >NIL: DFO: 10
COPY ENVARC: RAM:ENV ALL QUIET NOREQ
ASSIGN ENV: RAM:ENV
RUN >NIL: IPrefs >NIL:
ECHO "Customised ED Disk."
SYS:System/SetMap GB ; 2.04 machines ONLY!!
CD Textfiles
SYS:MEMACS NewFile
```

Note the line SYS:System/SETMAP is required for Workbench 2.04/5 machines only. Starting from V2.1, keyboard layouts are controlled by IPREFS.

Save and exit (<ESC-X>). The ED startup disk should now be complete.

The proof of the pudding ... Ensure the ED disk is located in the internal disk drive and reset your system by turning the power off and then on again after *at least* 20 seconds, or more simply by pressing <Ctrl><Amiga><Amiga>.

If all is well, the ED disk will boot the system and finish off by creating a new file called 'NewFile' in the 'TextFiles' directory. If booting does not go according to plan, note what command resulted in the error. You will have to re-boot the Workbench disk and re-edit the Startup-Sequence file from the ED disk (best done by copying it into the Ram Disk first of all). If you are sure that the file is correct, check that you have copied all of the commands across into the C directory of your ED disk.

Finally, when you quit from ED you are deposited into a standard Shell window. The Shell is not installed as part of your ED disk, however, how to do this is outlined below. Remember that only the commands copied into the C directory are available to you at this point.

Shell-Startup

The command NEWSHELL opens a Shell window using a special file called Shell-Startup. Like the Startup-Sequence file this must be located in S:. To revamp the ED disk, re-boot the Workbench disk and copy the various AmigaDOS commands as outlined above to the Ram Disk for ease of use.

These files, plus the various others introduced at this point, are discussed later. However, this shouldn't stop you copying them across and using them for now! Copy the following from S: to the 'S' directory on the ED disk:

Shell-startup

Copy the following from the 'System' directory of the Workbench disk to the same directory on the ED disk:

CLI

Shell

The final file to copy is the 'Shell' file from the root directory of the boot disk (SYS:) to the ED disk.

The final task required is simply to add a few extra lines to the Startup-Sequence script. These are the RESIDENT, MOUNT, NEWCON and ENDCLI commands and the final startup-sequence file.

You should modify your Startup script thus:

```
ADDBUFFERS >NIL: DFO: 10
COPY ENVARC: RAM:env all quiet noreq
ASSIGN ENV: RAM:env
RUN >NIL: IPrefs >NIL:
ECHO "Customised ED Disk."
SYS:System/SetMap GB ; Workbench 2.04 only!
NEWSHELL CON:10/10/500/100/Shell
CD Textfiles
RUN SYS:MEMACS NewFile
ENDCLI >NIL:
```

Save and exit (<ESC-X>)

On booting with this disk you will be left with a Shell window, and a new ED file called NewFile will be open and ultimately saved to the EDfiles directory.

Debugging

The term 'debugging' is used to sum up the process of finding out and correcting problems and errors that may occur when a program is run for the first time. Problems and errors associated with program are termed 'bugs'.

Like customising disks, the process of debugging is a personal one and that's not a cop out – honest! Effective debugging relies on experience and that cannot be taught, only acquired. However, there are a couple of ground rules that can be introduced now which should stand you in good stead.

Always take note of what is happening on screen after you have set a program into motion by whatever means. Note at what point the problems occur and also take heed of any error messages displayed by the Amiga. This is normally the point at which to start your investigation which should, in the first instance, be a re-check of what you have typed.

If the startup does not create much screen output and goes dormant before falling over – then make it produce screen output. This is best done at various points in the program with ECHO commands which print the name of the command being executed – or part of the program if it is rather long. For instance:

```
ECHO "Making Shell segment"
```

This should help you narrow the location of the bug, and these commands can be removed when the program is running as expected.

Finally, if all else fails, take a break. Programming is exciting, infuriating and satisfying. The most experienced programmers in the world make mistakes and can spend hours locating 'stupid' errors. Case sensitivity is a classic mistake which, thankfully, does not apply to most of AmigaDOS – oddly enough, it does affect the TOOLTYPES array on Workbench info. Anyway, an hour away from the keyboard is relaxing, refreshing and does wonders. Staying at the keyboard in such circumstances will make you want to tear your hair out. Not surprisingly, neither of the authors have much left – perhaps computers should carry a government 'pate' warning: computing can seriously reduce cranial hair...

Inside The AmigaDOS Disk

As promised earlier in this chapter, you will find a brief description of the function of each of the directories that are to be found on the Workbench disk. This will give you a better understanding of the way in which system disks are organised, which in turn should aid your understanding of the examples given above.

Before you start, a quick word about how this is organised. We've shown the path of each directory referenced from SYS: and the name of the logical assignment – if any – it normally receives by default. Remember SYS: is an assignment, it defaults to the root directory of the boot disk, but it can be anywhere. The default assignments are listed first, followed by the other (optional directories). Advanced users can alter the assignments or add new ones. See ASSIGN for more information.

The C Directory

Assignment: C:
Path: SYS:C
Contents: AmigaDOS commands

This is used to house the transient AmigaDOS commands. These are commands which are loaded, executed and unloaded from the system each time they are invoked. The same applies to all commands, however, C: is *always* in the search path (see PATH) so AmigaDOS always knows where to find its commands.

The Devs Directory

Assignment: DEVS:
Path: SYS:Devs
Contents: Hardware device drivers

The various device drivers are held here, including the drivers to enable AmigaDOS to communicate with printers and serial devices etc. Device drivers communicate with the hardware at a very low level they all have the name nnnn.device. Also found here are the device mount entries or the single Mountlist up to 2.1: see *Mastering AmigaDOS 3 Reference*.

The Clipboards Directory

Assignment: CLIPS:
Path: RAM:Clipboards
Contents: Vary

This directory is reserved for applications programs. It's an area of temporary storage where applications can store just about anything, from text to art file and sampled sound. The clipboard.device maintains this directory, all you have to do is make sure the CLIPS: assignment has been made in the Startup-sequence. For most applications, it's best to slap this in RAM: for speed.

The Keymaps Directory

Assignment: KEYMAPS:

Path: DEVS:Keymaps

Contents: International keyboard layouts

This directory is managed by the system, so you don't need to worry about it too much. It must contain the keymap necessary for your particular keyboard – UK users, for instance, need the GB keymap. Don't just assume the right file is in there – make sure. Most users will only need one keymap file in this directory. It must exist, however, because this is where the IPREFS and SETMAP commands look.

The Printers Directory

Assignment: PRINTERS:

Path: DEVS:Printers PRINTERS:

Contents: Printer drivers

Like keymaps, most users only need one file in this directory – the one which belongs to their printer. This file is determined *but not set by* Preferences. If you change the Preferences printer, then you must update this directory. If you don't have a printer this directory can be left empty. It should not be deleted.

The Fonts Directory

Assignment: FONTS:

Path: SYS:Fonts

Contents: Typefaces for applications programs

FONTS: contains only Topaz with remaining fonts plus the new Intellifonts being available via the Fonts disk. See the chapter on Intellifont/Fountain for more details on access to these.

The L Directory

Assignment: L:

Path: SYS:L

Contents: Device handlers

This is the handler library directory and it contains several important files to deal with the handling of virtual devices. These files are code segments loaded by MOUNT and started as tasks – *do not* tinker with them. Not all handlers are required by all applications.

The Libs Directory

Assignment: LIBS:
Path: SYS:Libs
Contents: Optional libraries

This is the applications library directory. Libraries are just groups of functions designed to manage specific parts of the system. The essential ones like the dos.library and graphics.library are part of Kickstart. The others are kept in here until something needs them. This is highly efficient and makes upgrading the system software a very simple task. The SETPATCH program is used to fix any bugs in the Kickstart libraries. Bugs in the external libraries can be corrected by replacing the library code. Many applications use private and Public Domain third-party libraries. Some of these have since been adopted as part of Workbench – asl.library and commodities.library are two examples.

The Classes Directory

Assignment: LIBS:
Path: SYS:Classes
Contents: Gadget libraries and translations

The contents of this Workbench 3 directory are a odd. Currently it includes two sub-directories:

Gadgets: drivers for the new colourwheel and gradientslider gadgets. You can find examples of both these in the Palette Preferences tool.

Datatypes: translation data for the multiview program. Each of these gives MultiView the ability to read or display a specific type of data: for instance ILBM is for graphics and 8SVX is for sounds. AmigaGuide may be required if an application uses AmigaGuide help (few do at the time of writing).

The S Directory

Assignment: S:
Path: SYS:S
Contents: Script and configuration files

This directory is used to house the various startup files used by the system when it is booted. The only required file is Startup-Sequence (although even that can be thrown away if you're not using DOS). The rest of this directory is dedicated to scripts and configuration files for the Shell, ED and so on. Third-party software often uses this directory to store their default setups. Note: EXECUTE looks here if it can't find the file to execute in the current directory.

The T Directory

Assignment: T:
Path: Usually RAM:T
Contents: Vary

This is used as a home for any temporary files created by software. It should be empty as software should carry out its own housekeeping to delete files from here when they are no longer required. This is not always true, of course – scripts can use the T: assignment to store small temporary work files. EXECUTE uses this directory to store the temporary command file created when a script using .KEY is executed.

The Prefs Directory

Path: SYS:Prefs
Contents: The preference tools on hard disk systems

Prefs is short for Preferences and this directory contains information used by Amiga to define how and what type of display you require, etc, etc.

The Environment Archive

Assignment: ENVARC:
Path: SYS:Prefs/env-archive
Contents: The preferences settings

This directory and associated assignment were introduced to cope with the greatly enhanced range of alterable settings provided by the Preferences tools. The safest way to deal with this directory is to copy the whole lot!

The System Directory

Path: SYS:System
Contents: System command files

As the name implies, this is used to house the various system related files including the CLI interface and commands such as FORMAT.

The REXX Directory

Assignment: REXX:
Path: SYS:S (by default)
Contents: ARexx script files.

This directory assignment is read automatically by ARexx when you enter a command such as:

1>RX GENSUMM

AREXX programs are ASCII text files and for this reason are stored in the S: directory and assigned REXX:. All ARexx programs should be suffixed ".REXX" to separate them from script files.

The WBStartup Directory

Path: SYS:WBStartup

Contents: Autostartup programs.

Icons are dropped in here by Workbench users when they need to get some software initialised. The ARexx master process usually lives in this directory, but that is one of the very few programs that work well from it. If an application launched by Workbench in this way does not exit after a short period of time, Workbench complains. A better solution for AmigaDOS users is to edit the User-Startup.

The Expansion Directory

Path: SYS:Expansion

Contents: Hardware drivers.

Icons are dropped in here for hardware that requires manual mounting by BINDDRIVERS. In practice, most hardware either Autoconfigures or uses separate drivers. This directory was intended to convenience users but very few programmers bother with it.

The REXXC Directory

Path: SYS:REXXC

Contents: ARexx Commands.

The ARexx commands are stored in here –except the ARexx Master process. ARexx commands are only required if you are running ARexx scripts either from Workbench on application; otherwise they are quite superfluous.

The Trashcan Directory

Path: SYS:Trashcan

Contents: Vary

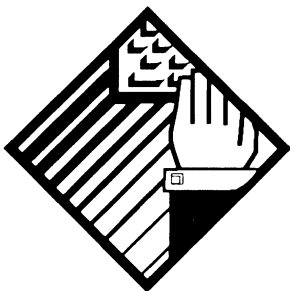
Files and directories which are to be discarded can be moved here. It is only of use from the Workbench. Shell based disks should not bother with this directory as a rule.

The Utilities Directory

Path: SYS:Utilities

Contents: Utilities

This should be a fairly familiar directory by now. It contains the standard Utility type programs.



41: Multi-User Machine

While we were compiling the original Mastering AmigaDOS 2 book, we often found we both needed to use the same machine at the same time. The simplest method would have been to buy another A3000, but at the time Amiga 3000s were in short supply and very expensive.

For the sake of speed, the only solution was to connect the two machines back to back using AmigaDOS 2 on both of them. Impossible? Well it might seem like that, especially when you realise the second machine wasn't even another Amiga!

It needn't be. In fact, just about any small computer system you happen to have lying around can be pressed into service. You will need the following items.

Remote Machine (Parasite)

- A small computer with monitor and serial interface.
- A null modem cable with connections for the Amiga (see note).
- A simple terminal package for the second micro.
- An assistant.

Host Machine (File Server)

- Any Amiga.
- AmigaDOS 1.3 or better.

Note: The Amiga 1000 and A2000 models have a non-standard serial interface. Although it is possible to use these, you must get the correct lead. This does not affect the B2000 machine.

Typically you can use: other Amigas, Atari STs, Amstrad PCW with serial option, MTX 500, Acorn BBC B, and most cheap PC clones. If you do not have any of these, you may find a cheap CP/M machine at amateur (HAM) radio rallies. HAMS use these for packet radio but a suitable second-hand setup can be had for as little as 30 quid.

Whatever you do, try not to spend too much money and read this chapter in its entirety before parting with any cash. Using the Amiga in this way is simple but raises some interesting problems. A debugging chart has been included at the end of this chapter – and the programs do work, but please read the following caution carefully.

Never, ever, plug or unplug a serial lead without first switching both computers off. Failure to comply with this caution can cause serious damage to your hardware and bank account. The authors and publisher cannot accept claims for damage or injury, however caused, arising from following the instructions detailed here.

Getting Started

Interfacing two machines in this way is relatively easy, but once the two machines are connected, you must decide on a serial protocol. That is the way the machines talk to each other. It's no use having the Amiga speaking Swedish to the remote and the remote trying to answer back in ancient Greek. The result will look something like this:

Take me to your leader

Take me to your leader

Protocols in serial communications are just like language, and so long as the two speakers agree to speak the same tongue there is no problem. There are four main parameters to consider here:

Baud rate	the transmission speed.
Word length	the number of bits in each data byte.
Parity	error checking.
Stop bits	the number of bits to send after each data byte.

Errors in these give rise to garbage.

For the purposes of this type of communication, a fairly fast data rate is required. In practice some setups refuse to work at speeds exceeding 4800 baud and the best speed can be achieved through

trial and error. The other parameters should be set to 8 data bits, no parity and 1 stop bit. (Comms nuts refer to this as 4800-8N1.) You should set these at the Amiga end with the Preferences tool. However, if your remote computer's terminal software does not support this (for instance a Prestel™ emulator) you can try 7 data bits and even parity for both.

Testing 1-2-3

Once the computers are configured correctly you can perform the initial test. Open a Shell on the Amiga and enter the following :

```
1>ECHO >T:msg "Hello World"
```

```
1>COPY T:msg to AUX:
```

alternatively you can enter:

```
1>ECHO >AUX: "Hello World"
```

However, the latter method has been found to lock the AUX device, thus preventing two way communication. The remote computer will echo the message *Hello World*. If it didn't, follow the debugging guidelines. If it did, you are now ready to enter the world of the multi-user Amiga. Note, if at any time the Amiga or the remote terminal freeze, you may have to reset the machine.

Now enter:

```
1>NEWSHELL AUX:
```

```
1>
```

Nothing should happen at the host machine (your Amiga) instead the new Shell will start on the remote terminal. Its screen will look something like this:

```
New shell process 2
```

```
1>
```

Now ask your assistant to enter LIST on the terminal. This will provide them with a listing of your Workbench disk *or* the currently selected volume (disk). The LIST command should be made resident to avoid troublesome disk swapping if you only have one drive. This is a nuisance in any case, but when two people are sharing one machine it can become a nightmare of Orwellian proportions.

Problems, Problems...

Before going any further, it is worth noting this technique is anything but perfect, but at least it's cheap. The main problem is it only works with commands that only affect the Shell window. In other words, programs which rely on Intuition (and that includes

the ED editor) will not work. Any Intuition programs launched on the terminal appear on the host machine – usually when the operator is in the middle of something!

Actually, it can be quite amusing to install a beginner on the Amiga, launch a Workbench hack such as Viacom from the remote and watch them squirm! This is because the remote terminal is purely operating as a keyboard and screen, not as a separate computer. Because of this, it is not strictly correct to call the host Amiga a network fileserver. Nevertheless, there is a vast range of Shell-based commands, not just AmigaDOS ones, which do work correctly.

What Next?

At this point you should be able to control many aspects of your machine remotely from your old hardware, so that's all right. But what about sending messages between the two machines? In a real situation the two terminals could be rooms apart, so chatting is out of the question. Or is it? We've already mentioned the remote terminal is, in reality, just an extension of the existing machine. So, provided you can send messages between two Shells, you can send messages between the two machines.

Listings 1 and 2 are the command scripts used to communicate using a form of Email, that is: you leave messages for the other party which can be collected later. The two programs are almost identical so We'll just describe *Mail-2-Remote* here.

1. This first line is very important because it determines how the script will react to command lines. In this case, the script is given two parameters – a message and a filename. As you will see, the script usually determines its own filenames but you can override this feature by entering a name here. You must enclose the message body text in quotes (speech marks) or the script will fail. Typical examples might look like this:

```
1>mail-2-remote "Hi Bruce! Nice weather huh?"
```

```
1>mail-2-remote "Ol' man river" name=river
```

- 2...3 Set the bracket characters to { and } respectively. You may remember these default to < and > which conflicts with the re-direction operators used extensively in this program.
4. This line comes into effect if no value is supplied to the name (filename) parameter. The default name is *ItsForYou.hsts*.

5. This line (which makes use of our favourite command) is used to read and delete any new mail messages from the remote terminal. It's very complex so we'll break it down into its component parts:

5.1 list

The LIST command itself.

5.2 >T:ItsForMe{\$\$}

Names the destination script file which is created at 5.4. The file will be placed in the T: assignment (usually RAM:T) and called ItsForMe[XX], where XX is the process number of the current Shell; this allows the script to create a unique name for itself in the multi-tasking environment.

5.3 T:#?.rmt

Selects any mail messages in T: which have been sent from the remote. This is done using the AmigaDOS *everything* wildcard #? with the extension .RMT. The .RMT extension is added to every message written by *Mail-2-Host*. This program does much the same thing at step 10 using the extension .HST.

5.4 lformat "TYPE %s%s*nDELETE %s%s*n"

Creates a temporary script program to read and remove the current messages. Assuming the remote's operator has used the name option to create a message called hello, the resulting program would look like this:

TYPE T:hello.rmt

DELETE T:hello.rmt

6. This runs the script created at Step 5, reading any pending messages and deleting them afterwards. This is done to prevent a lot of useless files jamming up the T: assignment.
- 7...9. These steps are used to determine if you have actually entered a message – if not, the program stops. This allows you to check for mail periodically. Thanks to the nature of the Amiga, it is possible to design a special version of the Mail-2-system which will periodically check for incoming mail after a short time – more of which later.
10. This determines if a message file already exists with the same name. This means the other terminal has not yet read the message so you are given the option to leave or update it...

11. ...here. Note the default operation (if you just press Return at the prompt) is to leave the message untouched. As an aide-memoire, the y/N? prompt indicates this with a capital N.
12. This determines what happens at the ASK statement (Step 11). ASK sets the WARN condition if Y was pressed and clears it otherwise. By negating the action of IF, with the NOT switch, the script branches to 14 when you enter Y.
13. If control reaches this point the script terminates immediately.
- 14...15 Close the two IF...ENDIF constructs opened at Steps 10 and 12 respectively.
- 16...18 These lines create a compound message using the current time to show when the message was posted. This allows the receiving party to determine when the message was posted. Note the use of the append re-direction operator >>. This tacks the output of any command onto the end of an existing file.

Opening the Mail

This system for reading your messages works, but it would be nicer to get the machine to read them for you and Listings 3 and 4 were devised to do just that. They work in a similar fashion to the others, but take more advantage of the Amiga's multi-tasking properties.

1. This defines the argument template for this script. Only one argument can be supplied here, the time delay in minutes. Unless you have a fast machine and a hard disk, do not set this below 10 minutes. If no time limit is supplied, the script will check messages every 30 minutes (defined at Step 4).
- 2...3 Redefine the bracket characters as before.
4. Sets the variable for the time limit if none is supplied to 30.
5. This label is supplied so the script has somewhere to return to during looping. In fact, this script has been designed to loop continuously until stopped; more of that shortly.
6. This line is identical to the one used in the Mail-2-Remote program. It displays and removes the current mail messages.

- 7...9 Check for the existence of the StopItNow environment variable. Actually, this could have been a temporary file placed anywhere, but it is more convenient here. Note this line is identical in both versions of this program so once the variable is set, both users will cease to get update messages.
10. Here the script is executed as a new process with the RUN command. Now this might seem a little strange, but a minor bug in AmigaDOS EXECUTE causes the SKIP at Step 12 to fail if this is not done!
11. This puts the script to sleep for the predetermined time – a default of 30 minutes in this case. You might want to change this to seconds (by substituting SECS for the MINS switch) but don't forget to change the default time value. If the delay is too short the machine tends to get very tied up attempting to read messages which simply aren't there.
12. After the WAIT at Step11 times out, this forces the script to go back and do it all again.

Using HostRead

This script is an unusual one because it is designed to multi-task – even though it starts its own tasks too. There are two ways of doing this, the obvious way:

```
1>RUN EXECUTE HostRead
```

and the less obvious way:

```
1>NEWSHELL
```

```
1>RUN EXECUTE HostRead
```

In the second case, you start a new Shell process before starting HostRead. This allows you to work as normal without the messages suddenly appearing in the middle of your screen. Note however, the second technique cannot be used on the remote terminal because the new Shell window will still appear on the host's terminal – phew. Also, once this script has been started, it can only be stopped by setting the StopItNow environmental variable. You can do this thus:

```
1>ECHO >ENV:StopItNow "ON"
```

The actual value is arbitrary, but once this has been done, the program will halt during its next loop.

Commanding Scripts

To get the best effect from the scripts described here, you should set the S protection bit of each so they work like real commands. Assuming you have stored the scripts in the S: directory assignment, the following will do the trick:

```
1>SPAT PROTECT S:#? +S
```

The command SPAT used here is not in the C: directory, it is a script in its own right which is used to add single pattern matching facilities to commands which lack them. Don't worry about how SPAT works, it just does.

Listing 1 – Mail-2-Remote

```
1. .key message,name/k
2. .bra {
3. .ket }
4. .def name ItsForYou.hsts
5. list >T:ItsForMe{$$} T:#?.rmt lformat "TYPE
   %s%s*nDELETE %s%s*n"
6. execute T:ItsForMe{$$}
7. if "message" EQ ""
8. quit
9. endif
10. if exists T:{name}.hst
11. ask "Message pending. Delete y/N?"
12. if not warn
13. quit
14. endif
15. endif

16. echo >T:{name}.hst "Posted on: " noline
17. date >>T:{name}.hst
18. echo >>T:{name}.hst "{message}"
```

Listing 2 – Mail-2-Host

```
1. .key message,name/k
2. .bra {
3. .ket }
4. .def name ItsForYou.rmts
```

```

5. list >T:ItsForMe{$$} T:#?.hst lformat "TYPE
   %s%s*nDELETE %s%s*n"
6. execute T:ItsForMe{$$}
7. echo >>T:{name}.rmt "{message}"

8. if "message" EQ ""
9.   quit
10. endif

11. if exists T:{name}.rmt
12.   ask "Message pending. Delete y/N?"
13.   if not warn
14.     quit
15.   endif
16. endif

17. echo >T:{name}.rmt "Posted on: " noline
18. date >>T:{name}.rmt
19. echo >>T:{name}.rmt "{message}"

```

Listing 3 – HostRead

```

1. .key time
2. .bra {
3. .ket }
4. .def time 30

5. Lab Start
6. list >T:ItsForMe{$$} T:#?.rmt lformat "TYPE
   %s%s*nDELETE %s%s*n"

7. if exists env:StopItNow
8.   quit
9. endif
10. run execute T:ItsForMe{$$}
11. wait {time} mins
12. skip Start BACK

```


Listing 4 – RemoteRead

```
1. .key time
2. .bra {
3. .ket }
4. .def time 30

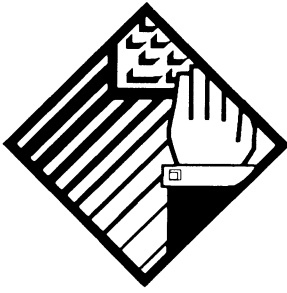
5. Lab Start
6. list >T:ItsForMe{$$} T:#?.hst lformat "TYPE
   %s%s*nDELETE %s%s*n"

7. if exists env:StopItNow
8.   quit
9. endif

10. run execute T:ItsForMe{$$}
11. wait {time} mins
12. skip Start BACK
```

Debugging Chart

<i>Problem</i>	<i>Possible cause</i>	<i>Remedy</i>
When connected to the remote machine, I get a load of garbage on the display.	Serial protocols not matched correctly.	Check protocols (baud rate, data bits, parity, stop bits) on both machines.
The <i>New Shell process...</i> message appears but I cannot get the Shell to respond.	The Shell has started on the host machine with output re-direction to AUX:	Try again using: 1>NEWSHELL AUX:
Every character I type appears twice, ie: 1>DDIIRR	Local echo enabled on remote terminal.	Either disable the local echo facility or use the CHAT (split window) function if the software supports it.
When I enter a command (like STATUS) it returns a message like <i>Unknown command SAU</i>	Two (or more) Shells are sharing the AUX: device and characters are getting lost.	Enter: EENNDDCCLLII <Space> <Return> If that fails, you must reset the host machine and start again.
The first command I enter works, but then it all falls apart.	The Amiga is receiving carriage return characters (ASCII 13) in addition to the line feeds (ASCII 10) it expects.	Set the output carriage return (CR) translation to LF only.
Some commands return chopped up results with odd characters missing.	The two machines are not handshaking correctly. This happens at speeds in excess of 4800 baud even with XOn/XOff handshaking.	Reduce the transmission speed rate to 4800 baud or less on both machines or use RTS/CTS handshaking if your serial lead and hardware support it.



42: Pipes

Implementing and experimenting with a dual-user system is fun. In Chapter 41, we described some file-based messaging programs and in this part, we'll take that a step further. You may recall those programs waited for a specific time and checked for any pending messages. However, it would be much nicer if messages could be instantly displayed – like the chat system found in better BBSs; CIX for instance. Although this is possible using pipes, there are a few minor limitations which are due to AmigaDOS's scripting language.

Using Pipes

Before going into the complexities of pipes and piping in general, the simplest way to consider a pipe is to think of it as a file on the Ram Disk. A pipe file can be created and read in exactly the same way as a RAM file. Try this:

```
1>COPY S:SPAT to RAM:
```

```
1>TYPE RAM:
```

AmigaDOS responds by copying the SPAT script into RAM and then displaying it in the current Shell window. Now try it using a pipe:

```
1>COPY S:SPAT to PIPE:
```

```
1>TYPE PIPE:
```

The result should be exactly the same. Although this is a useful way to show how simple pipes can be, it is not very useful. Now try this:

```
1>DIR PIPE:
```

AmigaDOS responds with an error. Why? Consider this example (with accompanying error) for a clue.

```
1>MAKEDIR PIPE:testing
Cannot create directory
Packet request type unknown
```

As you can see, directories are not supported by pipes. The error message (it varies with different AmigaDOS versions) has come from the pipe device – it does not understand what has been asked of it. The device does not support requests to create directories on it. Much the same effect can be gained by attempting to create a directory on another non-disk device such as the serial sub-system:

```
1>MAKEDIR SER:testing
Cannot create directory
Packet request type unknown
```

Common sense dictates this attempt is ridiculous, even though it seems almost logical. After all it works on many other devices such as RAM:, RAD: DFx:, DHx: to name just a few. Now try this:

```
1>COPY S:SPAT to RAM:
1>MAKEDIR RAM:SPAT/testing
Cannot create directory RAM:SPAT/testing
```

This error *Cannot create directory* occurs because you are effectively trying to create a sub-directory of a file. Files are stored in directories, not the other way round. This gives rise to the theory that, at AmigaDOS command level, pipes can be treated more or less like files.

Paying the Piper

A pipe is a FIFO (first-in, first-out) buffer. You drop little packets of information in at one end and they appear at the other end in the same order. This can be compared to a stack where the opposite is true and the last packet of data is always the first out (LIFO).

A stack can be thought of in several ways, but the most accurate analogy probably lies with the bullets in a rifle magazine. Bullets are added one at a time, and pushed down against a spring. As the bullets are removed either by firing, or manually, they are removed in reverse order. Figure 42.1 shows this more clearly.

Pipes (FIFO buffers) seem like confusing concepts when applied to computers but most of us use them in our everyday lives. When you stand in line at a bus stop, you are in a pipe buffer. The first passenger in the line is the first to board the bus: first in the queue,

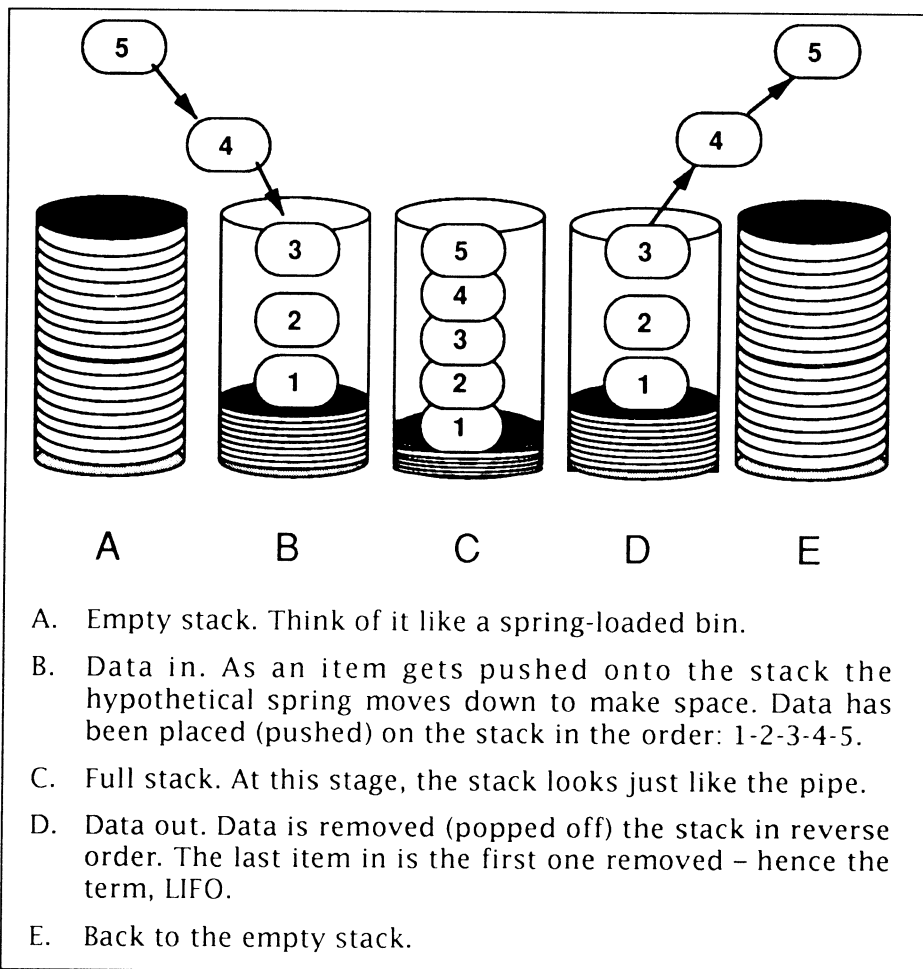


Figure 42.1. In a stack (LIFO) buffer, the last item in is always the first one out.

first out of it. When the bus is full, the person at the front of the queue has to wait until the next bus comes along, but will still be first on when it does

Pipes work in exactly the same way. Think of each passenger as an item of data waiting to be processed. When the device is ready to receive data (a bus arrives) data packets (passengers) leave the queue. If the device becomes full, the flow of data stops until the device signals it's ready to receive more data. Also, just as most people will refuse to join a very long queue, pipe buffers can get filled up. If this happens, the pipe makes the sender wait until some space becomes available. Figure 42.2 shows this in more abstract detail:

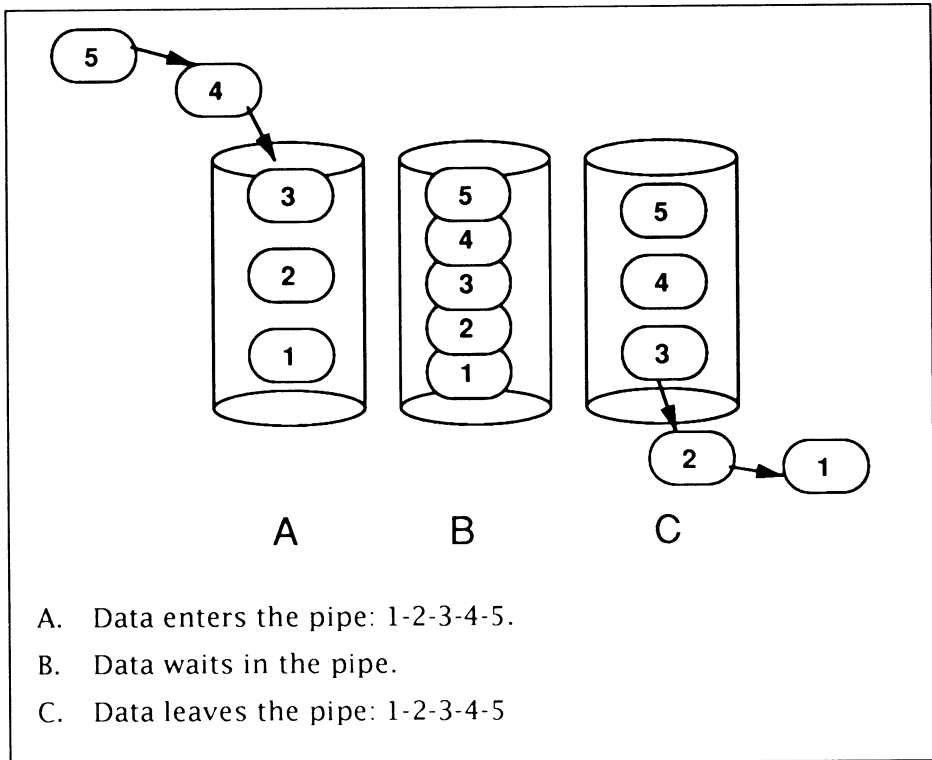


Figure 42.2. In a pipe (FIFO) buffer, data is let through in the order it comes in.

At a very simple level, pipes are a software device emulation, so as far as AmigaDOS is concerned, pipes behave exactly like all other devices. This means you can create, write to, and read from a pipe just as you would a normal disk-based file with three important differences:

Writing:

- To disk:** If a disk device is filled, the write operation stops and an error is generated.
- To pipe:** If a pipe device is filled, the write operation is suspended until some space becomes available and some data is removed from the pipe. Furthermore if a different process tries to write to a pipe, AmigaDOS returns an *Object in use* error. In fairness, it would be highly unusual for two processes to share the same pipe name.

Reading:

- From disk:** If a file does not exist, an error message is generated.
- From pipe:** If a pipe does not exist, the read operation waits until some data is written to the pipe.

Random Access:

- Disks: Support random access so a file pointer may be positioned and data read from any point.
- Pipes: Do not support random access. This does not affect the vast majority of users, but programmers should note this means the device does not support *Seek* functions.

Naming Pipes

AmigaDOS does not support unnamed pipes. In the first example above:

```
1>COPY S:SPAT to PIPE:
```

the pipe gets its name from the source file. However for normal piping operations it is good practice to give each pipe a name. Since pipes are temporary, that is they only exist until they have been read, a single letter will usually be sufficient. Therefore a typical pipe operation might look like this:

```
1>COPY RAM:MyFile to PIPE:A
```

```
1>TYPE PIPE:A
```

All this theory is fine, but it doesn't really explain what pipes are for. To use the jargon, pipes are intended for inter-process communication. Translated to English they offer a means of exchanging data between Shells and this is where they become invaluable in the dual-user environment. Using pipes, the new chat system is very short but still requires a script. Users with one machine can try these examples by running two Shells at once. In these examples, the *host* Shell's prompt is 1> and the *remote* Shell's prompt is 2>. Enter the two scripts (Listings 1 and 2) and set the S protection flag on each like this:

```
1>PROTECT HOST-CHAT +S
```

```
1>PROTECT REMT-CHAT +S
```

Now enter the following. If you are trying this on a single machine, you will probably find it easier to open two Shells before proceeding. Also, make sure you enter the commands in the correct Shell window.

```
1>RUN HOST-CHAT
```

```
2>RUN REMT-CHAT
```

```
1>ALIAS CHAT COPY * TO PIPE:B
```

```
2>ALIAS CHAT COPY * TO PIPE:A
```


The first two lines start the pipe-based chat system as an asynchronous process. Once activated it cannot be turned off, but the operation is completely transparent. Note also, both programs must be run before an inter-Shell conversation can take place. The last two lines might seem a little strange, but rely on a feature of ALIAS whereby aliases are local to the Shell process in which they were created. In other words, CHAT in Shell 1 will execute the command:

```
COPY * TO PIPE:B
```

and CHAT in Shell 2 will execute:

```
COPY * TO PIPE:A
```

This version of COPY may seem unfamiliar because the source file is an asterisk – normally used as a wildcard character in other systems. Under AmigaDOS, the * used in this way refers to the Shell window; specifically, to take input directly from the keyboard and copy it to the named pipe. When this command is executed, all keyboard input is directed to the pipe. To return to the Shell, you must enter the EOF (End of File) sequence by holding down Ctrl and pressing \. This forms the basis of the chat system.

Assuming you have started the chat scripts in two Shells as detailed above, you can now start chatting. Try this:

```
1>CHAT
```

```
Hello World
```

```
Is anyone out there?
```

[Press Ctrl-\ here]

```
1>
```

All being well, that message will appear instantly in the other Shell window:

```
2>Hello World
```

```
Is anyone out there?
```

The same command can be repeated from the other Shell to reply or send another message. Note the prompt (2>) will not appear in the receiving Shell after the message. This is quite normal and does not affect the Shell's operation. Pressing Return in the receiving Shell will return a prompt. An interesting feature of AmigaDOS is that, once you start typing, the chat system is disabled – it won't interrupt half-way through entering a command line.

The scripts themselves are extremely simple and are listed and documented in the following pages.

Program: HOST-CHAT

Purpose:

1. **lab start**
 2. **type pipe:A**
 3. **skip start back**
1. Defines an arbitrary label for the script to jump to when it loops.
 2. Serves two functions. First, if a message is waiting in the pipe, it is displayed (typed) immediately. Second, if no messages are waiting, TYPE halts until one appears. This is an action of the pipe device, not the command.
 3. The script only reaches this point after a message has been posted to the receiving pipe (above) and displayed. After this happens the script is sent back, via the label at Step 1, and waits for the next message at Step 2.

More Pipes

Unlike MS-DOS and UNIX, AmigaDOS does not support unnamed pipes where the output stream of one command can be connected to the input stream of another. Hence the MS-DOS construct:

```
TYPE READ.ME | MORE
```

is not valid in AmigaDOS and has no direct equivalent. The bar | symbol is used in MS-DOS to signify a pipe. The nearest alternative is to do the command in two steps thus:

```
1>COPY READ.ME PIPE:A
```

```
1>MORE PIPE:A
```

or, alternatively:

```
1>TYPE >PIPE:A READ.ME
```

```
1>MORE <PIPE:A
```

Note: the ability to use unnamed pipes is documented as part of the ARP 1.3 release and is claimed to work with the Shareware Shell replacement, Command. ARP 1.3 users may want to try this out for themselves.

The MORE program exhibits different behaviour depending on whether it is launched directly or as a separate process via RUN. Try this:

```
1>MORE S:SPAT
```

```
1>RUN MORE S:SPAT
```

Notice in the first instance how MORE uses the current Shell window, but when RUN it opens a window of its own. This feature may seem pointless, but it allows MORE to be used over the serial port. Moreover, it also gives rise to a variation on the CHAT theme. Listings 3 and 4 form a *Chat* system which is designed to be launched by the remote system and initialise all the pipes. It cannot set up the CHATTO alias but this can be done in Shell-Startup, by adding the following lines:

```
ALIAS ChatTo COPY * TO PIPE:[ ]
ALIAS ChatNow RUN EXECUTE CHATTY
```

To start the chat system, the remote operator enters (for example):

```
1>CHATNOW MARK BRUCE
```

The host terminal then receives a message (via MORE) like this:

```
Chat system opened as: Host=MARK Remote=BRUCE
```

Mark (using the host Amiga) can now start chatting to Bruce:

```
1>CHATTO BRUCE
Hello Bruce!
```

and sends the message by pressing Ctrl-\ as described earlier. Similarly, Bruce can chat back to Mark like this:

```
1>CHATTO MARK
Hello Mark! wonderful system, huh?
```

Program: CHATTER

Purpose: Start CHATTY; handle messages on host machine.

1. .key NAME1/A,NAME2/A
2. Echo >T:qwe<\$\$> "Chat system opened as:
Host=<NAME1> Remote=<NAME2>"
3. More T:qwe<\$\$>
4. Run Execute S:Chatty <NAME2>
5. Lab Start
6. Copy Pipe:<NAME1> T:msg<\$\$>
7. More T:Msg<\$\$>
8. Skip Start Back

1. This defines the script's argument template. Two inputs are required from the user: the name of the host and remote machines. Since the /A template option has been used, both arguments must be supplied or the script will fail to run.

2. Creates a file in the temporary files assignment T: containing the startup message which will be displayed on the host terminal. The arguments surrounded with angle brackets will be replaced by the user's input. Therefore if the command line was:

1>RUN CHATTER SIMON FERGIE

the file will contain the message: Chat system opened as: Host=SIMON Remote=FERGIE. The filename is determined adding the process number to *QWE*. Therefore if CHATTER was running as process 3, the filename would be *QWE3*.

3. This is a trick which relies on the ability of MORE to recognise when it has been RUN-launched. Normally, MORE would display T:QWE on the remote terminal, but since EXECUTE has been RUN-launched, the script is also running as a process and any commands it contains are also running asynchronously. The end result is that MORE pops up as a window on the HOST machine (much to the surprise of unwary operators).
4. Not a lot of people know this, but it is quite legal to RUN-launch scripts from within scripts – even those which have been launched with RUN in the first place. That's what happens here, CHATTY is RUN-launched from CHATTER. It must be started in this way because, as you will see later, it never returns. CHATTY is passed one parameter, the name of the remote terminal. (The parameter was missing in the original listing.)
5. This label defines the start of a loop which is called endlessly. Like CHATTER, this script never finishes.
6. Copies the contents of the host pipe (if any) to a temporary file. This forces the script to pause until some data appears at the pipe and prevents the script from needlessly looping.
7. Immediately displays the contents of the temporary file. If MORE was used like this to display the contents of the pipe directly the script would not pause correctly.
8. Forces the script to jump back to the label defined at 5 completing the endless loop. The result is that the program waits until a message appears on the pipe, displays it, and waits for the next one.

Program: REMT-CHAT

Purpose: Handle messages on remote machine.

```
lab start
type pipe:B
skip start back
```

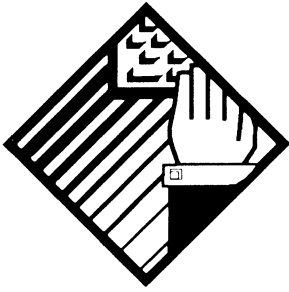
Program: CATTY

Purpose:

1. .key NAME
2. Lab start
3. Type pipe:<NAME>
4. Skip start back

This script is never executed directly, it works as a support script which is called by CHATTY. Typically, scripts of this size can be created by the script which calls them, but we'll leave that heady topic for another time.

1. Defines the argument template for the script. Although the argument would normally be required, it is not necessary to do that here since the correct syntax is assured by the calling script. The argument received by CHATTER is the name allocated to the remote terminal.
2. Defines a label which will be jumped to when the script loops.
3. Waits for data to be sent to the pipe on the remote terminal and prints it. Like COPY, TYPE waits for information to appear on the pipe before doing anything.
4. Loops the script back to Step 1, causing it to execute again. This script never stops, but because it's attached to an internal Shell (via RUN) it does not affect the machine's operation.



43: The Recoverable Ram Disk

Copious and effective use of the Ram Disk makes the use of AmigaDOS a much more pleasurable pursuit, even with the most basic of systems. However, its volatile nature means that every system reset requires the Ram Disk to be totally re-configured as its contents are not protected.

The device RAD: provides a new kind of Ram Disk which can be protected from erasure on a soft reset and can be used to re-boot the system in a much faster manner. Contents will be lost, however, when power is removed.

The standard MountList startup file located in the Workbench Devs directory contains all of the information required by AmigaDOS to be able to use RAD: with the use of the MOUNT command. MOUNT is examined in closer detail in *Mastering AmigaDOS 3 Reference*. Its main purpose in this case is to inform AmigaDOS that it is now in use.

Unlike RAM: the RAD: Disk is not a self adjusting device. It does not regulate its size to accommodate the files copied into it or deleted from it. The default size is 880k – enough to take the full contents of a system boot disk. This should be enough for most instances but it is possible to alter the size to suit your own needs and this is outlined below. If you only have a 1Mb Amiga then an 880k RAD Disk will not work due to lack of memory. In such cases you must reset the size as outlined later.

The RAD Disk will not be seen on the Workbench until it is activated, for instance selecting it as the current drive:

MOUNT RAD:

CD RAD:

When these two commands are executed the RAD Disk will be displayed and has the volume name RAM_0 and RAMB0 prior to this. This name can be changed, should you wish, by using the RELABEL command. For example:

RELABEL DRIVE RAD: NAME "RAD Disk"

DISKCHANGE RAD:

The command DISKCHANGE informs AmigaDOS that the name of a disk has been changed and this should be used as a matter of course after all relabelling to prevent AmigaDOS becoming confused. From the Workbench and indeed from AmigaDOS RAD can be treated like any other disk drive. From this point on the RAD Disk will survive any soft reset.

Try the soft reset and you will find that the RAD Disk has indeed survived the reset and its name now reflects that assigned to it above.

The Use of RAD:

Unless you have a large memory on your Amiga, (at least 2Mb) to enable you to set a RAD disk size of 880k, you will not be able to copy the entire Workbench disk contents into it. So the number of files that you can copy into the RAD Disk will be limited.

The main use, therefore, for the RAD Disk would be as a home for your files as you create them, thus ensuring their survival against any disasters.

Where you don't wish to have a full Workbench environment implemented and perhaps are using a smaller system, then the RAD Disk is a suitable home for it.

Because of the relative complexities involved in customising a startup file to boot in a practical sense from the RAD Disk, and also because there are many more commands we have to cover before we can do that, a practical worked example of a customised RAD Disk is left until later. A complete Workbench disk based on RAD: is possible.

Changing Size

The amount of memory allocated to RAD: is defined in the MountList file. This file contains information pertaining to the various devices that can be mounted under AmigaDOS. The section

that begins with RAD: contains the relative information and will display like this when loaded and found using a suitable editor such as ED.

MountList looks like this:

```
RAD:  
  Device = ramdrive.device  
  Unit    = 0  
  Flags   = 0  
  Surfaces = 2  
  BlocksPerTrack = 11  
  Reserved = 2  
  Interleave = 0  
  LowCyl = 0 ; HighCyl = 79  
  Buffers = 5  
  BufMemType = 1  
#
```

The amount of memory allocated to RAD: is defined by the value assigned to 'HighCyl' which is 79 by default. The value assigned to HighCyl, starting from 0, should be multiplied by 11k to obtain the size to be set. For example, setting:

```
HighCyl = 10
```

will create a RAD: device that is 110k large. To create a RAD: device that is 880k long use:

```
HighCyl = 79
```

The command INFO can be used at anytime to find out the current size of the RAD Disk.

Removing and Formatting

To remove a RAD Disk use the command:

```
REMRAD
```

When the system is next re-booted the RAD Disk will have been removed.

The RAD Disk can be formatted if so required though there is no need to do this unless the Fast File System is to be used. However, there is a further drawback here in that it is not possible to re-boot from the RAD Disk (though its contents will still survive a soft reset) and the following line must be added to the end of the MountList entry for it:

```
BootPri = -129
```


Bootting from RAD:

There are two ways to set up a bootable Ram (RAD:) Disk. The first involves modifying your Startup-Sequence as already described in an earlier chapter. The second is more usable since it gives you control over when RAD: is used. This saves having to answer questions like "Do you want to setup RAD?" in the Startup-Sequence. Machines should boot before they start asking questions and there's nothing more irritating than an Amiga which needs baby-sitting every time it's booted.

This solution is quick, simple and effective. So simple it could even be configured as a script! Let's assume you have just booted the machine and have the Workbench disk in drive 0. Now, open a Shell and type:

```
1>MOUNT RAD:
```

```
1>DISKCOPY FROM DF0: TO RAD:
```

That's that! The catch is you need at least 2Mb free to have a Ram Disk of this size and it takes up 880K (almost half) of that precious RAM. However, it does make the machine boot in about 30 seconds! This is useful if you're developing Startup-Scripts for example and you need to test them out. If something goes wrong, you just pop the original disk back in DF0: and boot from it. Mounted floppy disks always have priority over the rest of the system, including hard disks.

Multiple RAD Disks

Because of the way the RAD: device has been implemented it is possible to have multiple RAD Disks in operation, subject to having to have enough memory to do so. The process is straightforward and simply requires you to add extra device definitions to the MountList entry. For example, suppose you wished to create a second RAD Disk with a size of 123k. Using a suitable text editor such as ED make a copy of the RAD: definition and copy this immediately below it.

There are three simple changes to make:

1. Change the device name from RAD: to RAD1:
2. Change Unit = 0 to Unit = 1.
3. Change HighCyl to read 11.

The additional RAD entry in the MountList should now look like this:

```
RAD1:
```

```
Device = ramdrive.device
```

```
Unit      = 1
```

```
Flags      = 0
Surfaces   = 2
BlocksPerTrack = 11
Reserved   = 2
Interleave = 0
LowCyl     = 0 ; HighCyl = 11
Buffers    = 5
BufMemType = 1
```

#

Now re-boot the Amiga and from the command line you should be able to instigate two RAD Disk as follows:

```
MOUNT RAD:
```

```
CD RAD:
```

This displays the traditional RAD Disk. The second is forced into being with:

```
MOUNT RAD1:
```

```
CD RAD1:
```

Additional RAD Disks can be created in similar fashion. Just increment the RAD device number and the Unit number.

Removing Multiple Disks

The REMRAD command can be used to remove all RAD Disks. However, what if you wish only to remove one disk when there might be several? The answer is still the REMRAD command, but it now takes the form:

```
REMRAD <drive>
```

Where <drive> is the drive to remove. Thus to remove RAD1: use:

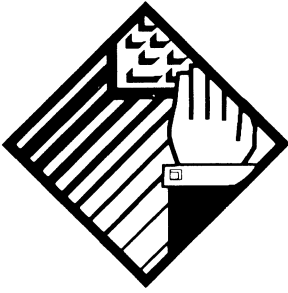
```
REMRAD RAD1:
```

Wot No RAD Disk!

Provided the MountList is correct a RAD Disk will only ever fail to display its colours if it does not have enough memory to run in. This usually manifests itself as follows. The disk can be mounted but when you enter:

```
CD RAD:
```

The prompt returns immediately and the relevant icon is not displayed on the screen. The solution is to reduce the size of the RAD Disk or get yourself more memory!



44: Commodities

The Commodities are a suite of programs that interact with the keyboard and mouse and allow you to alter or modify the manner in which they work. The Amiga has prior claim to their operation so they are unaffected by any application software that you may run.

At the time of writing there are eight Commodities Exchange programs supplied and these are located on the Extras disk. The programs are detailed briefly in Table 44.1 overleaf.

They are rather atypical of normal Amiga based programs in that four of them do not open windows while the remaining four can display windows but remain running even when these windows are closed! This latter technique is called *hiding*.

Exchange

This program can be thought of as the Commodities Exchange control panel, because it provides you with the tools you need to control what programs are operating and allow you to turn their operation on and off from within its window.

For most purposes you can call the program directly with a simple command:

EXCHANGE

although a better way of invoking the program is to use:

RUN EXCHANGE

Either displays the Commodities Exchange window.

<i>Name</i>	<i>Description</i>
Exchange	Monitors/controls CX Programs
AutoPoint	Automatically activates window under pointer
Blanker	Automatic screen blanker
ClickToFront	Brings window to front
CrossDOS	Facilitates data transfer to MS-DOS machines
FKey	Allows text assignment to function keys
MouseBlanker	Removes pointer during data entry
NoCapsLock	Temporarily disables Caps Lock key

Table 44.1. The Commodities Exchange Programs

As you run other Commodities programs they are added to the Exchange list and should you open the Commodities Exchange control panel again you will see them listed.

Exchange hangs around in memory so, unless you plan to kill it right away, the latter method of starting is a necessity otherwise you won't be able to use the Shell which launched it until you close it. To kill Exchange select the Quit gadget in the window.

We'll come back to the use of the Commodities Exchange program when we have had a look at what the Commodities programs themselves actually do. To use any of these you either need to relocate them by transferring them into the C: assignment on your AmigaDOS boot disk, into RAM: or use the long-winded method of giving the full path name as part of the RUN assignment. For example, to run AutoPoint from the Extras3.0 disk:

RUN Extras3.0:Tools/Commodities/AutoPoint

Equally you could add the Commodities drawer to the C: path list. Clearly the other two methods would be better to avoid disk swapping on a single drive system. If you want to stick to the above method it would be more convenient to give the path an alias:

ALIAS AP Extras3.0:Tools/Commodities/AutoPoint

Then typing:

RUN AP

will have the same affect.

Note that for the sake of simplicity, we'll assume you have already changed to the correct directory or set the correct path in the examples that follow.

AutoPoint

This is one of those tools that you'll either love or hate. It selects and activates the window over which the pointer sits. There is no need for you to have to physically select the window by clicking the mouse select button.

To set AutoPoint in operation use:

RUN AUTOPOINT

Note that AutoPoint does not open a window when you start it and it remains active until you Kill or Disable it using the Commodities Exchange program as detailed below. Alternatively you can kill it off simply by running the program again, ie:

RUN AUTOPOINT

Blanker

This Commodities provides a screen blanker which 'turns-off' the screen display if there has been no keyboard or mouse activity within a specified period of time.

To invoke Blanker use:

RUN BLANKER

On running a small window appears on the screen. This window displays the default blanking time which is normally 60 seconds. Thus if a period of 60 seconds passes without any keyboard or mouse activity Blanker blacks out the screen. You can change the blanking period simply by changing the Seconds setting. To set a period of two minutes delete 60 and type 120. Select the Hide button to run the Commodity but remove the window.

When a screen goes blank it appears black. The original display can be restored by pressing a key on the keyboard or mouse or just by moving the mouse. The Quit button can be used to remove the Blanker operation.

At first sight Blanker may appear to have no real value but in many respects it is an essential accessory if you leave your Amiga idle for more than a few minutes at a time. The idea is to prevent 'burn-in' on the monitor. Burn-in being the etching of characters into the screen phosphor. Although this affects mainly monochrome monitors, including the A2024, it can affect colour monitors as well.

As it runs in the background without any real hindrance it is worth running it though you may find it more convenient to increase the blanking period to 180 seconds or so.

You can change the default type period from 60 seconds to 180 if you wish by adding the Seconds option to the command line thus:

RUN BLANKER SECONDS=180

If you do not wish to display the Blanker window then redirect the command output to NIL: thus:

RUN >NIL: BLANKER

Alternatively you can use the POPUP extension to turn the window display off thus:

RUN BLANKER SECONDS=180 CX_POPUP=NO

To turn it on use:

CX_POPUP=YES

There is a hot key combination associated with Blanker – <Shift-F1>. If Blanker is running and you press the Shift and F1 keys together its window is displayed allowing you either to edit the blanking time or to kill it off altogether.

You can re-define the Hotkey using the POPKEY extension thus:

RUN BLANKER "CX_POPKEY=Alt F1"

Now Alt and F1 must be pressed to awaken Blanker. POPKEY extensions available are:

SHIFT	Either Shift key
Control	The Ctrl key
Alt	Either Alt key
F<n>	Function key number 'n'

ClickToFront

This accessory allows you to bring any window to the front simply by clicking the pointer in the selected window whilst depressing the left hand Alt key. This solves what can be an annoying problem when you have several windows open and have to sort through their depth gadgets. Used in conjunction with AutoPoint this is a very worthwhile utility.

This accessory does not open a window and is run simply:

RUN CLICKTOFRONT

By default the modifier key is the left hand Alt key (LALT) however there are a total of three modifier keys that can be used. These are the left and right hand Alt keys and the Ctrl key. The modifier key is defined in the icon's Tool Types. To change the key used first select the ClickToFront icon and choose Information from the Icons menu. The Tool Types gadget includes a definition similar to this:

QUALIFIER=LALT

To change the modifier key to the Ctrl key, click on the above text so that it appears in the text gadget immediately below. Delete LALT and insert CONTROL so that the line reads:

QUALIFIER=CONTROL

Select the Save gadget.

When ClickToFront is next run depress the Ctrl key while clicking in a window to bring it to the front.

When up and running ClickToFront can be disabled or removed via the Commodities Exchange program. Alternatively run the program again.

CrossDOS

CrossDOS allows you to read and write to MS-DOS format disks – those disks commonly used by PCs. This means that it makes it a simple matter to transfer files from a PC to an Amiga and vice versa. Of course, to make sense of the files you need to ensure that they are in a format that both machines understand. For example, consider a wordprocessor file on a PC. This can be read into a wordprocessor on the Amiga by one of three methods. First, by ensuring that you are using the same wordprocessor on both machines – for instance Protext. Secondly, by ensuring that the destination wordprocessor can recognise and translate the wordprocessor source file or thirdly, by saving the wordprocessor file as ASCII or Text from the source wordprocessor. Virtually without exception all wordprocessors allow you to save in ASCII or Text format and all wordprocessors can read this format. CrossDOS also supports it!

To use CrossDOS first run the command by typing:

RUN CROSSDOS

However, it will have no real effect until it is activated by mounting it thus:

MOUNT PC0:

However, this will not work as if the correct MountList will not be available and this will need to be appended to the MountList by copying the entry for PC0: and rebooting! There is an easier, more practical way of using CrossDOS – drag the PC0: icon into the Workbench DEVS:DOSDrivers directory from the DOS Drivers directory on the Storage Disk. In here you will also find other drive availabilities such as PC1: for an external drive.

Note that there is now a new device that refers to the PC disk called PC0:. This relates to the internal drive which may now be used for either AmigaDOS or MS-DOS disks.

You can copy files to and from an MS-DOS disk in the normal way simply by using the PC0: device name for the disk. For example, to copy a file called FILE1 from the Ram Disk to a PC disk in drive 0 use:

COPY RAM:FILE1 PC0:

MS-DOS file names are limited to seven characters plus an optional three letter extension. The three letter extension is separated from the main file name by a full-stop. For example, to copy a file called FILE1 from the Ram Disk to a MS-DOS disk renaming it FILE1.TXT (where TXT might indicate a text file) use:

COPY RAM:FILE1 PC0:FILE1.TXT

Like many Commodities CrossDOS has its own window and this can be displayed using the hotkey combination <Ctrl><Alt><C> once it has been activated. You can make this screen appear every time a disk is inserted into the drive by typing:

CROSSDOS CX_POPKEY=DISKINSERTED CX_POPUP=NO

FKey

This commodity allows you to assign a string of characters to a specific function key. It is intended to make repetitive tasks, like entering AmigaDOS commands, easier by assigning them to function keys. To launch the accessory and display the Fkey window use:

RUN FKEY

You can assign strings to every function key and shifted function key (that is the same function key pressed in combination with the Shift key), thus providing up to 20 functions for definition.

Note that the string '\n' is interpreted as Return and has the same effect as pressing the Return key. Notice that each function key definition is terminated by this character combination. If it is missed off then you will have to physically press the Return key after selecting the desired function key to implement the command.

Strings can be added simply by clicking the pointer in the required text gadget and typing in your text. For example to assign the AmigaDOS command to display the Ram Disk to function key F9 proceed as follows:

1. Display the FKey window.
2. Click in the text gadget and enter 'F9'.
3. Type:

DIR RAM:\n

When you have entered your desired strings select 'Save Defined Keys' from the menu to implement them and then the Hide gadget to remove the FKey window. To use a key to issue an AmigaDOS command first either select Execute Command from the Workbench menu or open an AmigaDOS Shell. Then press the desired function key.

If you want to add definitions via the command line when you run the commodity you can as this example shows:

```
RUN FKEY F1=DIR\N F2=STATUS\N SF3=MORE\N
```

Here F<n> is a plain function key and SF<n> the Shifted version.

You are not limited to using AmigaDOS commands – you can assign any text you require to the keys. Thus, if you are using MEmacs or similar and have trouble spelling certain words you could assign each to a function key and simply select the required function key at each point. In such a circumstance there would be no need for you to include the \n Return character at the end of each line.

The FKey strings that you enter into the text gadgets are not, by default, saved. Thus when you turn off or re-boot your Amiga any FKey settings are lost..

In all there are ten possible function key definitions (F1 to F10) and another ten Shifted function keys (SF1 to SF10).

We have seen that you can implement a Return in the strings by using the \n combination. There are several other such combination that implement a particular effect:

```
\r   Adds a Return
\t   Adds a tab
\0   Adds a zero
```

As with all Commodity programs the CX_POPUP option can be used to ensure that the Fkey window is not displayed on start-up:

```
RUN FKEY CX_POPUP=NO
```

The hotkey used to awaken Fkey is <Ctrl><Alt><F>. You can re-define the Hotkey using the POPKEY extension thus:

```
RUN FKEY "CX_POPKEY=Shift F1"
```

Now Shift and F1 must be pressed to awaken Fkey. POPKEY extensions available are:

SHIFT	Either Shift key
Control	The Ctrl key
Alt	Either Alt key
F<n>	Function key n

Finally, FKey assigns the function keys with their definitions before Intuition or any application software (such as a wordprocessor or desktop publishing package) has claim on them. Such software may well expect to use the function keys. If software you intend using does (if not sure then you can check by looking in the manual supplied with it) you should quit or disable the FKey accessory before running the software concerned.

The Use button in the Fkey window allows you to edit the function key strings and then make use of them. It does not save them for future use. Therefore, if you wish to edit the function key strings and use them only for that particular run of the program exit the window by selecting the Use and not the Save gadget.

MouseBlanker

A strange but very useful Commodity. When run it simply hides the mouse pointer until such time that you move the mouse again! This is merely a command of convenience (if there is such a thing) that was thought of to allow users to type text with the pointer out of the way.

RUN MOUSEBLANKER

NoCapsLock

This Commodity does not open a window. When run it effectively disables the Caps Lock key on the keyboard. Other keys such as the Shift key remain functional. When pressed and turned on Caps Lock forces all keystrokes to capital letters, thus if an 'a' is typed 'A' is produced.

With NoCapsLock in force capital letters can still be produced, by holding the Shift key down when pressing the appropriate letter on the keyboard.

To run the accessory use:

RUN NOCAPSLOCK

NoCapsLock can be disabled or killed from the Exchange window by running it again.

Using the Exchange Window

The Exchange control program window may well be self-explanatory to you at this stage, essentially it allows you to alter or unload currently selected Commodities program. For this purpose you can call the program directly thus:

EXCHANGE

or, if it has already been loaded you can bring its screen to life by pressing the hotkey combination <Ctrl><Alt><Help>.

Each commodity that is running is listed in the Available Commodities window which is on the left of the Exchange window itself. You can select any one of these simply by clicking on it once – details about the selected Commodity then appear.

The Exchange window displays four button gadgets. Show Interface and Hide Interface simply display or remove the window (where the Commodity has one) of the selected Commodity. The Remove gadget does just that while the selected Commodity can be made Active or Inactive by using the cycle gadget.

Commodity Tools

All Commodity programs share a common option that allows you to set the priority of the program. This option is called CX_PRIORITY. By using this you can determine which Commodity (and only Commodity) programs take preference over others. By default all are set to 0, but it can be changed by editing the value in the icons Information window thus:

CX_PRIORITY=<n>

where <n> is the new priority number.

By changing the priority setting you can determine which Commodity program reacts to a certain situation before another. Here's an example: of how to change the priority number of a Commodity at startup:

RUN FKEY CX_PRIORITY=5

Commodity programs that open windows, such as BLANKER and FKEY, have a further Tool Type associated with them. This is CX_POPUP and Table 44.2 lists the various keys that can be combined with the option.

If you should make an error in these definitions or use an unacceptable combination then the accessory will not load.

Here are a few more examples with brief descriptions:

CX_POPUP=no

This stops the program window from opening when the program icon is opened – the program itself will be activated though. This could be useful if you want to start the program from a script.

CX_POPUP=Numericpad7

This sets the popup key to the 7 key in the numeric keypad – the standard 7 key will have no effect.

CX_POPUP=Leftbutton RShift X

Defines a combination of pressing the Selection button on the mouse simultaneously with the right Shift key and X key.

<i>Qualifier</i>	<i>Key</i>
No	No window displayed
Alt	Either ALT key
RAlt	Right Alt key only
Shift	Either SHIFT key
LShift	Left SHIFT key only
RShift	Right SHIFT key only
LCommand	Left Amiga
RCommand	Right Amiga
Control	Ctrl
Numericpad	Use key on numeric key pad
Rbutton	Click mouse <i>menu</i> button
Leftbutton	Click mouse <i>selection</i> button
CapsLock	Caps Lock key

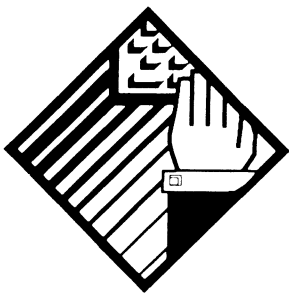
Table 44.2. Basic key combinations with CX_POPUP.

Last Words

Because the Commodities Exchange (CX) programs are useful it is common to include them in Startup-Sequences. If you do this then direct their output to the NIL: device, ie:

```
RUN >NIL: EXCHANGE CX_PRIORITY=NO
```

Note that they do not return error code conditions when an incorrect command line is entered. However, if in doubt, you can check they have launched correctly in scripts using STATUS.



45: Intellifont and Fountain

Intellifont (called Fountain in earlier releases) allows you to take advantage of Compugraphic outline fonts. Workbench 3 users do not get any example fonts, a retrograde step in our opinion. Workbench 2 users get three examples supplied on the Fonts disk. This program and the outline Fonts were first issued with release 2.04.

Fonts

The Fonts disk contains what we would call traditional Commodore bitmap or screen fonts and also outline fonts from Agfa's Compugraphic library. Notice that Commodore fonts are named after semi-precious stones. Fonts live on their own disk (Fonts:) because there is not enough room to fit them on the Workbench disk.

At this point, to improve the Amiga's potential and to take advantage of the extra space now available, new fonts were introduced.

Commodore fonts are all bitmaps: they are stored on disk in the form of pixels which are literally dumped to the printer when they are needed. This is relatively quick but it does not lead for a pleasant looking font especially when the font is scaled up. See Figure 45.1 overleaf

A new technology (to the Amiga) has been the introduction of outline fonts from Compugraphic called *Intellifont*. The data for these fonts is stored in the form of a set of co-ordinates which plots out the characters of each letter – a mathematical

representation of the font if you like. The great thing about this technique is that it allows any font to be printed in any size without loss of quality. The outline mathematics are simply scaled accordingly – simple for the Amiga. While the final output is massively improved the overhead is that the fonts take longer to produce.

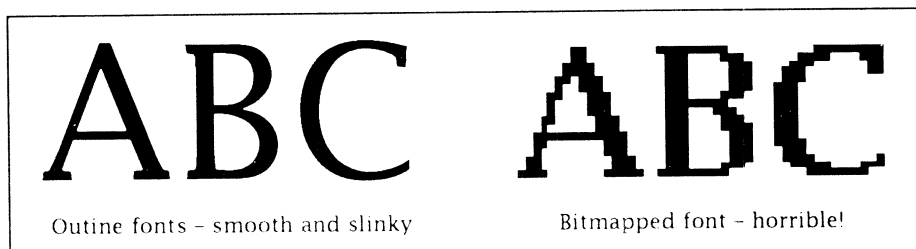


Figure 45.1 Outline fonts vs. bitmapped fonts.

Workbench 3 is not supplied with example outline fonts, and you will have to get any from other sources. Magazine coverdisks such as those on Amiga Shopper, CU Amiga and Amiga Computing are a good place to start. A better source in the long run though is PD libraries or specialist dealers. Commodore supply three example outline fonts on the Workbench 2 Fonts disk: CGTimes, CGTriumvirate and LetterGothic.

Using Intellifont

As we have seen, the Amiga has traditionally used bitmap fonts and this is primarily what the system software is geared for. For each font supplied on disk there is a separate file containing the data used to produce each available size for a particular style. Thus Topaz 8 and Topaz 9...

If you display the root directory of the Fonts disk you will notice that it contains a number of drawers named after the fonts. Look inside one or all of these and you will see that each contains a number of files each with a number – these are the bitmap fonts and the filename is their size. For instance:

DIR Fonts:Helvetica

produces:

11	13
15	18
24	9

One of the purposes of Intellifont is to convert outline fonts into files of this type which can then be used by existing Amiga software, the Fonts Preferences for example. But before this can

take place the outline Fonts must first be installed and this is another function of Intellifont. Incidentally, if you are not sure whether your application supports outline fonts consult the manual. Some such as Wordworth 2 will, others like Final Copy come with their own and others such as DPaint leave you stuck with the bitmap versions!

The installation and bitmap conversion processes are relatively straightforward but to do it you must have at least two floppy drives, a hard disk, or 2Mb of RAM. With an internal drive and less than 2Mb you cannot use IntelliFont. If your Amiga was supplied with a hard disk already fitted (such as the A600HD) then you might find that the fonts are already installed (check via Fonts Preferences).

Hard disk installation is straightforward – but there has been a great deal of confusion over getting the program to function on a dual floppy disk system. Certainly it created a large number of headaches for the authors of this particular tome!

However, the following solution does work! Here's a step-by-step guide to its use. You'd be advised to have a spare formatted Amiga disk available as well – you could name this ExtraFonts and use it to hold you installed Intellifont and bitmap disks. This can then ultimately be added to your assignment list.

Installing Outline Fonts

On machines fitted with Workbench 3 and a single drive, the best way to start Intellifont is to make a special version of the Workbench disk complete with a RAD Disk and Intellifont. Here's how:

1. Make a copy of the Workbench disk. (Use DISKCOPY or Workbench if you prefer.)
2. Delete the contents of the Utilities drawer – but leave the drawer itself on the disk. Remove the Format program from the System drawer too.
3. Copy the Intellifont program from the Extras disk to the System drawer of the duplicate Workbench disk.
4. Copy the RAD icon from the Storage:DEVS/DOSDrivers to DEVS:DOSDrivers.

That completes the disk – label it IntelliFont_Boot disk. You only have to do this once. The following instructions apply to each time you wish to start Intellifont.

1. Boot the machine with the IntelliFont_Boot disk.

2. DISKCOPY the boot disk to the RAD: device. You can do this from Workbench, remove the floppy and re-boot the machine from RAD.
3. Start IntelliFont and follow the on-screen prompts. You must replace the correct FONTS: disk when prompted because Intellifont requires extra bits (from _Bullet_outlines) to work.

Fountain on Workbench 2.0

Workbench 2's Fonts disk contains Version v37 of the diskfont.library. This must be copied to your boot disk as it has the necessary intelligence to use the outline font technology. The old version supplied doesn't.

1. Copy diskfont.library from your AmigaFonts disk into Devs: on your boot disk by using the following command:

COPY AMIGAFONTS:DEVS/diskfont.library SYS:DEVS

2. Re-boot your Amiga so that it has the new diskfont.library installed.

Start Intellifont which is located in the System directory of the Extras disk. Within a few moments the Intellifont window should appear and it will look somewhat similar to that illustrated in Figure 45.2.

Font Installation

In the large window located on the right hand side of the Intellifont window is a list of the existing fonts and typefaces. These are the ones on the Fonts disk supplied ready to use in bit-map flavour. What we now need to do is to locate the new fonts to be installed.

Click on the Outline Font Source gadget at the top of the window and an Outline Font Source disk requester are displayed. Use this to open the disk with your new fonts on. This presents a list of font directories, such as:

Caligula

CGTimes

Mira

To select a typeface for installation click the pointer on it. This should display a + to the left of the typeface name. When you have selected the typefaces you want to install, click on the Install Marked Typefaces gadget at the bottom of the window.

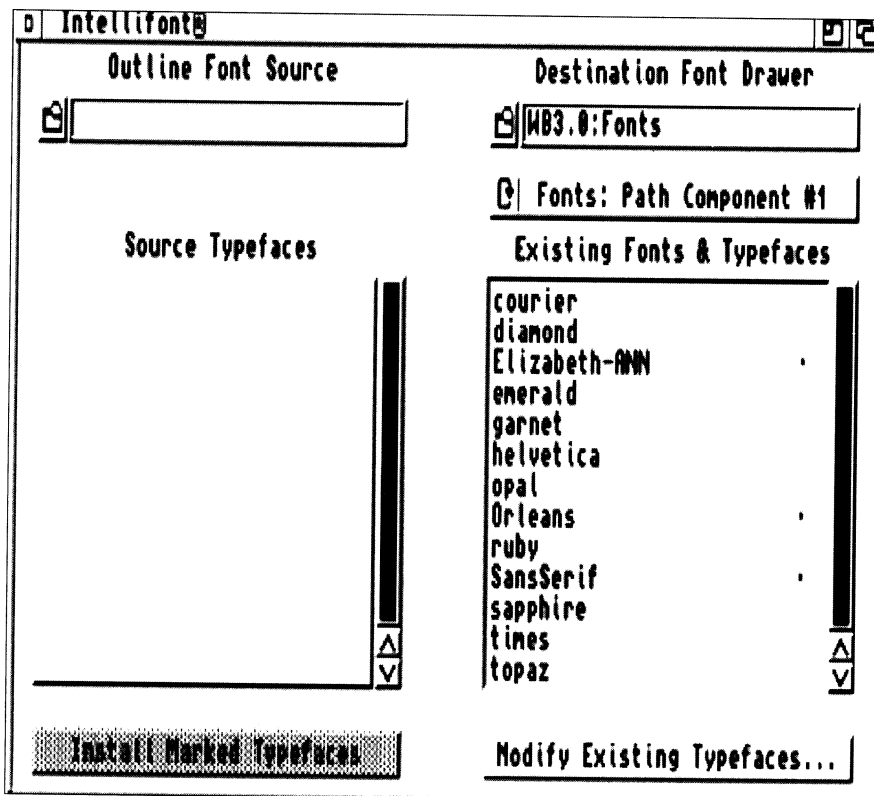


Figure 45.2. The Intellifont (Fountain) window.

Note however, you will *not* be able to install these fonts to the Workbench disk as there is no space available. Change the Destination Font Drawer gadget to – if you have pre-formatted it as suggested – the ExtraFonts disk. When you select the Install Marked Typefaces gadget the installation process starts. This takes some time depending on how many typefaces you plan to install. CGTimes on its own takes about 30 seconds.

On completion the Existing Fonts & Typefaces window will display the names of the installed Outline fonts. You have just installed the outline font(s) you selected and they can now be used.

Outline fonts listed in the Existing Fonts & Typefaces window are distinguished from any bitmap fonts that might be amongst the pack by having a small bullet (black dot) to the far right of their name.

If you have installed the CGTimes font you will find that the Fonts disk contains files called:

CGTimes.Font

CGTimes.otag

and a directory called:

_Bullet_Outlines

which itself contains a file called:

CGTimes.type

which is the outline information.

Bitmap Installation

Installation of bitmap equivalents of these new fonts such as CGTimes can now take place should you wish to use them on your Amiga and in particular within applications that don't support outline font technology. To proceed with this you need to be at the main Intellifont screen – the one displayed when the program is run as illustrated in Figure 45.2.

The outline fonts you wish to give the bitmap treatment should be listed in the Existing Fonts & Typefaces window (remember, these have been installed on the ExtraFonts disk if you have followed the procedure outlined above). Select the Modify Existing Typefaces gadget. This displays a window that looks similar to that illustrated in Figure 45.3.

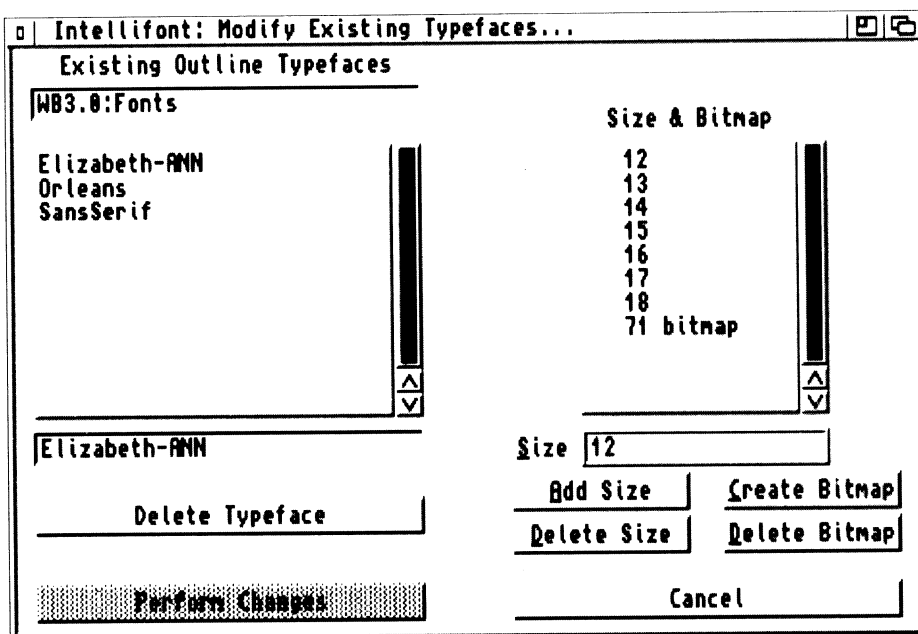


Figure 45.3. The Intellifont Modify Existing Typefaces window.

The main scrolling window on the left hand side of the screen presents a list of installed typefaces. Click on the one you wish to modify – the selected typeface being listed in the string gadget at the base of the window. To the right of this is the Size and Bitmap window. You can create new fonts sizes simply by entering a new

size in the Size gadget and selecting the Add Size gadget. Similarly you can delete a size by selecting it and then clicking on the Delete Size gadget. In either case the change or addition is implemented by selecting the Perform Changes button.

To create a bitmap, again select the typeface and then the size you wish to convert into a bitmap. It is advisable to convert only to bitmap those sizes you will actually need as each takes up disk space. Select the size you wish by clicking on it and then select the Create Bitmap gadget. The latter adds the word bitmap to the side of the font size in the window.

When you have selected the sizes you wish to convert to bitmap click on the Perform Changes button. This causes disk activity taking from half a minute to several minutes depending on how many bitmap fonts you are installing. The conversion is signified when you are returned to the main Intellifont window.

The new bitmap fonts are placed in an appropriately named directory and, if you examine the contents of this, you should find a file for each of the font sizes you have converted into bitmaps.

The fonts are now ready to use. You will need to use FixFonts to make the system aware that they are. If you are running these fonts from the ExtraFonts disk then do remember that you need to add this to the Fonts: Assignment list. You can do this by using the ASSIGN command along these lines:

ASSIGN FONTS:ExtraFonts ADD

The ADD options appends it to any fonts assignments present. Of course you can add this to your Startup-Sequence should you so wish.

If you are using a hard disk system then the installation of typefaces via Intellifont is very much simpler (assuming that they have not already been installed for you – check via the Fonts Preferences editor window).

Intellifont Environmental Variables

The question often asked at this point is “How does an application that supports outline fonts know what size of fonts I have available?” The answer is that the information, ie a list of sizes, is held in the Intellifont environmental variable:

ENV:Sys/Intellifont

When you use the Modify Existing Typefaces gadget part of the work done by Intellifont at this point is to alter the contents of this particular environmental variable to take in account font sizes that

have been added or deleted. If you have used Intellifont you will be aware from the sizes presented that the default list of sizes are 15, 30, 45, 60 and 75.

Rather than add or delete sizes via Intellifont you can make the changes directly by editing the variable contents directly in a text editor such as ED. The file is located in:

SYS:Prefs/Env-Archive/Sys/Intellifont

and must retain its ASCII format. The maximum number of fonts sizes you can hold in this environmental variable is 20.

In addition to the aforementioned variable Intellifont uses a second called *Diskfont*. This variable, stored at:

ENV:Sys/Diskfont

details the parameters used by Intellifont when it converts an outline into a bitmap font. The variable takes the following format:

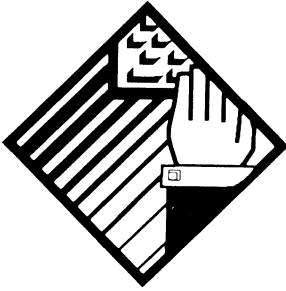
XDPI/N, YDPI/N, XDOTP/N, YDOTP/N

The first two parameters adjust the aspect ratio which by default is 1:1. These can be edited to take into account the display mode you anticipate using on your monitor. For example, for a Hires display mode, it would be visually better for the font to have settings of 100 and 50 to these two respectively. Using a text editor again this can be achieved by editing the file SYS:Prefs/Env-Archive/Sys/Diskfont as follows:

XDPI 100 YDPI 50

Note that you must specify both, not one without the other.

The other two parameters control the dot size percentage: this means the space a dot fills in relation to the screen resolution. The default value for XDOTP and YDOTP is 100. You need to make very large changes to these parameters to see any noticeable difference.



A: MEMACS

MEMACS is one of three text editors supplied with the Amiga. Pronounced 'Micro Emacs', it is the most sophisticated and versatile one. The other two being ED and EDIT.

MEMACS offers a wide and comprehensive range of features many of which are akin to those found on wordprocessors. However, it should be remembered that it is a text editor and is provided primarily as a tool for program development. As such it does not support features such as a spelling checker, a thesaurus and the like. That said, there is absolutely no reason why it cannot be used to "author" text.

The most striking features of MEMACS are that it has the ability to allow multiple file editing and macro (multiple command execution) facilities. The former is ideal for script development as it allows the documentation process to take place concurrently.

MEMACS presents itself as a full screen-orientated editor that can be operated using a full range of menus or by using key stroke sequences. This makes it easy to learn and ultimately very quick to use. During the learning process the menus can be used and dispensed with as the hot-key strokes are gradually learned. These will become second nature and are ultimately preferable as they allow you to keep your fingers on the keyboard without having to leave to drive the mouse.

Up and Running

You can launch MEMACS from both the Workbench and the command line. The icon for MEMACS can be found in the Tools Drawer and double clicking on this will launch the editor. From the Shell the syntax of the command to run MEMACS is:

MEMACS [<filename>] [goto <n>] [OPT W]

In its simplest form issuing the command:

MEMACS

opens a full screen window ready for use. Note that the screen window exhibits the features associated with over screen windows – such as Workbench, and can be dragged up and down or toggled with other screen windows using the back-front gadget in the top right hand corner of the screen.

If you have a file ready to edit – or if you wish to create a new named file – then this can be specified in the command line and will then be loaded straight into MEMACS. For example:

MEMACS Test

Here MEMACS searches the current directory and, if a file called 'Test' is located it is loaded. If no such file is found a new file is created. (This is not strictly true – it is perhaps more accurate to say that a potential new file is created. The file only becomes 'physical' when text is entered and then saved.)

If the file exists, you can move directly to a point in the file by specifying the line number after the filename, thus:

MEMACS Test 100

Assuming the file 'Test' exists then it will be loaded and the cursor will be placed at line number 100. If there is not a line number 100, then the cursor will be positioned at the start of the first line.

MEMACS has one option associated with it and this allows it to be opened in a Workbench window opposed to its own screen. The syntax for this is:

MEMACS OPT W

If this is used then the MEMACS window can be moved and manipulated in the same way that any other Workbench window can. For example, it can be re-sized and dragged around the screen.

Whatever method you use to launch MEMACS, the window that appears is divided into three distinct areas. The first of these is the Menu Bar – this runs across the very top of the window and provides access to the eight menus through which the various operations provided by MEMACS can be accessed.

The second region occupies the majority of the screen immediately below the Menu Bar – this is the text area into which you type and enter the contents of the file you are creating.

The third and final area is immediately below this at the bottom of the screen below a dashed line and this is the command area. From here the various MEMACS messages are displayed and additional information can be typed. The function of this region will become more apparent as we look at the various menus in turn. The dashed line also displays the name of the current file being edited after the 'MicroEMACS' title.

One item of jargon that you need to come to terms with, when using MEMACS, is the word 'buffer'. This simply means an area of memory that is set aside for something. When related to MEMACS we normally talk in terms of 'file buffers'. In this instance, a file buffer is an area of memory that is set aside and into which a file is read. For example, as we have already said, MEMACS allows you to work on more than one file. The only requirement for this is that you must have enough memory available to allow the files to be held in memory together. If two files were being edited together then two file buffers would be employed. File 1 would be held in one buffer and File 2 in a second.

Editing with MEMACS

The various commands associated with editing and moving quickly around file buffers can be found under the Move, Line and Word Menu details. However, you can move around the current screen of text by one of two methods:

- Using the cursor keys
- By clicking the mouse pointer once at the point you wish to reposition the cursor.

Scrolling quickly down through a file a window at a time can be performed by clicking the mouse pointer on the dashed line that separates the Text area from the Command area.

The maximum width of a MEMACS screen is 80 characters. If a line of text is longer than this a dollar ('\$') character is placed at the right hand end of the line. It is not possible to scroll across and see the text beyond this point though. Use the Open-line command (Line Menu) to split the excess text onto a new line.

The Project Menu

The first menu on the list is the Project Menu and this contains the various commands that are broadly associated with the overall manipulation of the file or files. There are twelve options in all, and these are listed in Table A.1 which also details the short or quick keys.

Note: When using any quick key shortcuts, take note of the case of the character(s) following either <ESC> or ^. Many commands are sensitive to this. If the letter is uppercase, ensure that the Caps Lock light is on before using the command. If it is a lower-case letter, then ensure that the Caps Lock light is extinguished.

<i>Option</i>	<i>Quick-Key</i>
Rename	^XF
Read-file	^X^R
Visit-file	^X^V
Insert-file	^X^I
Save-file	^X^S
Save-as-file	^X^W
Save-mod	^X^M
Save-exit	^X^M
New-CLI	^-
CLI-Command	^X!
Quit	^C

Table A.1. Project Menu Summary.

The first thing to notice is the way in which the quick key commands are implemented. Firstly, the caret symbol '^' stands for the Ctrl key. Wherever this is listed you should press and hold this down while pressing the next key. The Ctrl key can then be released. Note that some commands use two Ctrl key combinations and some only one, although a second key is also generally specified. We'll labour this point a little more as it can be confusing and may lead to undesired effects.

Looking at Table A.1 we can see that the Rename command can be invoked by pressing:

^XF

It is important to realise that the Ctrl key should only be held when the 'X' key is pressed. Once this has been done the Ctrl key should be released and the 'F' key pressed on its own. In this case, if you keep

the Ctrl key pressed as you press the 'F' key you will find yourself exiting from MEMACS! (Thankfully the file is saved prior to this exit so it's not disastrous if this happens – merely an annoyance.)

This is different from the save-file option whose short-cut is:

^X^S

Here, the Ctrl key must be held down when the 'X' and 'S' keys are pressed. Once again if you make a mistake you may get a different action. If the Ctrl key is not pressed when you hit the 'S' key you implement the MEMACS search routine. This is apparent as the line:

Search:

appears in the Command window. You can normally exit without effect from these commands simply by hitting the Return key.

Let's examine each of the Project Menu options in turn.

Rename

As its name implies, this option allows you to rename the current file being edited. When selected the message:

New file name:

appears in the Command area. Enter the new name and press Return. There are a couple of points to note here: First, the original file (provided it has been saved) remains intact; it does not have its name changed in the same way the AmigaDOS RENAME command works. In fact Rename simply alters the name of the working file in use. If you wish to save this newly named file then you must first make an editing change to it – this can be as simple as inserting a space. SAVE will then save the file using the new name.

Read-file

This command effectively swaps the current file with a new one. The command prompts you to enter the new file name, which is then read into MEMACS. If the file to be replaced has had changes made to it since the last save, the message:

Discard changes [y/n]

is displayed. Pressing 'n' loads the new file and any changes are discarded. If you wish to save changes press 'y', save the file and then execute Read-file once again.

Visit-file

This command lets you work with additional memory resident files. When you open a file its entire contents are placed in a buffer. When you use the Visit-file command you are asked to enter the name of the file to be visited. This is then read into a new buffer which becomes the current work file. You can visit the original file by using the command again and entering its name. This then

becomes the current file. The advantage is that you are reading the file from memory and do not have to re-load it from disk every time you need it. This is a quicker method of switching files.

It is possible to obtain details about current file buffers by using the List-buffers command in the Edit Menu. See under the heading Edit Menu below for more details of this command.

Insert-file

This command allows you to insert one file into another or in other words combine one file with another. To use this, move the cursor to the point immediately below the point where you wish the new file to appear. Execute the command and enter the name of the file to be inserted as requested.

Save-file

This simply saves the current file using the name assigned to it, ie, the one displayed above the Command area.

Save-as-file

In effect this is simply a 'Save as' command. When issued the message:

Write file:

appears. Enter the new name of the file and press Return. The current file is then saved under the new name – the original file remains intact providing it was previously saved – and the new name is then applied to the current file.

Save-mod

This command works on all active file buffers, saving each in turn, to the device in use, normally a disk. On completion you are returned to the current file buffer.

Save-exit

This performs the same task as Save-mod but, on completion MEMACS is closed and you are returned to either the AmigaDOS prompt or the Workbench, depending on how MEMACS was originally launched.

New-CLI

This command starts a new Shell process called 'Spawn Window'. From here you can issue AmigaDOS commands without interfering with MEMACS. Note that you cannot use the MEMACS from where you issued the New-CLI command until you close the Spawn Window with:

ENDCLI

(See *Mastering AmigaDOS 3 Reference* for a full description of this command.) However, providing you have the memory, you can run a new Shell or even MEMACS from the Spawn Window should you feel so inclined.

CLI-Command

When this command is issued you are moved into the Command area at the base of the MEMACS window. A '!' prompt appears and from here you can enter and execute an AmigaDOS command. The output of the command is displayed in a second window called 'spawn.output'. The spawn.window is treated just like another buffer and can be edited accordingly. You can get rid of it by clicking in the main window and selecting the One-window option.

Quit

This closes MEMACS and returns you to either the AmigaDOS prompt or the Workbench window. If you have modified any of the files currently held in buffers since a save command was last issued, then the message:

Modified buffers exist, do you really want to exit [y/n]?

is displayed. Pressing 'y' discards the current buffer contents and closes MEMACS. Pressing 'n' aborts the Quit command and returns you to MEMACS. You can then save and quit if you so wish.

The Edit Menu

This menu provides 16 options, all of which are associated in some way with the editing of files held in buffers. Table A.2 lists the options and their short cut key equivalents.

Each of these options are described below. We'll deal with the first three, for reasons that will become apparent later.

Set-mark

This command allows you to mark a position in a file, or more correctly, in a buffer; marks are not saved with the file. This marked position remains until it is reset by the issuing of another Set-mark command. The position of the mark is the position of the cursor when the command was issued. The position of the cursor as it moves through the file from that point is referred to as the 'dot'. The area bounded between the mark and the dot is called the 'region' or 'block'.

The easiest way to create a mark, though, is to double click the mouse where you want the mark to appear.

<i>Option</i>	<i>Quick Key</i>
Kill-region	^W
Yank	^Y
Set-mark	^@ or <ESC>
Copy-region	<ESC>W
Upper-region	^X^U
Lower-region	^X^L
List-buffers	^X^B
Select-buffer	^Xb
Insert-buffer	<ESC>^Y
Kill-buffer	^Xk
Justify-buffer	^Xj
Redisplay	^L
Quote-char	^Q
Indent	^J
Transpose	^T
Cancel	^G

Table A.2. The Edit Menu Options.

The Set-mark command is used to create a region of text that can then be manipulated by one of the other Edit menu commands or an associated command in the Move menu. You can toggle between the position of the mark and the dot, at any time, by using the 'Swap-dot&mark' command (see the Move menu).

Kill-region

This command 'deletes' (cuts) a marked region of text (see Set-mark). The deleted text is not lost; it is removed from the current file buffer and placed in the 'kill' buffer. This buffer always holds the last block of text, overwriting any previously killed region.

Note: there is only one kill buffer shared by all file buffers.

Yank

This command has the reverse action to Kill-region. It copies (pastes) the contents of the kill buffer into the current file buffer, placing it at the current cursor position. Note that it copies the kill buffer – the contents of which remain intact. It is therefore possible to Yank the contents of the kill buffer more than once.

By using Kill-region and Yank, it is possible to perform a variety of move options across one or more buffers.

Copy-region

This command works in much the same way as Kill-region. However, it does not delete the marked text but simply copies it to the kill buffer. Once in the kill buffer text can be Yanked in the usual fashion. Since the Kill buffer is shared, you can Yank text between different buffers.

Upper-region

This converts all the characters in a marked region into upper-case, ie, capital letters.

Lower-region

This converts all the characters in a marked region into lower-case – small letters.

List-buffers

This command is very useful if you are working on multiple files. When selected it lists details of all the current buffers in a separate buffer window. The buffer is displayed under the name:

`'MEMACS _ [List]'`

in its own buffer window. This window is not displayed in the list of active buffers. It should be remembered that the List buffers window is not automatically updated and, should you leave it displayed on screen and then create new buffers, these will not be listed unless the command is re-issued.

The list buffer window is broken down into four columns and will look something like this:

C	Size	Buffer	File
*	1234	Redo	ram:Redo
	14534	Redo.doc	ram:Redo.doc

A description of each of these columns follows.

- C** This stands for changed and, if a file has been modified since it was last saved, an asterisk will appear here. If no asterisk is displayed, the file has not been changed since it was last saved.
- Size** This is simply a count of the number of characters in the file.
- Buffer** This is the name of the buffer. It does not reflect the full path of the file, merely the name of the file.
- File** Gives the path and file description relative to the current directory. This is the path passed when MEMACS was started or the original file path.

Select-buffer

When you have several file buffers available, this command provides the mechanism which allows you to edit them in the currently selected window. When selected the prompt:

Use buffer:

appears in the command area. Enter the name of the buffer you wish to edit. If you wish to create a new buffer, simply use a new name. Pressing Return cancels the command.

Insert-buffer

This command works rather like Insert-file. It allows you to read the contents of one buffer into the current buffer at the cursor position. When selected the following message is displayed in the command area:

Insert buffer:

Enter the name of the buffer you require to be inserted, or press, Return to abort the command.

Kill-buffer

When run this command displays the message:

Kill buffer:

The named buffer is then deleted and removed from the buffer list. Note that if you do not wish to continue with the command you can abort it simply by pressing the Return key. Also it is not possible to kill the buffer that is currently being worked on.

Justify-buffer

This command works on the current buffer and justifies the text it holds between the two margins. Use with care, as there is no way of reversing this effect.

Redisplay

This command re-draws the current screen.

Quote-char

This command allows you to insert what are MEMACS control characters into your text. For example, you may wish to include an escape sequence within the body of the text, with a view to creating a special effect on an attached printer when you print out the file. This is done with the use of the ESC key. However, MEMACS uses ESC to set a marker. By selecting Quote-char MEMACS will interpret the next key press literally. So if the escape key was the first key pressed after this command was issued the code for ESC (^I) will be inserted into the text.

The command only works once. It must be used each time a MEMACS control character needs to be inserted into text.

Indent

If a line has been indented by a certain number of spaces, then this command will move automatically to the next line and position itself with a similar indent. This will normally be a position that is immediately below the first character in the indented line above.

If you use the command when the cursor is position within the body of the line, the line will be split at that point and moved to the indent position below as described.

Transpose

For ten-thumbed typists this command is a godsend. It swaps the position of the two adjacent characters. Thus if you typed the word:

rpogram

and then placed the cursor over the 'r' before issuing the command, the positions of the 'r' and 'p' are swapped to give 'program'.

Dyslexia rules KO!

Cancel

This terminates a menu command that may have an on-going effect. For example, a query search and replace (see the section headed 'The Search Menu' towards the end of this appendix).

The Window Menu

It is important to remember that a MEMACS window differs from that of a standard Workbench window. This should be apparent when you first load MEMACS, because many of the expected gadgets associated with windows are not present. This menu provides eight options which are listed in Table A.3.

<i>Option</i>	<i>Quick key</i>
One-window	^X1
Split-window	^X2
Next-window	^Xn
Prev-window	^Xp
Expand-window	^Xz
Shrink-window	^X^Z
Next-w-page	<ESC>^V
Pre-w-page	^Xv

Table A.3. The Window Menu Options.

One-window

When you first launch MEMACS, this is the mode of operation in use. It is used when you wish to display the current file in a single window that occupies the entire screen.

Split-window

This splits the current 'one-window' into two halves. The current file is displayed in both windows, thereby allowing you to work on two different portions of the same file at once. Ideal for long program development and editing. Changes in either window are applied to the file as a whole.

Next-window

This command moves the cursor down to the next file buffer window and makes it the current window for editing. The command continues to move the cursor down to the next window each time it is issued. The command is cyclic and, when the bottom-most window is reached, the cursor returns to the topmost window if issued again. A further such command returns the cursor to the top most page.

Prev-window

Moves the editing cursor 'up' to the next window. As with Next-window the Prev-window command wraps around in a circular fashion and, if the cursor is in the top most window, it moves to the bottom most window.

Expand-window

When a screen window is split to hold two windows, each window is of the same size. This command allows the size of the current window to be increased by a line at a time. The extra line is gained by reducing the second window by a line. If only one window is open then the message 'Only one window' is displayed in the Command area.

Shrink-window

The action of this command is the reverse of that of the previous command. It decreases the size of the current window by a line each time it is selected. In doing so it increases the size of the second window by a line. If only one window is open, the message 'Only one window' is displayed in the Command area.

Next-w-page

This command works when you are using split windows. It scrolls the contents of the 'other' window by a page, ie, a full window. For example, if you are using two windows and editing text in the top window, this command scrolls the contents of the bottom window.

Prev-w-page

This command is implemented like 'Next-w-page', however the window contents are scrolled back a single page.

The Move Menu

Not surprisingly the commands in the Move Menu all affect the position of the cursor, allowing you to reposition it in the current buffer quickly. Table A.4 lists the available options. The majority of these commands should be self-explanatory.

Top-of-buffer/End-of-buffer

These two commands position the cursor at the top and bottom of the current file buffer respectively.

Top-of-window/End-of-window

These commands reposition the cursor at the top or bottom of the currently active window. No text is scrolled.

Goto-line

MEMACS keeps track of lines by their line number, the first line in a file being line number one. When the option is used the message:

goto line:

appears in the Command area. Enter the number of the line required and press Return. The cursor is moved to the specified line. If the line number entered does not exist, the cursor is positioned at the end of the buffer.

<i>Option</i>	<i>Quick key</i>
Top-of-buffer	<ESC><
End-of-buffer	<ESC>>
Top-of-window	<ESC>,
End-of-window	<ESC>.
Goto-line	^X^G
Swap-dot&mark	^X^X
Next-page	^V
Prev-page	<ESC>v
Next-word	<ESC>f
Previous-word	<ESC>b
Scroll-up	^Z
Scroll-down	<ESC>z

Table A.4. The Move Menu Options.

Swap-dot&mark

The current cursor position is known as the 'dot'. If a mark has been set by the set-mark command, this option can be used to toggle the cursor between the position of the mark and the dot. If a mark has not been set, the message 'No mark in window' is displayed.

Next-page / Prev-page

This command scrolls the text by one full window page. Next-page moves the cursor forward through the text; Prev-page moves it towards the start of the file.

Next-word / Previous-word

Places the cursor at the next non-alphanumeric character after or before its current position.

Scroll-up / Scroll-down

Moves the position of the cursor up or down by one line respectively.

The Line Menu

The ten operations provided by this menu all work on single lines of text. Table A.5 lists the commands available. As the table shows, they are associated with cursor movement or deletion.

<i>Option</i>	<i>Quick key</i>
Open-line	^O
Kill-line	^X^D
Kill-to-eol	^K
Start-of-line	^A
End-of-line	^E
Next-line	^N
Previous-line	^P
Line-to-top	<ESC>!
Delete-blanks	^X^O
Show-line#	^X=

Table A.5. The Line Menu Options.

Open-line

This command splits the current line after the cursor. The text to the right of the cursor is moved down to the start of the next line. The cursor remains in its original position.

Kill-line

The line containing the cursor is deleted from the current file. Any text below is moved up a line to fill the vacant space. The deleted text is placed in the kill buffer. Note: some documentation of MEMACS suggests that when this command is used the line being killed is appended to any text in the kill buffer. This is not the case and any text in the kill buffer is replaced.

Kill-to-eol

'eol' stands for End Of Line. This command deletes text from the current cursor position to the end of the line containing the cursor. The deleted text is placed in the kill buffer.

If the kill buffer contains any text then kill-to-eol appends the text being deleted to the text in the kill buffer. The Yank function freezes the Kill buffer.

Start-of-line

As its name implies this command moves the cursor from its current position to the start of the line.

End-of-line

Moves the cursor to the end or right-most position of the line in which it is positioned. If the line of text exceeds 80 characters the cursor is placed at the right hand edge of the screen over the '\$' character (see the section headed 'Editing with MEMACS' towards the start of this chapter).

Next-line / Previous-line

These two commands move the cursor down or up one line respectively.

Line-to-top

The line containing the cursor is scrolled to the top of the current window.

Delete-blanks

This command removes blank lines. Any blank lines are deleted from the current cursor position to the next line containing text.

Show-Line#

This command provides information about the current cursor position. A typical response in the Command area when the command is used might be:

<Line 145 Column 1 (92%)

This indicates that the cursor is positioned on line 145. It is in column 1 (ie, the left most column) and is 92% of the way through the text.

The Word Menu

The six options provided by the Word menu are listed in table A.6 and all work on the word in which the cursor is currently positioned.

<i>Command</i>	<i>Quick Key</i>
delete-forw	<ESC>d
delete-back	<ESC>h or <ESC>
Upper-word	<ESC>u
Lower-word	<ESC>l
Cap-word	<ESC>c
Switch-case	<ESC>^

Table A.6. The Word Menu Options.

delete-forw / delete-back

These commands delete – in the direction implied – the current word or part of it. The character over which the cursor is positioned is also removed. Note that some manuals state that if the cursor is positioned over a space the command has no effect. This is not the case.

If the cursor is positioned on a space and delete-forw is executed all spaces are removed along with the next word. The space following the deleted word is not deleted and this is the resting position of the cursor.

If the cursor is positioned on a space and delete-back is executed the space is deleted followed by the word immediately before the space. The cursor is left positioned at the spot which was previously occupied by the first character in the word just deleted.

If the cursor is positioned at the end of a line and delete-forw is executed the invisible 'return' character is deleted and the line immediately below the cursor is concatenated onto the end of the current line.

If the the cursor is positioned at the start of a line and the delete-back command is executed then the 'return' character at the end of the previous line is deleted and the line moved up to the end of the previous line.

Upper-word / Lower-word

These two commands force the character case of the letters in the word where the cursor is positioned. Not all the letters in the word are affected, only those from the position of the cursor to the end of the word.

Upper-word forces the letters to capital letters regardless of their current state. Lower-word forces the case of characters to lower-case regardless of their current state.

Cap-word

This forces all the letters in the current word from the cursor position to the end of the word into upper case. For example, if the cursor was placed over the 'i' in:

amiga

and Cap-word executed, the result would be:

amIGA

Switch-case

This command toggles the case of the letters in a word starting from the position of the cursor. Thus:

AmiGa

becomes:

aMIgA

Where the command is issued with the cursor over the first 'A'.

The Search Menu

This menu provides the usual search and replace operations you would expect to find. There are two important points to bear in mind:

- In MEMACS version 1.4 (released with AmigaDOS 2) MEMACS regards as different words.
- When a replacement is made it is done in the case of the replaced text.

Table A.7 lists the five commands implemented under the Search Menu.

<i>Command</i>	<i>Quick key</i>
Search-forward	^S or ^XS
Search-backwards	^R or ^XR
Search-replace	<ESC>r
Query-s-r	<ESC>q
Fence-match	<ESC>^F

Table A.7. The Search Menu Options.

Search-forward

When you execute this command the following message is displayed in the Command area:

Search:

Enter here the string of characters (ie, word or phrase) that you wish to locate. Press Return to effect the search, which is done from the current position of the cursor down (forward) to the end of the file.

If the string is located the cursor is positioned immediately *after* it and this is normally in the space that follows. If the string is not located the message 'Not found' is returned.

Remember that this command searches the buffer from the position of the cursor – if the string is before the cursor it will not be located – use Search-backward for this.

Note that once you have used the command it 'remembers' the string and displays this in square brackets when you next use the command. If you wish to search for the next occurrence of the same string you simply have to press Return.

Search-backward

When you execute this command the following message is displayed in the Command area:

Search :

Enter here the string of characters (ie, word or phrase) that you wish to locate. Press Return to effect the search which is done from the current position of the cursor up (backwards) to the start of the file.

If the string is located the cursor is positioned immediately *after* it – this is normally in the space that follows. If the string is not located the message 'Not found' is returned.

Remember that this command searches the buffer from the position of the cursor – if the string is after the cursor it will not be located – use Search-forward for this.

When you have used this command it 'remembers' the string and displays it in square brackets when you next use the command. If you wish to search for the next occurrence of the same string you simply have to press Return.

Search-replace

This command works in a similar fashion to Search-forward. However, when the command locates *each* occurrence of the string it replaces it with the specified replacement string.

After you have entered the search string details a second prompt is displayed:

Replace:

Enter the replacement string here.

On completion a further message is displayed:

Replaced xx occurrences

'xx' is a number indicating the actual number of changes made.

See note at the start of this section regarding character case.

Query-s-r

The operation of the Search-replace command is absolute – that is it changes every single occurrence of the specified string from the point of the cursor. This command works in the same way but allows you to be selective in what you replace. Once you have entered the search and replace strings in the normal way the command locates the first instance of it – displaying it on screen – and then prompts:

Change string? [y/n/c/^G]

pressing one of these keys indicated prompts the following response:

y Yes, change and find next instance of search string.

n No, do not change this instance, move to next.

c Change this and all occurrences of the string.

^G Abort this command without changing any more occurrences.

On completion a further message is displayed:

Replaced xx occurrences

'xx' indicate the number of changes made. See note at the start of this section regarding character case.

Fence-match

This command reads the character under the cursor and then locates the next instance of that character, moving the cursor to that point. The command works in a forward mode from the current cursor position down the file.

The Extras Menu

The real power of MEMACS resides in this menu. It, in effect, allows you to define the way in which the text editor functions. In addition to this it also allows you to define your own commands using combinations of existing commands. A command built in this way is called a macro.

There are nine options in all – listed in Table A.8.

<i>Command</i>	<i>Quick Key</i>
Set-arg	^U
Set	<ESC>s
Start-macro	^X(
Stop-macro	^X)
Execute-macro	^Xe
Set-key	^X^K
Reset-key	<ESC>k
Execute-file	<ESC>e
Execute-line	^[^[

Table A.8. The Extras Menu Options.

Set-arg

This command allows you to repeat the action of a particular key multiple times. When you enter the command the following message is displayed in the Command area:

Arg: 4

The default value is always four. This is the number of times the action will be carried out. To change this number you simply type it using the numeric keys 0 to 9.

The action you wish carried out can then be typed. For example, to type a row of asterisks across the screen you would proceed as follows:

Move the cursor to the start of a blank line

Select Set-arg (From Extras Menu or by pressing ^U)

Enter 80 by pressing the '8' and '0' keys

Press the '*' key

The immediate effect should be the printing of a line of asterisks stretching across the screen. A number of blank lines can be entered simply by pressing the Return key when Arg: is displayed.

If the number you enter in response to Arg: is a multiple of four (the default) then you can enter this very quickly by pressing ^U.

It is also possible to enter minus numbers in response to the Arg: prompt. To do this, simply press the minus key on the main keypad (do not use the minus key on the numeric keypad as this does not show itself in the same way) followed by the number. For example, to enter -25 proceed as follows:

Select Set-arg (From Extras Menu or by pressing ^U)

'Press' -

'Press' 2

'Press' 5

Set

MEMACS has a number of settings which can be changed to suit your own requirements. This command allows you to alter them. When the command is executed it responds with the following message in the Command area:

Set what:

It then requires you to enter one of nine parameter names. These are detailed below.

Screen This toggles the MEMACS window between its Screen window and Workbench window states.

Interlace This toggles the interlace mode on and off.

Mode This produces a second prompt:

Mode:

There are two modes of operation, cmode and wrap.

cmode is designed for the editing of C programs – it provides automatic Fence matching.

wrap enables automatic word wrap thereby preventing text being typed off the right hand edge of the screen. This mode is best for text entry. If you select this mode you get a further prompt which is the character count across the screen where you wish the word wrap to take place. The letter 'W' is placed in the dashed line which separates the Text and Command areas at the point where the wrap will take place.

Left This allows the position of the Left margin to be set. When selected this parameter prompts you to enter a value which is the character count position in from the left where the left hand margin should be placed.

Right	This allows the position of the Right margin to be set. When selected this parameter prompts you to enter a value which is the character count position in from the left where the right hand margin should be placed.
Tab	This sets the number of character positions that should be typed when the TAB key is pressed. The default value is nine.
Indent	This is of interest when working in cmode and determines how far to indent each level of nesting. As with Tab you are prompted to enter a value to determine this.
Case	This allows you to define whether a search operation is case sensitive or not. The default is off.
Backup	<p>MEMACS has a backup save facility available which is normally switched off. When selected the following prompt is displayed:</p> <p>Backup:</p> <p>You can now enter one of three modes of operation. These are:</p> <p>ON: This gives the current file the same name but with the extension '.bak. added which is in turn saved to the T: directory.</p> <p>SAFE: This first tests to see if the file already exists in the buffer. If this is the case it will not overwrite it thereby preserving your original.</p> <p>OFF: The default option – MEMACS will not perform any sort of backup.</p>

Start-macro / Stop-macro

These two commands enable and disable macro recording. See the section headed 'Macros' at the end of this chapter for details about the use of macros.

Execute-macro

This command runs the currently defined macro. See the section headed 'Macros' at the end of this chapter for details about the use of macros. Set-key

The function keys can be used to hold macro definitions. This command allows you to do that and is covered under the section headed 'Macros' at the end of this chapter.

Reset-keys

If any of the function keys are redefined using Set-key, they can be returned to their default state by executing this command.

Execute-file / Execute-line

These two commands, and in particular Execute-file, are extremely powerful. A full description of using these commands to do text processing can be found under the heading 'MEMACS Command Files' which is towards the end of this chapter.

Miscellaneous Commands

In addition to the eight menus of command options provided by MEMACS there are an additional eight operations that can be performed by using quick key entries. These are listed in Table A.9.

Describe Key

This operation allows you to get details of the function a that specific key press performs. The name of the function appears in the Command area in the same format used to head each of the sections in this chapter.

For example, pressing <ESC>^D will produce the following message in the Command area:

Describe key:

You can now enter the quick key combination you require details of, for example:

<ESC>k

The Command area now displays:

Function: Reset-keys ESck

Where Reset-keys is, indeed, the operation performed by <ESC>k.

<i>Command</i>	<i>Quick key</i>
Describe Key	<ESC>^D
Bind Key	<ESC>^B
Echo	<ESC>^E
Move to Edge of Window	<SHIFT>+cursor key
Delete to Next Character	^D
Delete the Previous Character	^H
Move to Next Line	^M
Move Cursor Forward by x number of Characters	^F
Move Cursor Backwards by x number of Characters	^B

Table A.9. Non-Menu Command Options.

Bind Key

This allows you to bind a key to a function. In other words it allows you take one of the named options that appears on any of the eight menus and define which key sequence you would like to use to implement it.

For example, you may decide that you would like to use <ESC>C to perform the Cap-word function. To do this proceed as follows:

Press <ESC>^B

The message:

Bind:

appears in the command area.

Type in the name of the function to be implemented exactly as it appears in this section, followed by a space and then the key combination you wish to use to effect it. The line would look like this:

Bind: Cap-word <ESC>C

Note that in this case <ESC>C is typed letter for letter – it does not mean press the Escape key and then C. Press Return to complete the definition.

Unbind Key

This undoes the effect of Bind Key. When used you will be requested to enter the key sequence (spelt out not the actual combination) and then press Return. The message 'Key is not bound' will be displayed.

Echo

This command is normally used in MEMACS Command files and allows you to display a message in the Command area while the file is executing. It is not dissimilar to the AmigaDOS ECHO command. See the section headed 'MEMACS Command Files' towards the end of this chapter for more details on command files.

Move to Edge of Window

By holding down the SHIFT key and then pressing one of the four cursor arrow keys, you can move the cursor to the appropriate edge of the screen.

Delete the Next Character

Has the same effect as pressing the key. Deletes the character under the cursor.

Delete the Previous Character

Has the same effect as pressing the backspace key. Delete the character to the left of the cursor.

Move to Next Line

Inserts a return character at the point of the cursor and moves the cursor down one line.

Move Cursor by x Number of Characters

By pressing ^F or ^B you can move the cursor forward or backwards by a single character. One character is the default. A larger number of characters can be moved forward or backwards by using the Set-arg command. For example, to move 10 characters forward:

Press: ^U

Press: 1

Press: 0

Press: ^F

MEMACS Command Files

Each of the entries under the various menu headings given above start with the name of a command. The command name is written in a rather special and all important way. For example, the space between words is normally hyphenated and some words start with a capital letter. This format is very important as it allows you to execute a command by specifying the command name.

For example, the command Execute-line allows you to enter a command by name in the command area. Press <ESC><ESC> (ie, press the Escape key twice) and the message:

execute-line:

is displayed. Enter:

Set screen

press Return and the Set command will be executed with the screen parameter.

The Execute-file command takes this a step further and allows you to execute a MEMACS file containing nothing but commands. These commands act on the text held in the current buffer. The command file concept is also the reason why several of the various menu commands look a little pointless at first sight.

The best way to understand this is to work through this simple example.

Create a new MEMACS file called 'FirstUpper'. Enter the following exactly as shown:

Top-of-buffer

Set-mark

Next-line

Upper-region

Save this and create a new file and enter two or three lines of text into this. Now execute the 'Execute-file' command (via the Extras Menu or by pressing <ESC>e). The command will respond with:

File:

in the Command area. Enter the name of the command file, ie,

FirstUpper

and press Return.

The effect is almost instantaneous. You should see that the very first line of your text has been changed to all upper case.

This is just a very simple example of what can be done with command files – the potential is vast. You could even build your very own library of MEMACS Command Files if you intend to do a lot of work using MEMACS either for programming or text editing.

Macros

A macro is a set of actions or commands that are grouped together and executed in sequence. MEMACS macros are sets of two or more commands. However, macros allow you to go a step further. You create a macro by recording a sequence of actions, which may comprise the use of MEMACS commands plus other keyboard actions such as moving the cursor etc.

Try this example, which makes a copy of the first line in a text file and copy it to a new line at the end of the file. To do this we need to carry out the following steps:

1. Move to top of file
2. Mark here
3. Move to end of the line
4. Copy marked region into kill buffer
5. Move to end of file
6. Enter a new line (press Return)
7. Copy kill buffer to cursor position

These translate into commands as follows:

1. <ESC><
2. <ESC>-

3. ^E
4. <ESC>w
5. <ESC>>
6. <Return>
7. ^Y

(Note that in step 2 the <ESC> sequence has been used to set-mark. The ^@ shorthand form does not work within macro definitions.)

This may be a sequence of commands that you wish to use regularly. Rather than having to enter them in sequence every time you wish to carry out the operation, you can record them into a macro.

MEMACS has the ability to record a macro and hold it in memory. To start macro recording you can use the Start-macro command (Extras menu or ^X()). Now every action you perform will be recorded and remembered up and until you select the Stop-macro command (^X)). So to record the above in a macro select Start-macro, perform the commands in the list and then select Stop-macro. To execute the macro select Execute-macro (Extras Menu or ^Xe).

Macros and Function keys

When MEMACS is first run it loads the function keys with a set of default macros. These are quite useful and are listed in Table A.10.

<i>key</i>	<i>Default Action</i>
fkey 1	clone line
fkey 2	delete line
fkey 3	execute keyboard macro
fkey 4	next screen
fkey 5	previous screen
fkey 6	split window
key 7	one window
fkey 8	scroll window up
fkey 9	scroll window down
fkey 10	save file and exit
help	insert line and indent
keypad enter	insert line and indent

Table A.10. Function Key Default Macros.

All of these keys can be redefined to hold new macro definitions by using the Set-key command (Extras Menu or ^X^K) along with Shifted function keys. When you use the Set-key command the following message is displayed in the Command area:

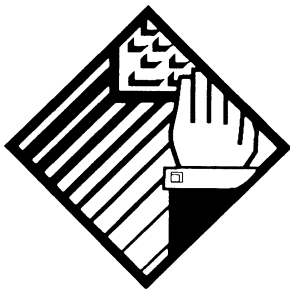
key to define:

At this point press the key that you wish to redefine. The command will respond to this by displaying the current definition of the command. Now you can enter the new macro definition by typing the relevant control characters. The macro can be up to 80 characters long.

Do remember that you must use the Quote-char function (^Q) to insert the function keystroke into the macro – see under the Edit Menu for further details regarding this should you need a refresher.

Startup Files

When MEMACS is first run, it looks for a command file called 'emacs_pro' in the current directory or, if not found, in the S: assignment. If located, this file is automatically executed by MEMACS. This allows you to create a customising file which could be used to define how you would personally like MEMACS set-up.



B: Original AmigaDOS 2 Startup- Sequence

The early release Amigas supporting AmigaDOS 2 included the A3000. The mass market version of AmigaDOS 2 was released with the A500 Plus and was revamped as version 2.04. AmigaDOS 2.04 is detailed in Chapter 38. For completeness this appendix gives details on the original *launch* version of AmigaDOS 2.

The full listing is given overleaf and a description of each line follows.

Explanation

1. Version usually just reports the version number of Workbench and Kickstart in use; and re-direction to NIL: stops anything being printed. In this release, it also creates two local environmental variables – Workbench and Kickstart – and sets those to the current version numbers. Since these are local variables, they will be lost when the process closes – ie, at the end of the Startup-Sequence – so they are picked up at 20 and 21.
2. This sets the failure level so high nothing short of a GURU (yes folks, they're still there), a non-standard return code or a *missing* command will stop the script. It seems this has been added because commands like SETCLOCK in release 2, can return fatal errors for some *apparently* trivial faults. SETCLOCK returns 20 (FAIL) when it cannot find a battery

backed clock – but machines running AmigaDOS 2 may not have clock cards. This is probably just a patch because AmigaDOS 2 was initially only supplied with high-end Amigas which all had clocks fitted at source.

3. This just loads the time and date from the internal battery backed clock. Output from the command is sunk to NIL: to stop error messages like “Can't find battery-backed up clock”. This has good and bad points. If you know the machine doesn't have a battery-backed up clock, this line can safely be removed.
4. This makes LIST resident for the rest of the script. Re-direction to NIL: prevents the command displaying any errors – this only affects the PURE switch and suppresses the “Pure bit not set message”. This error is not normally generated. The ADD switch causes LIST to be added to the resident list. That is, it will not replace one which is there already. A more detailed explanation of RESIDENT can be found in *Mastering AmigaDOS 3 Reference*.
5. Make COPY resident using the same considerations as LIST – see 4 above.
6. Make ASSIGN resident using the same considerations as LIST – see 4 above.
7. This line is very important and is concerned with the Preferences structure. In short, it copies the user's Preferences settings into a special directory on the RAM: disk – managed by the IPREFS daemon. (Beginners should leave this alone.) This starts a fairly complex stream of events so we will take them one at a time.

COPY >NIL:

- 7a This starts the copying operation and sinks all output to oblivion. Since the QUIET switch is active this might seem like overkill. Without this, if COPY generates an error – see 7d – it will still be displayed.

ENVARC:

- 7b Briefly this is where Workbench stores the default Preferences settings. This is passed to copy as the FROM argument so it can be created in the RAM: disk.

Like L:, DEVS:, S: and so on, ENVARC: is assigned to the boot disk (SYS:) when the machine is started. Its full path is, by default, SYS:Prefs/Env-Archive. The directory *must* exist on standard Workbench 2 disks or the startup can fail.

RAM:ENV

The AmigaDOS 2 Startup-Sequence

```
1  version >NIL:
2  Failat 21
3  SetClock >NIL: load
4  resident >NIL: c:List pure add
5  resident >NIL: c:Copy pure add
6  resident >NIL: c:Assign pure add
7  copy >NIL: ENVARC: ram:env all quiet noreq
8  mkdir ram:t ram:clipboards
9  assign T: ram:t ;set up T: directory for scripts
10 if exists sys:Monitors
11     list >t:mon-start sys:monitors/~(#{?.info) lformat="run
    >NIL: %s%s"
12 execute t:mon-start
13 endif
14 assign ENV: ram:env
15 run >NIL: iprefs >NIL:
16 wait >NIL: 5
17 addbuffers >NIL: df0: 15
18 echo "Amiga Workbench Disk. 2.0 Release Version $Workbench"
19 BindDrivers
20 setenv Workbench $Workbench
21 setenv Kickstart $Kickstart
22 resident c:Execute pure add
23 assign CLIPS: ram:clipboards
24 assign REXX: s:
25 mount speak:
26 mount aux:
27 mount pipe:
28 path ram: c: sys:utilities sys:rexxc sys:system s: sys:prefs
    sys:wbstartup add
29 if exists sys:tools
30 path sys:tools add
31     endif
32 rexxmast >NIL:
33 if exists s:user-startup
34     execute s:user-startup
35 endif
36 LoadWB
37 endcli >NIL:
```

- 7c This is passed to Copies TO argument. Used in this way, COPY will create a new directory in RAM: called ENV and copy the contents of ENVARC: to it. This directory becomes the ENV: assignment later on and will be read by IPREFS.

ALL

- 7d This switch causes COPY to copy all files and sub-directories from ENVARC: to RAM:ENV – creating a RAM-based copy of the current preferences settings.

QUIET NOREQ

- 7e This causes COPY to stop throwing up requesters like "Please insert volume XX in any drive" and generate a fatal (FAIL) error instead. Due to the way the script has been written, this switch will not stop the startup, nor will it generate an error. This feature has been added to allow the startup to continue when, for instance, the ENVARC: assignment does not exist.
8. This line makes use of the multiple argument parser to create the T and CLIPBOARDS directory in the Ram Disk.
9. This just moves the T: assignment from its normal place on the boot disk (T directory) to the Ram Disk. This has the same function it had in AmigaDOS 1.3.
10. At this point, the script checks for the presence of the optional monitors directory. If the directory is found, control resumes at 11, otherwise it jumps directly to 13. This directory is only used on machines with special monitor setups; multi-syncs and so on. Typically, the Amiga 3000 model.
11. Control only gets here *if* the monitors directory is present. It creates a script to bind the monitor hardware. Here is how it breaks down:
- list >t:mon-start**
- 11a Create a temporary file in T: called "mon-start"
- SYS:MONITORS/~(#?.INFO)**
- 11b Search the MONITORS directory of the boot disk and only select files which *do not* end in ".INFO". The pattern is used in this way to allow the user to add any extra drivers by simply dragging their icons from a driver disk into the appropriate drawer. Of course, you can't execute dot-info files so they have to be removed.
- LFORMAT="RUN >NIL: %S%S"**
- 11c The clever bit. Briefly, the resulting file depends on the number of monitor drivers located in the directory. If there were two, VGA.monitor and Multi-sync.monitor, the resulting script would look something like this:

RUN >NIL: Workbench2/VGA.monitor

RUN >NIL: Workbench2/Multi-sync.monitor

12. Executes the mon-start script created at line 11 and generates a new process for any new monitor drivers.
13. Marks end of the IF...ENDIF construct started at 10. Control passes through here no matter where it came from because ENDIF is just a closing marker – it must be present though.
14. This tells AmigaDOS where to find the ENV: assignment. ENV: is almost always assigned to RAM:env (created at 7) because it is shared by the global environmental variables and preferences.
15. Start the IPREFS daemon. This line looks a little strange because the NIL: device is used twice. The first >NIL: stops [Shell 2] appearing and appears to prevent the appearance of the "To reset Workbench screen please close all windows except drawers" message. Quite what the second one does is beyond us!
16. This forces the script to wait for 5 seconds while IPREFS reads and sets up the preferences. The NIL: is probably not necessary since WAIT does not generate any output.
17. Add 15 cache buffers (about 7.5K) to the internal drive.
18. Announce the Workbench disk and print the Workbench version. Note: the Workbench version number is read from a local environmental variable which is created by VERSION at line 1 (above).
19. Add any optional hardware drivers to the system. This is probably only required by non-auto-booting hard disks and some RAM expansions. Most standard Commodore equipment is mounted automatically by Kickstart.
20. Creates a new GLOBAL environmental variable containing the version of Workbench in use. This is read indirectly from the LOCAL variable created by VERSION (see 1 above). See Chapter 12 for more details on this.
21. Like 20, this creates a GLOBAL environmental variable containing the version of Kickstart in use. This is read indirectly from the LOCAL variable created at 1 above.
22. Add EXECUTE to the resident list. This is optional because execute is only called twice in this script – even in the worst case. Assuming of course, some user script does not call it recursively. This line should be left in if you use scripts much.

23. This adds the CLIPS: assignment to the Ram Disk. Very little software uses CLIPS: so this can be viewed as optional. Just assigning takes very little RAM.
24. This new assignment is for AREXX. It tells the interpreter where to look for REXX programs – the S: assignment. Note however: although you can assign one logical assignment to another assignment like this, the actual path will be absolute. Let's say S: was pointing to Workbench2:S. If you re-assign S: to DF1:S, REXX: would stay assigned to Workbench2:S. If you use REXX a lot, its probably better to give it its own directory.
- 25..27.

These optional pseudo devices do not take too much RAM. You can remove them if you want to. See Chapter 28, Devices, for more information.
25. Make the speech system available. Very little software uses this via AmigaDOS – but it is handy to have around.
26. Add the auxiliary serial device to the device list. This is only used for secondary terminals.
27. Add Matt Dillon's PIPE: device. This is an advanced FIFO buffer which can be used to pass messages between processes. Pipes are reasonably easy to use and often avoid the need to created temporary files.
28. This adds the following directories to the search (command) path. Workbench and the Shell share the path settings. See *Mastering AmigaDOS 3 Reference* for more information:

RAM:

The Ram Disk. Fast access to AmigaDOS commands.

C:

The AmigaDOS command directory. Forces searching of the BOOT disk's command directory even if C: has been assigned elsewhere.

Utilities

Various utility programs like MORE and the Commodities Exchange live in here. Only used by Shell.

Rexxc

The AREXX command directory. Only used by Shell.

System

System commands like FORMAT and DISKCOPY.

S:

The scripts directory. Only used by EXECUTE and internally by software like BRU.

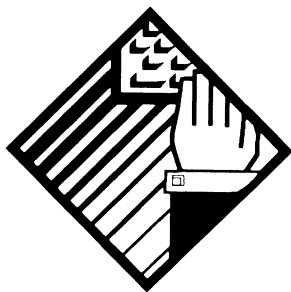
Prefs

The preferences directory.

WBStartup

This must be in the path or the auto-starting programs will not work.

29. If the TOOLS directory is present on the BOOT disk, then continue at 30; if not continue at 31.
30. Add the optional tools sub-directory to the search path.
31. This terminates the IF...ENDIF construct started at 29.
32. Start the AREXX resident process – re-direction to NIL: stops AREXX displaying its normal startup message. This line is not required unless you intend using AREXX. Note however, although REXX program examples are not supplied with the system – some of the system software – ED for instance– can use it.
33. Check for the optional User-startup script. If this is found, continue execution at 34, otherwise jump to 35.
34. Run the User-Startup script. This is a special file which you can add and edit and customise the system to your own needs. The idea behind this, is since Workbench 2's startup is so complex, it is better to allow you to make small changes and amendments in a separate file:
35. This terminates the IF...ENDIF construct started at 33.
36. Trigger the Workbench.
37. Close the bootstrap Shell and terminate the script.



C: ED's Commands

Appendix D is devoted to the use of the various ED commands. It does not contain any information that is vital to the following chapters. It is, of course, vital if you wish to learn how to be an EDspert!

Cursor Control

Like the Shell, ED provides a number of editing facilities and these are listed with brief descriptions in Table C.1.

Most of these command actions are fairly straightforward and easy to understand, however a few are worth a few words of expansion and this is given below. In all cases experiment with a simple ED file such as TestED.

<TAB>

ED does not contain any TAB stop positions that can be adjusted to suit your own needs. Instead it assumes a standard TAB setting of three character spaces and each time TAB is pressed it moves the cursor three positions to the right. The effect is exactly the same as having pressed the <right arrow> key three times in succession.

<CTRL-F>

This swaps the case of the character under the cursor, and moves it right one position. Thus if the letter under the cursor is an 'e' and <CTRL-F> is pressed, it becomes an 'E'.

<CTRL-G>

ED supports a number of 'extended' commands accessed with the ESC key (see below) – this repeats the last extended command.

<CTRL-O>

This command deletes the characters from and to the right of the cursor position up until the first space character. If the character under the cursor is a space it deletes all spaces to the right until the first non-space character.

<i>Key</i>	<i>Action</i>
Backspace	Delete character to left of cursor
DEL	Delete character under cursor
ESC	Enter extended command mode
RETURN	Split line at cursor and create new line
TAB	Move cursor to next 'Tab' position. (In fact this moves three spaces and does not use a Tab character.)
Cursor Up	Move cursor up one line
Cursor Down	Move cursor down one line
Cursor Right	Move cursor right one character
Cursor Left	Move cursor left one position
CTRL-A	Insert new line
CTRL-B	Delete current line
CTRL-D	Scroll text down 12 lines
CTRL-E	Move to top or bottom of screen
CTRL-F	Invert chase of character under cursor
CTRL-G	Repeat last extended command line
CTRL-H	Backspace and delete
CTRL-I	Move cursor right to next 'Tab' position
CTRL-M	Return
CTRL-O	Delete current word or spaces to right of cursor
CTRL-R	Move cursor to end of previous word
CTRL-T	Move cursor to start of next word
CTRL-U	Scroll text up 12 lines
CTRL-V	Rewrite screen
CTRL-Y	Delete to end of line from and including cursor position.
CTRL-[	Escape character (enter extended mode)
CTRL-]	Toggle cursor to end or start of line

Table C.1. The ED command set.

ED's Extended Commands

ED supports a wealth of extended commands. These are commands which generally require more than one letter used in combination with the ESC key, or require information to be supplied with the command.

For example the command:

<ESC-SH>

can be used to display the file information that ED keeps on each file it creates (this is saved with the file). The above command can be entered as <ESC-S> and <ESC-H> or as <ESC-S><H>. As with all ED commands extended commands are displayed in the bottom left hand corner of the screen and are not executed until the <Return> key is pressed. Table C.2 lists the various extended commands used by ED.

As with the standard CTRL commands, many of these command actions are fairly straightforward and easy to understand, however many are worth a few words of expansion and these are given below. In all cases experiment with a simple ED file such as TestED.

<ESC-A> Insert Text

This command can be used to insert text at the current cursor position. Its effect is really just like typing directly at the keyboard. For example:

<ESC-A>/ECHO "This is an example of ESC-A"

<ESC-BE> Mark Block End at Cursor

Blocks of text can be marked for various reasons. These are outlined below under the section headed 'Marking Blocks of Text'.

<ESC- BF> Find String Backwards Search

Specific strings of text can be located using ED's find facility. Searches can be carried in either direction from the cursor position. This command searches backwards, ie, from the cursor towards the start of the file. To search for the string 'ECHO' use:

<ESC-BF>/Echo

Note that by default the search is case dependent, ie, the case of letters must match exactly. For instance, searching for 'Echo' will not recognise 'echo' or 'ECHO'. The command locates the very first occurrence. Repeat command with <Ctrl-G>.

<ESC-BS> Mark Block Start at Cursor

Blocks of text can be marked for various reasons. These are outlined below under the section headed 'Marking Blocks of Text'.

<i>Command</i>	<i>Action</i>
A/text/	Insert text after current cursor position
B	Move to bottom of file
BE	Mark end of block at cursor
BF/string/	Backwards find
BS	Mark block start at cursor
CE	Move cursor to end of line
CL	Move cursor left one position
CR	Move cursor right one position
CS	Move cursor to start of line
D	Delete current line
DB	Delete block of marked text
DC	Delete character under cursor
E/x/y/	Exchange first instance of 'x' with 'y'
EQ/x/y/	Exchange first instance of 'x' with 'y' but query first
EX	Extend right margin
F/string/	Find string 'string'
I/text/	Insert 'string' before current cursor position
IB	Insert copy of marked block
IF/s/	Insert file s
J	Join current line with next
LC	Make searches case dependent
M <num>	Move to line number <num>
N	Move to start of next line
P	Move to start of previous line
Q	Quit without saving text
RP	Repeat following command until error occurs
S	Split line at cursor position
SA	Save text to file
SA/filename	Save text under new name
SB	Show block on screen
SH	Show information
SL <num>	Set left margin to <num>
SR <num>	Set right margin to <num>
ST <num>	Set Tab to <num> spaces
T	Move to top of file
U	Undo changes on current line
UC	Equate upper case and lower case in searches
WB/filename	Write block to file
X	Exit, writing text to disk

Table C.2. Extended ED commands.

<ESC-DB> Delete Block

This command deletes the marked block of text. If no block is marked the error message 'No block marked' is returned. See the section headed 'Marking Blocks of Text' below.

<ESC-E> Exchange Characters

This command finds the first occurrence of the specified character and replaces it with the second specified character. For example, a common mistake is to use an Oh in place of a zero. This can be rectified by moving the cursor to the top of the file and the proceeding as follows:

<ESC- E>/0/0/

<Ctrl-G>

When there are no more occurrences, or if an occurrence is not found, the error message 'Search failed' is displayed.

Note: All ED's extended commands use the slash (/) symbol to delimit characters from either the command or from each other. Slash means the same to ED as the space does to AmigaDOS so it's important to fix the distinction in your mind.

<ESC-EQ> Exchange Characters With Query

The above example of using <ESC-E> is fine but it does mean that each occurrence will be replaced. Of course you might have Oh's in your text which are vital. This command allows you to account for this. To search for occurrences of Oh and possibly substitute them for zeros use:

<ESC-EQ>/0/0/

On finding the first occurrence the following message is displayed in the bottom left hand corner of the ED window:

Exchange?

Pressing 'Y' (or 'y') effects the change. Pressing any other key aborts the command without any change. Pressing <Ctrl-G> will re- execute the command once again.

<ESC-EX> Extend Right Margin

By default the width of any text across the ED screen depends on the setting of the right margin position. By default this is 77 . The position of the margin can be changed with <ESC-SR>. Once reset text is displayed according to this. This command allows this boundary to be exceeded. When issued, the current line is allowed to exceed the right margin position regardless of settings.

<ESC-F> Find Text String

This command searches forward to find the first occurrence of the string, which may consist of a word or phrase. For example:

<ESC-F>/Hello there mate/

By default the search is case dependent.

<ESC-I> Insert Text Before Cursor

This command works like <ESC-A>, however the text is placed directly before the cursor as opposed to after it. For example:

<ESC-I>/This is before the cursor/ <Body text>

<ESC-IB> Insert Copy of Marked Block

This command locates the marked block of text and inserts a copy of it at the position of the cursor. See the section headed 'Marking Blocks of Text' below.

<ESC-IF> Insert Named File

This command locates the ED file named and reads it into the body of the current text at the position of the cursor. The file being read remains unaltered. For example to load a file called 'Tester' into the body of a current ED text file you could use:

<ESC-IF>/Tester/

<ESC-LC> Distinguish Character Case

This is the default state of ED and is used to countermand the action of <ESC-UC>. After being executed searches are carried out and are case dependent. Therefore the three words:

ECHO

echo

Echo

are 'seen' as three different words.

<ESC-M> Move To Line Number

This command becomes very useful when dealing with long ED files as it allows you to move to specific lines. For example to move the cursor to line number 10 use:

<ESC-M> 10

and press RETURN.

The line numbering starts from the top and the first line is always line number one.

<ESC-SA> Save Text To File

This command saves the current file but without quitting ED. A filename can be included to save the text under a new filename. For example:

<ESC-SA>/NewFile/

<ESC-SB> Show Block On Screen

When a block has been marked, issuing this command will move the start of the marked block to the top of the screen. See the section headed 'Marking Blocks of Text' below.

<ESC-SH> Show File Information

This command displays the various bits of file information kept by ED. Seven items of information are listed in Table C.3.

<i>Heading</i>	<i>Information Provided</i>
Editing file	The name of the current file.
Tab distance	The number of spaces moved by the cursor when Tab is pressed.
Left margin	The character position of the start of a line of text.
Right margin	The character position of the end of a line of text.
Block start	The first line of a marked block of text.
Block end	The last line of a marked block of text.
Buffer usage	The percentage of buffer space used by the text.

Table C.3. File information displayed.

<ESC-SL> Set Left Hand Margin

This allows the position of the left hand margin to be positioned. The default setting is 1. For example, to make the left hand margin five characters in from the normal position use:

<ESC-SL> 5

Once the cursor reappears, press <Return> to effect the change.

<ESC-SR> Set Right Hand Margin

This allows the position of the right hand margin to be positioned. The default setting is 77. For example, to make the right hand margin 20 characters in from the normal position use:

<ESC-SR> 20

Once the cursor reappears, press <Return> to effect the change. Changing margin settings does not affect any existing text, only that typed subsequently. The amount of text area that can be used is found by subtracting the left margin setting from the right hand setting.

<ESC-T> Set Tab Distance

This command allows the number of spaces to be skipped on, when Tab is pressed, to be altered. For example, to set the Tab jump to eight characters use:

<ESC-T> 8

The default value is three.

<ESC-U> Undo Changes To Line

ED stores a copy of the current line where you are typing. If you edit the line and decide that you wish to revert to the original line then this command does just that, providing <Return> has not been pressed. Pressing <Return> 'fixes' the changes.

<ESC-UC> Equate Character Case

If this command is issued then character case is ignored when search operations are carried out. Therefore the three words:

ECHO

echo

Echo

will be seem to be the same.

<ESC-WB> Write Block to File

This command writes the marked block of text to the named file. For example:

<ESC-WB>/Marked/

The current text is unaffected as is the current ED file.

Marking Blocks of Text

ED allows sections of text to be identified by enclosing them within bounded areas. These areas are defined by the use of two markers inserted into the text using the commands <ESC-SB> and <ESC-BE>.

The text to be marked can vary from just a few characters up to a few hundred lines. The markers are not shown on screen but you can identify the start of a marked block using <ESC-SB>. Note that markers only remain until you start typing into ED, they are not affected by ED commands however.

ED markers have several uses: they allow you to move a section of text from one part of a file to another; to copy a repetitive section of text quickly and simply; to allow you to delete a section of text; and to allow you to save a marked section of text to a file. So get used to the markers, they play a very big role in effective use of ED.

Inserting a marker is easy. Move the cursor to the exact position you wish the marker to be placed and then press <ESC-BS> (Block Start). The marker does not occupy any space on screen and may, and generally does, occupy the same position as a character from your text. The second marker is placed in the same way but, as it is marking the Block End, use <ESC-BE>.

Blocks that are to be copied to a new position are always copied to the current position of the cursor, so always reposition this first before executing the <ESC-IB> command.

Note: Markers affect complete lines – you cannot mark sections of a line in ED as you could in larger editors. However, you don't have to move the cursor to the start of a specific line.

ED Menus

ED is complemented by a series of menus. These provide instant access to a variety of the extended commands discussed in the preceding pages. They are used in the normal fashion – select the menu option required and its function is carried out. If the option requires further user input then this is requested at the base of the window just as though you had issued the command from the keyboard.

However, ED menus may be customised and this is determined by the use of an ED startup file in the S: directory. The file is called, not surprisingly, Ed-Startup

When ED is run it tries to run this file. If the file is not found then ED defaults to a full set of menus. On the other hand, if the file is located then the menus defined therein are utilised. Chapter 37 describes how to write your own ED-startup files. They are quite easy to do and worth a try.

Default Menus

Because the ED menus may be customised to suit requirements and Amiga packages it is difficult to give a full description of all possible combinations of menus. However, as each and every ED command has been detailed in this chapter you should have no problem using them.

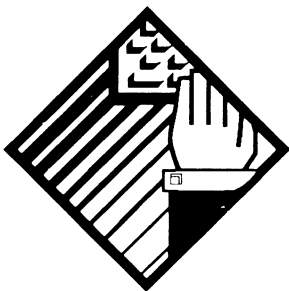
By default (ie, when there is no S:Ed-Startup file) ED starts with six menus. These are:

- Project
- Edit
- Movement
- Search
- Settings
- Commands

Despite what we have just said the Commands menu does contain one command not detailed. This is:

AREXX Command <ESC>rx

AmigaDOS provides an interface with the AREXX command language. This makes it possible to customise and interface programs that support AREXX. A full tutorial on AREXX would require a book the size of this one as such we have not been able to delve into it. However, should you decide to pursue the AREXX interface then it is worth knowing that this particular AREXX port exists. ED's AREXX port is called, wait for it, ED.



D: Mastering Amiga Guides

Bruce Smith Books are dedicated to producing quality Amiga publications which are both comprehensive and easy to read. Our Amiga titles are being written by some of the best known names in the marvellous world of Amiga computing. Below you will find details of all our currently available books for the Amiga owner.

Titles Currently Available

- Mastering AmigaDOS3 – Reference
- Mastering Amiga AMOS
- Mastering Amiga C
- Mastering Amiga Printers
- Mastering Amiga System
- Mastering Amiga Assembler
- Mastering Amiga ARexx
- Amiga A1200 Insider Guide
- Amiga A1200 Next Steps
- Assembler Insider Guide
- Amiga Gamer's Guide
- Mastering Amiga Workbench 3*

*This title is available late 1993. Other version 3.0 and A1200 specific books are planned so get yourself on our mailing list for further information.

Other titles which relate to Amigas with AmigaDOS and Workbench prior to version 3.0 are:

- Amiga A600 Insider Guide
- Mastering Amiga Workbench 2
- Mastering AmigaDOS2 Vol. 1
- Mastering AmigaDOS2 Vol. 2

Brief details of these guides along with review segments are given below. New publications and their contents are subject to change without notice. If you would like a free copy of our catalogue and to be placed on our mailing list then phone or write to the address below.

Our mailing list is used exclusively to inform readers of forthcoming Bruce Smith Books publications along with special introductory offers which normally take the form of a free software disk when ordering the publication direct from us.

Bruce Smith Books,
PO Box 382,
St. Albans, Herts, AL2 3JD
Telephone: (0923) 894355
Fax: (0923) 894366

Note that we offer a 24-hour telephone answer system so that you can place your order direct by 'phone at a time to suit yourself. When ordering by 'phone please:

- Speak clearly and slowly
- Leave your name and full address and contact phone number
- Give your credit card number and expiry date
- Spell out any unusual names

Note that we do not charge for P&P in the UK and endeavour to dispatch all books within 24-hours.

Buying at your Bookshop

All our books can be obtained via your local bookshops – this includes WH Smiths which will be keeping a stock of some of our titles, just enquire at their counter. If you wish to order via your local High Street bookshop you will need to supply the book name, author, publisher, price and ISBN number.

Overseas Orders

Please add £3 per book (Europe) or £6 per book (outside Europe) to cover postage and packing. Pay by sterling cheque or by Access, Visa or Mastercard. Post, Fax or Phone your order to us.

Dealer Enquiries

Our distributor is Computer Bookshops Ltd who keep a good stock of all our titles. Call their Customer Services Department for best terms on 021-706-1188.

Compatibility

We endeavour to ensure that all general *Mastering Amiga* books are fully compatible with all Amiga models and all releases of AmigaDOS and Workbench.

Amiga Gamer's Guide by Dan Slingsby

Everyone loves games and Amiga games are growing in sophistication, always setting new playing challenges while introducing ever more gasp-producing graphics and sound effects. Even the techies at Bruce Smith Books are not immune to the games phenomenon. This latest book for the discerning Amiga owner, is a highly illustrated guide to your favourite Amiga games, including classics like Shadow of the Beast and recent top ten hits like Putty, Formula One Grand Prix, Streetfighter 2 and Indiana Jones.

From sports sims to arcade adventures, Amiga Gamer's Guide gives you the hints and tips, hidden screens and puzzle solutions which you are looking for. Completed by a massive A to Z of tips and tricks for over 300 games, Amiga Gamer's Guide is the most masterful of games guides yet published.

Written by CU Amiga editor Dan Slingsby, Amiga Gamer's Guide contains a wealth of background information to the most popular Amiga games. The graphically appealing layout with hundreds of pictures used to illustrate the games and their story lines, makes this one of the most attractive Amiga books to be found on the bookshelves.

The games featured, with full scenarios, hints and tips and solutions are:

Another World; Chaos Strikes Back; Dungeon Master; Elvira 2; Epic; Formula One Grand Prix; Gobliins; Indiana Jones & the Fate of Atlantis; Ishar; John Madden American Football; Kickoff 2; Lure of the Temptress; Monkey Island 2; Populous 2; Project X; Putty; Robocod; Sensible Soccer; Shadow of the Beast 1, 2 and 3; Speedball 2; Streetfighter 2; Striker; Supremacy; Zak McKracken.

Amiga Gamer's Guide

by Dan Slingsby

ISBN: 1-873308-16-7, price £14.95, 368 pages.

Mastering AmigaDOS 3 – Reference

Mastering AmigaDOS 3 – Reference is the companion volume to this book. It is a detailed and comprehensive reference to all the AmigaDOS commands detailing their use and capabilities. In A to Z format for instant reference.

Here's what the press said:

"If you're a complete beginner or unsure of a few areas, this book is an amazingly informative read." Amiga Format.

"As a reference book it's very useful. So far as I know there isn't any similar book...If you need to know how every AmigaDOS command works get this book...it is a definitive reference" Amiga Format.

"The Reference book that Commodore forgot to commission" Keith Pomfret of New Computer Express.

"The book can be strongly recommended....and even more strongly to those having difficulty getting to grips with its various commands. You won't find a better guide to, or a more useful book on, the Amiga than this" Micronet AmigaBASE.

"No other authors have investigated AmigaDOS with the thoroughness of Smith and Smiddy and every page provides useful information. Put off getting that new game, and buy this instead. You won't regret it." Micronet AmigaBASE.

Mastering AmigaDOS 3 - Reference

by Mark Smiddy

ISBN: 1-873308-18-3, price £21.95, 416 pages.

For users of earlier versions of AmigaDOS 1.2, 1.3, 2.04 we publish two similar volumes with all the relevant information you require.

Mastering AmigaDOS 2 Volume One (Tutorial)

by Mark Smiddy and Bruce Smith

ISBN: 1-873308-10-8, price £21.95, 416 pages.

Mastering AmigaDOS 2 Volume Two (Reference)

by Mark Smiddy and Bruce Smith

ISBN: 1-873308-09-4, price £19.95, 368 pages.

Mastering Amiga C

C is without doubt one of the most powerful programming languages ever created, and it has a very special relationship with the Commodore Amiga. Much of the Amiga's operating system software was written using C and almost all of the Amiga technical reference books assume some proficiency in the language.

Paul Overaa has been writing about C and the Amiga for as long as the machine has been in existence. He knows the Amiga-specific pitfalls that can plague the beginner, knows how to avoid them, and above all he knows about C. Best of all he's prepared to share that experience. The result is a book which is guaranteed to get the Amiga owner programming in C as quickly and as painlessly as possible.

This introductory text assumes no prior knowledge of C and covers all the major compilers, including Lattice/SAS and Aztec. What is

more it also covers NorthC – the Charityware compiler – so that anyone who is interested in learning C can do so for just a few pounds. This book assumes no prior knowledge of C and features:

- Easy to follow tutorials
- All major C compilers
- Explanations of special Amiga C features
- Amiga problem areas
- Debugging and testing

Here's what CU Amiga thought of Mastering Amiga C: *"This book has been written with the absolute novice in mind. It doesn't baffle with jargon and slang".*

Writing in Amiga User International, Mike Nelson called Mastering Amiga C: *"Very thorough, Paul Overaa has gone to considerable lengths to keep up to date with developments the in real world of C and the ANSI Standards.....this book will go a long way to help you master C on your Amiga".*

Mastering Amiga C

by Paul Overaa

ISBN: 1-873308-04-6, price £19.95, 320 pages.

FREE Programs Disk and NorthC Public Domain compiler when ordered direct from Bruce Smith Books.

Mastering Amiga Printers

Next to the Amiga itself, your printer is the largest and most important purchase you're likely to make. It's surprising then, that so little help is available for those about to take this step, whether it be for the first time, or for the purpose of upgrading from an old, trusted but limited model to one of today's much more versatile and complex machines. The problem of course is that you can't take one home on trial to find out what it does.

Today's printers are extremely sophisticated and complex devices, with a wide range of capabilities, so it's all too easy to make a mistake at the stage of buying if you don't know what to look for, the right questions to ask and the sort of comparisons to make between similarly priced models from different manufacturers. Since a printer is such a large investment, quite possibly more expensive than the micro itself, choosing the right type and model for your needs is doubly important, because you'll have to live with your decision for a long time.

Unfortunately for the user, neither computer nor printer manufacturers see it as their responsibility to offer guidance or assistance to users in this important purchase.

Mastering Amiga Printers fills this gap perfectly. Making no assumptions about previous printer experience, the explanations begin with the basic principles of how printers work, including a run-down of the different types most commonly used with home and business micros.

After a comprehensive grounding in the abilities and methods of the different types of printer hardware you'll then learn how to install them in the Amiga. Preference selections and printer drivers are thoroughly explained for both Workbench one and two, so you'll know not only which choices to make, but what they mean. There's also a thorough grounding in the direct use of printers from the command line, which you'll need if you want to write your own programs.

Additional chapters take a logical approach to trouble-shooting and routine maintenance, vital to the newcomer. These chapters include the sort of information and knowledge which is normally only available after long experience, the very thing the new user lacks. *Mastering Amiga Printers* is a must for every user who wants to the best out of their Amiga and its printer.

Mastering Amiga Printers

by Robin Burton

ISBN: 1-873308-05-1, price £19.95, 336 pages.

FREE utilities and printer drivers disk when ordered direct from Bruce Smith Books.

Mastering Amiga Workbench 3

The Workbench is one of the most important aspects of the Amiga, yet so few users really understand how to use it to its full potential. From it you can access virtually all of the Amiga's functions and determine how your computer will operate from the moment it is switched on. With the advent of Workbench 3, running under the much enhanced AmigaDOS 3, the options open to the Workbench users are greater than ever before.

In this book Bruce Smith explains everything you will want to know about the Workbench version 3.x, using screen illustrations throughout for ease of reference. The book is geared towards all types of users, whether you have a single floppy disk or a hard disk to operate from.

Starting from first steps the book explains the philosophy of the Workbench and how it ties in with your Amiga. It then moves on to describe the best way to perform basic housekeeping tasks such as disk copying, file transfer and how to customise your own Workbench disks for different occasions and requirements.

The author works his way through each of the menu options with full descriptions of their use, providing many hints, tips and tricks *en route*. By this stage you will already be an accomplished Workbench user, but as the book enters its final stages you will make the transition to expert status as areas such as Preferences, Tools and Commodities are fully explained.

In effect *Mastering Amiga Workbench 3* provides you with a complete guide to your Workbench and Extras disks in an easy to read style guaranteed to upgrade you to full proficiency on your Amiga.

Mastering Amiga Workbench 3

by Bruce Smith

ISBN: 1-873308-31-0, price TBA, pages TBA.

Note: Mastering Workbench 3 will be available late 1993.

Mastering Amiga System

A complete tutorial to Amiga System programming with copious examples. A basic knowledge of C is required but the book begins with short examples which only later build into full-scale programs. Serious Amiga programmers need to use the Amiga's operating system to write legal, portable and efficient programs. But it's not easy! Paul Overaa shares his experience in this introduction to system programming in the C language, the natural language for getting the best out of the Amiga. And there's a free programs disk included in the price with lots of examples and utilities.

Readers will learn how to communicate with the system, how to handle tasks and processes and work with libraries and how to incorporate IFF graphics into their own applications. They will also find out how to harness the power of Intuition, the routines behind the Amiga's classic graphical interface. At no time is the reader left to produce code from general explanations. The author keeps it specific and presents skeleton programs which are fully documented so that they can be followed by the newcomer to Amiga programming. The larger programs are fully-fledged examples which can serve as templates for the reader's own ideas as confidence is gained.

Paul Overaa spells it out in a step by step fashion as he proceeds to explain devices and the custom chips which make the Amiga the all-time great graphical microcomputer. In dealing with a difficult subject, Paul Overaa has avoided merely duplicating standard documentation. Instead he has entered on a journey through the different aspects of the Amiga's system, finding the safest and most effective routes to practical programs. *Mastering Amiga System* is an invaluable purchase for the Amiga programmer who wants to master the system software.

The free disk which accompanies this book contains both source code and runnable versions for all of the programs discussed within the text. In addition to this, a number of document files have been provided which deal with compilers and compiling, ANSI C and K&R C (including notes on ANSI C <-> K&R C program conversion), header files and guru numbers. As well as the tutorial style examples, the disk also contains a number of utility programs including a MIDI message analyser, a generalised minterm expression evaluator, and a recursive search disk routine that could be used as the basis of a number of disk catalogue programs. Source code files for each of these utilities have been provided.

Mastering Amiga System

by Paul Overaa

ISBN: 1-873308-06-X, price £29.95, 398 pages.

FREE programs disk when ordered direct from Bruce Smith Books.

Mastering Amiga Assembler

Although the 68000 processor series is well-documented, the use of assembly language to write efficient code within the unique environment of the Amiga is only now explained in this hands-on tutorial. It teaches how to generate machine code from the popular Amiga assemblers, all of which are supported by the many program examples in this book. These programs also appear as source code and runnable programs on the free support disk.

The Amiga is a powerful machine but the sheer complexity of its operating system has provided a major obstacle to many programmers wishing to enter the world of Amiga assembly language programming. Mastering Amiga Assembler holds the reader's hand, introducing topics through short examples, with diagrammatic explanation where necessary. Longer programs of a practical nature come only later, as the required techniques are practised and mastered.

Topics covered include:

- Fundamental assembly language concepts.
- The 68000 processor and its important instructions.
- The 68000 addressing modes.
- Use of system header files and the official Amiga documentation.
- Working with the Amiga libraries.
- CLI/Shell and Workbench programming.
- Low-level Intuition and graphics programming.
- Introductions to some advanced topics including Exec interrupts and mixed code programming.

- Details of popular assembler environments including HiSoft's Devpac and the PD 68K assembler.

Mastering Amiga Assembler

by Paul Overaa

ISBN: 1-873308-11-6, price £24.95, 416 pages.

FREE disk with programs from the book and extra utilities when ordered direct from Bruce Smith Books.

Mastering Amiga AMOS

AMOS has very quickly developed into one of the most exciting and most accessible programming language environments on the Amiga. Its easy to use interface and familiar BASIC structure are augmented by powerful libraries for games and graphics programming.

Phil South provides a fascinating introduction, tries out all the main features and provides lots of example code for the reader to experiment with. He also brings you right up to date on 3D, graphics, demo programming and developments such as Easy AMOS, AMOS Professional and third party support.

If you are a novice programmer, or are looking for the next step after BASIC then AMOS is for you and you'll enjoy learning AMOS in the company of this step by step tutorial. Like any programming language AMOS has many shortcuts and clever ways of doing things, most of which are presented with examples in this book.

AMOS has also gathered around it a number of powerful utilities which can be used to augment the language in particular ways and provide materials such as windows, text and menus, screens, sprites and bobs, icons and screen blocks, sound and music, object movement, SpriteX, CText and TOME, AMOS compilers, 3D and advanced AMAL. These are covered in detail with information about how to get hold of them and how to join up with AMOS clubs and program libraries.

Anyone following the various series on AMOS which are appearing in the Amiga magazines will find Mastering Amiga AMOS a handy reference and source of programming ideas. The example programs listed have been selected to demonstrate a wide variety of programming techniques.

Whether you have a specific program that you wish to write or you intend to do some programming as a hobby, AMOS opens the door; this book takes you by the hand and leads you through it. Sit yourself down at the keyboard with Phil South's many examples and you'll soon find impressive results can be obtained easily.

Mastering Amiga AMOS

by Phil South

ISBN: 1-873308-12-4, price £19.95, 320 pages.

Mastering Amiga ARexx

ARexx is the Amiga version of the REXX programming language Commodore have adopted as part of Workbench 2/3. With this sort of official endorsement ARexx is set for a very bright future and Amiga users are now asking the questions: Is ARexx a replacement for AmigaBASIC? Can ARexx replace AmigaDOS? Can real programs be written with ARexx? How does ARexx allow programs to talk to each other? In short, people want to know what ARexx is and how it can be used. It is exactly these types of question for which this book provides the answers!

ARexx is in many ways an unusual programming language. For a start it is easy to learn and even those new to the Amiga soon find that simple ARexx script programs give added weight to an already powerful operating system.

The creator of ARexx is Bill Hawes and the general consensus is that his initial interpretation of the REXX language and the subsequent programming has been nigh-on faultless. This has resulted in ARexx being one of those relatively rare third party products officially endorsed by Commodore. ARexx is of course now provided as part of Workbench 2 and 3 but it has been happily running on the Amiga through both 1.2 and 1.3 versions of the operating system. Even nowadays you do not need to be a Workbench 2 user to benefit from ARexx because ARexx is still available as a separate package

This latest guide to ARexx is from programming expert Paul Overaa and is guaranteed to get the Amiga owner into the world of ARexx programming quickly, productively and enjoyably.

Topics covered include: how to install ARexx on any Amiga, introductions to the ARexx language and comparisons with languages like BASIC, in depth discussions of the ARexx language's main features, explanations of how ARexx is used to control other programs, details of ARexx's built-in functions and support libraries, methods for creating well structured ARexx programs, introductions to many advanced ARexx programming topics plus tips and tricks, programming and debugging guidelines, tutorials, and much more.

When ordered direct from Bruce Smith Books, Mastering Amiga ARexx also comes with a free support disk containing the many example programs from the book.

Mastering Amiga ARexx

by Paul Overaa

ISBN: 1-873308-13-2, price £21.95, 336 pages.

FREE disk with program examples and utilities when ordered direct from Bruce Smith Books.

Amiga A600 Insider Guide

A perfect companion for all A600 and A600HD users. This book provides you with a unique insight into the use of Workbench and AmigaDOS on all versions of the Amiga A600.

Assuming no prior knowledge it shows you how to get the very best from your machine in a friendly manner and using its unique Insider Guide steps (see A1200 description below).

Amiga A600 Insider Guide

by Bruce Smith

ISBN: 1-873308-14-0, price £14.95, 256 pages.

Amiga A1200 Insider Guide

Assuming no prior knowledge, it shows you how to get the very best from your A1200 in a friendly manner and using its unique Insider Guide steps. Configuring your system for printer, keyboard, Workbench colours, use of Commodities and much much more has made this the best-selling book for the A1200.

As well as easy to read explanations of how to get to grips with the Amiga, the book features 55 of the unique Insider Guides, each of which displays graphically a set of step by step instructions. Each Insider Guide concentrates on a especially important or common task which the user has to carry out on the Amiga. By following an Insider Guide the user learns how to control the Amiga by example. Beginners to the A1200 will particularly appreciate this approach to a complex computer.

The disks which come with the A1200 contain a wealth of utilities and resources which allow you to configure the computer for your own way of working. The step by step tutorials take you through using these point by point, anticipating any problems as they go. There are also fully fledged programs such as MultiView and ED which can seem impenetrable for the new user but which become clear when observed in use over the shoulder of author Bruce Smith.

Great new features such as the colour wheel, Intellifonts, using MSDOS disks with CrossDos and configuring sound are dealt with in detail. A useful appendix acts as a file locator so that any of the many files on the Amiga disks can be quickly found.

Amiga A1200 Insider Guide

by Bruce Smith

ISBN: 1-873308-15-9, price £14.95, 256 pages.

Amiga A1200 Next Steps

For those who have mastered the very basics of the A1200 this book is the ideal companion to our Amiga A1200 Insider Guide. Leaving the basics of the Workbench and AmigaDOS behind this book takes you the next step and shows you how to get the very most out of your A1200, using both the software supplied and other material readily available.

For example, learn how to use MultiView to write your own adventure game and edit a picture! Create your own fully recoverable Ram Disk, get better results when you print out, recover deleted files. We even show you how to add your own hard disk and run software from it! This is only the tip of the iceberg. Amiga A1200 Next Steps is worth its weight in gold!

Amiga A1200 Next Steps

by Peter Fitzpatrick

ISBN: 1-873308-24-8, Price £14.95, 256 pages.

Assembler Insider Guide

The Amiga Assembler Insider Guide has been written for the new user who wishes to learn to write programs in the native code of the Amiga computer – assembly language.

The approach taken to this, often daunting, subject is designed to achieve practical results. This is done through short examples which demonstrate different programming skills. Each program in the book can be assembled and run in under one minute so the beginner need have no fear of long impenetrable listings. This is programming for the novice, made all the easier though the mini Insider Guides which summarise important operations and fundamental concepts.

The author Paul Overaa writes many articles on programming and runs columns for beginners to programming so his style is carefully crafted for the programming newcomer. Possible stumbling blocks and areas which regularly cause beginners problems are taken head on. The issue of using Commodore libraries, and not using them is dealt with. No extra software is required to run the examples provided. After reading the book, the user will be able to confidently type in and edit source code, assemble it, debug it and run it.

The book is compatible with all the main assemblers on the market. A support disk is available from the publisher which contains the A68K assembler, all the listings in the book, additional utilities and examples (cost £2.00 P&P). With the Amiga Assembler Insider Guide learning assembler on the Amiga has never been easier.

Assembler Insider Guide

ISBN: 1-873308-27-2, Price £14.95, pages 256.

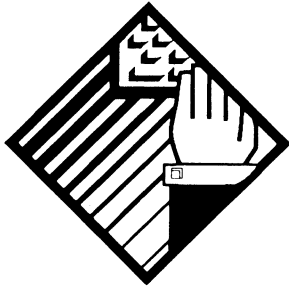
Over you'll find a compatibility chart so that you can check which books go with your version of the Amiga. Note that all our general titles, such as Assembler and System, are A1200 and A4000 compatible.

E&OE.

Book	A500	A500+	A600	A1200	A2000	A3000	A4000
Mastering AmigaDOS2 Vol. 1	Y	Y	Y†	N	Y	Y*	N
Mastering AmigaDOS2 Vol. 2	Y	Y	Y†	N	Y	Y*	N
Mastering AmigaDOS3 Reference	N	N	N	Y	N	Y#	Y
Mastering Amiga Workbench2	N	Y	Y	N	Y	Y*	N
Mastering Amiga Beginners	Y†	Y	Y	Y	Y†	Y	Y
Amiga Gamer's Guide	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga AMOS	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga C	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga Printers	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga System	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga Assembler	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga ARexx	N	Y	Y	Y	Y	Y	Y
Amiga A600 Insider Guide	N	N	Y	N	N	N	N
Amiga A1200 Insider Guide	N	N	N	Y	N	N	N
Amiga A1200 Next Steps	N	N	N	Y	N	N	N

Y† State if you have AmigaDOS 1.3 Y* Earlier versions with AmigaDOS2

Y‡ 80% compatible with 2.1 version Y# Latest versions with AmigaDOS3



Command Index

A

ADDBUFFERS.....104, 237
ALIAS64, 110, 141, 166, 189, 190
.....285, 288, 298
ASK206
ASSIGN.....
....110-116, 164, 228, 245-246, 260
AUTOPOINT299
AVAIL.....136

B

BINDDRIVERS242
BLANKER.....141, 299-300
BREAK160

C

CHANGETASKPRI159
CLICKTOFRONT300
CMD135
COPY19, 84-87, 111, 122, 124
....127, 134-135, 147, 164, 168, 171
....174, 180, 181, 202, 203, 204, 209
....210, 225, 228, 246, 255-260, 271
.....281, 285, 286, 287, 301
CPU136
CROSSDOS301-302

D

DATE.....103, 124, 187
DELETE89-90, 141, 191, 193, 194
DIR25-27, 49, 52, 57, 58, 78, 86
.....104, 112, 122, 140, 164, 186
.....187, 210, 230, 246, 281
DISKCHANGE292
DISKCOPY35-36, 64, 294

E

ECHO61, 118, 141, 153, 164-166
.....183, 187, 190, 191, 195, 199
.....202, 203, 218, 219, 271
ED131, 185-188, 248, 255
ELSE200, 202
ENDIF.....200, 202, 204, 219
EVAL167-169, 183, 218, 213-222

EXCHANGE.....297, 304-305
 EXECUTE.....146, 186, 190, 193
 194, 201, 202, 204, 228, 236, 247

F

FAILAT.....204
 FAULT.....60
 FILENOTE.....130
 FIXFONTS.....134, 260
 FKEY.....302-304
 FORMAT.....34-35, 64, 78, 256

G

GET.....162, 170
 GETENV.....162, 165, 170

I

ICONX.....209-211
 IF.....161, 163, 200-207, 218
 INFO.....105
 INSTALL.....255, 256
 IPREFS.....228, 232, 234

J

JOIN.....132

L

LAB.....199, 200, 204, 218
 LIST.....52, 97-102, 116, 124
131, 143-150, 172, 173
191, 193, 194, 247

M

MAKEDIR.....77-81, 86, 122
228, 256, 282
 MAKELINK.....122
 MEMACS.....315-314
 MORE.....125, 174, 191, 193
287, 171, 173, 231
291, 292, 301
 MOUSEBLANKER.....304
 MULTIVIEW.....92

N

NEWSHELL.....154, 163, 180
 NOCAPSLOCK.....304

NOFASTMEM.....136

P

PATH.....107
 PROMPT.....63, 162, 165
 PROTECT..118, 147, 190, 192, 285

R

RELABEL.....35, 292
 REMRAD.....293
 RENAME.....19, 87-89, 90, 209
 RESIDENT.....56-59, 83, 104
121, 256
 RUN.....125, 135, 137, 141
155-156, 157-158, 180
287, 297, 298, 299

S

SAY.....175-176
 SEARCH.....129-130, 205-206
 SET.....162, 169
 SETCLOCK.....103
 SETDATE.....123, 124
 SETENV.....152, 162-164, 167
169, 170, 197
 SETFONT.....133
 SETKEYBOARD.....134
 SETMAP.....134, 245
 SETPATCH.....228, 234, 239
 STACK.....159
 STATUS.....158-159

T

TYPE.....127, 140, 164-165, 174
218, 281, 285, 287

U

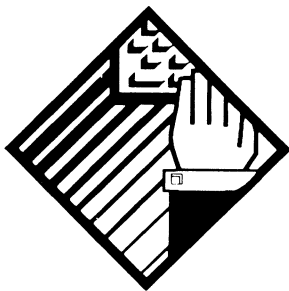
UNSETENV.....168

V

VERSION.....106, 227, 233, 259

W

WAIT.....157, 158
 WHICH.....59, 109
 WHY.....60, 161, 203



Index

String Conversion Operators

%-n.n (LIST).....	148-150
%0n (EVAL).....	216
%A (LIST).....	148
%B (LIST).....	148
%C (EVAL).....	216
%C (LIST).....	148
%D (LIST).....	148
%K (LIST).....	148
%L (LIST).....	148
%N (EVAL).....	216
%N (in the prompt).....	63
%P (LIST).....	148
%R (in the prompt).....	63
%S (in the prompt).....	63
%S (LIST).....	147
%T (LIST).....	148
%Xn (EVAL).....	216
' (EVAL conversion).....	215
0x (EVAL conversion).....	215
# (EVAL conversion).....	215

Script Operators

.BRA.....	196
.DEF.....	195
.DOLLAR.....	196
.DOT.....	196
.KET.....	196
.KEY.....	191-197

Pattern Matching

#?.....	71
?.....	72, 165
#.....	42, 72
~.....	72
'.....	42, 72
().....	72
[...].....	72
[...].....	72
#[...].....	72
#[...].....	72

Templates

"" (dummy argument)	86
/A	49, 194
/F	49, 194
/K	49, 194
/M	49
/N	49
/S	49, 194
{...}	48
(OR)	47
? command help	42, 49

Command Strings

*	42, <i>see also</i> CONSOLE:
*e (escape)	62, 65-66
*e[(CSI)	183
*n (newline)	61-62
` (insert command)	153

Environmental Variables

\$	163, 167, 243
\$Process	197
\$VER	227
<\$ \$>	197
RC	161, 169
Result2	161, 169

Symbols

/ (directory separator)	40, 42
.....	81, 91
: (root directory)	40, 42
< (re-direct CIS)	141, 166
> (re-direct COS)	140-141, 166
>> (append file)	141

A

A: (ASSIGN example)	116
ADDBUFFERS	104, 237
Adverbs (in commands)	46
AGA	20
ALIAS	64, 110, 141, 166, 189
.....	190, 285, 288, 298

ALL (DIR option)	26, 93
ALL (LIST option)	145
ALL (SEARCH option)	129
ALL (WHICH option)	109
Amiga-E	25
AmigaDOS 1.3	51
AmigaDOS 2.x	21
AmigaDOS 2.1	21
AmigaDOS 3	21
ANSI and CON:	182
ANSI function table	183
Archive flag	118, 120
ASCII	127
ASK	206
ASSIGN	110-116, 164, 228
.....	245-246, 260
Assignment list	114
Assignment names	110
AUTO (CON: option)	180
Auto-startup	247
AUTOPOINT	299
AUX:	178
AVAIL	136

B

B: (ASSIGN example)	116
BACKDROP (CON: option)	181
Backup	32-33
Batch files	20
Baud rate	270
Binary files (viewing)	128
Binary operators	214-215
BINDDRIVERS	242
BLANKER	141, 299-300
BLOCK (LIST option)	144
Boot	20, 23, 225, 254
Boot disk directories	263-268
Bootstrap code	255
BREAK	160

C

C directory.....58
 C directory.....264
 CATTY (script)290
 CD.....76-77, 111, 114, 140, 155
186, 187, 186, 187
 CES.....140
 CHANGETASKPRI.....159
 CHATTER (script).....288
 CIS140, 168
 Classes directory266
 Classes/DataTypes266
 Classes/Gadgets266
 CLI51
 CLICKTOFRONT300
 Clipboards (directory)70, 264
 CLIPS:112
 CLIPS:230
 CLOSE (CON: option).....180
 CMD135
 Co-operative multi-tasking.....153
 COLSTART (SORT option)131
 Command (C) directory257
 Command insertion (Shell).....152
 Command Line Interface51
 Command Sequence Introducer
127
 Commands (defined)23
 Commodities297-306
 Commodities in User-startup ..306
 CON:154, 179-183
 CON: parameters179-181
 Console Error Stream.....*See CES*
 Console Input Stream.....*See CIS*
 Console Output Stream.....*See COS*
 Console.device*See CON:*
 CONSOLE: .141,171, 172, *See CON:*
 COPY (to printer)134-135
 COPY (with PIPE:)127

COPY.....19, 84-87, 111, 122,124
147, 164, 168, 171, 174, 180
181, 202, 203, 204, 209, 210
225, 228, 246, 255-260, 271
281, 285, 286, 287, 301
 Copy and Paste in CON:.....181
 COS140
 CPU136
 CROSSDOS301-302
 CSI127, 183
 CTRL-\156
 Cursor.....52
 Customising disks253-269
 Cylinders31

D

DATE.....103, 124, 187
 DATES (LIST option).....145
 DCFS31
 Debugging boot disks.....263
 DELAY= (ICONX).....211
 DELETE..89-90, 141, 191, 193, 194
 Deleteable flag.....118, 119
 Devices (DEVS) directory258
 Devs directory264
 DEVS:111
 DF0:41, 57, 75
 DF0:????30
 DF0:Busy.....32, 33
 DF1:41
 DF2:41
 DIR (advanced)91-96
 DIR.....25-27, 49, 52, 57, 58, 78
86, 104, 112, 122, 140, 164
186, 187, 210, 230, 246, 281
 DIR Interactive options93-95
 Directories.....39, 78, 107-116
 see also: Sub-directories, Drawers
 DIRS (DIR option).....93
 DIRS (LIST option).....144
 Disk device41
 DISKCHANGE.....292

DISKCOPY35-36, 64, 294
 Diskloaded command56
 Dot-info (file)20, 78
 Drawers (Workbench)37-39
 see also: Directories
 DRIVE (option in FORMAT)34

E

ECHO61, 118, 141, 153
 164-166, 183, 187, 190, 191
 195, 199, 202, 203, 218, 219, 271
 ED131, 185-188, 248, 255
 ED-Startup248-252
 ED: Reference table351-360
 EDITOR (variable)169
 ELSE200, 202
 ENDIF200, 202, 204, 219
 Env (directory)70
 Env-archive directory261
 Env-archive directory267
 Env/multiview (directory)70
 Env/sys (directory)70
 ENV:112, 164, 178, 234
 ENVARC:112, 113
 Environmental variables...161-170
 ERROR (IF option)201
 Errs (INFO screen)105
 EVAL .167-169, 183, 218, 213-222
 EVAL in scripts217
 EVAL LFORMAT215-216
 EVAL operators214
 EXCHANGE297, 304-305
 Exec152
 Executable code83
 EXECUTE (lib. function)55
 EXECUTE146, 186, 190, 193
 194, 201, 202, 204, 228, 236, 247
 Executeable flag.....118, 119
 EXISTS (IF option)201, 202
 Expansion directory268
 External Command55

F

FAIL (IF option)201, 204
 FAILAT204
 FAULT60
 FIFO (buffer)282
 FILE (SEARCH option).....129
 File server269
 Filename extensions43, 91
 Filenames42
 FILENOTE130
 FILES (DIR option)93
 FILES (LIST option)144
 FIXFONTS134, 260
 FKEY.....302-304
 FONTS (directory)26, 260, 265
 Fonts (detailed).....307
 FONTS: (assignment)112
 FORM128
 FORMAT34-35, 64, 78, 256
 Formatting29-31
 Fountain310
 Free (INFO screen)105
 FROM disk.....32
 Full (INFO screen)105

G

GEM.....21
 GET162, 170
 GETENV162, 165, 170
 GUI.....38
 Handle139
 HELP:112
 HEX (option in TYPE).....128
 Host machine.....269
 HOST-CHAT (script)287
 HostRead (script)277

I

I/O handles156,
 *see also CIS/CES/COS*
 ICONX209-211

- IF (options)201
- IF.....161, 163, 200-207, 218
- IF with numbers206-207
- ILBM128
- INFO105
- INSTALL255, 256
- Intellifont.....307
- INTER (option in DIR)93
- Interactive console51
- Internal command55
- International mode31
- Intuition.....38
- IPREFS228, 232, 234
- J**
- JOIN132
- K**
- Keymaps directory265
- KEYMAPS:112, 230
- KEYS (LIST option)143
- Keywords (in commands)46
- Kickstart (variable)169
- Kickstart56
- L**
- L directory265
- L:112, 171
- LAB.....199, 200, 204, 218
- Launched152
- LFORMAT (LIST option).....145-150
- LFORMAT parameters147-148
- Libraries (LIBS) directory259
- Libs directory266
- LIBS:112
- LIFO (buffer)282
- Line history52
- Linking objects121
- LIST.....52, 97-102, 116, 124
.....131, 143-150, 172, 173
.....191, 193, 194, 247
- LIST in scripts.....189
- LIST ouput tail99
- LIST output body99
- LIST output header98
- LIST with dates99-101
- LIST with patterns101-102
- Load file.....128
- Local variables.....168
- LOCALE:112, 234
- M**
- Mail-2-Host (script).....276
- Mail-2-Remote (script)276
- MAKEDIR.....77-81, 86, 122
.....228, 256, 282
- MAKELINK122
- MEMACS.....315-314
- MEMACS: Command files..339-342
- MEMACS: Edit Menu321-325
- MEMACS: Extras Menu334-339
- MEMACS: Line Menu328-329
- MEMACS: Project Menu318-321
- MEMACS: Search Menu.....331-333
- MEMACS: Starting316-317
- MEMACS: Window Menu ...325-328
- MEMACS: Word Menu.....330-331
- Memory, FAST toggle.....136
- Memory, testing.....136
- MORE125, 174, 191, 193, 287
.....171, 173, 231, 291, 292, 301
- Mountlist291
- MOUSEBLANKER304
- Moving an object87
- MS-DOS20
- Multi-tasking151-156
- Multi-tasking in scripts196-197
- Multi-user debugging279
- MULTIVIEW92
- N**
- NAME (option in FORMAT).....34
- Name (INFO screen)105

NEWSHELL.....154, 163, 180
 NIL:140, 156, 166, 172, 234
 NOBORDER (CON: option).....181
 NOCAPSLOCK.....304
 NODRAG (CON: option).....180
 NOFASTMEM136
 NOHEAD (LIST option)143
 NOLINE (option in ECHO).....199
 NORES (WHICH option)109
 NOSIZE (CON: option)180
 NOT (IF option)201
 Nouns23
 Nursery English23

O

Objects (advanced)117-124
 Objects20, 45, 58, 83-90
 OPT (in DIR)25, 93
 Outline fonts: Installing ...309-312

P

PAR:177
 Parallel.device135
 Parity270
 Parsing.....45
 PAT (LIST option).....144
 PATH.....107
 Pathname40
 Patterns (in SEARCH)129, 130
 PC0301
 PIPE: (with MORE).....126
 PIPE:173-175
 Pipes281-290
 Pre-emptive multi-tasking152, 153
 Prefs directory.....267
 Printers directory265
 PRINTERS:112, 230
 Printing messgaes187
 Priority.....159
 Process169

Processor control.....136
 Project (Workbench)19
 PROMPT63, 162, 165
 PROTECT..118, 147, 190, 192, 285
 Protection flags117
 PRT:176
 Pure flag118, 120

Q

QUICK (FORMAT option)31
 QUICK (SEARCH option).....129
 QUIET compared to NIL:141
 Quitting ED186

R

RAD.....291-296
 RAD - booting from294
 RAD - multi-disk.....294
 RAD - removal293
 RAD - sizing292
 RAD:.....172
 Ram Disk67-70
 RAM:172
 RAW:177
 RC (system variable).....161, 169
 Re-direction.....139-141
 *see also* CIS/COS
 Readable flag118, 120
 Redundant copy83
 RELABEL35, 292
 Relative paths.....81
 Remote machine269
 RemoteRead (script)278
 REMRAD293
 REMT-CHAT (script)290
 RENAME19, 87-89, 90, 209
 Requester.....30
 RES (WHICH option)109
 Reset.....69
 RESIDENT56-59, 83, 104
 121, 256

- Resources152
- Result2 (system variable)
.....161, 169
- Returncode203-204
- REXX directory267
- REXX:.....112, 230
- Rexxc directory268
- Root directory39
- Root directory257
- Round-robin multi-tasking152
- RUN.....125, 135, 137, 141, 155
.....157-158, 180, 287, 297, 298, 299
- RUN with +.....155-156
- RUNBACK (ALIAS example)141
- S**
- S directory266
- S:.....112, 224
- SAY175-176
- Scatter load.....128
- SCREEN (CON: option).....181
- Script flag118, 120
- Scripts.....20
- Scripts directory261
- Scripts – flow control199-207
- Scripts — introduced.....189-197
- SEARCH (IF example)205-206
- SEARCH.....129-130
- SER:.....177
- Serial.device135
- SET162, 169
- SETCLOCK.....103
- SETDATE123, 124
- SETENV152, 162-164, 167
.....169, 170, 197
- SETFONT133
- SETKEYBOARD134
- SETMAP.....134, 245
- SETPATCH.....228, 234, 239
- Shell editing commands53
- Shell history searching.....53-54
- Shell-Startup262
- Siblings39
- SIMPLE (CON: option).....180
- SINCE (LIST option).....144-145
- SIT (command analogy)45-49
- Size (INFO screen).....105
- SKIP.....199, 200, 204, 219
- SMART (CON: option).....180
- SMUS128
- SORT131, 191, 193, 194
- SPAT.....227, 276
- SPEAK:.....175-176
- STACK159
- Stack (buffer)282
- Startup sequence 2239
- Startup sequence 2.1233
- Startup sequence 3.0227
- Startup sequences223-252
- Startup, Stopping.....226
- Startup-Sequence.....
.....20, 224, 255, 256
- STATUS158-159
- Status (INFO screen)105
- Stop bits270
- Stream139
- Stream handle.....139
- SUB (LIST option)144
- Sub-directories39, 79
- Synopsis34, 46, 50
- SYS:.....110
- System directory267
- System request30
- System-configuration224
- T**
- T (directory).....70, 267
- T:228, 234
- Task stack.....159
- Task switching.....152
- Template.....34

Template types.....	50
Text files (viewing)	127
TO (LIST option)	144
TO disk	32
Tool (Workbench)	19
Transient command	56
Trashcan directory	268
TYPE	127, 140, 164-165, 174
.....	218, 281, 285, 287

U

Unit (INFO screen)	105
UNSETENV.....	168
UPTO (LIST option)	144
Used (INFO screen)	105
Utilities directory	268

V

VAL (IF option).....	207
.....	<i>see also table on 201</i>
Variable indirection.....	170
Variables in calculations	167
Vault (verb example)	24
Verbs	23, 45
VERSION.....	106, 227, 233, 259
Version.library.....	259
VGAOnly	231, 236
Virtual terminal	51
Volume names	113

W

WAIT (CON: option)	180
WAIT	157, 158
WARN (IF option)	201
WBStartup directory	268
WHICH	59, 109
White space	47
WHY	60, 161, 203
Wildcard examples	73
Wildcards.....	71-73
WIMP	19, 38
WINDOW (CON: option)	181

WINDOW= (ICONX).....	210
Word length.....	270
Workbench (system variable)	169
Workbench.....	19, 29, 37, 68
Writeable flag	118, 119

X

Xerox PARC	38
------------------	----

Mark Smiddy and Bruce Smith

Mastering AmigaDOS 3 – Tutorial

Mastering AmigaDOS 3 is the de-facto two-volume guide to AmigaDOS versions 3, 2.1 and 2, covering the Amiga A1200, A4000, A600, A3000 and A500 Plus microcomputers. *Mastering AmigaDOS 3 – Tutorial* is a complete guide to learning and using AmigaDOS. Ideal for the novice, this volume starts from the first principles and leads you comfortably through to become an accomplished AmigaDOS user.

Co-authored by the world's *best known* AmigaDOS author, Mark Smiddy, and *beginner's champion* Bruce Smith, it adopts a modular approach with practical projects to get you started. Foundation-building early chapters lead you gently into the subject without sacrificing the subtle nuances that make AmigaDOS special. Climaxing with technically-advanced examples, it fully documents multi-tasking, multi-users, fonts, self-booting application disks and much, much more. Leave the Workbench behind and join the extraordinary world of AmigaDOS 3.

"No other authors have investigated AmigaDOS with the thoroughness of Smith and Smiddy and every page provides useful information. Put off getting that new game, and buy this book instead. You won't regret it."

– Bruce Smith Books –

Publishers of the World's Best Selling Amiga Books

£21.95

ISBN 1-873308-20-5



9 781873 308202