Covers AmigaDOS Releases 2 and 3

AmigaDOS Reference Guide

Fourth Edition

AmigaDOS

SHELDON LEEMON

The complete guide and tutorial to the convenience, flexibility, and power of AmigaDOS. For all versions of AmigaDOS including 1.3, and Releases 2 and 3.

Compute's AmigaDOS Reference Guide

Fourth Edition

Sheldon Leemon

COMPUTE Books
Greensboro, North Carolina

Editor: Stephen Levy

Cover design: Lee Noel

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

0-87455-268-0

COMPUTE Books, 324 West Wendover Ave., Suite 200, Greensboro, NC 27408, is a General Media company and is not associated with any manufacturer of personal computers. Amiga is a trademark of Commodore-Amiga, Inc.

Contents

Foreword

The Workbench, the graphic-based interface that offers icons, pull-down menus, and multiple windows, isn't the only way to operate the Amiga. A more direct way to control this personal computer is called the CLI or Command Line Interface.

Now in its fourth edition, this popular best-selling reference guide covers all versions of AmigaDOS including the new Release 2 and Release 3. *AmigaDOS Reference Guide, Fourth Edition* has been fully updated and shows you how to access this operating environment and how to use its commands.

When COMPUTE published the first edition of this guide we expected it to sell well—but we never expected the huge success it has seen. It has become the standard AmigaDOS reference guide for Amiga users around the world. Best selling author Sheldon Leemon has kept this guide up-to-date, adding new information and revising the text to meet the needs of all Amiga users. Included here is valuable, clear and concise information not only about the CLI, but also the Amiga file system, devices, batch commands, and the two editors, Ed and Edit.

This is the indispensable AmigaDOS guide and tutorial that has helped thousands of Amiga users become CLI experts.

Chapter 1

Introduction to AmigaDOS

The Workbench environment makes it extremely easy for first-time users to learn to use the Amiga personal computer. With its pull-down menus and its pictorial representation of files and subdirectories, Workbench insulates you from the harsh realities of a command-driven DOS (Disk Operating System) environment. But this ease of use has its price. In accepting the Workbench environment, you give up some of the flexibility and power afforded by a command-driven DOS. More importantly, the Workbench also hides much of the nuts-and-bolts details of what goes on at the DOS level. Leaving Workbench, users are somewhat unprepared when things don't work exactly as expected.

While the Workbench approach has its share of advocates, many users of the old-style DOS interface insist that they can run a program faster by simply typing its name on a command line than they could by opening a disk icon and double-clicking on the program icon. And for some Amiga users, the greater control offered by a command-driven DOS interface is a matter of substance, not style. Unless you are running Release 2 (or higher) of the operating system, there are some things that you just can't do from the Workbench.

Before Release 2, the Workbench could only create a *display* for disks, tools (program files), projects (data files), and drawers (subdirectories) for which there existed a corresponding disk file whose name ends in *.info* (for instance, *Preferences.info*). The .info files contain information about the type of object which the icon represents and the graphic representation of the icon itself. But there are many files on the Workbench disk that are *not* represented by icons. These files include a simple sorting utility program and a screen-oriented text editor. These programs could be well-used by many Amiga owners, but most don't even know that they're present since they are not visible from the Workbench unless you've turned on the "Show All Files" option of the Workbench 2 "Windows" menu.

Another feature of AmigaDOS that older versions of the Workbench do not directly support is the use of command sequence files (known in the MS/PC-DOS world as *batch files*). These allow you to automate a job which requires

several programs to be run in sequence, such as operating a compiler and linker in order to produce an executable program. And while it's not possible to send the directory of files on a particular disk to your printer from the pre-2.0 Workbench (unless you write a program specifically for that purpose), it is easy to do so from the CLI (Command Line Interface). Release 2 adds an "Execute Command" feature that allows you to perform the same tasks from the Workbench, but this feature merely invokes a CLI-like environment which can be best used by someone who understands that environment.

Unlike some personal computers, the Amiga does not limit you to one restricted operating environment, not even one so friendly as the Workbench. Its operating system was designed to provide alternative ways to use the computer, to meet the needs of as many kinds of people as possible. This philosophy is evident in the way Amiga programs allow you to substitute control key sequences for commands normally carried out by moving and clicking the mouse. Even the Workbench lets you use the keyboard instead of the mouse. It should come as no surprise, then, that the Amiga also offers the kind of command line interpreter that is so familiar to users of MS/PC-DOS and UNIX. On the Amiga, this environment is known as the CLI or shell. Just as Commodore has enhanced Workbench functions in Releases 2 and 3, so too has it made the shell even more useful in these new versions. *COMPUTE's AmigaDOS Reference Guide, Fourth Edition* will tell you how to find this operating environment and how to use its multitude of powerful and flexible commands.

What's Here

In addition, you'll find explanations of AmigaDOS's underlying concepts. These concepts will be helpful not only when you use the CLI, but will also expand your understanding of the Workbench and how to operate within it. If you have a single-drive system, for example, you've probably noticed that when you try to get a directory of the BASIC disk from BASIC, you're prompted to put in the Workbench disk. When you swap disks, you receive a directory of the Workbench disk instead. Knowing a little bit about how DOS operates and what files it looks for can eliminate a lot of this disk swapping. The RAM disk and recoverable RAM disk (RAD:) also offer computing power impossible through the Workbench alone. With a RAM disk, you'll have instant access to commands which normally must be read from the disk, such as the one which is used to produce a directory listing.

The introductory manual which comes with the Amiga personal computer assumes that AmigaDOS is of interest only to software developers. That's simply

not true. Thousands of people—people who don't write software for a living—are interested in knowing more about their computers and learning how to get the most out of them. If you fit that category, this book will help as you explore the power of your Amiga computer.

The Workbench Versus the CLI

The friendly Workbench environment that you see when you boot up your Workbench disk is actually an application—in other words, a program—and *not* part of the operating system. In fact, the computer *starts* in CLI mode and loads the Workbench program automatically through the use of a command file (you'll be hearing more about command files later).

Workbench's purpose is to interpret the choices you make when you move the mouse pointer to various icons and click the button. As such, the Workbench functions often have a close correspondence to DOS concepts. The drawer icons you see on the Workbench desktop, for instance, represent the normal subdirectories created by DOS. And the Trashcan icon represents a subdirectory named *Trashcan.* When you drag an icon to the trashcan, its corresponding file and that of its icon are transferred to the *Trashcan* subdirectory. When you select Empty Trash, the files which have been moved to the *Trashcan* subdirectory are deleted.

Some similarities between the Workbench and CLI environments are more superficial. When you double-click on a tool (program file), the Workbench prepares a suitable environment and runs the program. The same thing happens when you type ***RUN*** *program name* from the CLI. But some Workbench programs are not meant to be run from a CLI, and most CLI programs are not meant to be run from the Workbench. In fact, none of the CLI commands found in the *c* directory of the Workbench disk can be run from any version of the Workbench prior to 2.0. Part of the reason for this is that older versions of the Workbench recognize a file only if it has a corresponding file of icon information ending in .info. Since none of the CLI command files has an icon file, none of them shows up on a pre-2.0 Workbench.

But even if these *did* have icon files, the environment that CLI prepares for a program is different enough from the environment provided by the Workbench that these early CLI command programs still could not run under the Workbench. For one thing, from the Workbench you may pass instructions to a program to load a project (a data file that the program uses) by double-clicking on the project's icon. Programs that use the CLI expect you to pass instructions by typing them on the same line with the program name. The command line

```
COPY "oldfile" to "newfile"
```

for instance, tells the program named Copy which file to copy and what name to give the new copy.

Getting to the CLI or Shell Environment

If you're using a recent version of AmigaDOS, getting to the CLI environment is no problem. All you have to do is open the Workbench disk and double-click the icon for the "Shell" program, which creates an enhanced CLI window. If you are using an earlier version of the operating system, however, things are a bit more complicated.

Pre-1.3 versions of the Workbench do not have the Shell program, nor do they have visible icons for the CLI program (the *CLI.info* file in the *System* subdirectory has been renamed *CLI.noinfo.*). You must use the Preferences program to make the CLI icon in the System drawer of the Workbench visible. Start the computer with the Workbench disk (preceded on the Amiga 1000 by the Kickstart disk), and an icon representing that Workbench disk appears on the screen. Open this disk by double-clicking on the icon, or by selecting it and then selecting Open from the menu. A window will appear with icons representing the programs on the disk. Start the program called Preferences by double-clicking on its icon, which looks like an Amiga with a question mark on top of it. On the left side of the Preferences screen, you'll see a box marked *CLI,* just above the Reset Colors box and below the box where you choose between 60- and 80-column text.

The CLI box is divided into two parts, one marked *On*, the other *Off.* The Off box is highlighted in orange to show that the CLI icon is turned off. Click the On side of the box so that it turns orange. While you're at it, you can set your other preferences, such as text size and a printer driver, if you've not done so already. Save your new preferences by clicking on the Save box at the lower right of the screen. This renames *CLI.noinfo* as *CLI.info.*

Now, double-click on the System drawer to open its window. If you have already opened the System drawer before running Preferences, you must close the drawer and open it again in order to let your new preferences take effect since the Workbench checks for icon files only when it opens a drawer. The window which appears now contains an icon marked *CLI* (it looks like a box with the characters *1*> inside). Double-click on the icon. A window now displays on the screen, with the title New CLI Window in its title bar and the prompt 1> awaiting your command. (To get started, see Chapter 2.)

There's another, even easier way to get to a CLI window. During the boot-up process, and after you insert your Workbench disk, the screen turns from

white to blue (gray under Release 2), and a sign-on message appears which reads *Copyright (C) 1985 Commodore-Amiga, Inc.* When you see this message, hold down the CTRL key and press the D key at the same time. This stops the execution of the command file that loads the Workbench. ***BREAK - CLI* shows on the screen, and under this, the familiar 1> prompt.

Creating a CLI Workbench Disk

If you're planning to use the CLI environment often, this process of opening the Workbench and Shell icons (or the System drawer and the CLI icon) to get to the CLI can become tedious. Since computers are supposed to make things easier, doesn't it seem reasonable to expect the Amiga to do all this for you? With a bit of setup work on your part, your computer can automatically open a Shell or CLI window at startup time, or bypass loading and running the Workbench altogether.

In order to make a CLI Workbench disk, you should make a copy of your Workbench disk, change the command file that automatically closes the CLI window and loads the Workbench when the disk starts. To get you through this, the procedure is completely outlined for you below, step by step.

Make a Copy of Your Workbench Disk

You can do this either from the Workbench or from the CLI. Let's assume you'll use the CLI, since you presumably already know how to copy a disk with the Workbench. First, bring up the CLI or Shell by double-clicking its icon on the Workbench disk, or by booting the Workbench disk and then interrupting the loading process with a CTRL-D key combination when the blue (or gray) screen appears. From this point, the procedure is slightly different for single- and dual-drive systems.

Single-drive systems. When the CLI prompt (1>) appears, you may use the DISKCOPY command to copy the Workbench disk. Get out a blank, new disk for the copy. Remember, any information that's on this disk will be lost when you copy to it. Type:

```
SYS:SYSTEM/DISKCOPY df0: TO df0:
```

and press RETURN. The copy program will prompt you when to put in the disk to copy FROM (your original Workbench disk) and when to put in the disk to copy TO (your blank disk). You'll have to swap the FROM and TO disks a number of times with a single-drive system. The copy program will tell you when the copy process is complete.

Dual-drive systems. When the CLI prompt (1>) appears, leave the Workbench disk in the internal drive, and place a new, blank disk in the external drive. Type:

```
SYS:SYSTEM/DISKCOPY df0: TO df1:
```

and press RETURN. You'll be prompted to put the FROM disk in drive df0: and the TO disk in drive df1:, but since both disks are where they should be, merely press RETURN. The copy program will tell you when the process is completed. Place the disk which contains the copy of the Workbench into the internal drive.

Getting Going with CLI

Restart the computer with your new disk. Press the CTRL key and both Commodore/Amiga keys (the closed Amiga or Commodore key—on the left side of the space bar—and the open Amiga or Commodore key—on the right side of the space bar) at the same time to restart. Your new disk is now the system disk, which will save you some disk swapping later.

Bring up the CLI. Use the CTRL-D combination to stop the Workbench from loading during the boot process, or open the Shell icon (or System drawer and CLI icon). If you use the Workbench CLI, you may find it convenient to size this window to full-screen by moving it with the drag bar to the top left of the screen and pulling the sizing gadget down to the bottom right.

Edit the command file. This is used to load the Workbench automatically when you start the computer. You'll use the system screen editor program—called *ED*—to change the *startup-sequence* file in the *s* directory. To start the editor, enter

```
ED s/startup-sequence
```

at the 1> prompt (whenever you see text in `this font`, press the RETURN key at the end of the line). A new screen appears, showing the contents of this text file. A text cursor shows at the top left corner. If you haven't changed the default system colors, it will be orange (Release 2 users will have a blue cursor on a gray screen). Use the down-arrow cursor key to move this cursor down to the second from last line of the file (it should be resting on the first letter of the line that reads *LoadWB.)*

If you wish to open a Shell or CLI window automatically and are using Workbench 1.3 or above, type in the line

```
NEWSHELL
```

and press RETURN. Users of older versions should type in the line

```
NEWCLI
```

and press RETURN. If you want to avoid opening the Workbench, you may also delete the line that reads *LoadWB*. Make sure that your cursor is at the beginning of that line, hold down the CTRL key, and press the letter "b". Not opening the Workbench can be helpful if you are short on memory, but otherwise, you probably want to leave this line as it is.

After you have made the desired changes, press the ESC key (found in the upper left of the keyboard). An asterisk appears at the bottom of the screen, and the cursor is now next to it. Type the letter "x" and press Return. This exits the Ed program and saves your revised file to disk.

Make a backup copy of the disk right now, and put the original away so that you can make clean copies of the disk in the future (unless you want to go through these six steps every time). If you have only one drive, you'll find it particularly convenient to have all of the CLI commands on the same disk as your application programs. To make a new disk that contains both the CLI commands and the application program, simply copy those application programs onto duplicates of this master CLI disk. If you're really pressed for space, you may have to delete some of the less useful commands, printer-driver files for printers that you don't have, character font files, and so on. To determine which files you can afford to delete, see Appendix A, which lists all the directories on the Workbench disk.

Chapter 2

The CLI Environment

When you turn on the computer and insert the Workbench disk into the disk drive, the Amiga's operating system sets up a *task* (one of the programs that can run simultaneously under a multitasking system such as AmigaDOS) called a *CLI process.* The job of the CLI is to accept commands to run a program. When the CLI finds the program, it loads the program, prepares its environment, then passes control to the program. After the program finishes, control is passed back to the CLI, which waits for the next command. Although the system starts up only one CLI, you may start others yourself to run multiple tasks simultaneously.

The first thing that the initial CLI process does is to check whether there is a command file in the *s* directory called *Startup-Sequence.* If there is, the commands listed in that file are executed automatically (see Chapter 5 for more detailed information about command sequence files). On the standard Workbench disk, this file contains commands to load and run the Workbench and end the CLI process. But if there's no command present to load the Workbench, once the command file is executed, the CLI process prints its prompt message (1>) and waits for further orders.

The AmigaDOS CLI is limited to the simplest of functions. It starts in interactive mode, which means that it prints its 1> prompt and waits for you to type something. It simply sits, letting you type until it sees that you have entered a special editing character or pressed RETURN. The editing characters invoke some minor screen-editing functions which are described below. But when you press RETURN, the CLI looks at the whole line that you've entered.

It interprets the first word (a series of characters that end with a space) as a filename. The CLI then tries to load a program file with that name. If it cannot find the file, it prints an error message followed by another 1> prompt. If it finds the file, the CLI tries to load it as a program. Since program files have a structure that the CLI recognizes, it can tell whether or not the file is an executable program. Again, an error message and the 1> prompt are displayed if the file is not an executable program. If the file exists *and* is an executable program, the CLI

loads the program into memory, prepares a stack area for the program to use as workspace, tells the program where to find the rest of the text on the command line in case it wants that text as instructions, and passes control to the program. Once this happens, the CLI cannot accept user input until the program passes control back to it.

Let's break this simple task into its component parts and examine them in detail. We'll start with the process of accepting text characters that you type in.

The CLI Console

The console device that the CLI uses to accept keyboard input and display the results operates much like an old-fashioned Teletype terminal—it can deal with only a single line of text at a time. This *command line* may be up to 512 characters long (255 if you're using an older version of AmigaDOS). It's possible, therefore, that a single command line can occupy more than one line on the screen. As far as the console device is concerned, you're still entering text on the same line until you hit the RETURN key. When you've typed in 512 characters (more than eight or so screen lines, depending on the column width of the screen), the console refuses to accept any additional keyboard input.

One of the less pleasant aspects of using a simple line-oriented editor such as the console device is that you cannot use the cursor keys to move to another command line on the screen, edit it, and use the revised line. Each time you issue a new command you have to enter the entire command line from scratch. In fact, if you use one of the older Workbench versions, you cannot even use the cursor keys to edit the line you're on. If you make a mistake at the beginning of a line, you have to erase the whole line and start over. To remedy this situation, Workbench 1.3 added a new console device called Newcon:, which performs line editing functions. It also added a shell handler that introduced features like aliases and command history. We'll be discussing these features, which became a standard part of the operating system with Release 2, a little later on.

CLI Editing

Because of its limited line-editing capabilities, the console device recognizes only a very few special characters as editing commands. Some of these are useful for working with the CLI, while others merely enable you to control the color and appearance of the text that the console device prints to the screen (see Chapter 4 for more about this device). In summary, here are the editing commands:

Useful Editing Features

Key(s)	Function
BACK SPACE or CTRL-H	Erases the character to the left of the cursor
CTRL-X	Erases the entire current line (cancels the line)
CTRL-L	Clears the screen (form-feed)
RETURN or CTRL-M	Ends the line and executes the command
CTRL-J	Moves the cursor to the next line, but doesn't execute the command
;	Marks the start of a comment
CTRL-\	End-of-file indicator

As you can see, on pre-1.3 systems, the only way to correct your typing mistakes is to delete them with the BACKSPACE key (or hold the CTRL key and press X if you want to erase the whole line) and retype. If you press the CAPS LOCK key, the red light on the key appears, and all alphabetic keys will be capitalized. This is of little practical significance since the CLI does not discriminate between lowercase and uppercase, or even mixed case.

The RETURN key is the CLI's signal to process your command line. The linefeed character (CTRL-J) moves the cursor to the beginning of the next line, just like RETURN, but it doesn't cause the CLI to process the line until RETURN is pressed. This means that you can type a list of commands separated by CTRL-J and have the CLI perform them one by one. For example, if you type

```
DELETE old file <CTRL-J> DIR
```

the CLI first deletes the file named in *old file*, then feeds the next instruction to the following CLI prompt, which displays the new directory listing.

Though not really an editing character, the semicolon (;) is significant to the CLI. The CLI interprets anything following a semicolon as a *comment* and ignores the entire rest of the line. Comments may not be too useful for immediate mode commands which you enter at the keyboard, but they can be extremely helpful in documenting command sequence files (see Chapter 5).

The last character in the summary table of useful commands, CTRL-\, will probably make more sense after you've read Chapter 4, which covers devices. Briefly, it sends an end-of-file character to the console device. This is helpful

Text Output Features

Key(s)	Function
TAB or CTRL-I	Moves the cursor one space to the right (inserts a tab character)
CTRL-K	Moves the cursor up one line (vertical tab)
CTRL-O	Switches to the ALTernate character set (shifts out)
CTRL-N	Switches back to the normal character set (shifts in)
ESC-[1m	Switches to bold characters
ESC-[2m	Switches character color (to color 3)
ESC-[3m	Italics on
ESC-[4m	Underline on
ESC-[7m	Reverse video on
ESC-[8m	Switches character color (to background color—invisible)
ESC-[0m	Switches to normal characters
ESC-c	Clears the screen and switches to normal characters

Note: When using the ESC key combinations, just press the ESC key and then enter the one to three additional characters.

because the Amiga is flexible about letting you use one device in place of another. For instance, you can use the COPY command (program) not only to copy one file to another, but also from one file to another device, such as the printer. Likewise, you can COPY from the keyboard component of the console device to a disk file. Unlike a disk file, the console device does not have a natural limit to its input—you can keep typing and typing until you're too tired to type. The CTRL-\ character, therefore, lets the console device know when you've come to the end of the "file" so that you can stop using the console as an output device and start using it for your CLI input again. PC/MS-DOS users will recognize that this is the equivalent of the CTRL-Z (or F6) character used by that operating system. Under AmigaDOS Release 2, you can also close the current CLI window by sending it the CTRL-\ key combination.

Most of the other special command key combinations represent output formatting commands that you may find amusing or learn to avoid. Their functions are really a by-product of the fact that the console device supports certain standard codes which are usually applied to printer devices. The TAB key, for

example, moves the cursor over one space as the space bar does. But it leaves a tab character in its wake, which early versions of the command line interpreter didn't like at all. If you use a tab instead of a space, you may receive an error message.

CTRL-O acts like an ALT-lock which permanently switches you to the ALTernate characters (you can think of these as the *Other,* or *Oddball,* characters to remember the CTRL key combination). The alternate characters normally appear only when you hold the ALT key down as you type. These characters, which include accented vowels and other international symbols, are interesting to look at if you want to see what characters the standard Amiga set contains, but they're of little practical use here since the CLI doesn't recognize them. If you get into this mode by mistake, type CTRL-N (for *Normal* characters) to get out of it. You can also return to the normal character set by pressing ESC and the C key, which both clears the screen and changes the character set. In pre-2.0 versions of AmigaDOS, when the screen clears, you don't get your prompt back automatically—you must hit RETURN to get a new command line. If you just want to clear the screen, CTRL-L (Linefeed) does the job.

The console device also recognizes a series of ESCape codes which change the typeface of the font printed on the screen. For example, if you press the ESC key, then the [key, 1 key, and m key, the screen text changes to boldface. Likewise, the ESC-[2m combination changes the color of the printing, ESC-[3m turns on italics, and ESC-[4m turns on underlining. These special features are cumulative. In other words, if you change to bold, then turn italics on, the result is text in bold italics. To disable all these special features and return to normal text, use the ESC-[0m combination. Pressing ESC-C clears the screen and also resets the text to normal characters. Note that although these features affect the display, CLI pays no attention to special typefaces. This sampling of escape codes was listed primarily to acquaint you with the fact that the console device responds in many ways like a standard ANSI terminal. The codes are by no means the only ones to which the console device responds. For instance, it also accepts a wide range of cursor positioning commands. These commands, however, are of little use to the average CLI user and are of greater interest to programmers who wish to use the console device in their programs.

1.3 and Release 2 Console Enhancements

The original console device used by the CLI window was fairly primitive—it couldn't even match the line-editing capabilities of the Commodore 64! This

situation was remedied by a new console handler that was introduced with Workbench 1.3. This new console did not become a built-in part of AmigaDOS until Release 2, however. In version 1.3, this device (called NEWCON:) resides in a file in the L: directory called Newcon-Handler, and must be added to the system with the MOUNT command (the default startup-sequence file on Workbench 1.3 mounts this console handler for you automatically). To start a CLI window with the NEWCON: handler under 1.3, you must specify it as your console in the NEWCLI command:

```
NEWCLI NEWCON:0/0/600/100/
```

Note, however, that if you use the NEWSHELL command (discussed below) under 1.3, NEWCON: is used as the default console handler.

The enhanced console device adds two important enhancements to the standard console handler. First, it adds line editing features. With the old console, the only way to move back through a command line is with the Backspace key, which erases everything as you go. The new one lets you use the left and right cursor arrow keys to move left and right through the line, without erasing any text. When your cursor is in the middle of a line, any additional characters that you type will push ahead the existing characters to the right, without erasing anything. The Backspace key can be used to erase the character to the left of the cursor, while the Delete key erases the character under the cursor. The extended editing features of the NEWCON: handler (and the Release 2 CON: device) are summarized below:

The new console handler introduces one more subtle editing change. Control characters are now printed in reverse video, and are not acted on imme-

Key	Function
Del	Erases the character highlighted by the cursor
CTRL-A	Moves the cursor to the beginning of the line (or Shift-LeftArrow)
CTRL-K	Erases everything from the cursor forward to the end of the line
CTRL-U	Erases everything from the cursor backward to the start of the line
CTRL-W	Deletes the word to the left of the cursor
CTRL-Y	Replaces the characters deleted with CTRL-K
CTRL-Z	Moves the cursor to the end of the line (or Shift-RightArrow)

diately (though they will be passed through if the screen output is directed to a file or device). This means that typing the sequence `ESC-c` will no longer immediately clear the screen, although if you press RETURN after typing it, the screen will clear when the 1.3 CLI prints the error message Can't open <ESC-c>. Under Release 2, this error message changes to :Unknown command, and the message stays on the screen after it is cleared.

In addition to enhanced editing functions, the new handler provides a command history buffer. This buffer saves each command line as you enter it, up to a maximum of 2,048 characters (when the buffer fills, the oldest command line is deleted each time a new one is entered). To retrieve a previous command line, you merely press the UpArrow key. Each time you press this key, the next oldest command line appears. If you go past a command line, you can move forward through the buffer with the DownArrow key. To move all the way forward to the most recent (bottom) entry in the buffer, type CTRL-B (or Shift-DownArrow). This leaves you on a blank line. The command history also has a search feature which allows you to find a past command by typing part of it and then pressing Shift-UpArrow. For example, if you wanted to repeat the command:

```
Copy Work:Wordprocessing/Documents/MyDoc to df1:Backup
```

and you know you'd typed the command recently, you could just type:

```
Co <Shift-UpArrow>
```

and NEWCON: would display the last line that you typed that started with those two letters. If the command line you'd like to repeat showed up, you could just hit RETURN to re-issue the command. Note, however, that in 1.3, the search function is case sensitive; if you type `Co<Shift-UpArrow>`, NEWCON: won't find a command line that starts with copy. The search function of the Release 2 console is not case sensitive, and will find *copy* when you type in `Co`.

The command history of NEWCON: can come in very handy when you're performing repetitive tasks. For example, if you wanted to format a series of disks, you would only have to type the Format command once. For each succeeding disk, you would only have to press the UpArrow and Return keys.

A new feature that is unique to the Release 2 console device is the ability to copy text from a console window and paste that text elsewhere in the same console window, or even in a console window that belongs to an entirely different program such as the Ed text editor. To copy text, you move the mouse pointer to the beginning of the text, hold down the left mouse button, and drag

the mouse over the text until the desired portion is highlighted. Then, hold down the right Amiga key and press C to copy the text to the buffer. When you wish to paste the text, move the cursor the appropriate spot and press the right-Amiga-V combination. The same clip of text can be repeatedly pasted.

By default, the console device uses its own copy-and-paste buffer. When you run the CLI program called CONCLIP, however, it uses the clipboard device for these copy operations. This means that any program can participate in these transfers, even if it doesn't use a console window (as most programs do not), just by supporting the clipboard device. Since the default startup-sequence script automatically runs CONCLIP when you turn the computer on, the console uses the clipboard unless you specifically prevent it from doing so.

Pausing and Restarting

Another aspect of the console device that you should be familiar with is pausing and restarting screen output. The CLI (and the command programs that use its console device) constantly watches the console for input from the keyboard. If you type a character while one of the command programs is running, the program will stop its own output to the screen so as not to mix it with your input. Even if the command program prints no messages of its own, you'll not get the CLI prompt (1>) back until you restart output. The way to do that is either to erase the line that you're typing (by using the BACKSPACE or CTRL-X keys) or finish the line by entering a RETURN.

The pause is really a function of the CLI's type-ahead feature. The CLI can keep track of up to 255 characters of command instructions while it's busy running a command program and will execute these instructions after it's finished. In practical terms, however, it means that you can pause a display of, say, a directory listing, by pressing the space bar and restart it later by pressing the BACKSPACE key. This roughly corresponds to the function performed by the CTRL-S, CTRL-Q combination on MS-DOS machines. Under AmigaDOS Release 2, however, you can use CTRL-S to suspend output from the CLI, and CTRL-Q to resume, just as with MS-DOS.

If you use the RETURN key to complete the line rather than erasing it, you should be aware that the command line that you've just entered will be saved by the CLI and will be executed after it finishes with the current command.

If you prefer to terminate output entirely instead of just pausing it, you can use the BREAK function. Hold down the CTRL key and press the C key, and you'll see the message ***BREAK* as the CLI prompt appears once again. You

may also interrupt an EXECUTE command sequence with the CTRL-D combination (see Chapter 5 for details on command sequence files). AmigaDOS reserves the CTRL-C, CTRL-D, CTRL-E, and CTRL-F combinations for interrupt functions, but the CLI uses only the first two. Other programs may use the latter two as they see fit.

As you'll see later on, it's possible to have more than one CLI window open at a time. Using the CTRL-C or other break key combinations only work for the CLI window that's *currently* active. To interrupt others, you must either make them the active CLI and use the break keys or use the BREAK command. This command interrupts the other process just as if you'd made it active and then used the break keys.

The Shell

Another CLI enhancement first offered in Workbench 1.3 is a slightly "smarter" command shell. Where the earlier CLI would only execute the command whose name appears as the first word of the command line, the Shell knows a few additional tricks. Since the Shell did not become the default command handler until Release 2, however, 1.3 users must add it to the system by loading the *L:Shell-Seg* file on which it resides. To do this, use the command:

```
RESIDENT CLI L:Shell-Seg SYSTEM pure
```

Executing this command, however, will not convert the CLI from which the command was issued into a Shell CLI. Only CLI windows that are opened after the command is given will have the properties of a Shell. The default Startup-sequence script file on Workbench 1.3 executes the necessary RESIDENT command, so that all CLI windows that you open from the Workbench have the Shell characteristics. The command NEWSHELL not only opens a new Shell process (if the Shell-Seg has been loaded), but opens it with a NEWCON: window as well. Remember, these distinctions apply to 1.3 only. As of Release 2, all CLI windows are Shell windows with the new console features.

The first enhancement offered by the Shell is recognition of a new PROMPT argument. When you use the characters *%S* in a prompt string, they will be replaced with the name of the current directory, whatever that happens to be at the time. For example, the command:

```
Prompt "%N.%S> "
```

might lead to a prompt string that looks like this:

```
1.Workbench 1.3>
```

Whenever you change your current directory (more about that next chapter), the prompt will change to reflect the new location. For more details on the PROMPT command, see the reference section.

The second new feature of the Shell is much more substantial. This is the power to create *aliases* for commonly used AmigaDOS command. An alias is an "assumed name" for an existing command, usually shorter than the actual filename of the command. For example, you could give the command `Makedir` the alias `md`, and then, whenever you wanted to create a new directory, you'd only have to type `md`, instead of `Makedir`. To create an alias, you use the Alias command, which is recognized only by the Shell (it can be thought of as an internal command, one the Shell knows about without reading in a command file). The format for this command is:

```
ALIAS name command
```

where *name* is the new name by which you wish the command to be known, and *command* is the command that you want executed when that name is given. In the example above, you'd give the `Makedir` command its alias by saying:

```
ALIAS md Makedir
```

Alias is good for more than just changing a command's name, however. It can be used as a handy shortcut for an entire command line. For example, you could use it to condense the command Format drive df0: name Empty noicons down to the command fmt with the Alias command:

```
ALIAS fmt Format drive df0: name Empty noicons
```

Not only can you enter command parameters into an alias, but you can even enter in a partial list of parameters, and specify a place where substitutions will be made with a pair of square brackets []. For example, if you wanted the above alias to be usable with any floppy drive, not just the internal one, you could give the command

```
ALIAS fmt Format drive df[]: name Empty noicons
```

When you give the `fmt` command, the square brackets are replaced by whatever number you type as a command parameter. For example, to format drive 1, you would type

```
FMT 1
```

There are a few more facts that you should know about the ALIAS command. The command ALIAS entered by itself will list the current aliases. To remove an ALIAS, type `ALIAS name` with no command, where *name* is the

alias you wish to remove (under Release 2, use the UNALIAS command). Finally, each set of aliases is known only to its own particular shell process. If you start a new Shell, it won't know about the old Shell's aliases. The same is true for the enhanced prompt. If the old Shell had a prompt that contained the %S character, the new Shell will inherit a prompt that reflects the directory of the old Shell at the time the new one was created. This prompt will not automatically change to reflect the new Shell's path. Since it would be rather inconvenient to enter a new prompt string and a new set of aliases for each new Shell that you create, AmigaDOS lets you create a script file that automatically does it for you. If you start a Shell with the NEWSHELL command, it will automatically execute all of the commands in the file called *S:Shell-Startup* at the time of its creation.

The last two features of the 1.3 Shell will be covered in greater depth later on, since they concern subjects which we haven't covered yet. We'll mention them here, just for the sake of completeness. The first has to do with sequence file execution. Normally, you use the EXECUTE command to execute a command sequence file. With the Shell, however, its possible to execute sequence files just by typing their name, if the file's S protection bit has been set. The second feature is a new kind of output redirection. Normally, when you redirect output to a file, it creates a new file that overwrites the old file of the same name. With the Shell redirection operator >>, new information is appended on to the end of the old file, if any.

In addition to the benefits introduced in the 1.3 Shell, the Release 2 Shell adds some new features of its own. One important new feature is the ability to include the text output of one command as input for another command. Let's say that you wanted to use the ECHO command to print the total amount of free memory in a sentence such as *The total amount of memory available is xxxxxx bytes*, where the x's represent the number of bytes. This number may be obtained as the output of the AVAIL command. By enclosing it with the special backwards apostrophe character (also known as a back tick, it's the key right above the TAB key), you can insert the AVAIL command right into the ECHO command as follows:

```
ECHO "The total amount of available memory is `AVAIL
    TOTAL` bytes"
```

When this command is executed, the AVAIL TOTAL command is performed first, and the results are printed as part of the output of the ECHO command:

```
The total amount of available memory is 965,432 bytes
```

When using a back tick command in an alias, you should remember that embedded commands are executed at the time the alias is formed, not at the time the alias is executed. If you wished to turn the above example into an alias, you might try the command:

```
ALIAS free ECHO "The total amount of available memory
    is 'AVAIL TOTAL' bytes"
```

Unfortunately, the command `AVAIL TOTAL` will only be evaluated at the time when the ALIAS command is issued. If there were 965,432 bytes available at that time, that is the number you will see whenever you issue your new FREE command, regardless of how much memory is actually free at that time. In order to have the embedded command executed when the aliased command is executed, you must use the asterisk escape character in front of each back tick character. By changing the ALIAS command to:

```
ALIAS free ECHO "The total amount of available memory
    is *'AVAIL TOTAL*' bytes"
```

you'll get an accurate tally of free memory whenever you use the FREE alias.

The Release 2 Shell is also sensitive to environment variables. These are stored text strings whose contents are retrieved each time the Shell executes a command that refers to them. The two types of environment variables, local and global, differ only in their scope. Local variables are created using the SET command, and are significant only to the Shell in which they were created. Global variables are created with the SETENV command, are stored in the ENV: directory, and can be accessed by any Shell, or by any program. If you create the environment variable Editor with the command:

```
SETENV editor Memacs
```

then any program will be able to determine that your preferred editing program is Memacs. Moreover, any Shell command that uses the expression $Editor (the dollar sign is the character that identifies an environment variable) will have that expression replaced by the string "Memacs". Therefore, if a Shell command or script uses the line:

```
RUN $editor textfile
```

the Memacs program will be run, and it will load textfile. If you decided at some other time that you liked a different text editor better, you could use SETENV to

change the Editor variable to the name of another program, and the above command would run that program instead.

Under Release 2, certain environment variables are maintained by AmigaDOS. These are KICKSTART (contains Kickstart version), WORKBENCH (contains Workbench version), ECHO (if set to ON, commands are echoed to the screen before execution), PROCESS (contains the process number), RC (contains the return code of the last command), and RESULT2 (contains the error number for last command).

Running a Program from a CLI

Once the CLI has evaluated the text that you've typed in, the next phase of its operation is running a program. Since the CLI interprets the first word you type as the name of a program file, running a program from a CLI is simple—all you do is type the name of the program at the prompt, followed by pressing the RETURN key. If the program needs further input to run, you type that input on the same line as the filename. For example, to create a duplicate of one file under another name (on the same disk and in the same directory as the original), type:

```
COPY old file TO new file
```

In this command line, the word COPY is the name of the copy program file, and the rest of the line tells that program what to do.

The Complete Location

In actual practice, running a program is not quite as simple as typing its name. That works only if the program is located in the current directory of the current disk, or if it's located in the current command directory or command path. These concepts will be discussed in detail in Chapter 3, which deals with the directory structure, and Chapter 4, which explains the use of virtual devices. Generally speaking, however, when you start up the system, the current directory is the root (topmost) directory of the Workbench disk that's in the internal disk drive (DF0:) or the boot partition of your hard drive, and the command directory is the *c* subdirectory found on that disk. If your program is anywhere else, you have to specify its complete location by typing in the name of the disk and/or the subdirectory on that disk. For instance, to run a program called *WordWizard* located in the *Wordprocessing* subdirectory of the external floppy drive (DF1:), you would enter the command:

```
DF1:Wordprocessing/Wordwizard
```

There are other reasons why a program may not run when you type its name. There may be typing errors in either the program name or the instruction line that follows it. The file may not be in the executable load format that AmigaDOS requires, or the disk itself may be damaged or write-protected. In most cases, AmigaDOS gives you complete error messages and may even give you a chance to remedy the error without having to redo the command. In some cases, however, these messages may not be satisfactory. You can get more information about a failure by using the WHY command—just type `WHY` after receiving an error message. Only rarely will you receive a more cryptic message, such as *Error code 218.* To find this error code's meaning, use the FAULT command. Typing FAULT *n*, where *n* is the error code number, will usually yield a clearer explanation. If all this fails, or if you're simply curious, consult Appendix B, which explains the various errors you might receive. Release 2 users will be happy to know that starting with that version, error messages are much less cryptic.

When a program ends, it usually exits with a return code. This code may be zero if the program exits normally, or higher if it exits because of an error or problem. This feature is sometimes used to form a conditional block of statements in a command sequence file, as we will see in Chapter 5. Under AmigaDOS Release 2, the system maintains the RC and RESULTS2 environment variables to help you keep track of the last return code and error number.

Room to Work

One rare problem which you might encounter concerns the environment that the CLI provides to the program that it's running. As mentioned earlier, after the CLI successfully loads the program, it prepares a stack area for the program to use as working storage. The initial allocation for this stack area is 4000 bytes. Usually, this will be enough, but in some cases there won't be enough stack space for the program to run. If you try to run the ABasiC program that was supplied with the first Amigas from a CLI, for example, you'll receive a message that there's not sufficient stack space. If you first increase the stack space to 8000 bytes, however, with the STACK command (STACK 8000), the program runs. Other programs, like early versions of the SORT command—which needs a lot of working space if it's to sort a large file—may run out of stack space and cause the computer to hang up. If you are using an early version of AmigaDOS, increase the stack space before sorting a large file.

Though most programs written for the CLI will not need their stack increased, some programs that were written for the Workbench environment may need a stack increase when run from the CLI. To see the stack size that's required, click once on the program's icon, then select the Info item from the Workbench menu (Information is in the Tools menu in Release 2). There will be a number in a box marked "Stack" which will tell you the required stack size.

A final note about the simple nature of the CLI process. Some disk operating systems (like MS/PC-DOS) have a set of intrinsic commands, which the DOS recognizes and executes as soon as the user types them in on the command line. In Workbench versions prior to Release 2, CLI commands are all disk-based programs, so you must have the CLI disk in the drive before you can use any of these commands. This is not as inconvenient as you might expect. For one thing, you can transfer the commands that you use most often to the RAM device and add that directory to the current search path. Starting with Workbench 1.3, you can also make commands resident by using the Shell and the RESIDENT command (more on these later). This gives you the equivalent of a custom-tailored set of intrinsic commands which occupy no more user RAM than is really necessary. Another nice thing about having all the commands as program files is that you can rename any command to suit your preference (though for the sake of compatibility, you'll probably want to make a copy of the program with a new name, while retaining the file with the old name as well). For example, if you're used to MS/PC-DOS, you might want to use the word *ERASE* instead of *DELETE*. If you type `COPY c/delete TO c/erase`, you'll be able to use either form of the command. The *alias* feature of the Shell (discussed above) provides similar capabilities, without requiring multiple copies of the file.

Starting with Workbench 2.0, over 30 CLI commands have been removed from the C: directory and have become internal commands that are always available. You can get a list of names by issuing the RESIDENT command. Although these internal commands are part of the CLI, they can still be replaced using the REPLACE option of the RESIDENT command.

Starting Additional CLI Processes

Though AmigaDOS is a multitasking operating system, each CLI can run only one program at a time. To run several programs simultaneously, you must create additional CLI processes. The command program NEWCLI takes care of this nicely. When you type NEWCLI, a new interactive CLI window opens up in front of the current window. This window displays the message *New CLI task 2* (or *New Shell Process 2*), followed by its prompt, 2>.

This should solve the mystery of why the prompt in the first CLI window is 1>. The number in the prompt is the task number of the CLI. By the way, you're free to change the prompt to anything you want, in any CLI window, by using the PROMPT command. For instance,

```
PROMPT "What is your wish, O Master?"
```

will change the prompt to this verbose phrase. Even ALT characters, such as foreign language accented characters, can be used in a prompt string. In fact, you can use the text output commands (within quote marks) to change the color of your prompt, or to make it appear in reverse video, as with the command shown below.

```
PROMPT "<ESC>[7mReverse prompt>> <ESC>[0m "
```

When you create a new CLI window, it becomes the active window. You can tell which window is active by looking at the title bars. The title bar of the window that's currently active is a solid color, while the title bars of the other windows are dotty (or *ghosted*, as it's called). To change a window from inactive to active, just move the mouse pointer inside the window and click the mouse button. Whenever you type anything at the keyboard, the printing always appears in the active window. The other rules for system windows apply to CLI windows as well. You can use the normal system gadgets to change the size of the new CLI window, drag it around the screen, and move it in front of or behind other windows. You can keep opening as many as 20 CLI windows, provided that there's enough available memory (this 20-window limit was removed in Release 2).

Your Own Windows

If you just type `NEWCLI`, the operating system decides at what position on the screen to create the window and how large the window will be. These sizes are measured in *pixels* (picture elements), which are the individual dots used to create the display. The standard DOS screen is 640 pixels across and 200 pixels high. Versions 1.3 of AmigaDOS creates new CLI windows that start at the top left corner of the screen, and are 640 pixels wide by 100 pixels high. All new CLI windows are created in the same place, in the same size, unless you specify otherwise. This means that the third CLI window appears atop the second, and you'll have to drag one of them out of the way to use both.

You can create a new CLI window in a particular location and size by describing the console device output window. The description for this device follows the format

```
CON: hpos/vpos/width/height/windowtitle
```

where *hpos* is the horizontal position of the top left corner of the window (expressed as the number of pixels in from the left edge of the screen), *vpos* is the vertical position of the top left corner of the window (expressed as the number of pixels down from the top edge of the screen), and *width* and *height* give the size of the window in pixels. The maximum size for a CLI window is the screen size, which for the default Workbench screen is 640 × 200 pixels. The minimum is 90 × 25 pixels. The last entry, *windowtitle*, is optional. It allows you to enter the text of a title to appear in the title bar. If you don't enter any text, the title is left blank. To create a new CLI window that occupies the full screen, you would type

```
NEWCLI CON:0/0/640/200/
```

Note that the last slash mark is *required*, even though you didn't specify the title.

A title can contain special characters, such as the space character (which AmigaDOS usually interprets as separating one command word from another), but if you use them, you must put the *entire device name* in quotation marks:

```
NEWCLI "CON:40/40/200/100/A Standard Window"
```

Starting with AmigaDOS Release 2, the CON: window description can have one more item added, an *options* entry that appears after the title, and is separated from it by a slash. Actually, one or more of these options can be used, each one separated from the previous one with a slash. The relevant options available for a Shell window are:

CLOSE	A close gadget is included in the window border. This is the default case if no options are selected.
BACKDROP	The window type is changed to backdrop, which means that it appears behind all other windows on the Workbench screen. You cannot depth-arrange, move, or resize it (except by using the zoom gadget).
NOBORDER	No visible line is drawn around the window, although the zoom and close gadgets will still appear above it. If you zoom this window to full size, those gadgets will disappear, and you will have a full-screen window that can't be sized or moved.
NODRAG	The window can't be dragged. It will have a zoom, depth, and size gadget, but no close gadget.
NOSIZE	The window will not have zoom, size, or close gadgets. Only a depth gadget will appear.

SCREENname	The window will open on the public screen whose name is *name*. For example, to open the CLI window on the public screen named Fred, you would add /SCREENFred to the end of the window description. This option only works if a public screen of the specified name is already open.
SIMPLE	This option chooses the simple window refresh scheme. If you enlarge such a window, the text expands to fill the available space, allowing you to see more information, including information that had scrolled off the screen. This is the default refresh type for Release 2 Shell windows, if no option is specified.
SMART	This option chooses the smart window refresh scheme. If you enlarge this kind of window, existing text is not redrawn. This is the kind of CLI window used by AmigaDOS 1.3 and earlier.

To open a new Shell as a borderless backdrop window on a public screen named PubScreen, you would use the command

```
NEWCLI "CON:40/40/200/100/Shell Window/BACKDROP/
    NOBORDER/SCREENPubScreen"
```

Another difference between the Release 2 Shell and previous models is that the window uses whatever font is chosen for System Default Text using the Font preferences editor. This must be a fixed width font such as Courier. You may also change the font in each individual Shell window by using the new SETFONT command.

There's one more feature of NEWCLI that you should know about. When the first CLI window opens, it automatically executes a command sequence file called s:startup-sequence. We'll talk all about command sequence files in Chapter 5, but for now, let's just say that it's a file that executes a series of CLI commands. If you want your new CLI window to automatically execute a series of commands when it starts up, you can specify a command sequence file in the NEWCLI command:

```
NEWCLI FROM StartupFile
```

where *StartupFile* is the name of the command sequence file that you want executed. Under Workbench 1.3 and higher, if no *FROM* file is specified, the sequence file *S:CLI-Startup* is automatically executed when a NEWCLI com-

mand is issued. Likewise, if no *FROM* file is specified when a new Shell window is opened, the file *S:Shell-Startup* will be executed instead.

Going Away

Any time you want to eliminate one of your CLI windows, make that window active by clicking the mouse button inside its borders, and type ENDCLI. The message *CLI task n ending* (where *n* is the number of the CLI task) is briefly printed, and the window closes. (In fact, the message prints so quickly that you probably won't see it.) In Release 2, there are a couple of additional ways to close a Shell window. You can use the close gadget in the upper left corner of the window, and you can also press the CTRL and backslash keys at the same time (which give the "end of file" signal to the CLI).

If the Workbench isn't open, always leave yourself at least *one* open CLI window—if you close the final window, you won't be able to issue any commands. You'll have no choice but to warm start the computer by pressing CTRL and both Amiga keys at the same time. In fact, it's not a bad idea to keep an extra CLI around, just in case. There are programs, like the public domain "PopCLI," which allow you to open a new CLI window just by pressing a hot key combination.

If you're using one program and want to start another, you can switch back to the Workbench screen (the one on which the CLIs reside), either by using the depth-arrangement gadgets at the top right of the screen, or by using the Amiga-N key combination to bring the Workbench screen forward and Amiga-M to send it back. (The Amiga key combinations move entire screens, not just individual windows.) This gives you access to your open CLI so that you can run another program or use one of the DOS command programs.

If you have a number of CLI tasks running at the same time, some whose windows do not appear on the Workbench screen, you may lose track of them all. The STATUS command prints a list of all of the current CLI tasks and the command programs that they're running.

Running Programs In a Noninteractive Process

When you want to run a program as a separate task, but don't need the interactive features (and the memory overhead) of another CLI window, you can use the RUN command program. When you type RUN followed by a command you would normally type in a CLI window, a new CLI process is created which executes the command, but it doesn't open a CLI window. Let's say that you

want to run a word processor program without losing your current CLI window. If you normally would type `wordprocessor` to start the program, type `RUN wordprocessor` instead. The RUN command prints a message like *[CLI n]* (where *n* is the next unused CLI number) and then runs the word processor. This saves you the trouble of typing NEWCLI before entering the command and of getting rid of the CLI with ENDCLI after you're finished. It also saves you the memory that would ordinarily be taken up by the CLI window. When you finish with the word processor and exit the program, it leaves nothing behind.

Even though RUN does not provide you with a command window, it does offer a way to send additional commands to the process. At the end of the first command, type a plus sign (+) and press RETURN. You may then enter a second command on the next line. If you want to add a third, type the plus sign and RETURN at the end of the second line and add the new command on the third line. At the end of your last command line, just type RETURN. The RUN command executes each of the command lines in sequence, just as if you had typed them in a CLI window, one after the other. For example, if you want to send a sorted list of BASIC program files to the printer, enter

```
RUN LIST #?.bas TO ram:temp+
SORT ram:temp TO prt:
```

This runs the LIST program, which sends a list of all files with the characters *.bas* in their filenames to a file on the RAM disk. After LIST has finished, the CLI runs the SORT program, which sorts the lines and sends them to the printer. This CLI process doesn't disappear until the last task is finished.

> Throughout the rest of the book, you'll occasionally come across lines to be entered on the Amiga which, because of the book's formatting, are split on the page. A continued line is indented—do not press RETURN at the end of the first physical line, but simply continue typing with the indented characters.

Chapter 3

The Filing System

Each Amiga personal computer comes with an internal double-sided, double-density (or high density) 3½- inch disk drive. Each double density 3½- inch disk can hold a total of 880K bytes or 901,120 characters of information. Of this total, about 837K bytes is available for user storage (879K if the disk is formatted using the Fast File System under AmigaDOS Release 2). Before a disk can be used to store information, AmigaDOS must first write certain information to it in order to prepare it for use with the Amiga filing system. This is called *formatting* the disk and is performed by the FORMAT command. Its syntax is

```
FORMAT DRIVE df0: NAME Volume name
```

Volume Names

When you format a disk, the program notifies you as each of the 80 cylinders (tracks) on the disk is formatted (written), then verified (read) to make sure that the formatting information is correct. If you want to format the disk on a drive other than the internal drive, just substitute the device name of that drive (for example, df1: refers to the first external 3½- -inch drive on an Amiga 500 and 600, or the second internal drive on a 2000 or 3000). Notice that after the name of the drive, the command specifies NAME *Volume name*. AmigaDOS requires you to give each disk a name, known as the *volume name*. You must use the keyword NAME before entering the name. To name a disk as *Wordprocessing*, you'd use *NAME Wordprocessing*. It's a good idea to use a name which identifies the disk as precisely as possible. AmigaDOS is able to identify a disk by its volume name as well as the device name of the drive in which it resides. Therefore, if you remove the *Wordprocessing* disk from the drive and DOS wants to access something on that disk, it will prompt you to *Please insert volume Wordprocessing in any drive.* (This message is not always accurate. Sometimes the disk *must* be placed in a specific drive, normally the one which it was in earlier. If you put the disk in the wrong drive, the message will reappear.) You

can change the volume label of a disk at any time with the RELABEL command program. To change the name of the disk in the above example to *Spreadsheet,* for instance, you'd type

```
RELABEL df0: Spreadsheet
```

Identification

Besides the volume name, AmigaDOS also writes an identification number on each disk. It tries to make each of these ID numbers unique, so even if two disks both have the same volume name, the disk operating system can tell them apart. The disk-duplication programs provided on your Workbench disk do not reproduce the old ID number on the new disk, so even exact copies can be distinguished from the original. Only disks that are duplicated by a commercial mass-duplicating machine, or by a software program designed to exactly duplicate copy-protected software, will have IDs that match the original.

Info

After a disk is formatted, the INFO command shows that it contains 1758 blocks of usable storage space, each containing 512 bytes. Note that this is two blocks short of 880K—the disk operating system reserves these for its own purposes. In addition, DOS uses two of these 1758 blocks, leaving you with 1756 free blocks (878K) on a newly formatted disk. If you want to verify this, you can use the command program INFO to display the amount of storage used on the disk and the amount of remaining free space. Type *INFO* using version 1.3, and you'll see a display which looks like this:

```
Mounted disks:
Unit    Size    Used    Free    Full    Errs    Status      Name
DF1:    880K    1317    441     74%     0       Read/Write  Workbench
DF0:    880K    2       1756    0%      0       Read/Write  Empty
Volumes available:
Empty [Mounted]
Workbench [Mounted]
```

This display tells you the size of the total storage space on each disk currently in each drive (mounted), how many blocks have been used, how many

are free, the percentage of disk space that's used up, how many errors were encountered in reading from the disk, whether or not the disk is write-protected, and the volume name of each disk.

The 1.3 INFO results are somewhat misleading, because they fail to indicate that under the original Amiga filing system, only 488 bytes out of each 512-byte block are actually used for storage. The other 24 bytes are used to hold duplicate information about the structure of the files themselves, making it somewhat easier to recover files that have been corrupted by a bad write operation, or by problems with the disk medium itself. Under AmigaDOS Release 2, the INFO command shows 837K under the ***Size*** column for a normal Amiga floppy.

AmigaDOS Release 2 makes it possible to format a floppy disk using the Fast File System, a new filing system that is most commonly used for hard drives. To format a Release 2 disk with this new organization, use the FFS switch in the format command:

```
FORMAT DRIVE df0: NAME FastDisk FFS
```

A disk formatted using this file system will be read a little quicker, and will hold a little more information than a standard Amiga floppy. These advantages are outweighed, however, by the fact that an FFS disk can only be read on a system that has AmigaDOS Release 2 installed. For this reason, few users bother to format floppy disks using the Fast File System option.

Installing

There are a couple of other things that you should know about formatting a disk. First, it's not necessary to format a disk before you perform a DISKCOPY to it. The DISKCOPY program both formats the new disk and copies all of the information from the source disk to this disk. Second, the system will not accept a newly formatted disk if it's inserted at the prompt which tells you to put in the Workbench disk (it just keeps asking for a Workbench disk). In order to make a newly formatted disk bootable, you must use the INSTALL program. To install the boot information on drive df0:, for example, enter

```
INSTALL df0:
```

The INSTALL program doesn't prompt you to put the disk into the drive—it does the installation immediately. This makes it difficult to use INSTALL on a single-drive system because without an opportunity to swap disks, you must have the INSTALL program on the disk that you want to install. If you don't want to

copy that program to the disk, you can make use of the command template feature to pause the command after it is read off of the disk, allowing you to swap disks at that point. This feature is invoked by typing the command, followed by a question mark. In this case, with the Workbench disk in the drive we type:

```
INSTALL ?
```

When the INSTALL program is loaded up, it responds with the command template that shows the command line requirements and options. In the case of the Release 2 INSTALL command, it responds:

```
DRIVE/A,NOBOOT/S,CHECK/S,FFS/S:
```

which tells us that a drive name is required, and the options include NOBOOT (make it non-bootable), CHECK (check for boot block viruses), and FFS (write a Fast File System bootblock). After the command prints its template, it waits for you to enter the rest of the command information. At this point, you can pop out the Workbench disk, insert the disk you want to install, and enter "DF0:" at the prompt. The newly formatted disk you have inserted into the internal drive will be the one that is INSTALLed.

Once the INSTALL process is completed, you may put that disk into the internal drive when the system prompt for the Workbench appears on the screen, and the disk will boot and show the CLI 1> prompt. Unless you put the DOS command files on that disk, of course, you cannot use the commands just by typing their names.

The CHECK option that was added as of Workbench 1.3 allows the INSTALL program to see if the information on the boot block of the disk is the standard Commodore-Amiga code, or some non-standard code which may indicate that the disk contains a virus (a hidden program which may adversely affect your computer's performance). Using the INSTALL program to re-write the boot block to the disk will destroy most virus programs, but it can also cause some copy-protected programs to cease functioning.

Files and Their Characteristics

The basic unit of information stored on a disk is called a *file*. A file is just a group of characters of information which are stored together on the disk under a common filename. A file can represent a computer program, or a collection of data used by that program, such as the text of a document created by a word processing program. To see the contents of a file, use the TYPE command.

To print a text file called *document* on the screen, for example, enter the command `TYPE document`. You may remember from the previous chapter that you can pause output to the screen at any time by striking a key, such as the space bar, and restart output by using the BACKSPACE key to erase that keystroke. TYPE is really only helpful for seeing the contents of text files. If a file contains the numeric code for a computer program, the TYPE command will print out what seems like a jumble of nonsense characters.

Each file has a number of characteristics associated with it. These include the name of the file, the number of characters it contains, the number of disk blocks it uses, the protection level, the date and time of its creation, and comments (if any). To display a *directory* (a list of the names of files on a disk, sorted into alphabetical order), use the DIR command.

The LIST command displays a list of files and all of their characteristics. You can LIST all the files in a directory, a selected portion of the files, or even a single file. There are a number of variations on this command (see the "Command Reference" section for details). The simplest form is

```
LIST
```

which displays information about the files and directories in the current directory. As with other displays, you can pause it by pressing a key, such as the space bar, and resume it by pressing the BACKSPACE key.

In the sections below, we'll examine in detail each of the file characteristics displayed by the LIST command program.

Filenames

The most important characteristic of a file is its name, since you must know the name in order to access the information a file contains. A filename may be up to 30 characters long and may contain almost any character, with a few exceptions. A filename can't contain a slash (/) or colon (:); DOS uses these to identify the directory to which a file belongs (see the section below on directories for more information). A filename cannot use nonprinting characters (like TAB). In earlier versions of AmigaDOS they also may not contain characters from the alternate character set (which appear when you hold down the ALT key and type a character).

If you want to use the special characters that the CLI recognizes as command modifiers in a filename, you'll have to jump through some hoops. To use the space ( ), equal (=), semicolon (;) or a trailing plus (+) in a filename, you must put the whole filename in double quotation marks. For instance,

```
COPY SOB TO "Son of a Blitter Object"
```

If you include the device name and/or directory name as part of the file specification, the whole file specification must appear in quotation marks, like this:

```
"DF1:Programs/My Program"
```

Not like this:

```
DF1:Programs/"My Program"
```

By using the double quotation mark for this purpose, you've made it an exception to the naming rules. So what if you want to have a filename which includes quotation marks? You'll have to use an asterisk (*) in front of the double quotes as an escape character to tell DOS that you want the quotation mark to appear in the name and not just set off a chunk of text that contains space characters. This means that you would type the filename *"So-Called" Facts* like this:

```
"*"So-Called*" Facts"
```

Confused? It gets worse. Now you've made the asterisk an exception, too. That means in order to use the asterisk in a name, you must use another asterisk in front of it. The name **void where prohibited* must be typed as

```
"**void where prohibited"
```

To summarize:

- Filenames may be up to 30 characters long.
- They may not contain a colon (:), slash (/), or nonprinting character (nor an ALTernate character in older AmigaDOS versions).
- If you want to use characters like the space, plus (+) at the end of a name, equal (=), and semicolon (;), all of which have special significance to CLI, you must put the entire filename in double quotation marks (*"A Special File"*).
- If you want to use double quotation marks or an asterisk in a filename, you must precede them with an asterisk (*"*"Confusion**10*"* " for *"Confusion*10"*).

In the examples above, some of the filenames appear in lowercase characters, some in a combination of upper- and lowercase. Any combination can be used in naming a file. When you LIST the filenames, they'll be printed using the same combination of uppercase and lowercase used when the file was named. The CLI, however, does not distinguish between cases. You can refer to a file named CAPITAL as *capital* or *Capital* or even *CAPital,* and the CLI reads them

all identically. Since you cannot have two files with the same name in the same directory, a single directory cannot contain files named *Test* and *TEST*, because to the CLI each name looks the same.

Filenotes

Though the name of a file is your chief source of information about its contents, AmigaDOS provides another source as well. Using the command program FILENOTE, you can attach a comment of up to 80 characters to a file. This comment can be used to note what's in the file or show how this file differs from other files with similar names. When you use the LIST command to obtain information about the files on a disk, the FILENOTE comment is displayed right beneath the name of the file.

Not all files have filenotes attached. (No filenote is automatically attached to the file when it's created.) You must enter it yourself with the command FILENOTE, which uses this format:

```
FILENOTE filename COMMENT "This comment tells you
    about the file"
```

The use of the keyword COMMENT before the comment is optional. The rules for using special characters (such as spaces) within comments are the same as those for using such characters within filenames. If you use spaces within the text of the comment, the entire comment must be enclosed within quotation marks, and if you want to include quotation marks or an asterisk in the comment, you must precede them with an asterisk.

An interesting characteristic of filenotes is that they remain firmly attached to the file to which they're appended. The comment does not change or disappear when you rename the file. If you copy the contents of a file to one which has a filenote, the filenote stays attached, even though its contents have changed. If, however, you copy a file with a filenote to a new file, the filenote is not copied along with the contents (unless you use the COM or CLONE options with COPY). It sticks like glue to the original. In earlier versions of AmigaDOS, there is no way to delete a filenote alone. If you want to get rid of it, you have to change the comment to something innocuous, like a single blank space, or copy the whole file and delete the original. Under AmigaDOS Release 2, using the FILENOTE command without any comment will delete the comment.

File Size

The LIST command displays a number after the filename. This number represents the size of the file in bytes (characters). This number should not be confused with the number of disk blocks that the file uses. Even though each block can hold 512 bytes of information, every file uses a *minimum* of two disk blocks. This means that a file only one character long uses up 1024 characters of disk space.

To test this, type `INFO` to see the number of free blocks on your disk. Now type

```
COPY * TO test
```

Press the Return key, and then the CTRL key and the back slash key (\) at the same time.

This copies from the keyboard of your console device (represented by the asterisk) to a disk file named *test.* The CTRL-\ key combination is the end-of-file character, which signals the end of output from the console device and stops the copying process. What you end up with is a file that contains only one character.

If you enter `LIST test`, you'll see that the file length is really one character. But if you type `INFO` again, the number of free blocks has decreased by two. Keep this in mind—numerous small disk files may take up more space than if the same information was stored as one long file. Even an empty file uses up one block of storage.

Protection Level

On the display provided by LIST, there's space for eight characters next to the size of the file. These characters—hsparwed—represent the eight protection status flags associated with each file. The letter codes for these status flags are derived from the first letter of the words that describe the status flag functions: Hidden, Script, Pure, Archive, Read, Write, Execute, and Delete.

Some of these flags actually control the functions that AmigaDOS will perform on the files, while others are merely advisory, and are enforced only by programs that recognize them. AmigaDOS versions prior to 1.3 recognize only four of these flags, the ones that determine whether or not you may Read, Write, Execute or Delete a file. *Read, write,* and *delete* are fairly self-explanatory—if set, these flags allow you to read from the file, write new information to it, and delete the file completely. *Execute* affects only program files—it allows DOS to execute (run) the program. If the Execute flag is not set on a file, AmigaDOS

will not run it from the CLI or Shell, even if it really is an executable file (though you still can run it from Workbench if it has an icon). Of course, even if the Execute flag is set, the file must be actually executable for AmigaDOS to load and run it.

The Hidden, Script, Pure, and Archive bits were added for AmigaDOS 1.3. Only Script and Pure settings are enforced by AmigaDOS. The S (or Script) flag is used to tell the Shell that a file is a script file that would normally be run with the EXECUTE command (Chapter Five covers such script files in great detail). If the Script flag is set, you can type in the name of a script file at a Shell prompt and the Shell will execute the script even though you don't type in the EXECUTE command name. Under Release 2, if you set the script bit of an ARexx program file, you can run the program just by typing in its name, without using the RX command.

The P (or Pure) flag is used by the RESIDENT command, which loads a CLI command into memory and uses it as if it were a "built-in" AmigaDOS command. Only "pure" programs (those that can be run from multiple CLI's at the same time, using only one copy of the program) can be made resident. Therefore, the Resident command will only make a program resident if its Pure bit is set, indicating that it is a suitable candidate, unless you use the Force option. It is not a good idea to make a program resident if you aren't sure that it is pure, because if you do so (either by setting its Pure flag yourself or by using the FORCE option), the program may function unreliably, or you may even experience a software failure.

The final two flags, Archive and Hidden, are always advisory. The Archive flag is used by some hard-drive backup programs to allow what is known as an incremental backup. Once you have made a copy of your hard drive, the program will set the archive flag on all of the files that you have copied. Any new file that is subsequently created or any old file that has been changed will not have this flag set. Therefore, the next time you go to back up your hard drive, you need only copy the files whose archive flags are not already set. AmigaDOS does not automatically set the archive bit every time you copy a file, however. That is strictly a function of the hard drive backup software. The final flag which marks a file as Hidden is reserved for future use. Currently, AmigaDOS pays no attention to this flag.

When a file is created, the RWED flags are set, and the four characters (*rwed)* appear in the LISTing of the file name (along with four dashes, representing unset flag bits). To change the protection status of a file, use the PROTECT command program. The form of this command is

```
PROTECT filename FLAGS rwed
```

where *filename* is the name of the file whose status you wish to alter, and *rwed* are the letters for the flags that you wish to enable. For example, if you want to remove just the deletion flag from a file called *LifesWork,* you'd enter

```
PROTECT LifesWork FLAGS rwe
```

This would allow you to read or execute the file (if it is a program), but not to delete it (or to write to it, which would require its deletion as an intermediate step). As of Workbench 1.3, the PROTECT command allows you to use the + and - characters to add or subtract one or more protection flags. For example, to set the Script flag on a file called *Execute.Me,* you'd enter

```
PROTECT Execute.Me +s
```

File Dating

The final item displayed by the LIST command program is the date and time that the file was created. The Amiga 2000 and 2500 come equipped with a battery-powered clock/calendar module, from which the time and date is read at power-on, using the SETCLOCK command. The Amiga 1000 does not come with such a clock module, and is optional on the Amiga 500. If your computer does not have a clock/calendar, it's up to you to set the correct time and date each time you turn on the machine, or reset the computer. You can find out what time and date the Amiga is currently using by checking the time setting in the Preferences program or by entering the command filename DATE. You can set the time from the Preferences program that comes with the Workbench disk or by using the DATE command program.

To set the time using the DATE command, use the form

```
DATE HH:MM:SS
```

where *HH* is a two-digit number for the hour, *MM* is a two-digit number for the minute, and *SS* is a an optional two-digit number for the second. If you don't specify the seconds, the Amiga uses 00 for you (if you don't specify seconds, you don't need to include the final colon). Note that hours are expressed in a 24-hour format, in which 1:00 p.m. is referred to as 13:00, and midnight as 00:00.

The DATE program expects the date in the format *DD-MMM-YY,* where *DD* is a two-digit number representing the day of the month, *MMM* is the first three letters of the name of the month, and *YY* is the last two digits of the year. For example, to set the date to September 30, 1991, you'd type

```
DATE 30-Sep-91
```

It's possible to set both the date and time with one command:

```
DATE 16-May-89 14:56
```

Besides the *DD-MMM-YY* format, AmigaDOS also understands some common ways of expressing the date, such as Yesterday, Today, Tomorrow, and the days of the week, such as Monday, Tuesday, Wednesday, and so on. You can use these expressions in place of the *DD-MMM-YY* format anytime you want to change the current date to one *within the coming week.* For example, let's say that you just turned on the Amiga and used the DATE command to find out the current time and date setting. If today is Sunday, November 30, 1986, and you last wrote a file to the disk the day before, you may find that the setting is *Saturday 29-Nov-86 20:20:02.* To make the date current, you need only type

```
DATE tomorrow
```

or

```
DATE Sunday 10:00
```

Either form advances the setting one day.

Remember that using the name of a day of the week (you can't use abbreviations here—you must use the full name of the day) will always set the date *forward* to that day. In the example above, if you'd typed `DATE Friday`, it would have set the date to Friday 05-Dec-86 instead of Friday 28-Nov-86. The only date word that sets the date backward is *Yesterday.* The *DATE Yesterday* command moves the date back by one day.

AmigaDOS also uses these words in its LIST display, so don't be surprised if you see recent files with dates like Yesterday or Today. The meaning of such terms in the LIST display is somewhat different from the DATE command, however. DATE expects that the new date you're setting will be later than the current date that's shown. So if you use day names like Tuesday, it sets the date to the Tuesday following the current date. LIST, however, assumes that files on an existing disk must have been created previously, so when LIST says Tuesday, it means the Tuesday *before* the current date. If you put in a disk that wasn't in the drive when you booted up the Amiga, and there's a file on the disk with a date later than the current date, LIST will show its date merely as Future. To see the actual date of such a file, you would have to change the current date far enough to the future so that it's *later* than that of the file.

If you've set the correct date, expressions like Today or Wednesday can be helpful in quickly picking out new files from old ones. But what date does the Amiga use if you haven't set the correct date? AmigaDOS sets aside a place on each disk where it records the latest date and time that a file was created. This latest date is updated with the current date and time every time you write to a file (provided that the current date is later than the latest date). When you boot up the computer, AmigaDOS checks the latest date recorded on the boot disk (and on the disk in the external drive as well, if one's inserted). It sets the current date and time just a little later than the latest date found (AmigaDOS appears to advance it by 11 seconds). That way, even if you forget to set a new time and date when you boot up, your files will still appear in correct chronological order. You won't be able to tell the exact date and time a file was created, but you *will* be able to tell which was created most recently.

This time-stamping feature of AmigaDOS can be a great aid when you're trying to identify one file among several. In fact, it's so convenient that if your computer doesn't have a clock/calendar, you may want to alter the startup command file so that it prompts you to enter the correct date and time whenever you turn the computer on. An example of such a file can be found in Chapter 5, which explains command sequence files. If you have a 1 meg Amiga 500, a 2 meg 600, or an Amiga 2000 or 3000, however, your machine comes standard with a hardware clock. In that case, you normally need only to set the hardware clock to the correct time and date, and from then on, the computer will read the time and date from the clock hardware when you turn the computer on. Setting the hardware clock under AmigaDOS versions 1.3 and earlier is a two-step process. First, set the correct time and date using the DATE command. Then, save this setting to the hardware clock using SAVE option of the SETCLOCK command:

```
SETCLOCK SAVE
```

Your startup script will automatically read the time and date from the hardware by using the LOAD option of the SETCLOCK command:

```
SETCLOCK LOAD.
```

Under Release 2, using the hardware clock is even easier. The Time preferences editor sets the hardware clock/calendar automatically when you hit the Save button, and the Release 2 operating system reads the clock whenever you reset the computer automatically, without need for the SETCLOCK LOAD command.

While normally you only need to set the date and time once, you may find at some point that the clock has stopped working, and SETCLOCK LOAD either yields a nonsense time and date, or the message <<unset>>. Many people assume that this indicates a hardware problem such as a battery failure, when most often, it really means that some errant program has written nonsense commands to the clock hardware. To correct this problem, issue the command SETCLOCK RESET from AmigaDOS 1.3 or higher, and then repeat the normal clock setting procedure.

Directories and Subdirectories (and Sub-subdirectories...)

With 880K of storage space, it's quite possible to have over a hundred entries in you list of file name. Working with that many names in a single list can be cumbersome—just finding a single name in such a large list can turn into a major chore. This problem becomes much worse when you start to work with a hard disk that has upwards of one hundred million bytes of available storage space.

AmigaDOS's answer to this problem is to provide multiple directory levels, which branch out from the highest directory on down. This allows you to group lists of related filenames into their own directory, where they are isolated from the other, unrelated files on the disk. Using smaller sub-lists of filenames makes finding and working with files much easier when the disk you are using contains a large number of files. Your Workbench disk, for example, contains directories like *c,* which contains command program files, and *devs*, which contains files for device drivers like the one that makes your printer work. Some of these subdirectories, such as *Utilities*, even have icon files associated with them which make them appear on the Workbench screen as drawers.

Root and MAKEDIR

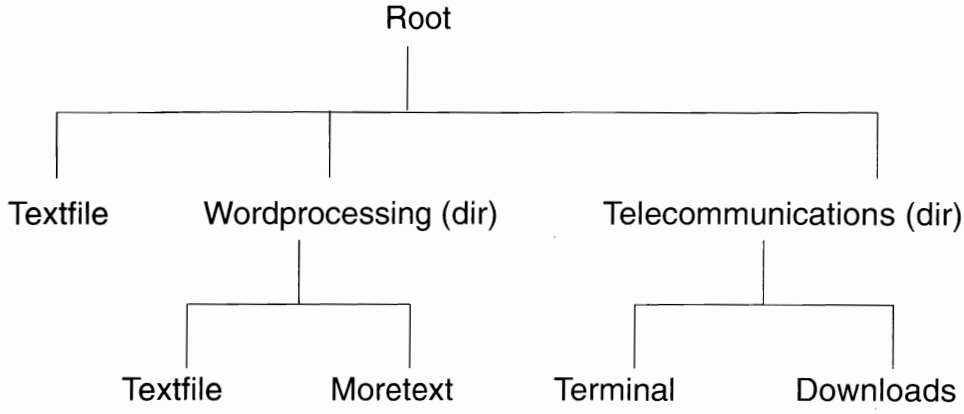
When you create a new file structure by formatting a disk, there's only one list of filenames on the disk. This is the highest level, or *root,* directory. When you write files to this disk, the names of these files are automatically recorded in the root directory. If you wish, however, you can create new directories, known as *subdirectories*, within the root directory. Let's say that you're going to use part of the disk for storing word processing files and part of the disk for telecommunications files. You could create separate subdirectories for each kind of file by using the MAKEDIR (make directory) command program. Just type *MAKEDIR*,

followed by the name of the directory. The rules for naming directories are the same as for naming files (see above for more information). Using the names in the example above, you'd type

```
MAKEDIR Wordprocessing
MAKEDIR Telecommunications
```

After you've put a few files into each of the directories, your directory structure might look like this:

A Typical Directory Structure



This structure is similar to what you might see when you draw a family tree. At the top level is the *root directory*, which contains a file (a data file called *Textfile*) and two subdirectories (*Wordprocessing* and *Telecommunications*). These subdirectories in turn contain their own files. The *Wordprocessing* directory contains the files *Textfile* and *Moretext,* and the *Telecommunications* directory contains the files *Terminal* and *Downloads.*

You'll notice that the root directory and the *Wordprocessing* directory both contain a file named *Textfile.* You probably already know that you can't have two files of the same name in the *same* directory. If you tried to create a new file with the same name as an existing one, the new file would overwrite and replace the existing one. But as you can see, there's no problem having two files of the same name in *different* directories. Each directory can be thought of as its own small disk except that a directory doesn't have a fixed size. A directory takes up as much space as required to hold its files and subdirectories.

Just as the root directory can contain either files or subdirectories, the subdirectories themselves may contain files or subdirectories. For instance, if you have a large number of document files in the *Wordprocessing* directory, you may wish to group them into subdirectories, such as *Personal Letters, Business Letters, Proposals,* and *Speeches.* There's no limit to the number of directory levels you can create—again other than the space available on the disk. Most people find, however, that about four or five levels down is as far as they care to go. A large list of subdirectories can be as bad as the large list of filenames that it was meant to prevent.

If you want to see the complete contents of a disk, including files within subdirectories, you can do so by adding the phrase OPT A or the word ALL to the DIR command. If you examined the sample disk illustrated above with the command DIR OPT A, you'd see the following display:

```
Telecommunications (dir)
        Downloads           Terminal
Wordprocessing (dir)
        Moretext            Textfile
Textfile
```

Gaining Access

You can gain access to files within subdirectories in one of two ways. One method is to specify complete information about the file, including each of the directory levels between it and the root directory (this is known as the *full pathname*). You do this by naming each of the directories, in order, from the root down, separating the name of each directory with a slash (/). If the disk described above is in the internal drive, you could refer to the file *Textfile* in the *Wordprocessing* directory as *DF0:Wordprocessing/Textfile.* Specifying the entire path from the top down always works, but it can be a bit tiresome (particularly with a file like *DF0:Wordprocessing/Personal Letters/Aunt Charlotte—Thank You*).

An easier alternative uses the concept of the current, or default, directory. If you refer to a file without specifying a device or directory path, AmigaDOS looks for that file in whichever directory is currently the default directory. When you first start up the computer, AmigaDOS sets the root directory of your boot disk (the one in the internal drive or the boot partition of your hard drive) as the

current directory. You're free to assign a new current directory at any time. Just type **CD** (for the Current Directory command program), followed by the name of the directory (or directory path, if you're going down more than one level). Release 2 users don't even have to include the CD command—just typing the name of the new directory at the command line is enough to make it the current directory. Using the same example, you could make the *Wordprocessing* directory the current one by typing

```
CD Wordprocessing
```

From then on, when you want to use the file *Textfile,* you can refer to it by name, instead of as *Wordprocessing/Textfile.* If you use the command DIR after changing the current directory to *Wordprocessing,* you see only a list of the files in that directory.

Up and Down

It's possible to skip down more than one level at a time. If you want to change the current directory from the root directory to the *Business Letters* subdirectory of the *Wordprocessing* directory, enter

```
CD "Wordprocessing/Business Letters" (quotes needed
    for names with spaces)
```

Unless otherwise specified, the CD command assumes that the name you give it is of a directory or path that lies *below* the level of the current directory. To move up to a higher level, you must use one of two special characters. The first is the familiar slash (/). A slash *in front of* a directory name is the signal to move *up* a level to the directory which contains the current directory. The slash alone works—you don't have to specify the name of the higher directory—since each directory has only one directory immediately above it. To change the current directory to the one immediately above, just type

```
CD /
```

You're not limited to a single slash. You can use as many slashes as there are directories above the current one. Thus,

```
CD //
```

moves you up two directories.

Nor are you limited to going in one direction at a time with CD. Assume that your current directory is the *Letters* subdirectory of the *Wordprocessing*

directory, and you want to change to the *Telecommunications* subdirectory of the root directory. You could use the command form

```
CD //Telecommunications
```

The first slash takes you up to *Wordprocessing,* the second slash takes you up to the root directory, and *Telecommunications* takes you down one level to make that directory current.

If your goal is to return to the root directory, however, it's not necessary to enter a slash for each level. You can use the colon (:) to indicate a move directly up to root level. For instance,

```
CD :
```

changes the current directory to the root directory of the current volume, while

```
CD :Telecommunications
```

changes the current directory to the *Telecommunications* subdirectory of the root, no matter how far down you were when you entered the command.

CD is not the only command that interprets an initial slash as a signal to move up one directory level, and you can use the colon to refer to the root directory at any time. Commands such as

```
DIR :
DIR :Wordprocessing
DIR /
DIR /Wordprocessing
```

all work, as long as the directories referred to really exist.

If you wish to change the default directory to one located on another disk, you must specify the device name or volume name when using CD. To switch to the root directory on the disk in the external 3½- -inch drive, for example, you'd use

```
CD df1:
```

Note that when you switch the current directory to another disk, AmigaDOS internally refers to that disk by its volume name and not by the device name of the disk drive in which it's mounted. This means that when you put a disk with volume name CLI in drive df1: and type `CD DF1:`, AmigaDOS changes the current directory to the root directory of volume CLI. If you take that disk out of the external drive and replace it with another, AmigaDOS will not recognize the root directory of that disk as the current one. Use DIR with the

new disk in the drive, and DOS won't comply. It will put up a requester box asking you to replace volume CLI in any drive. That's because AmigaDOS identifies the current directory as the root of the specific disk named CLI, not just any disk that happens to be in the external drive. When you wish to replace that disk with another, you should change the current directory to one of the disks you'll be using. In the example above, once you replaced the CLI volume with another disk, you could issue the command `CD DF1:` once again, making the root directory of that volume the new current directory. Then if you issued the DIR command, you would not be prompted to swap disks. If you're ever unsure which is the current directory, simply use the command `CD` (and that's all) to display the current directory name. For more information on device names, logical devices, and volume names, see Chapter 4.

File Manipulation Commands

Some of the most commonly used CLI commands are those which copy, delete, rename, and join (combine) files.

COPY

The COPY command is used to create a duplicate of a file in the same directory, in another directory, or even on another disk.

```
COPY vitalstuff TO vitalstuff.backup
```

This creates a duplicate copy of the file in the same directory with another name.

```
COPY programfile Programs/programfile
```

While this command line creates a copy of the file with the same name in the subdirectory named *Programs*.

```
COPY filename df1:
```

And this command makes a copy of the file (with the same name) in the root directory of the disk in drive df1:.

RENAME

The RENAME command program changes the name of a file or a directory. When you RENAME a *directory*, you change its position in the directory structure:

```
RENAME program TO program.old
```

This changes the name of the file *program* to *program.old*.

```
RENAME df1:c/delete TO df1:c/erase
```

While this command line changes the name of the command program *delete* on disk drive df1: to *erase*, also on disk df1:.

```
RENAME Wordprocessing/Letters TO :WordWiz/Textfiles
```

And this example moves the directory *Wordprocessing/Letters* and all of its contents to the directory *WordWiz/Textfiles*.

While renaming can be used to change the location of a file or directory on its current volume, it cannot be used to move it to a different volume.

DELETE

DELETE removes a file from the disk. Once you delete a file, the information contained in it is lost forever. DELETE lets you name up to ten files to delete at a time. Separate each filename with a space.

```
DELETE oldfile
```

This permanently erases the file *oldfile*.

```
DELETE oldfile1 oldfile2 oldfile3
```

And this sample erases all three of the named files.

DELETE can also be used to erase a directory, but *only* if it does not contain any files or subdirectories. You can use the same DELETE command first to erase the files in the directory, then to delete the directory, or you can use the keyword ALL. For example:

```
DELETE Wordprocessing/lonefile Wordprocessing
```

This first deletes the only file in the *Wordprocessing* directory, then deletes the directory. Or you can use

```
DELETE Wordprocessing ALL
```

which deletes the directory and all files that it contains.

JOIN

The JOIN command file takes the contents of from 2 to 15 files and combines them into a new and larger file (the Release 2 version of JOIN is no longer limited to joining 15 file). The original files are unchanged.

```
JOIN firsthalf secondhalf AS bothparts
```

This creates a new file called *bothparts* which contains all of the information of both *firsthalf* and *secondhalf.*

Pattern Matching (Wildcards)

Sometimes it's possible to specify one or more filenames which have a common characteristic without typing the entire filename. This technique, called *pattern matching*, lets you perform tasks like listing all files with names ending in the characters *.bas* or deleting every file in a directory at once.

AmigaDOS pattern matching is similar to the concept of *wildcard* characters used in MS/PC-DOS, but there are important differences. In PC-DOS, the asterisk (*) can be used to substitute for any string of characters in a filename. In AmigaDOS, the asterisk is used as an escape character, to allow quotation marks (and other asterisks) in a filename. Also, as you'll see in the next chapter, the asterisk is used to refer to the currently active console device.

PC wildcards can be used with more commands than AmigaDOS pattern matching, which is mostly confined to COPY, DELETE, DIR, and LIST (though the range of commands that allow the use of patterns is greatly expanded in Release 2). AmigaDOS patterns, however, are much more flexible. They allow you to match names starting with the same group of characters, end with the same group of characters, or have the same characters in the middle, preceded by any number of characters and followed by any number of characters. Such flexibility makes the system somewhat complex to learn, but well worth the time and effort required.

? and

The most important pattern matching characters are the question mark (?) and the pound sign (#). The pound sign followed by a single character matches any number of repetitions of that character (including none). For example, *#STUTTER* matches *STUTTER* (#S substitutes for one *S*), *SSSSTUTTER* (#S substitutes for four consecutive *S*'s), and *TUTTER* (#S can also substitute for zero occurrences of the letter *S*).

The question mark is used to replace any single character (but not the null string, or no character). Thus, *?LA?S* matches *FLATS* (first ? replaces *F,* second replaces *T)* or *2LAPS* (first ? replaces 2, second replaces *P),* but not LAPS (first ? must replace an actual character).

When you put these two special characters together (# ?), they become a powerful pattern which can match any number of any characters (or no characters at all). For example, you could use *PART#?* if you wanted a pattern to match all filenames starting with the letters *PART*. If you wanted to LIST all of the icon information files (whose names always end in *.info*), you could use the pattern *#?.INFO* to find them. You could also use a pattern like *PART#?.INFO* to match any file starting with *PART* and ending with *.INFO,* with anything (or nothing) in between (like *PARTICLE.INFO, PARTYANIMAL.INFO, PART47ZYC-332.INFO,* and even *PART.INFO*). Likewise, you could use a pattern like *#?CAT#?* to match a filename which had the letters *CAT* anywhere in it (like *CATAPULT, SCAT, SCATTER,* or "*I SNEEZE AT CAT HAIR*").

( )

In addition to the pound sign and question mark, there are three other characters which have special meaning when used for pattern matching. Parentheses ( ) may be used to group a number of characters together into a single pattern element. If you follow a pound sign with a group of characters within parentheses, for instance, it will match any number of repetitions of that pattern group (including none). Thus, *#(YO)* matches the filenames *YO, YOYO, YOYOYOYO,* and so on. If you didn't use the parentheses, *#YO* would match *YO* and *YYYYO,* but not *YOYO,* because the #Y could substitute only for repetitions of the single letter *Y*. Parentheses let you become creative, like using *#(P?NG)* to match the filename PINGPONG.

|

The vertical bar (|), entered by pressing the SHIFTed backslash key, is used when you want one of two or more patterns to match the characters in the filename. *A | B* matches either the letter *A* or the letter *B*. The pattern *GOOD | BAD* matches either a file named *GOOD* or one named *BAD*. And pattern *MO(B | N)STER* matches both *MONSTER* and *MOBSTER* (note how the parentheses were used to set off the *B | N* as a distinct pattern). AmigaDOS versions prior to Release 2 understand the vertical bar as applying only to one part of the pathname. For example, the command

```
COPY df0:c/COPY|DIR|LIST to ram:
```

would be understood to mean "copy any file named COPY, DIR, or LIST from the C directory on df0: to the root directory of the RAM disk," and not "copy the

file named COPY from the C directory of df0:, or DIR or LIST from the current directory to the RAM disk." This aspect of the vertical bar comes in quite handy when dealing with several files in a complicated path at one time. Another peculiarity of early versions is that prior to Workbench 1.3, there was a limit of 31 characters in any one wildcard expression, which limited the number of files you could join with the vertical bar. This limit was removed in 1.3.

With AmigaDOS Release 2, the rules of using the vertical bar changed somewhat. To copy multiple files from a single directory, you need to enclose the filenames in parentheses. The above example must be changed to:

```
COPY df0:c/(COPY|DIR|LIST) to ram:
```

in order to work properly. Since this does the correct thing in all versions of AmigaDOS and is easier to understand, always use this form of the command when you want to access multiple files from the same directory.

%

The percentage sign (%) represents the null string (no character). You've already seen how a pattern starting with the pound sign matches any number of repetitions of the following character, including none at all. The pattern *S#HIN,* for example, matches *SHIN, SHHHIN,* and *SIN.* But if you want to match only a single appearance of the character or none at all, you can use the form *(H | %),* which stands for either *H* or the null character (no character at all). Therefore, *S(H | %)IN* would still match *SHIN* and *SIN,* but would not match *SHHHIN,* which repeats the *H* character more than once.

Combining the percentage sign with the question mark in the form *(? | %)* creates an expression that will match *any* character or no character at all. Using a previous example, you could substitute the pattern *(? | %)LA?S* to match either *2LAPS* or just plain *LAPS.*

'

There's another special character used to address a problem created by the other special characters. Since those characters have special meanings in the language of pattern matching, it makes it difficult when you want to match a name which contains one of those special characters as part of the filename. In order to match a filename that contains a question mark, for example, you must precede the question mark with an apostrophe (') to let the pattern matching mechanism know that you want to match an actual question mark, without using the question

mark as a substitute for any other character. For instance, you could use the pattern *?OW'?* to match filenames like *HOW?* and *COW?*.

Naturally, since the apostrophe is now a special character, you must use two apostrophes to represent an apostrophe which might be part of a filename. A pattern like *?ONT''T* is needed to match filenames like *DON'T* and *WON'T*. If these rules remind you of the rules for naming files, all the better. The same rules apply to pattern substitution, too. If you're using a pattern containing space characters, for example, you must enclose the entire pattern with double quotation marks.

~ and -

Release 2 of AmigaDOS introduces some very handy new pattern characters, the tilde (~) and the dash(–). The tilde functions as a *not* operator, selecting any file that doesn't match the pattern. For example, if you wanted a directory listing of all files except icon files (those whose name end in .info), you could use the command *DIR ~(#?.info)*. The other new character, the dash, is used in an expression set off by square brackets to indicate a range of characters. Thus, the expression **[W-Z]#?** would select any file beginning with W,X,Y or Z, just like the expression **(W|X|Y|Z)#?**. Be sure that to use the square brackets (the keys just to the left of the top of the Return key), not the parentheses. To copy all files from the c: directory that begin with the letters A-D, you could use the command

```
COPY c:[a-d]#? ram:
```

Pattern Matching Summary

#c	Matches any number of repetitions of the character *c* (including none) *N#O* matches *N, NO, NOO,* and *NOOOOOOOOOO*
#(group)	Matches any number of repetitions of *group* (including none) *#(TOM)* matches *TOM* and *TOMTOM*
?	Matches any single character (but not the null character) *K?NG* matches *KING* and *KONG* (but not *KNG*)
#?	Matches any number of repetitions of any character (including none) *#?.BAS* matches any *filename* ending in *.BAS*

P1 | P2	Matches either pattern *P1* or *P2* *B(A | O)Y* matches *BAY* and *BOY* *df0:c/LIST|DIR* matches *df0:c/LIST* or *df0:c/DIR*
%	Matches the null string (no character) *(S | %)TOP* matches *STOP* or *TOP*
(? | %)	Matches any character or no character *(? | %)LOT* matches *SLOT, CLOT,* and *LOT*
( )	Used to set off a group of characters as its own distinct pattern *(M | P)A* matches *MA* or *PA* *M | PA* matches *M* or *PA*
'	Used in front of one of the special characters to show that you want to match it, not invoke its special meaning *?ON"T* matches *WON'T* and *DON'T*
[C1 - C2]	Matches any character in the range from *C1* to *C2* **AmigaDOS Release 2 and higher only.** *C:[t-w]#?* matches *TYPE, VERSION, WHICH* and *WAIT*
~P	Matches any file that does not fall within the definition of pattern *P* **AmigaDOS Release 2 and higher only.** *~(?.info?)* matches all files in the current directory **except** the icon files whose names end in *.info*

Chapter 4

Devices

The main function of a disk operating system like AmigaDOS is to let you control disk storage devices. But there are several other kinds of devices with which the Amiga can communicate, and the CLI also provides ways of interacting with them. Some of these devices are physical devices, like the console screen and keyboard, hard disks, streaming tape drives, printers, and modems. Others are "software" devices like the RAM disks and pipe handler. AmigaDOS can even treat a disk directory as a *logical* device.

Some devices are a standard part of the operating system, and as such are automatically recognized by AmigaDOS. Others use the Amiga's capability to auto-configure expansion devices to have their device driver software automatically added to the system. Still others must have their driver or handler software added to the system by use of the MOUNT or BINDDRIVERS command.

Disk Drives

Every Amiga comes with an internal disk drive. This device is known as DF0: (device type = Drive, drive type = Floppy, drive unit = 0). Optionally, you can connect an external 3½-inch drive (or in the case of the 2000, 2500, or 3000, a second internal drive), known as DF1:.

Although the Amiga supports up to four floppy drives, the power supply that comes with the Amiga 500 really only provides enough power for one external drive. If you want to run more drives than that, you should find a drive with an independent power supply, or buy one of the larger replacement power supplies that are available for the 500.

The Amiga 2000 or 3000's power supply is quite sufficient for four floppies, and even the 1000 should be able to handle three or four of the newer drives, which consume much less power than previous models.

The 5¼-inch drive which Commodore used to offer for use with MS-DOS disks is self-powered, but is not automatically recognized by AmigaDOS. The MOUNT command must be used to add this device to the system (a sample entry

can be found in the DEVS:Mountlist file that comes with the Workbench for this device, under the name DF2:). When used this way by AmigaDOS, however, the 5¼-inch drive can only store 440K, half of the amount stored on a 3½-inch drive.

A disk drive does not accept a single, indivisible stream of information like a printer does. Rather, its storage area is divided into a number of different files, whose filenames are stored in one or more directories. Therefore, you'll most often use the device name DF0: or DF1: only as part of a file or a directory description.

A Complete Description

The most complete kind of file description contains the disk device name, followed by the names of each succeeding directory level (which are separated by slashes), then finally the name of the file. The name *DF1:WordWiz/Letters/Formletter* is a good example. The filename is *Formletter*, which is stored in the directory *Letters*, which in turn is in the directory *WordWiz*. All are found on device *DF1:*, the external (or second internal) disk drive. While this kind of description tells AmigaDOS exactly where to look for the file, it takes quite a few keystrokes to get there.

Fortunately, you don't always have to give a complete description of a file. AmigaDOS also recognizes references to a file which are relative to the *current*, or *root,* directory. One directory is always recognized as the current directory. When you power an Amiga up, it uses the root directory of the disk it starts from (either an internal floppy or hard drive) as the default directory. Therefore, if you boot from a floppy and immediately refer to a file as *Myprogram,* AmigaDOS interprets this as *DF0:Myprogram.* If you change the current directory to *C,* for example, using the CD command, a reference to the file *Dir* will be taken to mean *DF0:C/Dir.*

You can use the colon (:) to indicate the root directory of the disk on which the current directory is located. Therefore, even when *DF0:C* is the current directory, you can specify a file in the *S* directory with the description *:s/startup-sequence,* which is equivalent to *DF0:s/startup-sequence*. Note that AmigaDOS ignores case in these names. Any combination of uppercase and lowercase can be used, as long as the letters themselves match.

You may always use the volume name of the disk itself in place of the device name of the drive in which it's mounted. For example, if you had a file called *program.bas* located on a disk whose volume name was *Extras,* you could describe the file as *Extras:program.bas.* In fact, such a description is often

preferable to using the device name of the drive, since it is valid regardless of which drive is used for the *Extras* disk.

In some cases, it is necessary to refer to a disk by its volume name. Let's say that you have only one disk drive and want to list a directory of a disk which doesn't contain the DIR command program. The volume name of this disk is *Stuff*. When you insert the *Stuff* disk into the drive and type `DIR`, the system prompts you to put the disk containing the commands into the drive. When you do, the Amiga lists a directory of *that* disk, not *Stuff*. But if you enter *DIR Stuff:*, you'll be prompted first to put in the disk with the commands, then to put in *Stuff*. Now you'll get a listing of the *Stuff* disk. Of course, there are other solutions to this problem—you could copy the DIR file to *Stuff*, or you could copy your commands to the RAM: disk device (see below). But if you want to specify operations on a particular disk, using the volume name assures you of the correct result. In fact, AmigaDOS keeps track of the disk with the current directory in just this way. If you take the disk out and type in a command, DOS prompts you with the volume name of the disk it wants you to insert.

Hard Disks

Most hard disks have fixed media that can't be removed from the drive like ordinary disks. The drives either fit inside the computer itself, or in a box that sits next to the computer. Although hard drives are more expensive that floppy drives, they can read and write information much more quickly, and can store as much information as dozens or even hundreds of floppy disks. These advantages are particularly important to Amiga owners, since the Amiga floppy drives are relatively slow, program and data files are getting larger all of the time, and some of the Amiga system software (such as text fonts) must be read in from disk. As we will see in the section on logical devices, software may need to read operating system files from the Workbench disk at any time, so it can be quite handy to have that disk constantly available. With the 1.3 Kickstart ROM, Commodore introduced auto-booting to the Amiga, which means that it is possible for the computer to start up from the hard drive, without having to read a Workbench disk in the floppy drive.

Although hard drives were not well supported by AmigaDOS until version 1.2, there are now a number of different interfaces that can be used to add a hard disk to your system. Early hard drives added their device control software to the system through the use of the MOUNT or BINDDRIVERS command, or by means of a special mounting program like *SupraMount* (for hard drives manufac-

tured by Supra Corporation) or *DJMount* (for drives connected to the A2000 via the IBM compatibility option, or Bridge Board).

Newer hard drive controllers which follow Commodore's Rigid Disk Block specification, store all of the mounting information on the hard drive itself. This makes it possible to mount all of the drive partitions automatically, and to boot the computer from any of these partitions. It also makes it possible to connect just about any drive to any controller, and have it work without reformatting the drive to conform to special layout requirements of that controller.

Traditionally, DH0: (for Hard Drive 0) is the device name used for a hard drive. There are many exceptions, however. For example, SCSI (Small Computer System Interface) drives connected to Commodore's own hard drive controller are numbered starting at DH2:. AmigaDOS partitions on a hard drive that is connected to a PC Bridge Board on the Amiga 2000 are known to AmigaDOS by names such as JH0: and JH1: (which stands for Janus Hard drive, after the Janus software which allows the Amiga and IBM-compatible sides of the computer to work together). Since the introduction of Rigid Disk Blocks, you can give a hard drive any device name you want, such as *MyHardDrive:*. Since some programs have problems with long device names, however, it is often best to stick with names that are three characters long, such as PRG: for a program drive, or DAT: for a data drive.

Hard drives are much larger storage devices than floppy disks, holding as much information as dozens of floppy disks. For this reason, most hard drive interface software allows the user to partition the drive into smaller, logical drives. A 105 megabyte drive, for example, might be divided into a 15 megabyte partition called DH0:, a 40 megabyte partition called DH1:, and a 50 megabyte partition called DH2:. The details of partitioning depend on how the device driver software is added to the system. Hard drives that use the MOUNT command usually have an entry describing each partition in the devs:MountList file. Hard drives that follow the Rigid Disk Block standard generally come with partitioning software, such as Commodore's *HDToolbox*, which writes the partition information to the hard drive itself. IBM-compatible hard drives that are interfaced through the Bridge Board use a program on the IBM side called ADISK which creates AmigaDOS partitions on the MS-DOS hard drive.

Using a hard drive on the Amiga is very similar to having a large floppy disk drive, so any information in this book concerning floppies (DF0: and DF1:) generally applies to hard disks as well. Transferring programs to hard disk from floppies, however, can present a number of difficulties to novice users. While a program can generally rely on the volume name of its floppy disk, it has no way

of knowing the name of the hard drive partition to which it has been copied. For that reason, some programs require a logical device name be assigned to the hard disk subdirectory from which they are run. Another problem that complicates hard drive installation is that some programs need support files in addition to the main program file. These support files may need to be copied to the same directory as the program file, or they may need to go into one of the system directories. When in doubt, consult the program's instruction manual. If the program is capable of running from a hard drive, there are likely to be quite detailed installation instructions. Also, check to see if the software includes a hard drive installation utility. Many packages come with such a program, which completely automates the installation process.

Commodore introduced a new piece of system software in Workbench 1.3 called the Fast File System (FFS). This is a new AmigaDOS disk software interface which provides much quicker disk access than the old filing system. The FFS comes standard with Release 2, but must be loaded from disk in versions 1.3 and below. For that reason, you can't boot from an FFS floppy under 1.3. In order to be compatible with all systems, therefore, almost everybody uses the old file system for floppies. Non-removable media, however, like hard disks and RAM disks don't face this problem.

Most hard drives can be formatted using the new layout, resulting in speed increases of 500% or more. The new filing system software is located in a file called *FastFileSystem,* which is located in the *L* directory of the Workbench disk. (It's built in on Release 2 and up.) Details of using the Fast File System vary from drive to drive, so you'll have to consult your hard disk interface manual for instructions on installing in on your hard drive. In most cases, the hard drive preparation software automatically installs the FFS.

Release 3 introduces a new file system, the Directory Caching File System (DCFS). This file system is designed to make directory searches, and thus file reads, somewhat faster. On a fast hard drive system, however, it does not speed up disk reads as much as it slows down disk writes. It is therefore more suitable for use with floppies, but only those disk that will be used exclusively on a system with Release 3 or higher.

PC0: and PC1:

These two new MOUNTable devices, added in version 2.1 and higher, are software-only file system extensions that permit you to use your normal 3½-inch floppy disks to read and write IBM/MS-DOS compatible disk. The software is

licensed from Consultron, who sells it independently under the names CrossDOS and Cross/PC. It includes a device driver that is able to handle the physical format in which MS-DOS stores information on a disk (devs:mfm.device), and a file system handler that lays out the information in the same format as that used on PC's and compatibles (l:CrossDOSFileSystem). Because the file system is mounted as a normal Amiga device, you can read or write to it directly like any other device. In other words, if you have PC0: mounted, you can insert a 720K IBM 3½-inch floppy into the first internal drive and read a text file into your word processor by selecting drive PC0: in the file requester, and write out the modified file the same way. Versions 2.1 and higher of the operating system include enhanced versions of the DISKCOPY and FORMAT commands which recognize and can deal with IBM format floppies.

There are some limitations to keep in mind with these devices, however. First, the normal-density 3½-inch drive used in most existing Amiga models can only handle 3½-inch disks that have been formatted at 360K or 720K densities. Only the newer high-density disks that come with some Amiga 3000, 3000T and 4000 models can handle the 1.44M format disks used by most new IBM compatibles (and by Macintosh models that include a Superdrive). Also, in order to assure complete compatibility, you must observe MS-DOS filename conventions. While AmigaDOS files can have very long names, MS-DOS filenames can be no longer than eight characters, followed by a period and three additional characters (e.g. "MYLETTER.TXT").

When PC0: or PC1: is MOUNTed, it "shares" the drive with DF0: or DF1:. This means that if you have an MS-DOS disk mounted in the first internal drive, the operating system may flash a requester that tells you that the disk in DF0: is "Not a DOS disk." There is no cause for alarm however—by definition, a disk that is right for PC0: can't be read by DF0:, even though both devices use the same drive.

The RAM: Disk

There's another disk drive available to all Amiga users. AmigaDOS allows you to reserve a section of memory for use as a super-fast electronic disk drive, known as the RAM: device. The RAM: device does not exist when you first start up the computer. It is automatically created by AmigaDOS at the first reference to the device. For example, when you COPY a file to RAM:, AmigaDOS creates the device if it doesn't already exist. But you don't have to move any information to RAM: in order to create the device. Typing a command like `CD RAM:`,

which changes the current directory to the root directory of RAM:, works as well.

Though AmigaDOS understands references to RAM:, the actual device handler for RAM: (the program which routes information to the device) must be loaded in from disk before the device can be used. This handler is located in a file called *Ram-Handler* in the directory of the system disk. If this file is not available when the first reference to RAM: is made, the device cannot be created. Once it's loaded, however, the system doesn't have to refer to this file again when using the RAM: device. The "startup-sequence" file on Workbench disks whose version number is 1.2 or higher refer to RAM: in the initial command sequence, so the RAM disk is automatically created when the Workbench is started.

You can read, write, execute, and delete files from RAM: just as from any other disk device. There are, however, a few important differences. The most significant is that RAM: is a *temporary* storage device. Its files disappear when you turn off the power or when you warm start the computer with the CTRL-Amiga-Amiga key combination. *If you store files to RAM:, then, remember to copy them to a physical disk device before you turn the power off or reset the computer.*

Another difference between RAM: and the physical disk drives is capacity. The 3½-inch disks have a fixed storage capacity of 880K, but RAM: is limited to available free memory. Unless you have substantial expansion memory, you won't be able to store as much in the RAM: disk as on the physical drives. In fact, you should avoid storing too much information in the RAM: disk. It's possible to crash the system if you take up all available memory. Even if things don't reach that stage, however, you may not have enough room to run application programs if your RAM: disk is too full.

One of the best ways to put the RAM: disk to use is to copy all or some of your CLI command programs to it and use the ASSIGN command (explained below) to make it the new command directory. The simplest way to do this is

```
COPY C: RAM: ALL
    ASSIGN C: RAM:
```

This is discussed at greater length in the section "Logical Devices," later in the chapter.

Communications Ports

The Amiga personal computer comes with two communications ports—one serial and one parallel. The serial port can be used for transferring information to or from a modem (or another computer), a MIDI musical device (with the proper interface), or to a serial printer. The communication speed for this serial interface can be set from the Preferences program at speeds ranging from 110 to 19,200 bits per second (bps). The parallel port is initially set up by the system as a Centronics-type printer interface, which can be used only to send information to a printer. Application programs (but not AmigaDOS) can configure this parallel port so that it can be used to input information as well. For example, external devices such as audio digitizers, video digitizers, and scanners, all use the parallel port for input rather than output.

AmigaDOS allows you to write information to either of these devices just as you would to a disk file. For example, if you wish to transfer the contents of a disk file named *wordfile* to a parallel printer, you could use the command `TYPE wordfile TO PAR:` or `COPY wordfile TO PAR:`. You could send the contents of the file to a serial printer or modem with the same commands by substituting the device name SER: for PAR:. You may also use the redirection operator (>) to cause the output from one of the disk commands to be sent to the parallel or serial devices (see the section on redirection below).

The software handlers which actually send output to the communications ports are not an integral part of AmigaDOS. They reside on disk files named *serial.device* and *parallel.device* in the *devs* directory of the Workbench disk. The first time that AmigaDOS tries to open these devices, it must read the proper handler file from disk. If it cannot find the file, it cannot open the device. Once the handler is loaded, it stays in memory, and DOS doesn't need to access the file again.

Using PRT:

Although you can control a serial printer directly through the SER: device and a parallel printer via the PAR: device, there's a better way. The device called `PRT:` can be used to send output to the printer, regardless of whether you have a serial or parallel printer connected. In the old Preferences scheme used in Workbench 1.3 and below, the PRT: device gets its information about which type of printer is connected from the *system-configuration* file in the *devs* directory. This is the file which the Preferences program uses to store the preference settings. In the new Release 2 preference scheme, PRT: gets its information from the

printer.prefs file which is stored in the *prefs/env-archive/sys* drawer on the boot disk, and moved to the *ram:env/sys* drive before the Workbench is loaded. In order to route information through the printer device, DOS must first load a handler that's stored in the disk file *printer.device* in the *devs* directory of the Workbench disk. This handler itself must refer to a specific printer-driver file in the *printers* subdirectory of the same *devs* directory. The PRT: device uses the information stored there to translate control codes (such as those used to start and stop underlining) to equivalent codes used by your printer. In addition, the PRT: device translates the linefeed character (CTRL-J or ASCII 10) to a carriage-Return character (CTRL-M or ASCII 13), plus a linefeed character. If you wish to use PRT:, but don't want a carriage Return added to the linefeed, you may specify the device PRT:RAW.

Diagnosing problems with the PRT: device can be difficult, because there is both a hardware and software component to the device. Printer problems may be due to the hardware connection between the printer and the computer, or they could be due to problems with the printer driver software. When testing a printer, it is helpful to first try sending output to the PAR: or SER: device first (depending on whether the printer is connected to the parallel or serial port). A command like "COPY s:Startup-Sequence to PAR:" should print the file to the device (though the line spacing may not be right, due to the absence of carriage Return characters). If this procedure doesn't work, the problem is with the hardware connection between your printer and computer (either the printer isn't ready to print, or you've got a bad cable). If it does work, however, you know that the problem lies with the printer driver software.

To summarize the AmigaDOS devices names that can be used to send information to the printer:

Device Name	**Function**
PAR: or SER:	Sends data directly to the printer, with no translation.
PRT:RAW	Sends data to the printer, translating printer codes, but does not add a carriage Return to each linefeed.
PRT:	Sends data to the printer, translating printer codes, and adds a carriage Return to each linefeed.

Console and Others

The console device is used to accept input from the keyboard and the mouse, and to print the characters on the screen. Output goes to a window on the screen,

known as the *console window*. The console device accepts input from the keyboard a line at a time. At any point before you press the Return key, you may edit the line using CTRL-H or the BACKSPACE key to delete characters, and CTRL-X to delete the entire line (see Chapter 2 for more information about the editing capabilities of the console device). When the console receives a line of text, it translates the keystrokes into ASCII and extended ANSI codes. As noted in Chapter 2, the console device responds to a number of ANSI escape codes which control things like cursor positioning, screen scrolling, line insertion and deletion, and the like. Workbench 1.3 comes with an enhanced console device called NEWCON:, which introduces new features like line editing and command history. Since this device was not automatically recognized by AmigaDOS until Release 2, we'll cover it more fully in the treatment of MOUNTable devices, below.

Each CLI comes with its own console window (it's the window in which the >n prompt appears). When you use the NEWCLI command to start a new CLI process, you may specify the starting position, size, and title of its console window (see Chapter 2 for more information on starting a new CLI process). If you don't specify these characteristics, a default console window is used.

It's possible, however, to create your own console windows which are not related to any existing CLI process. To do so, you refer to the device as

```
CON:hpos/vpos/width/height/windowtitle
```

where *hpos* is the horizontal position of the top left corner of the window (expressed as the number of pixels in from the left edge of the screen), *vpos* is the vertical position of the top left corner of the window (expressed as the number of pixels down from the top edge of the screen), and *width* and *height* give the size of the window in pixels. The maximum size for a console window is the screen size, 640 × 200 pixels in the default Workbench screen. The minimum is 81 × 25 pixels. The *windowtitle* entry is optional and allows you to enter a title which will appear in the title bar. If you don't enter a title, the title bar is left blank. Note that the final slash is required, even when you don't specify a title.

Each console window comes with a sizing gadget to change its size, but in versions prior to Release 2, the window doesn't redisplay the current data after you change the window size. This means that if you make the window smaller, the text in the area the window previously occupied is wiped out. If you later make the window larger again, the new area of the window will be blank, rather than holding its old contents. Besides the sizing gadget, each console window has the depth arrangement gadgets in the upper right corner, which let you send

the window to the back of the screen or bring it forward on top of another window. Console windows also have a drag gadget (which coincides with the title bar) that lets you change the position of the window on the screen.

Like the RAM: device, you create a new console window by referring to its device name. For instance, to LIST the directory to a new console window, you could type

```
LIST TO CON:0/0/640/100/
```

Try this, and you'll see that although a new console window is created and the listing prints within it, it disappears as soon as the command is completed. Though you can pause the display before it disappears by hitting any key (use the BACKSPACE key to restart), the short-lived nature of such a window limits its usefulness as an output device.

A number of options were added to the console window in AmigaDOS Release 2 to expand its usefulness. The CON: window description can have one more or more *options* entries appearing after the title, each separated by a slash:

```
CON:hpos/vpos/width/height/windowtitle/option1/
    option2...
```

These options include:

AUTO	The window opens automatically when the program that opened it requires input or produces output.
CLOSE	A close gadget is included in the window border. This is the default case if no options are selected.
BACKDROP	The window type is changed to backdrop, which means that it appears behind all other windows on the Workbench screen, and you cannot depth-arrange, move or resize it (except by using the zoom gadget).
NOBORDER	No visible line is drawn around the window, although the zoom and close gadgets will still appear above it. If you zoom this window to full size, those gadgets will disappear, and you will have a full-screen window that can't be sized or moved.
NODRAG	The window can't be dragged. It will have a zoom, depth, and size gadget, but no close gadget.
NOSIZE	The window will not have zoom, size, or close gadgets. Only a depth gadget will appear.

SCREENname	The window will open on the public screen whose name is *name*. For example, to open the CLI window on the public screen name Fred, you would add /SCREENFred to the end of the window description. This option only works if a public screen of the specified name is already open.
SIMPLE	Choses the simple window refresh scheme. If you enlarge such a window, the text expands to fill the available space, allowing you to see more information, including information that had scrolled off the screen. This is the default refresh type for Release 2 Shell windows if no option is specified.
SMART	Choses the smart window refresh scheme. If you enlarge this kind of window, existing text is not redrawn. This is the kind of CLI window used by AmigaDOS 1.3 and earlier.
WAIT	The window does not close automatically when the program that created it terminates. Rather, it waits until its close gadget is selected (if it has one), or the user enters the CTRL-\ key combination.

By using these options, you can create a more useful list window with CON:

```
LIST TO CON:0/0/640/100/ListWindow/CLOSE/WAIT
```

This window differs from the previous one in two important ways. First, if this window is too short to display all of the information, you can see more by increasing its size. Second, it doesn't disappear when the LIST program ends, but rather waits until you click on the close gadget.

Console as Input

The console window can also be used as an input device. In this role, it can act as a mini text editor, which can be used to create small text files or printed documents. For example, you can create a text file on the RAM: disk by typing

```
COPY "CON:40/40/200/100/File Creator" TO RAM:text
```

The new console window appears and is the active window. Start typing text, using the BACKSPACE key to delete errors. When you've finished a line, press the Return key and that line is sent to the file. When you're done, enter a CTRL-\ character to signal AmigaDOS that you are at the end of the file. Enter this character by holding down the CTRL key and pressing the backslash (\) key,

located next to the left of the BACKSPACE key. When you end the file, the window disappears and the disk file is closed. To see the contents of that file, enter

```
TYPE RAM:text
```

The console device gives you a handy way to create a small file (like the command sequence files discussed later). You can also send input from a console device to any other device (even another console window). For example, type

```
COPY CON:40/40/200/100/Typewriter TO PRT:
```

and each line that you type in the window is sent to the system printer (as soon as you press Return). Again, use CTRL-\ to end the session.

In addition to the new console windows which you create, you can also use the existing console windows belonging to your CLIs. You do this by referring to the active console device, named * (asterisk). This use of the asterisk should not be confused with the universal wildcard character used by MS-DOS (and optionally by AmigaDOS Release 2) or the asterisk used as an escape character before quotation marks in a filename. As an output device, * is more durable than CON: since the window doesn't vanish after each command. Unfortunately, it's not much more useful, since most commands output to the current console window anyway. It, too, can be used as an input device, and as such, it's even handier to type

```
COPY * TO textfile
```

than specifying a long CON: device name. This is a quick and easy way to create a short text file.

RAW

There's one more window device available to AmigaDOS, but it's really only suitable for application programs and not for general use by the CLI command programs. This device is called *RAW:*, and it's an apt name. A normal console window heavily filters what comes through it. You'll notice, for example, that the cursor keys have no effect when you're typing in a console window. The RAW: device, on the other hand, doesn't filter anything. Thus, it would be nice to use if you wanted to create a file which contained characters other than the standard letters, numbers, and punctuation marks—such as cursor movement codes. But, alas, RAW: passes through the CTRL-\ without interpreting it as an end-of-file character. So while a CON: disappears before you're through with it,

there's no way to close a RAW: window from CLI and therefore no way to close the file to which it's writing (not even the Release 2 close gadget works). If you really want to play with RAW:, remember that once you create the window, the only way to get rid of it is terminate the command that created it with the CTRL-C key combination, or to warm start the computer by pressing the CTRL-Amiga-Amiga key combination. A fairly safe experiment for the incurably curious is to type

```
COPY RAW:0/0/100/50/Input TO RAW:0/50/640/100/Output
```

Click in the Input window to activate it and start typing. Everything you type shows up in the Output window, including cursor movement keys. You can now warm start the computer, secure in the knowledge that you've tried everything at least once and that RAW: is as useless for ordinary purposes as we say it is.

It's NIL

Speaking of useless, the last device to investigate does absolutely nothing. True to the British origins of AmigaDOS, it's called *NIL:*. When used as an input device, NIL: just sends the end-of-file character. When used as an output device, NIL: accepts the output, and *does nothing with it.* Still, it's not as useless as it may seem at first. Programmers sometimes have a use for such devices in testing I/O routines. And even for the casual user, there are occasions where it's useful to get rid of output without showing it to anybody. For example, if you examine the command file called *startup-sequence* in the *s* directory, which is normally used to load and run the Workbench, you'll find that the last line of the file reads *endcli > nil:*. (You can look at this file by warm starting the Amiga, then putting the Workbench disk in the drive [the Workbench disk, *not* the CLI disk you've probably created], opening the System drawer, double-clicking on the CLI icon, and typing `TYPE s/startup-sequence`.)

The ENDCLI command usually prints the message *CLI task n ending* or *Process n ending* (where *n* is the task number), just before the window disappears. Apparently, the developers didn't want that message to print when the Workbench loaded and so used output redirection (which is discussed at the end of this chapter) to send the offending character string to limbo. Suppressing output from commands becomes more important under Release 2, where the first part of the startup-sequence is known as the "silent startup-sequence." If any of the commands outputs text before the IPrefs program sets the screen and font preferences, those preferences won't be put into immediate effect. Another

practical example of using the NIL: device is shown in Chapter 5, which deals with command files, where the output from DATE ? is sent to NIL: as a way of allowing you to enter the date without seeing the command template as a prompt.

In versions 1.3 and above, NIL: performs another useful role. Normally, a program that is RUN from a CLI depends on that CLI console window for its standard input and output streams. That means that unless the program takes special pains to detach itself from the CLI console, you won't be able to close that window as long as the program is running. Clicking on the close gadget will produce the *CLI task n ending* message, but the window won't close. This problem is most apparent with programs that are run from the initial CLI, which ordinarily is closed when the Workbench is opened. It can often be avoided by using the redirection operators to specify NIL: as the default input and output device:

```
RUN <>NIL: Program
```

Logical Devices

In addition to physical devices like the disk drive and printer, AmigaDOS also supports a variety of pseudodevices known as *logical devices.* Logical devices provide a way of giving a short device-like name (ending in a colon) to a particular disk directory. For example, if you assign the logical device name *let:* to the directory *df0:Wordprocessing/personal/letters,* you could refer to a file in that directory as *let:AuntMartha* rather than as *df0:Wordprocessing/personal/letters/AuntMartha.* This makes it easier to shorten the reference to a directory, without having to make that directory the current one. Logical devices also allow a program to have access to a file without knowing its exact physical pathname. For example, a word processing program may need to read a dictionary file in order to perform spell checking functions. It can assume that the dictionary is in the current directory, but that makes it difficult to store the dictionary in RAM: for faster access, or to place it in a directory of your choosing on your hard drive. If the program looks for the dictionary in a logical device called DT:, however, you can store the file wherever you want. By assigning the logical device name DT: to whatever directory you use to store the file, you can make it accessible to the word processing program, regardless of its actual physical location.

You can use the ASSIGN command program to assign logical devices to directories. When used for this purpose, the command format is ASSIGN

```
Volumes:
Extras [Mounted]
Workbench1.3 [Mounted]
Directories:
CLIPS       RAM DISK:clipboards
ENV         RAM DISK:env
T           RAM DISK:t
S           Workbench1.3:s
L           Workbench1.3:l
C           Workbench1.3:c
FONTS       Workbench1.3:fonts
DEVS        Workbench1.3:devs
LIBS        Workbench1.3:libs
SYS         Workbench1.3:
Devices:
PIPE    AUX     SPEAK    NEWCON    DF1
DF0     PRT     PAR      SER       RAW
CON     RAM
```

devicename directory. The assignment given in the first example above could be accomplished by the command

```
ASSIGN let: df0:Wordprocessing/person/AuntMartha
```

Assigning for Itself

User-created logical devices are not the only kind found on the Amiga. AmigaDOS itself uses these devices to solve the problem of having too much system software to fit in the Kickstart ROM. Since some of the system software is used only occasionally, it is automatically loaded from disk whenever necessary. The most obvious example of this process is the CLI commands themselves, which all reside on disk and must be loaded before they can be used. The handlers for the RAM: disk, the parallel, serial, and printer devices, all must be brought in from disk. As you'll soon see, the list of disk files which contain information significant to the operating system is quite long. AmigaDOS recognizes that it would be foolish to assume that each of these files is always in the

current directory. Therefore, it uses logical devices as a means of providing an alternative place to search for these important files. When you start up the Amiga, DOS assigns a number of logical device names to certain directories. When DOS needs to find one of the system files, it first looks in the current directory, but if it doesn't find the file there, it searches one of the logical devices.

To see a list of the logical devices which DOS creates, use the ASSIGN command name by itself. This command program displays a list of all logical devices, both the ones assigned by the system and those assigned by you. If you've not assigned any logical devices, a typical display produced by ASSIGN is shown on page 182 (assuming a dual-drive system and disks in the drives with volume names of *Extras* and *Workbench1.3*):

S:

There are seven directories to which DOS assigns logical device names (and at least three more to which names are assigned by the startup-sequence file described below). S:, the first logical device, is a directory used to hold command sequence files (batch files). When the EXECUTE command is told to execute a sequence file, it first looks for the sequence file in the current directory. If it doesn't find the file, it tries the directory to which the logical device name *S:* has been assigned. If the boot disk contains a directory called S, AmigaDOS automatically assigns it the logical device name S: at startup time.

The Workbench disk contains a file called *startup-sequence* in its S: directory. This sequence file is automatically loaded and run when the Workbench disk is inserted, and it in turn loads the Workbench program and runs it. If the Workbench 1.3 Shell has been installed, script files can be run directly, without using the EXECUTE command, if the *s* bit of the file protection flag is set. When you run them that way, however, AmigaDOS searches for them as if they were commands, and not scripts. See the description of the logical device named C:, below, for more information on command searches.

L:

If the boot disk contains an L subdirectory in its root, AmigaDOS automatically assigns it the L: logical device name. This is where AmigaDOS looks for handler software that controls communication with various hardware and software devices, such as the *Ram-Handler* file which controls the RAM: device. The most necessary of these is the *disk-Validator*, which is used to check if disks are

in the proper AmigaDOS format. Many of the MOUNTable devices, such as NEWCON: and SPEAK:, have their device handlers stored in the L: directory also.

C:

The command device C: is assigned to the C directory on the boot disk, or to the root directory if no C directory exists. This is one of the most significant logical devices, especially to CLI users. Whenever you issue a command to the CLI, DOS first looks in the current directory for a filename matching the first word of the command line. If it doesn't find the command in the current directory, it then searches the C: device directory. Although C: is in the default search path for commands, you can extend this search path with the PATH command. For example, the command *PATH RAM: Sys:System ADD* will cause the two directories named to be searched for commands after the current directory, and before the C: directory.

Since the C: directory is always in the command search path, if you don't keep the disk that contains it in one of your drives, you may be in for a lot of disk swapping. Every time you issue a CLI command not found in the current directory, you'll be prompted to insert the volume which contains the C: directory. One way around this dilemma (if you have sufficient RAM) is to transfer the command files to the RAM: disk and assign the C: device to it. The easiest way to do this is

```
COPY C: RAM:
ASSIGN C: RAM:
```

This copies all the command files to the root directory of the RAM: device. If you'll be using the RAM: device for other files as well, you may wish to create a *c* subdirectory first, move the files to this directory, and then assign C: to it with

```
MAKEDIR RAM:c
COPY C: RAM:c
ASSIGN C: RAM:c
```

You may find it convenient to place this sequence of commands in the batch file *startup-sequence* on your boot disk. (Remember that this file automatically executes every time you turn the computer on.) Notice, however, that there are 64 command files in the *c* directory of the Workbench 1.3 disk. If you copy

all of them, the RAM: disk takes up well over 240K of memory. That's most of the free memory available on a 512 Amiga system.

This doesn't mean that you can't have AmigaDOS search for commands in RAM: if you only have 512K in your Amiga. It just means that you'll have to be a little more selective. Move only the most frequently used command files, like COPY, DELETE, DIR, and LIST to RAM:, and use the PATH command to add RAM: to the search path, so that AmigaDOS will look there before assigning the C: device name. This way, you can create a custom-tailored list of intrinsic commands which are always available. A sample script to accomplish this task would look like this:

```
COPY c:(list|dir|delete|copy) ram:C
PATH ram:c ADD
```

Users of version 1.3 or higher may save memory by making a subset of CLI commands RAM-resident, using the s. This command only works in the Shell, so make sure that you're working from a Shell window, and not a normal CLI window. To make the DIR command resident, you'd type `RESIDENT DIR`. This command loads the DIR program into memory, and keeps it there. From then on, whenever you give the DIR command, the Shell executes that command from the copy that is already in memory. This eliminates the need to read the command in from disk, eliminates the duplication that occurs when you keep a command in the RAM disk (the "disk" copy in RAM: is loaded into memory a second time in order for the program to execute), and allows command programs to run instantly.

There are, of course, some drawbacks to making commands resident. First, just as with storing commands on RAM:, each command that you make resident reduces the amount of free memory that you have. Second, not every program can be made resident. Each time the program is executed, it is run from the same copy that was initially loaded into memory. Therefore, only programs that are re-executable (can be run a second time without being unloaded and loaded again) and re-entrant (can be executed from separate Shell windows at the same time) qualify as resident programs. Because of this, the RESIDENT command will only load programs with the *P* bit set on the their file protection flag. You will find that most of the CLI commands in the C directory of the Workbench disk have the pure bit set, and can be made resident.

FONTS:

This device is usually assigned to the FONTS directory of the boot disk, and contains the files for the various text fonts for the Amiga. These are the disk-loaded fonts that can be used from programs like Notepad, painting programs, and desktop publishing programs. When a program calls the system routine *OpenFonts,* which must be done whenever a new font is used, the operating system tries to find the new font in this directory if it's not already loaded into memory.

Release 2 of AmigaDOS adds support for the Compugraphic system of outline fonts, which can be scaled to any size on the screen. This font system requires Compugraphic font information to be copied to the _Bullet and _Bullet_Outlines directories of FONTS:, and a new diskfont.library file to be copied to the LIBS: directory. Because of the rather large size of these files, outline fonts are practical only on a system that includes a hard drive and lots of memory. The outline font installation procedure is performed automatically by the AmigaDOS Release 2 hard drive installation software. Release 2 also includes support for Colorfonts, which use multiple colors to give a "textured" look to the type face.

DEVS:

The DEVS: name is normally assigned to the DEVS directory of the boot disk. It contains device drivers for of the devices we've already discussed—the serial device, the parallel device, and the printer device, as well as the recoverable RAM disk device that we'll be covering a bit later. It also contains drivers for devices which the CLI commands do not use directly, like the narrator (speech synthesizer) and the clipboard. The first time that a program tries to open a device using the system routine *OpenDevice,* AmigaDOS looks in the DEVS: device directory for the device driver if it's not already loaded. In addition to device drivers, DEVS: also contains the *system-configuration* file containing the 1.3 preference settings, the *Mountlist* file used by the MOUNT command, and the printer drivers for the various printers supported by the system (these latter files are within the *printers* subdirectory).

LIBS:

The logical device where system library files can be found, LIBS:, is usually assigned to the LIBS directory of the boot disk. System library files hold the "extra" operating system function libraries that don't fit in the Kickstart ROM.

AmigaDOS Logical Device Assignments

Device Name	Assignment
S	Assigned to directory *s* of the boot disk. AmigaDOS looks for sequence files to EXECUTE here if not found in the current directory. Some application programs store their configuration files here.
L	Assigned to directory *l* of the boot disk. AmigaDOS looks here for its own extensions, like the *Ram-Handler, if* they're not already loaded. Some expansion devices (typically software devices) require their device handlers to be stored here.
C	Assigned to directory *c* of the boot disk. AmigaDOS looks for CLI command files here if not found in the current directory.
FONTS	Assigned to directory *fonts* of the boot disk. The *OpenFonts* operating system routine looks here for fonts if they're not already loaded. Some application programs require their own fonts to be stored here.
DEVS	Assigned to directory *devs* of the boot disk. The *OpenDevice* operating system routine looks here for device drivers if they're not already loaded. Some expansion devices (typically hardware devices) require their device drivers to be stored here.
LIBS	Assigned to directory *libs* of the boot disk. The *OpenLibrary* operating system routine looks here for system library extensions if not already loaded. Some application programs require their own shared library files to be stored here.
SYS	Assigned to the root directory of the boot disk. Can be used as a short way of referring to the system disk.

These include the function libraries which support features such as text-to-speech conversion (the *translator.library* file), disk-loaded text fonts (*diskfont.library*), and floating-point math calculations (*mathtrans.library, mathieeedoubbas.library, etc.*). Whenever a program calls the *OpenLibrary*

routine, the operating system looks to this device for the library file, if the library is not already resident. Although the system software libraries are the ones used most frequently, some applications programs require their own shared library files to be installed here.

SYS:

The final assignment which DOS makes is the SYS: device. This is assigned to the root directory of the disk which was used to boot up the system. Since it's a reasonable assumption that you'll use a disk which includes all the system files when you boot up, it gives you a handy way of referring to that system disk. In the example above, after you'd transferred only some of the CLI command files, and then assigned C: to RAM:, you used the volume name of the Workbench disk to access a command located in *Workbench/c*. You could also have specified the command directory as *SYS:c,* without having to know the volume name of the boot disk. Even if you did know the volume name of the boot disk, the device name of SYS: is shorter. The assignment of SYS: has some impact on the Workbench, since the program that is run when one disk icon is dragged over the other has the default path *SYS:System/Diskcopy*.

The logical device assignments made by AmigaDOS are summarized on page 72.

CLIPS:, ENV:, and T:

In addition to the logical device names that are created by the system at power-up time, there are some other logical device names that are assigned by the the startup-sequence script to various directories which that script creates on the RAM: device. They are CLIPS:, which is assigned to RAM:Clipboards, T:, which is assigned to RAM:T, and ENV:, which is assigned to RAM:Env. The CLIPS: directory is created for use by the Clipboard, a low-level device that is used by programs to exchange information with one another, and not directly by the end user. The Clipboard wasn't used much by programmers before its functions were expanded in Release 2, but Release 2 uses it to provide "copy-and-paste" operations in console windows.

ENV: is another logical device whose role increased in Release 2. Previously, it was used primarily by script files (see Chapter 5) as a means of passing data between script commands. In Release 2, ENV: started to be used more as an "environment" device, where information that might be used by several pro-

grams could be stored. One example is Release 2 preference files, which are stored in the ENV:sys drawer. You can find more information about environment variables in the Reference section under the SET and SETENV commands.

T: is used by a few programs for temporary storage. For example, the system screen editor, ED, stores a backup of the original text file that's being edited in this directory.

New ASSIGN Options for Release 2

Several important new options were added to the ASSIGN command in AmigaDOS Release 2. The first of these is the ability to assign several directories to one logical device name. In the past, only the command path could be assigned to several different directories via the PATH command (which in effect expanded on the function of the C: assignment). In Release 2 and 3, however, you can assign multiple directories to any logical device simply by listing the directories in the ASSIGN command:

```
ASSIGN FONTS: sys:fonts sys:colorfonts sys:videofonts
    sys:DTPfonts
```

This capability is quite handy for hard drive owners, who may, as the above example indicates, accumulate so many fonts that they may wish to store them in separate subdirectories. AmigaDOS Release 2 also gives you the capability to add directories to an existing logical device assignment, using the ADD option:

```
ASSIGN FONTS: sys:colorfonts sys:videofonts ADD
```

Another option that Release 2 owners have is to defer AmigaDOS' search for the directory that is being assigned. Normally, when you try to assign a logical device name to a directory, AmigaDOS immediately checks to see if the volume is mounted and the directory is present. If the volume is not mounted, it prompts the user to mount it. If the volume is present but the subdirectory does not exist, the assignment fails immediately. With Release 2, you can defer this verification process until a program actually tries to access the logical device, by using the DEFER option:

```
ASSIGN Dictionary: WordPro/Dictionary DEFER
```

Using this example, you could make the necessary assignment to run your WordPro program from the startup-sequence file on your Workbench, without having the WordPro disk actually mounted in the drive.

Like the DEFER option, the PATH option does not look for the assigned volume until the logical device name is used. PATH goes one step further,

however, by re-evaluating the assignment every time a program tries to use a file on that logical device. For example, the command `ASSIGN C: df0:c` will assign the C: device name to the c directory of whatever disk happens to be in the internal drive when you make the assignment. The command `ASSIGN C: df0:c DEFER` will assign the c directory of whatever disk is in the drive when the C: device is next used (typically, the next time that you type the name of a command file that isn't in the current directory). The command `ASSIGN C: df0:c PATH`, however, looks for df0:c each time a program tries to use a file on the C: device. That means as long as the disk in the internal drive has a command directory on it, AmigaDOS will never ask you to insert the Workbench disk if it needs to find a command. The one drawback to the PATH option is that it cannot be used as part of a multiple assignment for a single device. In the case of C:, however, you can use the PATH command to assign other directories to the search path.

File Assignments

Not only can you ASSIGN a device name to volumes and directories, but you can also ASSIGN a device name to program files. This allows you to create short "aliases" for program names, if you are using a version prior to 1.3 that does not include a Shell. While names like *EXECUTE* and *DELETE* may not seem so hard to type, it's more convenient to type names like *X:* and *D:*. If you `ASSIGN X: EXECUTE:`, then you can type

```
X: program
```

instead of

```
EXECUTE program
```

It may seem like a small savings in keystrokes, but time saved by ASSIGNs and Shell ALIASes can add up (particularly if you're not a crack typist). If you place these ASSIGN statements in the *s/startup-sequence* file, the logical device names which you ASSIGN will always be available to you.

Notes

Before leaving the subject of logical devices, there are a few final points to note:

- The logical device assignments apply to *all* CLIs, regardless of which was used to make the assignment (as opposed to Shell ALIASes, which apply only to one Shell).

- The ASSIGN command can be used to *remove* an assignment. The form for this is ***ASSIGN*** *devicename*. Note that it's possible to delete the assignments which the system makes. ***ASSIGN C:***, for example, removes the command directory assignment so that commands must be located in the current directory in order to be executed. Obviously, some caution should be exercised in removing the assignments that AmigaDOS has made.
- It's not possible to use the ASSIGN command to change the names of physical devices like SER:.
- You cannot delete a file or directory to which a logical device name has been ASSIGNED without first removing the assignment.

MOUNTable Device Drivers and Handlers

One of the most important features added to the 1.2 version of the Workbench was the MOUNT command. The MOUNT command allows almost any type of hardware or software device imaginable to be added to the system by reading information that describes the device and its driver or handler software from a text file in the DEVS: directory called Mountlist (see the MOUNT entry in the reference section for more information on the Mountlist file). Mountable device drivers can allow AmigaDOS to communicate with external hardware devices like a hard drive. "Devices" may also consist merely of software drivers that use existing system resources in a new way, as with the RAM disk device.

The 1.2 Workbench contained only one Mountlist entry for adding the A1020 5¼-inch drive as an AmigaDOS device. Workbench 1.3 added many more new standard devices to the system.

RAD:

Perhaps the most interesting of the new devices is one called RAD:. RAD: is a RAM disk device, which, like the familiar RAM: device, uses part of the computer's working memory as an electronic disk drive. RAD: is an additional device, not a replacement for RAM:, as there are many important differences between the two.

RAM: is an integral part of the AmigaDOS system; you create the RAM: device merely by referring to it in an AmigaDOS command. RAD:, on the other hand, must be added to the system with the MOUNT command, and its device driver, which is contained in a file called RAMDRIVE.DEVICE, must be present in the DEVS: directory.

RAM: automatically adjusts its size according to its contents. As you add more files to it, it grows in size, and that size is limited only by the amount of memory that's available. RAD: has a fixed size, which is determined by the Mountlist entry used to MOUNT it.

Because it is of a fixed size, RAD: acts more like the Amiga's floppy disks than its RAM disk. Like the standard 3½-inch floppies, RAD: is set up as a double-sided drive, with 512 bytes per sector, and 11 sectors per track. This means that each track (or cylinder as it is sometimes called) uses 11K of memory (512 * 2 * 11). The number of tracks used for RAD: is determined by the LowCyl and HighCyl entries in the mountlist. An entry of `LowCyl = 0 ; HighCyl = 21`, for example, allocates 22 tracks, at 11K per track, for a total memory usage of 242K. This is enough to store all of the files in the C directory on the 1.3 Workbench disk. If you have a couple of megabytes of fast memory on your Amiga, you might even set the HighCyl value to 79, for an 880K RAD: drive, the exact same size and layout as a floppy disk. When the RAD: drive is the same size as a floppy, it's possible to use the Diskcopy command to copy an entire floppy to RAD:, or vice versa. It's even possible to format the RAD: drive, something that can't be done with RAM:.

The most important difference between RAD: and RAM:, however, is their duration. Since both use the computer's memory to simulate disk storage, the contents of both devices is lost when the computer is turned off. A warm boot situation is a different story, however. While RAM: loses its contents whenever you press the Ctrl-Amiga-Amiga key combination, or your computer takes a trip to the Guru, RAD: is a recoverable RAM: disk. As long as the error which caused the Guru Meditation didn't scramble the contents of memory as well, with the 1.2 Kickstart ROM you can MOUNT RAD: again, and it will re-appear with its contents intact. If you are using version 1.3 or higher of Kickstart, you can not only recover the contents of the RAD: drive after rebooting, but you can even reboot from the RAD: device itself, so long as you've assigned it a boot priority higher than any other devices that are present (such as a hard drive). Even if a boot disk is used for a warm start, Kickstart 1.3 automatically restores RAD: upon warm boot, with no need to re-MOUNT it. Sometimes, RAD: can be a little too persistent. Since it doesn't disappear when you warm boot, like RAM: does, the only way to remove it short of turning off the power is to use the REMRAD command. This deletes the contents of RAD:, shrinks it down to the smallest possible size, and makes it non-recoverable, so that it disappears at the next warm boot.

Since RAD: is a MOUNTable device which can be formatted, 1.3 users can format it for the Fast File System (FFS) by adding two items to the RAD: entry in the DEVS:MOUNTLIST file. These two lines are:

```
GlobVec = -1
    FileSystem = l:FastFileSystem
```

These lines can be added anywhere after RAD: and before the # which ends the entry. In addition, the FastFileSystem file mentioned in the second line must appear in the l: directory on your Workbench disk. After you've changed the Mountlist entry, you may use the command MOUNT RAD: to mount the drive. Because you're using a different file system than the default one, you must format the drive before you use it. You can, however, use the new QUICK option of the FORMAT command, which shortens the process:

```
:SYSTEM/FORMAT drive RAD: name Speedy QUICK
```

Using the FFS on the recoverable RAM disk does speed up operations somewhat, though since it is a RAM drive, those operations are fairly quick without it. But in versions prior to Release 2, you lose the ability to reboot from the RAM drive if you format it with the FFS (you should add the line `BootPri = -129 to the RAD:` entry in the Mountlist file to tell AmigaDOS not to try rebooting from this device). In fact, with the 1.3 ROM, you can't even recover the contents of the RAM drive when you reboot if you have it formatted with the FFS, since the drive is automatically mounted on warmstart as a normal DOS filesystem device. Since Kickstart is expecting the RAM drive to be in the old AmigaDOS format, it thinks that it is not a DOS disk. Using the 1.2 Kickstart, however, you must MOUNT the drive again after a warmstart, and so even though you can't reboot from the RAM drive, you can recover its contents even if it is formatted with FFS.

AmigaDOS Release 2 allows you to create multiple recoverable RAM disks at a time. To create a second unit, edit the DEVS:Mountlist file with a text editor such as ED (see Chapter 6). First, make a duplicate copy of the RAD: entry (everything from the name RAD: to the first #). Then, change the name to something like RAD1:, the unit number (in the line Unit =) to 1 instead of zero, and the HighCyl to the appropriate number for the size you have selected. Now, you can MOUNT RAD1: after you have MOUNTed RAD:. To remove the new unit, use the unit number after the REMRAD command, like `REMRAD 1`.

PIPE:

Next on the list of device handlers added by Workbench 1.3 is one called PIPE:, which is mounted during the standard startup-sequence. This device emulates the pipes feature of UNIX and MS-DOS, which allows the user to transfer the output of one program directly to the input of another. Let's say, for example, that you wish to display a large disk directory on screen. The DIR command may not be suitable, since it outputs filenames in a continuous stream, and doesn't pause when the screen fills up. By piping the output of DIR to the MORE program, which displays text a screen full at a time, you get the information you want, in the format you prefer. In UNIX or MS-DOS, this can be accomplished with a command like DIR | MORE. Unfortunately, the Amiga command shell doesn't recognize the | operator, so it's necessary to simulate pipes to achieve the same result.

Amiga owners have always been able to get similar results by using file redirection to a temporary storage area on the RAM: disk. In the case of the example discussed above, you could use the command sequence

```
DIR >ram:temp
MORE ram:temp
DELETE ram:temp
```

to achieve the same result. Still, there are instances in which you may not have enough room on the RAM: disk to create the intermediate file. For example, if you create a hex dump of a 25K program file using the TYPE OPT H command, you may end up with a text file that is 100K or longer. In such a case, you may wish to MOUNT PIPE:, whose handler is found in the file l:pipe-handler.

The PIPE: devices acts as a conduit, directing the output of one program to the input of another. One process writes to the pipe, assigning it an arbitrary file name (such as pipe:temp). Each pipe name uses a 4K buffer, which means only that much may be written to the pipe before the writing process is blocked. When the second program reads that 4K buffer (by accessing the same file name as was written), the first program can write 4K more of data, until all of the output is transferred. In the example above, you could pipe the output of DIR to more using the command sequence

```
RUN DIR >pipe:temp
MORE pipe:temp
```

assuming, of course, that you had first used the MOUNT PIPE: command (the normal Startup-Sequence file on Workbench 1.3 mounts this device automatically). Note that in the above example we used RUN to spin off a separate process for DIR. Both commands can't use the same CLI process because if the directory output is larger than 4K, DIR won't terminate and give back the CLI prompt until MORE has read all of its output.

The roundabout method that PIPE: uses to simulate pipes may not be as simple as that available on other systems, but it does have some unique advantages. In addition to the traditional pipe situation described above, PIPE: can be used for its buffering capabilities alone. Many terminal programs, for example, download files in a synchronous fashion. This means that they receive a block of data, send it to the disk, wait until the disk write is finished, and then ask to receive the next block. Each intermediate disk write causes a slight delay in the transmission. You can always avoid this delay by downloading to a file in RAM:, but in doing so you run the risk of filling up the RAM disk before the file transfer is completed, or of forgetting to copy the file to a floppy before turning off your computer. A better solution is to use the command:

```
COPY pipe:temp TO df0:downfile
```

before running your terminal program, and then downloading to the file pipe:temp. That way, large amounts of data are buffered before any writes actually take place, which means fewer delays. At the same time, you avoid the risks associated with downloading to RAM:, since when the download process is terminated, your file is stored safely on disk.

AUX:

AUX:, another device handler introduced in AmigaDOS 1.3, is also mounted during the startup-sequence. The AUX: device, whose handler is located in the file L:Aux-Handler, transfers data through the serial port, much like the SER: device. While SER: buffers its output, however, sending it out only after a 512-byte block has accumulated, AUX: provides unbuffered communication with the serial port. The main use for an unbuffered serial device is to create a CLI window that uses the serial port for its input and output. Such a window can be set up by MOUNTING the AUX: device (this is normally done automatically by the Startup-Sequence file), and typing the command:

```
NEWCLI AUX:
```

This procedure lets you hook up another computer or terminal to the Amiga, and give AmigaDOS commands from that machine over the serial port,

or even over a modem. While you can not, of course, run Intuition-based windowing programs on your remote terminal (at least not yet), you can use CLI commands like DIR and INFO to gain information about the Amiga disks, and the TYPE command to send files to the remote screen (where they can be captured to a buffer file). There is even a public domain program called CANCEL! which automatically hits the Cancel button whenever a system requester pops up. While serial-port CLIs don't exactly make the Amiga a multi-user system, they do come pretty close to it.

SPEAK:

In an effort to make the built-in speech synthesis feature of the Amiga even more accessible, Commodore has added the SPEAK: device handler to the 1.3 Workbench. SPEAK: is similar to the SAY program in the Utilities drawer, in that it converts text input into speech which is output through the audio channels. Like SAY, it uses the translator.library file from the LIBS: directory to convert the text to phonemes, and the narrator.device from DEVS: to output the phonetic speech. While SAY takes input only from the keyboard, however, SPEAK: is mounted as a device, which means that it can take its input from any source that can write to a disk file. For example, you can COPY a file to SPEAK: from the CLI, save a file to SPEAK: from a text editor or word processing program like MicroEMACS or Notepad, or even open SPEAK: as a capture file for a terminal program, so that text is spoken as it is received from the modem.

As with the SAY program, it is possible to adjust SPEAK: in order to vary the spoken output. You may change the pitch and speed of the speech, choose male or female voice characteristics, and select natural or robot (monotone) speech inflection. To add a voice setting, you include it as part of a SPEAK:opt/ path name when you access the device. For example, to listen to a file with a female voice at a pitch setting of 200, you could use the command:

```
COPY filename to SPEAK:opt/f/p200
```

The full list of voice options which can be added to the SPEAK:opt/ pathname is shown below:

P###	Set Pitch. (### is a number from 65 to 320)
S###	Set Speed. (### is a number from 30 to 400)
M	Use male voice characteristics
F	User female voice characteristics
R	Robot speech (uninflected monotone)

N	Natural speech (natural inflection)
O0	Do not allow option settings in input stream
O1	Allow option settings in input stream
A0	Turn off phoneme input mode
A1	Turn on phoneme input mode
D0	Determine sentence breaks by punctuation alone
D1	Determine sentence breaks from Carriage Return and Line Feed, as well as punctuation

When option settings are allowed in the input stream, you can change the voice characteristics with commands given in the data that is sent to the device. For example, if you give the command

```
COPY * to Speak:opt/O1
```

and then type `This is a test`, the phrase is pronounced in the default voice. If you then type the lines

```
opt/f/p200
This is a test
```

the phrase is pronounced in a high female voice. When option settings are not allowed in the input stream (the default), the options settings are read aloud (Opt slash eff slash pea two zero zero).

NEWCON:

One of the major complaints that users always had about the CLI command environment was that it didn't support command line editing. If you made a typing mistake in the first word of a command line, you had to erase the whole line and start over again. A device that was added in 1.3 called NEWCON: finally provided a console window that not only allowed editing with the cursor keys, but also added a 2K command history buffer. After mounting the NEWCON: device, (whose handler is located in the file L:newcon-handler), 1.3 users may open a CLI window that uses this new console device with the command:

```
NEWCLI NEWCON:x/y/w/h/name
```

where *x* and *y* specify the position of the upper left corner of the window, *w* and *h* specify its width and height in pixels, and *name* designates an optional window name. With a NEWCON: window, you can edit a command line by using the left and right arrow keys to move the cursor back and forth across the line one space

at a time, or in combination with the Shift key to take you to the beginning or end of the line. The up and down arrow keys implement a command history feature. Each time you enter a command line, that line is stored in a 2K circular buffer. Pressing the up arrow key retrieves the previous entry in the buffer, which appears at the command prompt. Pressing the down arrow key moves you forward through the buffer (Shift-down arrow takes you to the bottom of the buffer). If you don't want to cursor through each previous command, you can use the command history's search feature. Typing a partial command line, and then pressing Shift-cursor up, initiates a search for the last command line that matches the partial string.

With Release 2, all of the features of the NEWCON: device (and then some) were integrated into the standard CON: device. This means that all console windows in Release 2 have the advanced editing capabilities of NEWCON:, without having to load an additional handler.

Redirection of Input and Output

Ordinarily, AmigaDOS accepts input from the keyboard and outputs it to the current console window. These are known as the *standard input* and *standard output* devices. In some cases, you may redirect input to a program so that it comes from a device other than the console keyboard, and you may redirect output from the program so that it goes to a device other than the console display. Redirection of command input/output (I/O) is accomplished through the use of the redirection operators < and > (the angle brackets—you may be more familiar with them as the *less than* and *greater than* signs—which are entered by pressing a SHIFTed comma and SHIFTed period, respectively). The left angle bracket (<) is used to redirect input, and the right angle bracket (>), to redirect output. You can easily remember which is which, because the direction in which the angle bracket is pointing indicates the direction in which the information is going (*from* is left, *to* is right).

You use a redirection operator, followed by the name of the device or file which you wish to use for input or output, directly after the command name. For example, if you wish to send a directory listing to the printer, you could type

```
DIR >PRT:
```

You can use one redirection operator or both for a particular command, but in versions prior to 2.04, the operator(s) must come right after the command name, not after the command parameters:

```
DIR >PRT: OPT A
```

is correct, but

```
DIR OPT A >PRT:
```

is incorrect because DIR will interpret >PRT: as the root directory of a volume named >PRT. In versions 2.04 and higher, however, even this latter format is acceptable.

Note that several commands, such as COPY and LIST, allow you to specify a destination device to which output is directed. Therefore, you don't have to use the redirection operator to specify the output for those commands.

Redirection of input is a little trickier than redirection of output, since the CLI commands generally take all their input directly from the command line rather than waiting for someone to type it in at the keyboard. One way of getting around this is to use the question mark (?) as a command parameter. When you put a question mark after the command name as its only parameter, AmigaDOS prints out a command template and waits for you to enter the command parameters. For example, if you first redirected the output of ECHO to a file named *textfile*:

```
ECHO >textfile "*"This is a test*""
```

you'd end up with a one-line text file which starts and ends with double quotation marks. Then, you could use ECHO to print the contents of the file by typing

```
ECHO <textfile ?
```

ECHO first prints out a colon (:)—its command template—then gets the input to print from *textfile*. Notice that this works only with short files, since ECHO can only take a character string shorter than 256 characters. Another handy use for input redirection is with commands that require a carriage return to continue. If you wish to use the DISKCOPY command to automatically copy your boot disk to the recoverable RAM disk on startup, you'll find that DISKCOPY prompts you to hit Return to start the copy. To avoid having to manually enter a Return (and thus defeating the automatic nature of the script), you could use the command:

```
DISKCOPY <NIL: DF0: to RAD:
```

The input from NIL: will satisfy DISKCOPY's desire for a carriage return.

Redirection applies only to the command in which the operators are used. Subsequent CLI commands will use standard input and output.

The 1.3 Workbench Shell provides an additional output redirection operation, which uses two angle brackets (>>) instead of one. If you use the standard operator to direct output to a file that already exists, the existing file will be replaced by a file consisting of the new output. If you use the new redirection operator on an existing file, however, the new material will be added to the end of the current file (appended). If you try to use the append redirection operator on a file that doesn't exist, the command will fail under version 1.3. Under Release 2, however, AmigaDOS will create the file if it doesn't already exist.

Chapter 5

Command Sequence Files

Running individual command programs from the CLI is easy—you just enter the name of the command. You may find, however, that some tasks require you to enter several CLI commands, one after the other. You may find that you even use a particular sequence of commands again and again. AmigaDOS offers a way to simplify this process—it allows you to enter each of the commands into a text file and to use the EXECUTE command to execute that sequence of commands whenever you want. Such a text file is known as a sequence file, a batch file, or a script file. When EXECUTEd, a batch file will run a sequence of commands, one after the other, just as if it had been typed at the console keyboard. The sequence will continue to run until all of the commands have been carried out, or until a command fails, or the user manually breaks out of the sequence by pressing the CTRL-D key combination.

Sequence files can do more than just execute a fixed series of commands. There are provisions for testing certain conditions and for issuing alternative series of commands depending on the outcome of those tests. They also allow you to substitute text within the command file so that the commands operate with options you specify in the EXECUTE command, not just with a fixed set. Finally, the special command file named *startup-sequence* (or the *user-startup* file in Release 2) lets you automatically execute a number of commands whenever you turn on the computer.

Batching Simple Commands

In order to use the EXECUTE command, you must first create a text file containing the command statements you want to execute. You may use either of the system editors, ED or EDIT, to create the command file (see Chapters 6 and 7, which explain the use of the editors), or the Memacs editor in the Tools drawer on the Extras disk. You may also use any word processor or text editor which can save a text-only file, one without imbedded command characters in the text. (To create such a file using *WordPerfect,* for instance, you must choose the Save

Text File option on the Project menu.) Another handy method of creating short command sequence files is to use a console window as a mini text editor. Chapter 4 showed you how to do this.

The file you create should contain one or more lines of CLI commands, one command to a line, with a RETURN character at the end of each line. The format should look like this:

```
ECHO "The current date and time settings are:"
DATE
ECHO "*N*E[3mThe current device assignments are:*E[0m"
ASSIGN  SAY All, done.
```

This file contains the ECHO command, with which you may not be familiar. ECHO prints out the text string enclosed in quotes (in Release 2, the quotes aren't always necessary). It's really only useful when included in command sequence files. By placing ECHO statements in command files, you can let the user know what the command file is doing. ECHO uses the asterisk (*) as a special escape character. The asterisk causes ECHO to treat subsequent characters as formatting commands, rather than as text that it should print. In the example above, the *N combination causes ECHO to skip a line. The *E combination is used for the ESC character, so that console escape sequences, like the one that changes to italic print, can be used (see Chapter 2 for more on console escape sequences). The AmigaDOS Release 2 ECHO only treats the asterisk as special if the text is enclosed in quotes.

The SAY command, found in the last line above, is similar to the ECHO command except that it uses the Amiga Narrator device and Translator library to actually speak the words typed on the command line. Unlike the ECHO command, you can't use quotation marks around the text with SAY—if you included them, your Amiga would try to speak them. The comma after the word All isn't a typo, either. Punctuation marks like the comma and period can change the timing and inflection of the speech that SAY produces.

Let's assume that you've created a disk file in the current directory named *Report,* which contains the lines of text listed above. You could then type ***EXECUTE Report***, and each of the commands in the file would be executed in sequence, producing the following screen output shown on page 88.

In this example, it's assumed that the file *Report* was in the current directory. If it were in another directory, you could have used the full pathname to identify its location (*EXECUTE df1:Utilities/Report*). But there's another way to

Output of the Report Sequence File

```
The current date and time settings are:
Saturday 15-Nov-86 18:27:01
The current device assignments are:
Volumes:
Extras [Mounted]
Workbench1.3 [Mounted]
     S          Workbench1.3:s
     L          Workbench1.3:l
     C          Workbench1.3:c
     FONTS      Workbench1.3:fonts
     DEVS       Workbench1.3:devs
     LIBS       Workbench1.3:libs
     SYS        Workbench1.3:Workbench
Devices:
     PIPE       AUX       SPEAK     NEWCON    DF1
     DF0        PRT       PAR       SER       RAW
```

make the EXECUTE command execute a sequence file that isn't located in the current directory. As you may remember from the previous chapter, the system assigns the logical device name S: to the *s* directory on the boot disk when you turn on the computer. The EXECUTE command first looks for the command file in the current directory, but if it doesn't find it there, it looks in the S: directory. By saving your command files to the S: device, therefore, you can be sure that EXECUTE will always be able to find it, regardless of which directory is current.

If you are using the 1.3 Shell window instead of an ordinary CLI, it's possible to execute script files without using the EXECUTE command. If the script file has the S protection bit set, you may execute the sequence just by typing the name of the file. To do this with the file in the example above, you would first set the file's S bit with the command:

```
PROTECT Report +S
```

Afterwards, you could execute the sequence just by typing ***Report.*** Since the Shell executes script files as if they were CLI commands, it doesn't use the normal sequence file search path to find them. Instead, it looks for them in the normal command file search path. If you use a lot of script files from the shell,

and these scripts are stored in the S: directory, you may want to be sure that S: has been added to your command search path (as it normally is by the default startup-sequence).

Startup-Sequence: The Autoexecuting Command File

As has been mentioned several times already, AmigaDOS recognizes a special command sequence file located in the S: directory called *startup-sequence*. The sequence of commands contained in the *startup-sequence* file is executed whenever you turn on the machine or reset it by holding down CTRL and both Amiga keys. To see the standard command file which comes on the Workbench

Workbench 1.3 Startup Sequence File

```
c:SetPatch >NIL: r ;patch system functions
Addbuffers df0: 10
cd c:
echo "A500/A2000 Workbench disk. Release 1.3.2 version 34.28*N"
Sys:System/FastMemFirst ; move C00000 memory to last in list
BindDrivers
SetClock load       ;load system time from real time clock (A1000 owners
                    ;should
                    ;replace the SetClock load with Date
FF >NIL: -0 ;speed up Text
resident CLI L:Shell-Seg SYSTEM pure add; activate Shell
resident c:Execute pure
mount newcon:
;
failat 11
run execute s:StartupII ;This lets resident be used for rest of script
wait >NIL: 5 mins ;wait for StartupII to complete (will signal when done)
;
SYS:System/SetMap usa1 ;Activate the ()/* on keypad
path ram: c: sys:utilities sys:system s: sys:prefs add ;set path for Work-
bench
LoadWB delay ;wait for inhibit to end before continuing
endcli >NIL:
```

disk, enter `TYPE S:startup-sequence`. For Workbench 1.3, the displayed file should look like the display on page 89.

The first command installs software fixes for Kernel ROM bugs (SETPATCH is the standard method used to make updates between ROM revisions, and should always be run first to insure those fixes take place immediately). The second allocates additional disk buffers to speed up floppy disk access, and the third makes C: the default directory. The next command uses ECHO to send a message to the screen. The FASTMEMFIRST command is run to give priority to external expansion memory if any is present.

Next, the *BINDDRIVERS* command adds in any device drivers that are in the expansion drawer, such as the one needed to integrate the PC Bridgeboard into the system. *SETCLOCK* is used to set AmigaDOS' system clock from the hardware clock in the 2000 and 500's and 600's equipped with the clock option. Next, the FF command is used to speed up text printing. The first RESIDENT command loads the program needed for the 1.3 Shell, and the next makes the EXECUTE command resident, after which the NEWCON: device is mounted to add enhanced console functions to Shell windows.

The FAILATcommand is used to set the failure level higher, so that the script won't end abruptly if one of the subsequent commands should fail. Next, RUN is used to EXECUTE a separate batch file which does some ASSIGNS and other startup tasks. A WAIT command is used after this to stop the current script until the other script is done. This prevents the two scripts from trying to access the same disk at the same time, a situation which slows down disk access, and causes a lot of unnecessary disk seeks. This WAIT command will wait for five minutes, or until the other script sends it a Break, whichever comes first. After the other script has signalled that it is done, the SETMAP command is used to install a new keymap, one which recognizes the extra keypad keys on the 500/2000 that weren't on the 1000. The PATH command is used to set some extra default search paths for commands. LOADWB is used to start the up the Workbench environment, and ENDCLI is used to terminate the initial CLI window.

Although the sequence of commands is somewhat different in the Release 2 startup file the process is much the same (there is a line-by-line explanation of Release 2 startup file starting on page 7-56 of the "Using the System Software" manual that comes with Release 2). The Release 2 startup-sequence, however, automatically executes another script called *User-startup* if it is present. If you plan to make any additions to the startup-sequence, you should always put those additional commands in the *User-startup file* instead. That way, you'll know which commands are the ones normally required by the system, and which you've added.

The *startup-sequence* command file is a powerful tool because it lets you specify what happens every time you turn on your Amiga. For instance, you can choose to load the Workbench every time or stay in CLI mode, or have both interfaces available at the same time. You've already seen that to stay in CLI mode, all you have to do is leave out the last two lines of the standard file. But if you want to load the Workbench *and* keep a CLI window, you can insert the line

```
NEWCLI con:20/20/200/100/
```

or for 1.3 users

```
NEWSHELL Newcon:20/20/200/100/
```

right before the *LoadWb* line. This starts up a smaller CLI window that will stay on the Workbench screen after the Workbench is loaded. (See Chapter 6 for details on how to edit a file such as *startup-sequence*. Briefly, though, to add this line, type ***ED s/startup-sequence***, which puts you in the screen editor. Use the cursor keys to move to the *LoadWb* line, press RETURN, cursor up to the empty line, and enter what you see above. Press ESC, then enter ***X***, and press RETURN. The new *startup-sequence* file will overwrite the old. Wait until all disk activity has ceased, then warm start your Amiga by pressing the CTRL key and both Amiga keys at the same time. The Workbench should appear, along with a CLI window.)

There are a number of other things you may want to do automatically at startup time. If you're using a hard disk or other other external peripheral device, you may need to run a program to integrate them into the system. One of the most useful sequences of commands to include in the *startup-sequence* file is one that sets up a RAM disk directory containing a collection of your most frequently used commands and ASSIGNs it as the default command directory. The simplest sequence to use is

```
MAKEDIR RAM:c
COPY SYS:c ram:c ALL
ASSIGN c: RAM:c
```

This is faster than copying each individual file since DOS doesn't have to read the COPY program from disk each time. The disadvantage is that you end up using a lot of RAM to hold command programs you seldom, if ever, use. Taking up over 128K of RAM for command programs is wasteful on a 512K system and prohibitive on a 256K system. The alternative is to copy files selectively, like this:

```
C:CD C:
MAKEDIR RAM:c
COPY C:copy RAM:c
RAM:c/copy assign|cd|delete|dir|diskcopy RAM:c:
RAM:c/copy echo|ed|endcli|info|join|list RAM:c:
RAM:c/copy Makedir|newcli|run|type RAM:c:
PATH RAM:c Add
```

This script copies the COPY program to RAM:, then uses that version to copy the rest of the files. This reduces the time spent reading the COPY program from disk. Some other steps are taken to shorten the time required to execute this script. Another shortcut is to change the current directory to C:, to make the CLI look there first for each command, instead of wasting time looking in some other default directory. When a command in another directory is needed, we use the complete pathname. To avoid loading the COPY command for each operation, we use the "OR" wildcard operator (the vertical bar) to move several files at a time. When all the required files are copied, we add the RAM:c directory to the default command search path with the PATH command.

Shell users will find it more beneficial to use the RESIDENT command to keep often-used commands loaded in memory than copying those files to the RAM disk. A resident program is always loaded in memory, ready to go, and Shell windows search the Resident list before looking in any of default search paths. Resident commands may also be given new, shorter names, like DEL for Delete, and MD for Makedir. The RESIDENT command doesn't allow the use of wildcards, however, so you would have to change the above example to include a separate line for each command that you wish to make resident.

Another common task you can perform at startup time is setting the system clock and calendar. If you've purchased an optional hardware clock/calendar for the 1000, it probably came with a program for setting the system clock from the hardware clock. The command to run this program should be part of your *startup-sequence* file (A SETCLOCK command for 500 and 2000 owners is already in the default startup file). If you don't have a hardware clock for your 1000 or 500, you should set the time and date manually each time you start the system. The original *startup-sequence* file on older Workbench disks print a message telling you to set the date and time from the Preferences program. If you prefer, you can give yourself the opportunity to set the time and date as part of your *startup-sequence* file. The following example demonstrates one technique for doing this:

```
ECHO " "
ECHO "The current setting of the date and time is:"
DATE
ECHO " "
ECHO "Enter the correct date and/or time now."
ECHO "Use the form DD-MMM-YY for the date (format as
    09-Sep-86)."
ECHO "Use the form HH:MM:SS or HH:MM for the time (for-
    mat as 14:55)."
ECHO " "
```

The next command is tricky. It uses the question mark form of DATE to prompt you with the command template and wait for input. It uses redirection to send the prompt text down a black hole. The result is that it accepts input and sends it to DATE.

```
DATE >nil: ?
ECHO " "
ECHO "The new date and time settings are:"
DATE
DATE >now
```

As the comments in italics explain, this example uses the question mark form of DATE. Normally, when you type `DATE ?`, the DATE command prints out its command template and waits for you to enter input in that format. By redirecting the output of the command to NIL:, which does nothing with it, you suppress the command template and instead provide more detailed instructions as reminders to yourself. Redirecting the output to NIL: performs an additional function as well. If you decide that you don't want to change the date and just press RETURN, the DATE command doesn't get any instructions about what date or time is to be set. In such a case, the command normally prints out the *current* date or time. Here, that would be inappropriate and would confuse the display. Luckily, the redirection to NIL: prevents this text from being displayed so that if you just press RETURN, nothing happens.

Notice that the last command in the new *startup-sequence* file redirects the output from DATE to the file *now*. This kind of date stamping can be helpful, for the Amiga looks to the most recently modified or created file to set the time (if you don't do it yourself manually). Thus, if you haven't altered or created any

files since the last time you booted the computer up, it looks to *now* for the current date.

Passing Instructions to Commands

As convenient as it may be to EXECUTE a sequence of fixed commands stored in a file, it limits you to working with the same specific files and directories every time. That's why AmigaDOS has a mechanism for passing words from the EXECUTE command line to the command file and substituting them in the commands. This lets you create command files which do different things, depending on what you type in the EXECUTE command line.

Since this concept is much easier to demonstrate than to explain, let's take a very simple example. Suppose you want to create a command file which makes a backup copy of a file. You need some way of specifying the name of the file so that you won't be continually backing up the same file. The following short command file, named *Backup,*does just this:

```
.KEY filename: (.K filename is also acceptable)
COPY <filename> TO :Backups
```

To use this command file, type `EXECUTE Backup Mydata`. The result is that the file named *Mydata* is copied to the *Backups* directory (this assumes that the *Backups* directory already exists in the root directory of the current disk). If you typed `EXECUTE Backup Program`, the file named *Program* would be copied to *Backups.* The key to this process is in the first line of the file. The line starts with the word *.KEY,* which is not a normal CLI command, but rather a sequence file directive which tells the EXECUTE command how to operate. The .KEY directive tells EXECUTE that the command template which follows should be used to determine what commands can be passed to this command file. In this case, .KEY tells EXECUTE that if a word is entered on the EXECUTE command line after *Backup*, that word is to be referred to as *filename.* Anytime *<filename>*appears in the *Backup* file, the word appearing on the command line after *Backup* is substituted. So when you type `EXECUTE Backup Mydata`, EXECUTE takes the command line *COPY <filename> to :Backups* and substitutes *Mydata* everywhere that *<filename>*appears. The result is the command line *COPY Mydata TO :Backups.*

If you don't enter any command words after the name of the command sequence file, there's nothing to substitute for the keyword specified by the .KEY (or .K) directive. In the above example, the command *EXECUTE Backup* translates to the command line *COPY TO :Backups*, which copies everything in

the current directory to the *Backups* directory. This may not be the result you wanted. Fortunately, AmigaDOS provides a way to prevent this from happening. It allows you to specify a default text string to be substituted for the keyword if the user (yourself, more than likely) doesn't enter a substitution value. There are two ways of specifying the default value.

The first way to provide an alternative text string is to use the .DEF directive, followed by the substitution value. When you use this directive, the default value is substituted wherever the keyword appears in the absence of a normal substitution. Let's change the *Backup* command file to look like this:

```
KEY filename
.DEF filename #?.bas
ECHO "Copying <filename> to the Backups directory"
COPY <filename> TO :Backups
```

Now, if you type `EXECUTE Backup`, the pattern expression *#?.bas* is substituted for the keyword, and the command becomes *COPY #?.bas TO :Backups.* The pattern matches any file whose name ends in the characters *.bas,* so any file fitting that description is copied to *Backups.* An ECHO command was added to tell you what's happening. The default value is substituted in that command as well, so ECHO prints the message *Copying #?.bas to the Backups directory.*

While the .DEF directive substitutes every instance of the keyword in the file with the default value, another directive, the dollar sign ($), causes the substitution to be made only in the line in which it appears. Using this directive, we can create a *Backup* file which looks like this:

```
.K name
ECHO "Copying <name$all BASIC program files> to the
    Backups directory"
COPY <name$#?.bas> to :Backups
```

Using this version of the *Backup* command file, the command *EXECUTE Backup* still copies all files ending in *.bas* to the *Backups* directory. This time, however, the default value is only substituted in the COPY command. A different value is substituted in the ECHO command. The message printed by ECHO is *Copying all BASIC program files to the Backups directory.* Notice that you didn't have to put quotation marks around the phrase *all BASIC program files,* even though it contains spaces. The substitution value replaces the keyword with

the exact string of characters which appears in its definition. Also, don't confuse the $ substitution character which always appears between angle brackets (or other BRA and KET characters), with the $ which is used to retrieve the value of an environment variable (which doesn't appear in brackets).

As of 1.3, the EXECUTE command recognizes one additional substitution directive. Two dollars signs together ($$) within brackets are substituted by the number of the current CLI. For example, the string file_number<$$> would be interpreted as *filenumber1* in a batch file that was executed from CLI 1. This substitution is useful for creating temporary files with unique names, that won't be overwritten if the same batch file is run from two separate CLIs at the same time.

The EXECUTE command doesn't limit you to substituting a single word on the command line. The .KEY directive can specify a template which contains as many keywords as you like (up to a total of 255 characters). The only restriction is that the template must be in the same format as the command template which prints when you type a command name followed by a question mark (see the beginning of the "AmigaDOS Command Reference" section for more information on command templates). This means, among other things, that the keywords must be separated by a comma, with no spaces between them. It also means that you can use /A after the name to show that this argument is required. For example, let's say you wanted to be able to back up two named files each time you executed *Backup.* The following command file shows how you can substitute both filenames:

```
.K name1/a,name2/a
COPY <name1> to :Backups
COPY <name2> to :Backups
```

Using this *Backup* command file, if you type `EXECUTE Backup document letter`, both the document and letter files will be copied to the *Backups* directory. If you do not specify at least two files on the command line, however, the command will fail.

As you've seen from earlier discussions of filenames and pattern matching, the use of special characters can cause some problems. The EXECUTE directives are no exception. What if you want to use the default message *Copying files —> thisaway* in the above example? Because there is a right angle bracket imbedded in the text, EXECUTE will interpret the entered command `ECHO "Copying <name$files –>> thisaway> to the Backups Directory`, and

print *Copying files —> thisaway> to the Backups Directory,* not *Copying files —>> thisaway to the Backups Directory,* as you wanted.

A similar problem may occur when you try to use the redirection operators (< for input redirection and > for output redirection) in a script file that contains keyword substitutions. To avoid these problems, AmigaDOS provides directives that let you redefine the directive characters. For example, you can change the left angle bracket character to a left square bracket character with the directive *.BRA [.* (the final period after the bracket character is necessary). Likewise, to change the right angle bracket to a right square bracket, use the directive *.KET].* (again, the period after the bracket is mandatory). The *.DOLLAR* or *.DOL* directive is used to change the character which introduces the default substitution value. The directive *.DOL #* changes the dollar character to a pound sign, for example. And finally, the *.DOT* directive allows you to redefine the period character that appears in front of most of the directives.

Testing Conditions with IF

A command sequence file which always does the same thing can't cope very well with unforeseen conditions. For example, the *Backup* file described above assumes that a directory named *Backups* exists in the root directory of the current disk. If it doesn't exist, the COPY command will copy all of the files to a single file named Backups. One solution would be to insert a MAKEDIR command to create the directory. But if the directory already exists, then the MAKEDIR command will fail, and the script will terminate unexpectedly.

The solution is to let the command sequence file test whether or not the directory exists, then act accordingly. If the directory exists, the copy can take place. If it doesn't exist, then MAKEDIR can create it before the copy takes place. EXECUTE uses the IF command to make such decisions. IF can be used to test a number of conditions. If the condition it tests is true, then the subsequent commands will be executed. If the condition is not true, then none of the subsequent commands in the file will be executed (until the ENDIF command is given).

One of the conditions which you can test with IF is whether or not a directory or file exists (the EXISTS option added in 1.3 to the ASSIGN command is better for testing volumes, since no "Insert Volume xxx in any drive" requester appears if the volume doesn't exist). The form of this command is

```
IF EXISTS name
```

You can use the keyword NOT to reverse any test. The condition *EXISTS name* is true if the object called *name* exists, and *NOT EXISTS name* is true if it doesn't exist. Applying these facts to the problem raised at the beginning of this section, you can come up with this new, improved *Backup* command file:

```
.KEY filename
IF NOT EXISTS :Backups
MAKEDIR :Backups
ENDIF
COPY <filename$#?.bas> to :Backups
```

Now when you EXECUTE *Backup* (by typing `EXECUTE Backup` *filename*), the command file first checks whether the *:Backups* directory exists. If not, it creates the directory. But if it does exist, the command file skips the MAKEDIR command and copies the file. Users of Workbench 1.3 and higher should note that this process isn't always necessary using the newer Copy command; it creates the target directory automatically when you copy a directory.

The IF-ENDIF sequence also allows for an ELSE clause. Commands which come after the ELSE command will be executed *only* when the IF clause is *not* true. Let's look at the following sample command file:

```
.Key from/a,to/a
IF NOT EXISTS <to>
COPY <from> AS <to>
ELSE
ECHO "Sorry, there's already a file named <to>."
ECHO "If I copy <from> to <to>, it will wipe out
        <to>."
ENDIF
```

Call this command file *safecopy*. It's a cautious version of the COPY command. The COPY command is pretty reckless—if you tell it to copy the file *ordinary* to another file called *important,* and there's already a file *important,* then the contents of *ordinary* replaces the *important* file. You've just lost *important*'s contents. This version first checks to see if there's a file with the target. If not, it executes the COPY command and then skips from ELSE to ENDIF. But if the file already exists, it skips the COPY command and instead executes the command sequence starting with ELSE, which politely explains why it can't make the copy. Note the use of the command template characters "/a" after the

Directive	Function
.KEY *value1,value2* or .K *value1,value2*	Uses the command template (value1,value2) for substituting command values
	Substitutes the first command value here
	Substitutes the first command value here, but if none is given, substitutes default
	Substitutes the current CLI number (Workbench 1.3)
.DEF value1 default	*If the command value was not entered, substitutes* default*for* <value1>*everywhere*
.BRA character	*Replaces the left angle bracket (<) with* character
.KET character	*Replaces the right angle bracket (>) with* character
.DOLLAR character or *.DOL* character	*Replaces the dollar sign ($) with* character
.DOT character	*Replaces the dot (.) with* character

keywords. These characters indicate that both keywords are required. If you try EXECUTing the file without specifying a "from" and a "to" file on the command line, the sequence will fail.

Another condition that IF can test is whether two text strings are the same. The keyword used for this test is EQ. The format of the test is *IF string1 EQ string2*. One use for this test is to determine what string was substituted for a word designated by the .KEY directive. You can even test it to see whether any substitution was made. Let's look at an example:

```
.KEY name
IF <name>q NOT EQ "q"          ;if name was entered, this is
                               true
IF EXISTS <name>               ;check to see if the file
                               exists
RUN EXECUTE Backup <name>      ;you can nest EXECUTEs
ELSE                           ;matches IF EXISTS <name>
ECHO "I can't find a file called <name> "
ENDIF                          ;matches IF EXISTS <name>
ELSE                           ;matches IF <name>q NOT EQ"q"
ECHO "You did not enter the name of a source file"
ENDIF                          ;matches IF <name>q NOT EQ"q"
```

As you can see, this is a bit more complicated than the previous examples. There are two IF statements, one nested within the other. The first IF tests whether any value was entered on the command line to be substituted for the keyword *name.* It does this by seeing if what was substituted for *name,* plus the letter *q,* is equivalent to the letter *q* alone. If any substitution was made, the two strings will not be equal, and the condition is true. If no substitution was made, the EXECUTE command branches to the ELSE clause, which prints *You did not enter the name of a source file.*

After the command file has tested to see whether the name of a source file was entered, it must still test to see whether that file exists. The second IF statement takes care of that, using the EXISTS keyword as the test. If the file exists, the *EXECUTE Backup* command is run as a background process to back up the file. This demonstrates that you can both run an EXECUTE sequence as a background process and that you can use one command file to EXECUTE another. If the file doesn't exist, execution skips to the ELSE clause, which prints the message *I can't find a file called <name>.*

Several new comparison operations were added to IF in Workbench 1.3. It can test for all possible string comparison cases, with the addition of GT (greater than) and GE (greater than or equal) keywords. To test for less than or and less than or equal conditions, use NOT GE and NOT GT. These string comparisons normally use alphabetical order to test whether the value of one string is greater than or less than another. Using the new VAL keyword, however, it is possible to test the strings by numeric order instead. For example, when you execute the script

```
IF "44" GT "14"
ECHO "44 is greater"
ELSE
ECHO "14 is greater"
ENDIF
```

it prints ***14 is greater.*** If you change the script to read

```
IF VAL "44" GT "14"
ECHO "44 is greater"
ELSE
ECHO "14 is greater"
ENDIF
```

however, you come up with the correct answer. Another change to the IF command in Workbench 1.3 is the use of the dollar sign ($) as a substitution character. A string that starts with a dollar sign will be replaced by an environment variable of the same name (in the Release 2 environment variables are substituted anywhere within a Shell command line). An environment variable is a string that is stored with the SETENV command, and can be retrieved with the GETENV command (local strings that are recognized only by the current Shell can be created with SET and retrieved with GET under Release 2). For example, the command "SETENV name Fred" stores the value "Fred" in an environment variable called name. If this is the case, the statement

```
IF $name EQ "Fred"
```

will be true.

The last condition which IF tests is the return code left by the previous command. The return code is a number passed to the CLI by a program when it finishes. The code indicates whether the program was successfully completed or whether an error occurred. Programs normally use a return code of 5, 10, or 20 to indicate that an error happened. The higher the return code, the more serious the error. IF lets you test for each of these codes with the keywords WARN, ERROR, and FAIL. IF WARN is true if the last return code was 5 or greater, IF ERROR is true if the code was 10 or greater, and IF FAIL is true if a code of 20 or greater was returned.

Some CLI commands are designed almost exclusively to set return codes for scripts. The ASK command, for example, allows a script to print a question, and then obtain a yes or no answer from the user. If the user responds with a Y,

the return code is set to 5, and WARN is true. If the user responds with a N or carriage return, the return code is 0, and WARN is false. Similarly, the VERSION command can be used to test the current version of the operating system. This makes it possible to have a script perform one way if running under 1.3, and another if running under Release 2:

```
VERSION >nil: graphics.library 36   ;Is OS version 36 or
                                    greater (2)?
IF WARN                             ; No, so it must be 1.3
                                    or lower
EXECUTE s:1.3.script                ; Execute the 1.3
                                    script
ELSE                                ; Yes, 2.0 or higher
EXECUTE s:2.0.script                ; Execute the script
                                    appropriate for 2.0
ENDIF
```

Normally, if a serious error occurs during a command sequence, the entire sequence is immediately terminated. The default cutoff point is a return code of 10 or higher. Using this default setting, it's impossible to test for a FAIL or ERROR condition, since the sequence terminates before the test can take place. It's possible to change the point at which a command sequence fails, however, using the FAILAT command. Entering the command `FAILAT 25`, for instance, insures that the sequence doesn't terminate unless a program returns an error code of 25 or higher. The new failure threshold applies only to the current command sequence. Once it has finished executing, the default value is restored.

In most circumstances, you'll want to terminate the command sequence if a serious error is encountered. Changing the FAILAT threshold and testing for the error yourself gives you an opportunity to present the user with a message that clearly explains what happened. For example, you could change the command file allowing the user to input the date and time to read:

```
ECHO " "
ECHO "The current setting of the date and time is:"
DATE
ECHO " "
ECHO "Enter the correct date and/or time now."
ECHO "Use the form DD-MMM-YY for the date (09-Sep-
    86)."
ECHO "Use the form HH:MM:SS or HH:MM for the time
    (14:55)."
ECHO " "
FAILAT 25
DATE > nil: ?
IF ERROR
ECHO "You did not enter a correct date and/or time
    setting."
ECHO "The current settings remain in effect."
ELSE
ECHO "The new date and time setting are:"
DATE
DATE >now
ENDIF
```

If this example was part of a larger *Startup-Sequence* file, there's a good chance that you would not want the entire sequence to terminate if the user didn't enter the date or time correctly. Using FAILAT to reset the failure threshold and IF ERROR to test for errors, you can tell the user that the attempt was not successful and continue with the rest of the sequence.

Even if you've used FAILAT to change the failure threshold, you may exit from a command sequence at any time by using the QUIT command. QUIT also allows you to leave a specific return code. The command QUIT 20, for example, terminates the command sequence immediately and leaves a return code of 20.

To summarize, the IF command uses the following keywords for making its test:

Keyword	Function
EXISTS *name*	Is true if the volume, directory, or file exists
*string1*EQ *string2*	Is true if the text of the two strings is the same (ignoring uppercase and lowercase)
*string1*GT *string2*	Is true if the ASCII value of string1 is greater than the ASCII value of string2 (Workbench 1.3)
*string1*GE *string2*	Is true if the ASCII value of string1 is greater than or equal to the ASCII value of string2 (Workbench 1.3)
*VAL string1*GT *string2*	Is true if the numeric value of numeric string1 is greater than that of numeric string2 (Workbench 1.3)
WARN	Is true if the previous program left a return code of 5 or greater
ERROR	Is true if the previous program left a return code of 10 or greater
FAIL	Is true if the previous program left a return code of 20 or greater
NOT	Reverses the result of the test

Note: Under Workbench 1.3 and higher, IF evaluates $VAR as the string contained within the environment variable named VAR. If there is no environment variable of that name, its value is the text string "$VAR".

Branching and Looping with SKIP

For most simple cases, IF-ELSE branching is sufficient. But if you're making a number of tests, the SKIP command can make things easier. It allows you to use the results of the IF test to jump to a subsequent command line. The LAB command is used to designate the line at which you wish execution to resume. This is the general format:

```
IF test       ;If the results of test are true;
SKIP Next     ;Executition jumps
ENDIF
command
command
command
LAB Next      ;to here
command
```


An ENDSKIP command was added in Workbench 2.0 to designate the end of a SKIP block.

SKIP is particularly useful where you may wish the same thing to happen after a number of tests are completed, regardless of their outcome. Rather than writing the commands over and over in the body of the IF-ELSE-ENDIF clause, you can have each command jump to the same labeled line. The following command file demonstrates this principle. It copies both a file and its associated icon file to another volume or directory:

```
.Key from,to
IF<from>q EQ "q"
SKIP Missing
ENDIF
.
IF <to>q EQ "q"
SKIP Missing
ENDIF
.
IF NOT EXISTS <from>
SKIP Missing
ENDIF
.
COPY <from> <to>
IF EXISTS <from>.info
COPY <from>.info <to>
ENDIF
SKIP Done
.
LAB Missing
ECHO "You must enter the name of an existing file to
    copy,"
ECHO "and the volume or directory to copy it to."
LAB Done
```

Prior to Workbench 1.3, it was not possible to use SKIP to move backward within the command file. With version 1.3, a new option, BACK, was added. When the BACK option is used, SKIP starts searching for the LABEL at the beginning of the file, instead of at the current line. Although you can not use BACK to SKIP past a prior EXECUTE command, it is still useful in many cases for executing instructions in a loop. You can even use an environment variable as a loop counter, as the following example shows:

```
SETENV Count 1
LAB Loop
ECHO "This is pass number " NOLINE
TYPE ENV:Count
EVAL <ENV:Count >NIL: to=T:Temp<$$> value2=1 op=+ ?
COPY T:Temp<$$> ENV:Count
IF VAL $Count NOT GT 10
SKIP Loop BACK
ENDIF
DELETE >nil: T:Temp<$$> ENV:Count
ECHO "All Done!"
```

When you execute this script, the output looks like this:

```
This is pass number 1
This is pass number 2
This is pass number 3
This is pass number 4
This is pass number 5
This is pass number 6
This is pass number 7
This is pass number 8
This is pass number 9
This is pass number 10
All Done!
```

Due to Workbench 1.3 limitations, this script is somewhat clumsy. It takes two commands to print each line, for instance, because ECHO has no way of including the environment variable in its output. It also takes two commands to increment the environment variable Count. The EVAL command takes the contents of the ENV:Count file, adds one to its value, and copies the result in the temporary file T:Temp<$$>. Then, we must copy this value back to the environment variable, Count, in order to have it evaluated by the IF command. With version 2.0 enhancements, this script becomes much simpler:

```
SET Count 1
LAB Loop
ECHO This is pass number $Count
SET Count 'EVAL $Count + 1 '
IF VAL $Count NOT GT 10
SKIP Loop BACK
ENDIF
UNSET Count
ECHO All Done!
```

Under Release 2, ECHO can include the value of the environment variable in its message. And using the backtick feature, we are able to combine the three lines the 1.3 version of the script requires to increment the value of Count into one line. This line takes the result of the streamlined EVAL command, and feeds it directly to SET.

EXECUTEing from a Command Sequence File

It is possible, and sometimes quite useful, to use the EXECUTE command from within a command sequence file. A command file can even EXECUTE itself. This permits a limited form of looping. For example, let's say that you have a number of disks to copy, and you want to write a command sequence file that continuously prompts you to insert source and destination disks, and then copies one to the other. To avoid having to swap in the Workbench disk when DOS wants to read the commands, let's copy them to RAM:

```
COPY Sys:system/DiskCopy to RAM:
COPY c:Execute to RAM:
MAKEDIR RAM:T ;Only necessary if the startup script
    hasn't done it already
CD RAM:
```

Now let's create a file called *RAM:ConCopy* that continuously executes the DISKCOPY command:

```
DiskCopy df0: to df1:
ASK "Copy another disk? (Y/N)
IF WARN
EXECUTE ConCopy
ENDIF
```

When we EXECUTE *ConCopy,* it runs DISKCOPY once, then ASKs if we want to make another copy. If the user types in a "Y" reply, the script EXECUTEs itself all over again. Note that we created a directory :T in RAM:. The EXECUTE command often needs to create a temporary file, and tries to store this file in the :T directory. If there is no :T directory, the error message *EXECUTE: Can't open work file ":T/Command-0-T01"* appears and the command fails. As of Workbench 1.3, however, EXECUTE will first try to create its temporary files in T: if it is assigned, and if not, will then go to :T.

With some of the new options added in Workbench 1.3, it's even possible to use scripts to build other scripts and execute them. Prime examples are the SPAT and DPAT scripts that come in the S: directory, which allow you to apply commands to any files which meet your wildcard substitution criteria. Let's look at SPAT, which you use with commands that take a single command parameter:

```
.key com/a,pat/a,opt1,opt2,opt3,opt4
.bra {
.ket }
failat 21
list >t:q{$$} {pat} lformat="{com} *"%s%s*" {opt1}
    {opt2} {opt3} {opt4}"
IF NOT FAIL
execute t:q{$$}
ELSE
echo "{pat} not found"
ENDIF
failat 10
;do wildcards for single arg command
```

SPAT uses the LFORMAT option of the LIST command to create a temporary file called T:q{$$}, where {$$} represents the process number of the Shell from which SPAT was run. This temporary file contains a command line for

each file that matches the pattern. Therefore, if you give the command "EXECUTE SPAT PROTECT Sys:System -d" from a Shell whose process number is 1, SPAT will create and execute a command file called q1 whose contents look like this:

```
PROTECT "Sys:System/Fountain" -d
PROTECT "Sys:System/Fountain.info" -d
PROTECT "Sys:System/.info" -d
PROTECT "Sys:System/Setmap.info" -d
PROTECT "Sys:System/Setmap" -d
PROTECT "Sys:System/RexxMast.info" -d
PROTECT "Sys:System/RexxMast" -d
PROTECT "Sys:System/NoFastMem.info" -d
PROTECT "Sys:System/NoFastMem" -d
PROTECT "Sys:System/Format.info" -d
PROTECT "Sys:System/Format" -d
PROTECT "Sys:System/FixFonts.info" -d
PROTECT "Sys:System/FixFonts" -d
PROTECT "Sys:System/DiskCopy.info" -d
PROTECT "Sys:System/DiskCopy" -d
PROTECT "Sys:System/CLI.info" -d
PROTECT "Sys:System/CLI" -d
PROTECT "Sys:System/BindMonitor.info" -d
PROTECT "Sys:System/BindMonitor" -d
PROTECT "Sys:System/AddMonitor.info" -d
PROTECT "Sys:System/Addmonitor" -d
```

This makes it possible to protect every file in a directory from deletion with one command from Workbench 1.3 (since the Workbench 2.0 PROTECT has an ALL option, it can perform the same task without a script).

Debugging a Script

Under Workbench 1.3 and below, debugging a long script can be difficult, because you can't be certain what command the script is executing when it fails. About the best you can do to pinpoint the problem is to add ECHO command

along the way, saying things like "Got to point 1" and "Got to point 2 without problems."

Release 2 adds a handy new feature, however, that makes debugging a snap. The Release 2 Shell recognizes an environment variable called ECHO. By default, ECHO is set to OFF, but when you set ECHO to ON, each command line is typed out before it is executed. This makes it easy to see which commands in a script are executing without problem, and which cause the script to fail.

Executing a Script from Workbench

If you've written a script that you find useful, you may want to share it with others. Unfortunately, the novice users who would be most helped by such a script often do not know how to execute a script from the CLI. It would be nice if you could assign an icon to the script, and have the user execute it just by clicking on the icon. You can't do that with the normal EXECUTE command, because it only works from the CLI or Shell. Workbench versions 1.3 and higher, however, include a variation of EXECUTE called ICONX (short for Icon eXecute) that is made to run scripts from the Workbench.

To create an icon for ICONX to use, you must copy a Project icon to a file whose name is the same as the script plus the letters *.info*. Any old icon won't do—it must be of the proper type (the Shell icon is a good one to use, since it is a Project icon). If the pathname of your script file was *DF1:Script*, for example, you could use the command `COPY Sys:Shell.info DF1:Script.info` to copy the icon file from the Shell to the script. Next, you must change the default tool of the Project icon to c:Iconx. You can do this by clicking once on the icon to select it, and choosing *Information* from the Workbench menu. Don't be surprised if you can't find the new icon you created right away. If the window into which you copy the icon is open on the Workbench, it will not update its display to show changes that you make with the CLI until you close it and open it again. Once you have the Information window for the icon displayed, you will see an entry for Default tool somewhere in the left middle part of the window. If you had copied the Shell icon, the default tool will read *Sys:System/CLI*. Click in the text window, and change the text to read *C:Iconx*, then click on the Save button. You will now have a Project icon assigned to your script file, whose default tool is ICONX. By double-clicking on the icon, you will execute the script as if it had been run from a CLI whose current directory is the one in which the icon resides. For more information, see the ICONX listing in the reference section.

Some Common Script Commands

Certain CLI commands are used more frequently in scripts than outside of them while other commands can only be used in scripts. Keep the following commands in mind while writing your scripts.

ASK	Asks a question and receives a yes or no response from the user that can be used by the IF command.
ECHO	Prints a single line of text.
TYPE	Prints multiple lines of text from a text file.
FAILAT	Temporarily changes the failure threshold to allow testing of ERROR and FAIL conditions.
IF	Tests a condition and executes a sequence of commands if true.
ELSE	Executes an alternate command sequence if the condition is false.
ENDIF	Ends a conditional sequence of commands.
LAB	Labels a section to SKIP to.
SKIP	Diverts execution to the LABeled sequence of commands.
ENDSKIP	Ends a SKIP sequence (2.0 and up only).
LIST	Lists a set of files. Can produce command lines with embedded file names for automatic script building.
EVAL	Evaluates numeric expression. Can be used to do math in scripts.
VERSION	Used to test the version number of libraries and devices. Sets WARN flag if lower than specified version.
AVAIL	Gives the amount of free RAM, which can then be tested by script conditions.
CPU	Can be used to test whether the computer is running a 68000, or an advanced processor like the 68030 or 68040. Can also test whether a math co-processor is present (2.0 only).
BREAK	Used to stop another program from a script.
STATUS	Can find the process number of the program BREAK wants to stop.
SET	Sets a local environment variable (2.0 only).
GET	Retrieves a local environment variable (2.0 only).
UNSET	Removes a local environment variable (2.0 and higher).

SETENV	Sets a global environment variable.
GETENV	Retrieves a global environment variable.
UNSETENV	Removes a global environment variable (2.0 and higher).

AREXX

Release 2 of the Workbench incorporates a powerful new command language called AREXX, which is based on the IBM command language called REXX. AREXX can be used not only to run commands from the Shell, but to send messages to programs while they run, providing those programs include AREXX message ports. Many of the newer programs on the Amiga permit you to perform almost any function from an AREXX script that you could normally perform with a keyboard or mouse.

In addition to its inter-process communication capabilities, AREXX is a complete programming language in its own right. Because of its sophisticated input and output functions, conditional execution functions, and looping, AREXX can be used to create truly complex script functions that would be impossible to duplicate with EXECUTE scripts. The AREXX language is far too complex to be covered even briefly in this chapter. Indeed, there are several large books devoted exclusively to AREXX. For our purposes, it is enough to say that if you are trying to create a large script that appears to strain the abilities of the EXECUTE command, you should consider writing it in AREXX instead.

Chapter 6

ED, the System Screen Editor

The screen editor program, ED, is located in the c directory of the Workbench disk, but it has nothing to do with disk or file management like the other CLI commands. Rather, it is a full-screen text editor which can be used to create or edit a text file, such as the script files we discussed in the previous chapter.

ED differs from the other text editor program included on the Workbench disk, EDIT, in a number of ways. EDIT is a line-oriented editor, which means that you must first select the line you want to change. ED, however, is a screen-oriented editor, which displays a whole screen of text at a time and lets you move the cursor around the screen, adding or deleting text as you see fit. While EDIT can be used to alter files which contain binary code (such as program files), ED is designed to edit text-only files. And, finally, ED files always end with a linefeed character, which ED adds if it doesn't find one already present.

To start ED, type ED, followed by the name of the file you wish to edit. If the filename doesn't describe an existing file, ED assumes that you want to create a new file. To exit the ED program, type ESC-Q to quit or ESC-X to save the current file and exit.

ED starts with a maximum workspace of 40,000 characters, which is normally plenty of room for script files and the like. Unless you change the size of the workspace, you're limited to editing files of that size. To change the size of the workspace, use the keyword SIZE on the command line which you use to run ED, followed by the number of characters you want in the workspace. For example, entering the command `ED Windbag SIZE 100000` lets you edit a file called Windbag which can contain up to 100,000 characters. It's a good idea always to specify a size somewhat greater than the exact size of the file.

Release 2 of ED provides a few more startup options. The WINDOW option allows you to specify the characteristics of ED's console window, using the standard window description format described in previous chapters:

```
CON:hpos/vpos/width/height
/windowtitle/option1/option2...
```

You can also use the WINDOW option to specify an alternative console device to use, other than opening a new console. For example, the command `ED Ram:Temp WINDOW *` will use the current CLI console window as ED program window. The Release 2 version of ED recognizes the keywords WIDTH and HEIGHT as specifying the number of characters to display horizontally and vertically, which can be helpful when using an alternative console type with which ED is not familiar. The keyword TAB can be used to set the tab stops. When you press the TAB key, the cursor will be advanced to the position past next column that is an even multiple of the TAB setting (the default is three).

There are two ways of issuing commands to ED—immediate mode and extended command mode. In immediate mode, you give ED its commands by pressing nonprinting key combinations. In extended command mode, you first type the ESC character, which places your cursor on the command line at the bottom of the screen. You may then type in one or more command strings. Command strings are not executed until you press RETURN.

Release 2 of ED is able to read and execute a series of extended commands that are saved in a text file with one command per line of text. The WITH option lets you name a command file for ED to execute as soon as it starts. The command `ED Ram:Temp WITH s:Commandfile`, for example, causes ED to execute all of the commands in *s:Commandfile* at the beginning of the program. Additionally, ED will execute the command file *S:Ed-Startup* if such a file exists. This file can be used to set up custom menu and function key assignments, as we will see later on. If no S:Ed-Startup file exists, ED 2.0 will open up with a default set of menus.

Immediate Mode

The ED program starts in immediate mode. Here, the characters you type are inserted into the text document. To edit, just move the cursor to the appropriate place and either erase existing text or add new text. In versions prior to 2.0, you can only move the cursor by using the arrow keys, or commands. Release 2 lets you place the cursor by moving the mouse pointer to the desired position and clicking the left button. In addition to text characters, there are a number of control commands which can be entered from immediate mode. These commands are executed by holding down the CTRL key, then pressing another key. (The notation CTRL-x will be used to refer to these commands. This indicates that you're to hold down CTRL, and press the key specified by x.) All CTRL character commands are executed as soon as you press the key combination.

Cursor Commands

The cursor is a colored block which indicates the position where additional characters will enter the text buffer. If you're using the default set of colors, it appears as a blue block highlighting the current character in Release 2, and as an orange block in all other versions. You can move the cursor in any direction by pressing one of the cursor arrow keys to the right of the RETURN key. If there's more text in the buffer than appears on the screen, moving the cursor to any edge of the screen and pressing the corresponding cursor arrow key shifts all text (scrolls) to show part of the hidden text. For example, if you move the cursor to the bottom line of the first screen of a long document, then press the down-arrow key, the cursor moves down to the next line and reveals the hidden first line of the next screen. What was formerly the top line scrolls up and out of sight. By using the down and up arrows, you can move forward and backward through the text file.

Other immediate commands allow you to move the cursor in larger increments. The CTRL-T combination moves the cursor right to the first character of the next word. CTRL-R moves the cursor back to the space at the end of the previous word. CTRL-] moves the cursor to the end of the current line, scrolling the screen if the line is longer than the screen width. If the cursor is already at the end of the line, CTRL-] moves it back to the beginning of the line. If you press CTRL-] a number of times, the cursor alternates between the first and last characters of the line. Likewise, CTRL-E moves the cursor to the beginning of the first line on the screen. If, however, the cursor is already at the start of the first line, CTRL-E moves it to the end of the last line on the screen. In Release 2, you can hold down the Shift key to increase the range of the arrow keys. Shift-up and Shift-down take you to the top and bottom of the file respectively, while Shift-left and Shift-right take you to the beginning and end of the current line.

The scroll commands don't change the absolute position of the cursor, but rather move the text itself. CTRL-U scrolls the screen up, which appears to make the cursor move down toward the end of the document. CTRL-D scrolls the screen down, which in effect moves the cursor toward the beginning of the document. Either command causes the whole screen to be redrawn from the top, making the scrolling action rather slow.

Note that in ED, the TAB key is strictly a cursor movement key. When you press TAB (or CTRL-I), the cursor moves to the next TAB position, which is one greater than an even multiple of the TAB setting. For example, if you're using the default TAB setting 3, the TAB key moves the cursor from column 1 (the left

edge of the screen) to column 4, then column 7, column 10, column 13, and so on. You can change the size of the TAB stops with the extended command ST (see below), or in Release 2 use the TAB command line option. Unlike some editors, ED doesn't insert characters into the text when you press TAB. The TAB key leaves neither a TAB character nor spaces in the text, though if it passes over a blank portion of the line, the space characters it bypasses remain in the text. Note also that when you load a text file which contains TAB characters into ED, ED replaces each with a number of spaces.

Character Deletion/Insertion

When you've moved the cursor to the text location you want to edit, there are several immediate mode commands which you can use to delete or insert text. The BACKSPACE key (or CTRL-H) moves the cursor one character to the left, deleting the character. The DEL key deletes the character under the cursor and moves the text to the right one position to the left.

You can also delete characters in larger chunks. The CTRL-O command's actions depend on whether the cursor rests on a character or a space. If the cursor is on a space, CTRL-O deletes all spaces it finds until the first character of the next word. Otherwise, CTRL-O deletes the current character and all characters it finds until the next space between words. Thus, CTRL-O can be used alternatively to delete whole words or the spaces between words. CTRL-Y deletes everything from the current character position to the end of the line. CTRL-B deletes the entire current line, regardless of the cursor position.

Unlike many screen editors, ED doesn't let you delete the RETURN character at the end of a line. This means that once you've split a line with a RETURN, the only way to join it together again is with the extended command J (see below).

The ED editor is always in insert mode. This means that any new characters that you enter push the existing text to the right, rather than overwriting characters. Thus, no special character insertion commands are needed. ED does have an immediate mode command, CTRL-A, which allows you to insert a blank line below the current line and moves the cursor to the beginning of that line.

ED supports lines wider than the screen display. To see different parts of such lines, scroll the text horizontally by moving the cursor left or right. Each line has a maximum of 255 characters—ED won't let you insert characters in a line of maximum length. Normally, though, ED tries to keep all of your text within the right margin by using a form of word-wrap. If a right margin is set, and you're typing a word which extends past that margin, ED automatically ends

the current line with a RETURN character at the space before that word and moves the start of the word down to the next line. This word-wrap feature applies only when you're typing at the end of a line. If you insert characters into the middle of the line, forcing the line over the margin, ED won't break the line. You can disable this feature for the current line only by using the extended command EX, which acts like the margin release on a typewriter (see extended commands below). You may also use extended commands to change the left and right margins from their default positions of 1 and 77 respectively.

Miscellaneous Immediate Commands

The CTRL-F command flips the case of the current character and moves the cursor one position to the right. This means that if the current character is in uppercase, it changes to lowercase and vice versa. If the current character is not a letter, it doesn't change, but the cursor still moves to the right. If the cursor is positioned at the first letter of the word *this,* and you press CTRL-F four times, the word changes to THIS.

CTRL-V redraws the screen. Since ED itself refreshes the display if you size the window or move it or scroll it in any direction, this command will be useful only on rare occasions.

CTRL-G is used in conjunction with the extended mode commands. It repeats the last extended mode command you issued. The usefulness of this command will soon become apparent, as the discussion turns to the extended commands.

Extended Mode Commands

Although immediate mode commands are faster and more convenient to use, the extended mode commands are more powerful. Generally, you may use extended commands to execute any of the cursor movement and deletion functions of the immediate commands. In addition, you may use extended mode commands to delete, copy, or move whole blocks of text, to save and load text files, to find and replace text strings, and to perform various other functions. You can even issue a number of commands at one time or indicate that one or more of these commands is to be executed a number of times. As of Release 2, you can also store a whole sequence of extended mode commands in a text file, so that you can execute the sequence without typing it in each time.

To issue an extended command, you first press the ESC key (or CTRL-[). When you do, an asterisk appears on the bottom line of the screen, and the cursor moves to the space following the asterisk. This indicates that you've moved to

the command line, and any text you enter is to be interpreted as an editor command, not as text to be inserted into the document. After entering the command(s), pressing the RETURN key executes the command. If you just press RETURN without entering a command, no command is executed and you return to immediate mode.

For instance, let's say you want to use the T command (explained below) to go to the top of the file. You first press ESC, and the line at the bottom of the screen shows an asterisk:

```
*
```

You then type T and press RETURN:

```
*T <CR>
```

The command line disappears, and the display moves to show the top of the file.

Extended mode commands are made up of one or two letters. Case is not important, and you can put more than one command on a line by separating them with semicolons.

Sometimes a command requires an *argument,* such as a number or a text string. A text string must be set off with characters known as *delimiters* so that it won't be confused with a command string. The delimiter character can be anything except letters, numbers, spaces, semicolons, or brackets. Double quotation marks are the most common delimiters, though if you want to type a string with double quotation marks in it, you must use something else (like the slash or exclamation point). Strings may appear properly in commands in the form *"this is a string"* or */this is a "string"/* or *!c:sub/"so called"!.*

Cursor Movement Commands

The **CL** (Cursor Left) and **CR** (Cursor Right) commands work just like the left- and right-arrow keys, moving the cursor one space to the left or right. As explained below, however, you can add a repeat count. For example, the command **4CL** moves the cursor four spaces to the left. The *N* command (Next) moves the cursor to the start of the next line, while the *P* command (Previous) moves the cursor to the start of the previous line.

CS (Cursor Start) and **CE** (Cursor End) move the cursor to the start and end of the line respectively. **T** (Top) and **B** (Bottom) move the cursor to the top or bottom of the document, while **M** (Move) moves the cursor to an absolute line number. For example, `M 662` moves the cursor to the start of line 662. This can

be extremely helpful when used with compilers which identify the line numbers where errors occurred.

Release 2, with its greater emphasis on program control, adds *many* new cursor movement commands. The **PD** (Page Down) and **PU** (Page Up) commands scroll the page down twelve lines or up twelve lines, respectively. The **EP** (End Page) command places the cursor at the beginning of the last line of the display. **TB** (Tab) moves the cursor to the next tab position. **WN** (Word, Next) moves the cursor to the first letter of the next word, while **WP** (Word, Previous) moves it to the space after the previous word.

Deletion/Insertion Commands

The **DC** command works just like the DEL key, deleting the character under the cursor. The **D** command functions like the immediate mode command CTRL-B and deletes the entire current line.

I (Insert line) is used to insert a string of text as a new line above the current line. The string follows the I command, as in

```
*I"This goes above the current line"
```

The **A** (Add after) command is similar to the **I** command, but adds the new line after the current line.

S (Split) and **J** (Join) are used to split one line into two and join two lines into one. The **S** command acts just like a RETURN character, which ends the current line and moves the text to the right to a new line below. In effect, the **J** command deletes the RETURN character at the end of the current line, thus joining it with the next line.

The Release 2 version of ED includes some new extended mode insertion and deletion commands. **DL** (Delete Left) works like the BACKSPACE key, deleting the character to the left of the cursor. **DW** (Delete Word) deletes to the end of the current word, while **EL** (End of Line) deletes everything from the current cursor position to the end of the line. **FC** (Flip Case) turns uppercase letters to lowercase and vice versa, just like the immediate mode CTRL-F command.

Search and Replace (Find and Exchange)

Another way of scrolling the screen to a particular place in a document is with the **F** (Find) command. The **F** command is issued along with the text string you want to find:

```
*F"Intuition"
```

Once issued, F searches the document for the exact text specified, from the current cursor position forward to the end of the file. A complementary command, **BF** (Backwards Find), searches from the current cursor position to the beginning of the file. By default, both find commands are case-sensitive and will find a match only if both text strings contain exactly the same combination of uppercase and lowercase letters. You may, however, change this default so that searches ignore differences in case by using the **UC** command. Once you've issued this command, all searches ignore case differences until you reset the default with the **LC** command.

Sometimes you wish both to locate a phrase and replace it with another. The **E** (Exchange) command does just this. When using E, you must first specify the phrase to find, then follow it with the replacement phrase, like this:

```
*E"Intuition"User Interface"
```

This example looks for the word Intuition and replaces it with the phrase User Interface. The **E** command only looks forward, so if you want to catch all occurrences of the search phrase, first move the cursor to the top of the file with the **T** command.

The **EQ** (Exchange with Query) command is a variation on E. Instead of making the substitution automatically, it prints the message Exchange? on the command line. If you press the Y key, the exchange takes place, but if you enter N, the cursor moves past the string.

Both the find and exchange commands lend themselves well to the repeat features of ED. For example, once you've set up a search string with F, it's a simple matter to find the next occurrence of the string by using the immediate command CTRL-G. And it's just as simple to replace every occurrence of a search string with a command like

```
*RP EQ"me"myself"
```

which repeatedly replaces the word *me* with the word *myself* after verifying that you want to make each change. For more on repeating commands, see the section "Multiple and Repeat Commands" below.

Block Transfers

Among the most powerful commands are those which manipulate an entire block of text at once. With these commands, you can delete, copy, or move whole blocks of text.

A block is made up of one or more adjacent lines of text. You use the **BS** (Block Start) and **BE** (Block End) to mark the beginning and ending of a block of text. Blocks always consist of whole lines. When you issue the **BS** command, it marks the beginning of the block at the first character of the current line, regardless of where the cursor is positioned in the line. To complete the block, you must cursor down to the final line and issue the **BE** command. This marks the end of the block at the end of the current line. Both the **BS** and **BE** commands are required to successfully mark a block, and the start of the block must always be above the end. (In other words, you cannot mark the start of the block near the end of the file, then move the cursor up and mark the end of the block.) You can mark the start and end of the block on the same line, however, as with the command

```
*BS;BE
```

which marks the entire current line as a block.

You can only mark entire lines as blocks. **BS** always starts marking at the beginning of the current line, and **BE** always marks to the end of the current line. If you want to mark only parts of a line, you must first use the RETURN key to split the line. Also note that the block stays marked only so long as you don't make any changes to the text. Once you make any editing changes to any part of the text (not just the marked lines), the block markers disappear.

After you've marked a block, you can insert copies of the block by moving the cursor to where you want the block inserted, then using the **IB** (Insert Block) command. You can insert as many copies as you wish, as long as you perform the inserts immediately after marking the block and don't edit text in between insertions.

You can delete the entire block with the **DB** (Delete Block) command. Unlike some editors which retain a deleted block in a special buffer and allow you to retrieve it, ED simply discards a deleted block. Once you've deleted it, it's gone. You can move a block of text, however, by first duplicating it with the **IB** command, then deleting the original block with the **DB** command.

The **WB** (Write Block) command lets you save a marked portion of text to a named file. This allows you to split a large file into two smaller parts or generally manipulate portions of a file. The **WB** command must be followed with the name of the file to which the marked portion is to be written. This filename must be enclosed by the normal string delimiters, such as quotation marks:

```
*WB"RAM:tempfile"
```

The final block command is **SB** (Show Block). This command helps you identify the currently marked block by moving its text to the top of the screen.

In Release 2, you also can copy blocks of text in ED by using the console copy-and-paste feature. Select a block of text by moving the mouse pointer to the beginning of the block, holding down the left button, and moving the mouse pointer to the end of block. Let go of the button, and you will see the block that you've selected is highlighted. Hold down the right Amiga key and press the letter C. This copies the text to the clipboard device. Then, move the cursor to the place where you want the text pasted, hold down Right Amiga, and press V. This inserts the text at the current cursor position.

File Management (Save/Load/Exit)

Versions of Ed prior to AmigaDos Release 2 have no Load command, since you must specify a file to edit when you start the program. However, you can insert text from a disk file within the current text file with the **IF** (Insert File) command. When you type

```
*IF "filename"
```

filename is inserted under the current line, and the rest of the text in the document is moved down.

ED won't let you start editing a file which contains binary (nontext) characters. If you try this, ED ends with the message *File contains binary.* It's interesting to note, however, that in some versions of ED, you may start by editing a blank file, then use IF to merge a file which does contain such characters. This isn't recommended, however, as such characters don't appear correctly on the screen, making it hard to do accurate editing.

The **SA** command is used to save a current copy of the document to disk. If you don't add a filename, the document is saved to the filenamed when you started ED. It's recommended that you periodically save your work to disk (every half hour or so is best) to protect yourself against the perils of power outages. Speaking of backups, you should be aware that ED creates a backup of your original text file in the T directory of the document disk (or T: if assigned), in a file called ED-Backup.

If you use the **SA** command with a filename, you can save a copy of the current document to a file other than the one named when you started the program. This allows you to keep several copies of the document, each varying slightly. The format for this command is

```
*SA"filename"
```

There are two ways to exit the ED program in version 1.3 or below. The first is with the **Q** (Quit) command. Q just quits, without saving your text. If you try the **Q** command after you have changed the text of the document, however, without saving these changes, you'll receive a prompt saying *Edits will be lost - type Y to confirm:*. This gives you an opportunity to save the changes—press any key and the quit is stopped. If you press Y, however, the program ends without saving the changes.

The other way to exit ED is with **X** (eXit). **X** both saves the current document and exits the program. Think of it as first performing an **SA**, then a **Q** command.

Release 2 adds a few additional file-management commands. It lets you exit by clicking on the window's close gadget, or by using the **XQ** (eXit with Query) command, both of which exit unless the file has been changed since the last save, in which case a requester appears asking if you really want to exit without saving. It also allows you to clear the existing file and create a new file with the **NW** command (if the file has been changed the message *Edits will be lost–type Y to confirm* allows to preserve the present contents by pressing any other key). The Release 2 ED also allows you to load a file in place of the one you are currently editing, with the **OP** (OPen file) command. As with Save, you can specify the name of the file in a delimited string that appears right after the OP command:

```
*OP"filename"
```

Or, by adding a question mark after the **OP** command, followed by a string of text to appear in the requester window, you can bring up a file requester that allows you to choose from a list of existing files:

```
OP ? "Open a File"
```

Tabs and Margins

The **SL** (Set Left) and **SR** (Set Right) commands are used to set left and right margins. As explained above, the right margin is used for the purpose of word-wrapping. This means that as you add characters to the end of a line and force it over the right margin, a RETURN character is inserted and the word past the margin is moved to a new line below. Word-wrapping occurs only when you add characters to the end of a line. If you insert characters in the middle of a line, you can cause the end of the line to go past the margin without wrapping. If you wish

to disable the word-wrapping feature for the current line, use the **EX** (EXtend margin) command. This works like the margin release on a typewriter, allowing you to add characters to the end of the line past the right margin. The **EX** command extends the margin only for the current line, however. If you wish to extend the margin permanently, change the right margin setting from its default value of 77.

You can also set a left margin with the **SL** (Set Left margin) command. The default setting is 1 (the leftmost column). When you change this setting, each new line begins at the position indicated. The preceding character positions will be filled with space characters. This left margin is not a hard margin. You don't have to use the **EX** command to move past it. You may use the backspace character to move to the left of it. The **CS** command moves you back to column 1 as well.

The **ST** (Set Tabs) command is used to set the distance between tab stops. The default setting is a stop every three characters.

Miscellaneous Commands

The **U** command (Undo) gives you a very limited undo capability. When you start to edit a line, ED saves the original contents of the line. As long as you stay on that line, you can restore its contents by issuing the **U** command. However, once you move off that line, you cannot undo the changes. Moreover, U cannot restore a line once you remove it completely, either with the immediate command CTRL-B or with the D command.

The **SH** (SHow information) command displays information about the current editing session. When you use the **SH** command, a number of lines appear at the top of the editing screen, showing the name of the file you're editing, the tab setting, the left and right margins, the first and last few characters in the block (if any is marked), and the percentage of the edit buffer that's filled. This display disappears as soon as you type a character.

The **SM** (Status Message) command is new for Release 2. It allows you to print a text message on the status line at the bottom of the screen. This is used mainly to allow command or AREXX scripts to furnish prompts or messages to the user.

Multiple and Repeat Commands

When in extended command mode, you're not limited to issuing one command at a time. Several commands may be placed on the same command line, separat-

ed by semicolons. For example, if you want to search for the first occurrence of the word Amiga in a text file, you could use the command sequence

```
*T;F"Amiga"
```

which moves the cursor to the top of the document, then starts the search. In addition, you can specify that a command should be repeated a number of times by placing that number in front of the command. For instance, the command

```
*4D
```

deletes four lines in a row, starting with the current line. You can also use the special repetition command RP to specify that you want the command repeated until an error occurs. Let's say that you frequently misspell the word separate as seperate. If you want to change every occurrence of the word seperate to separate, you could use the following command series:

```
*T;RP E"seperate"separate"
```

The first command, T, moves the cursor to the top of the document. The next command, RP, specifies that you want to repeat the following sequence until an error occurs. Finally, the E command causes the second string to be exchanged for the first. The result of all this is that ED searches for seperate and replaces it with separate until it can't find the string any longer. When that happens, an End of file error is issued, which causes RP to stop. (Notice that you do not separate RP and the following command to be repeated with a semicolon.)

You can stop any command or series of commands by pressing any key. ED always exits the extended command mode as soon as you press a key, and displays the message *Commands abandoned* on the command line.

Using a repetition count or the RP command only repeats the very next command on the line. Sometimes, however, you need to repeat a whole series of commands. Let's say that you're editing a double-spaced file, in which every other line is blank, and you wish to delete all blank lines. One strategy would be to position the cursor at the top of the file (assuming it's not a blank line), then repeatedly move the cursor to the next line and delete it. You could try the command

```
*T;RP N;D
```

but this wouldn't work. The cursor first moves to the top, but only the N command repeats so that the command just moves the cursor to the last line of the file where it encounters an End of file error.

To counter this problem, ED allows you to group commands together in parentheses. When you do this, the repetition count applies to all of the commands enclosed in the parentheses. Thus, the command

```
*T;RP (N;D)
```

does just what you want. It moves the cursor to the top of the document and then repeats both the N and D commands, again and again.

Keyboard Macros and Custom Menus

The ED program that comes with the Release 2 Workbench has two important features that are not found in previous versions. The first is pull-down menus. ED supports menus that allow beginners to save and load files without first learning the necessary commands. ED also allows the user to customize those menus, to have them provide whatever functions he likes. Second, ED now allows the user to assign extended mode commands to function keys and control-key combinations. This means that you can redefine the keys used by the immediate mode commands, or create your own.

Normally, when you run ED, there is a set of pull-down menus you can access by holding down the right mouse button and moving the pointer to the menu bar. This set of menus is generated by commands that are issued by the S:Ed-Startup file, the command file that is run automatically when you start ED (see the section on command files below). If you delete the S:Ed-Startup file that comes on your Workbench disk, ED starts with an expanded set of default menus.

ED provides two commands to let you custom tailor your own set of menus. The **SI** command lets you define each menu heading and menu item, and the **EM** command enables the menus you defined. These commands can be given from immediate mode, from the S:Ed-startup file, or from another command file. The format for the **SI** command is:

```
SI itemnumber type "Heading Text" "Command"
```

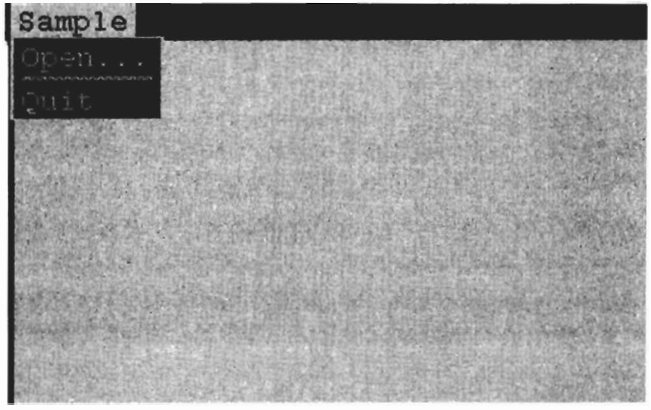
where *itemnumber* is a number from 0 to 120 which specifies the position of the menu heading or item. The *type* is a code which specifies what kind of entry we're describing. Varying types require differing amounts of additional information. The types are:

Type Number	Meaning of Type Number	Additional Text Strings Required
0	End of Menus	No additional text
1	Menu Heading	Heading Text only
2	Menu Item	Heading Text and Command Text
3	Submenu Heading	Heading Text
4	Separator Bar	No additional text

The End of Menus and Separator Bar entries do not require any additional information. For Menu Headings and Submenu Headings you must add the text that will be shown on the menu. In addition, Menu Items require an extended mode command string to be issued when the menu is selected. The commands required to generate a simple example menu look like this:

```
SI 0 1 "Project"
SI 1 2 "Open..." "op ? /Open File/"
SI 2 4
SI 3 2 "Quit" "q"
SI 4 0
EM
```

The menu that these commands generate looks like this:



If you have a problem getting your menu's to appear in the bar, you should remember two points. First, your menus must have an entry whose type number

is zero, and that entry must have the highest itemnumber. Second, none of your SI commands go into effect until the EM command is given.

Another way in which ED allows you to customize the program is by assigning extended mode command functions to the function keys or special key combinations. This lets you create your own immediate mode commands, as well as modifying or deleting the default immediate mode commands. You define command keys by using the **SF** (Set Function key) command. The format for this command is:

```
SF KeyNumber /Extended Mode Command String/
```

The KeyNumber is used to identify the key or key combination used to execute the command. The **SF** command recognizes 57 key numbers, of which four are reserved for future use. Numbers 1 through 10 are used for the function keys F1 to F10, while numbers 11 to 20 are used for the Shifted function keys. Key numbers 27 to 52 are used for CTRL-A through CRTL-Z. When defining a control key combination, you can use the caret symbol (^) plus the letter instead of the number. For example, you can assign the Delete function to the CTRL-B key combination like this:

```
SF 28 /D/
```

or like this:

```
SF ^B /D/
```

The command string part of the **SF** command consists of any valid extended mode command line, in delimited string format. You can even use multiple commands in a line, separated by semi-colons. You should be careful with string delimiters, however, since some commands also require the use of strings which must have their own distinct delimiters. If you want to create a key combination that saves the present file under the name *RAM:Sub/File,* you could not use the command:

```
SF ^Z /SA/RAM:Sub/File//
```

You'll need to use separate sets of delimiters for the SF and SA commands, and in neither case could you use the slash character, since it is part of the filename. You can correct the above command by substituting quotes for one pair of slashes, and exclamation marks for the other, like this:

```
SF ^Z "SA!RAM:Sub/File!"
```

There are two additional commands dealing with command keys. The **DF** (Display Function key) command displays the command string attached to a

particular key. For example, DF 28 would normally display the letter D on ED's status line, to show that the CTRL-B key combination calls the extended mode Delete command. The **RK** (Reset Keys) command can be used to reset all of the key definitions to their default values. The chart on page 131 shows both the key combinations and the default extended mode command assigned to each key number.

Editing under Program Control— Script Commands and AREXX

Another feature that was added to the AmigaDOS Release 2 version of ED program was support for executing a sequence of commands, either from a command script or an AREXX program. The concept of a command script is very similar to that of an AmigaDOS script. Just as an AmigaDOS script is a text file which contains a series of AmigaDOS commands, an ED command script is a text file which contains a series of ED extended mode commands. When you run an ED command script, each command is executed as if it were typed in at ED's command line, just as when an AmigaDOS command file is EXECUTEd, each DOS command is run as if it were typed in. Finally, if there is a file in the S: directory called Ed-Startup, that command file is automatically executed when the ED program starts, just as the S:Startup-sequence file is run automatically when you turn the computer on, or the S:Shell-Startup file is run when you start a new Shell process. This file is often used to set up menus and keyboard macros automatically.

There are actually three ways of executing an ED command file. As indicated above, the *S:Ed-Startup* file is run automatically unless otherwise specified. To execute another file when ED begins, you may use the WITH option of the command. If you start ED with the command

```
ED S:Script WITH S:CommandFile
```

then *S:CommandFile* will be executed instead of S:Ed-Startup. Since ED scripts can use all of the extended mode commands, including ones that quit the program, it is possible to run ED with a command script that opens a file, performs several editing operations, saves the file, and then exits the ED program, all without any user intervention at the keyboard.

The third way of executing an ED command script is from within the ED program itself, using the extended mode **RF** (Run File) command. The command requires a text string after it, specifying the name of the command file to run (for example **RF S:CommandFile**).

Key Number	Key or Combination	Default	Command
1-10	F1-F10	None	
11-20	Shift F1-F10	None	
21	Shift left arrow	CS	Cursor to start of line
22	Shift right arrow	CE	Cursor to end of line
23	Shift up arrow	T	Cursor to top of file
24	Shift down arrow	B	Cursor to bottom of file
25	Del	DC	Delete character under cursor
26	Reserved	None	
27	CTRL-A	A//	Insert Line
28	CTRL-B	D	Delete Line
29	CTRL-C	None	
30	CTRL-D	PD	Page Down (cursor down 12 lines)
31	CTRL-E	EP	Cursor to top or bottom of screen
32	CTRL-F	FC	Flip Case
33	CTRL-G	RE	Repeat last extended command sequence
34	CTRL-H	DL	Delete character to left of cursor
35	CTRL-I	TB	Move cursor to next tab position
36	CTRL-J	None	
37	CTRL-K	None	
38	CTRL-L	None	
39	CTRL-M	S	Split line (carriage return)
40	CTRL-N	None	
41	CTRL-O	DW	Delete Word or spaces
42	CTRL-P	None	
43	CTRL-Q	None	
44	CTRL-R	WP	Cursor to end of previous word
45	CTRL-S	None	
46	CTRL-T	WN	Cursor to start of next word
47	CTRL-U	PU	Page Up (cursor up 12 lines)
48	CTRL-V	VW	Re-display window
49	CTRL-W	None	
50	CTRL-X	None	
51	CTRL-Y	EL	Delete to end of line
52	CTRL-Z	None	
53	CTRL-[	CM	Enter command mode (escape)
54	Reserved	None	
55	CTRL-]	CT	Move to start or end of line
56	Reserved	None	
57	Reserved	None	

ED's AREXX support provides a flexible way of controlling the editor by a program. Each time you run the ED program, it opens up a message port through which it can communicate with AREXX scripts, or even other application programs. The name of this message port for the first copy of ED you run is ED, but each subsequent copy that you run takes a slightly different port name so that you can run multiple copies of ED and communicate with each individually. The second copy of the ED that you run has a port named ED_1, the third ED_2, and so on.

When you run an AREXX script from within ED, you can determine the port name of the current copy of ED with the AREXX address() function. For example, if you execute an AREXX script from the second copy of ED which contains the line ***HOST=address()***, the value ED_2 is placed in the variable HOST.

External programs can send any extended mode command to ED through its AREXX port. For example, you can send the cursor to the top of the file loaded into the first copy of ED with the following command line in an AREXX script:

```
address 'Ed' 't'
```

In addition to sending commands to ED, AREXX programs can also obtain information from ED about current ED settings and even the contents of the current text file. This is accomplished by sending ED the special **RV** (Rexx Variables) command, along with the stem name of the compound variable in which to store the information. For example, if an AREXX program sends the first copy of ED an RV command with the program line

```
address 'Ed' 'RV/Ed_status/'
```

ED will assign the following values to fifteen variables:

Variable Name	Variable Contents
Ed_Status.LEFT	Current left margin (1 is left edge), from SL command
Ed_Status.RIGHT	Current right margin, from SR command
Ed_Status.TABSTOP	Current tab stop, from ST command
Ed_Status.LMAX	Line number of last visible line on screen
Ed_Status.WIDTH	Width of screen in characters
Ed_Status.X	Physical X position of cursor (1 is left edge)
Ed_Status.Y	Physical Y position of cursor (1 is top line)
Ed_Status.BASE	Window base offset (non-zero when screen scrolls to right)
Ed_Status.EXTEND	Extended margin flag 0 = normal margins 1 = margins extended with EX command
Ed_Status.FORCECASE	Case sensitivity flag 0 = Search is case sensitive (LC)
Ed_Status.LINE	Current line number in file (1 is first line)
Ed_Status.FILENAME	Name of the text file currently being edited
Ed_Status.CURRENT	Text of the line on which the cursor is positioned
Ed_Status.LASTCMD	Last extended command that was issued
Ed_Status.SEARCH	Text of last search string

Using this information, it is possible to read in an ED text file line by line into an AREXX program, and use both ED and AREXX string commands to manipulate it.

It is possible to launch an AREXX program directly from ED, using the RX extended command. The format for RX is:

```
RX /ArexxFile/
```

where ArexxFile is the name of the script file to execute. If you create this file in the REXX: directory, and end it in the letters .ed, you can easily identify it as an ED script, and you do not have to refer to the pathname or extension in the RX command. If you have a file called Script.ed in your REXX: directory, for example, the extended command ***RX /Script/*** will be sufficient to run that AREXX program. You can even attach the RX command to a function key combination, to create very complex new immediate mode commands.

ED Command Summary

Keyboard Commands

Cursor Movement

Cursor keys	Move cursor one character up, down, right, or left
Shifted Cursor keys	Move cursor to top or bottom of file, or start or end of current line (Release 2 or above only)
TAB, CTRL-I	Move cursor right to next TAB position (no characters inserted into text)
CTRL-T	Moves cursor to start of next word
CTRL-R	Moves cursor to end of previous word
CTRL-]	Moves cursor to end or start of line (alternates)
CTRL-U	Scrolls text up (moves cursor down) a page
CTRL-D	Scrolls text down (moves cursor up) a page
CTRL-E	Moves cursor to top or bottom of screen (alternates)

Insert/Delete

BACKSPACE, CTRL-H	Delete character to left of cursor
DEL	Deletes character under cursor
CTRL-O	Deletes next word or spaces before next word (alternates)
CTRL-Y	Deletes to end of current line
CTRL-B	Deletes entire current line
CTRL-A	Inserts a new line below current line

Miscellaneous Commands

CTRL-F	Flips case of character under cursor (and moves cursor one character to the right)
CTRL-V	Verifies (redraws) the screen
ESC, CTRL-[	Enter extended command mode
CTRL-G	Repeats last extended command

Extended Mode Commands (Press ESC to Enter Command Mode)

Cursor Movement

CL	Moves cursor left one character
CR	Moves cursor right one character
N	Moves cursor to start of next line
P	Moves cursor to start of previous line
CS	Moves cursor to start of line
CE	Moves cursor to end of line
T	Moves cursor to top of file
B	Moves cursor to bottom of file
EP	Moves cursor to top or bottom of screen (alternates) (Release 2 or above only)
PD	Moves cursor down 12 lines (Release 2 or above only)
PU	Moves cursor up 12 lines (Release 2 or above only)
WN	Moves cursor to start of next word (Release 2 or above only)
WP	Moves cursor to space after previous word (Release 2 or above only)
TB	Moves cursor to next tab position (Release 2 or above only)
M*linenum*	Moves cursor to line number linenum

Insert/Delete

DC	Deletes character under cursor
D	Deletes entire current line
DL	Deletes the character to left of the cursor (Release 2 or above only)
DW	Deletes to the end of the current word (Release 2 or above only)
EL	Deletes to the end of the current line (Release 2 or above only)

FC	Switches the case of the current character (Release 2 or above only)
I/*string*/	Inserts string as a new line above current one
A/*string*/	Inserts string as a new line below current one
S	Splits current line at cursor position (same as RETURN)
J	Joins current line with next line (deletes RETURN at end of current line)

Find and Exchange (Search and Replace)

F/*string*/	Finds string in following text (forward search)
BF/*string*/	Backward find (searches previous text for string)
E/*string1*/*string2*/	Exchanges (replaces) string1 with string2
EQ/*string1*/*string2*/	Exchanges (replaces) after query string1 with string2
LC	Requires searches to match both uppercase and lowercase
UC	Ignores case differences in searches

Block Transfers

BS	Marks a block starting at start of current line
BE	Marks a block ending at end of current line
DB	Deletes current block
IB	Inserts copy of the block below current line
WB/*filename*/	Writes the block to file filename
SB	Shows the block onscreen

Save/Load/Exit

IF/*filename*/	Inserts file filename at the cursor (and moves rest of file down)
SA/*filename*/	Saves file to disk (to filename if given; if not, to current file)
X	Exits, saving text file to disk
Q	Quits without saving text
XQ	Exits unless changes have been made to file (same as clicking close gadget) (Release 2 or above only)

NW	Creates a new file, replacing the existing one (Release 2 or above only)
OP/*filename*/	Opens the file filename (OP ? // presents file requester to choose name)
RF/*filename*/	Loads and executes the commands in filename (Release 2 or above only)

Tabs and Margins

SLcolnum	Sets left margin to column number colnum
SR*colnum*	Sets right margin to column number colnum
EX	Extends right margin
ST	Sets distance between tab stops

Miscellaneous

SH	Shows information on filename, tab stops, margins, block markers, and buffer usage
U	Undoes changes to current line
;	Executes another command on same command line number
number	Repeats following command number times
( )	Groups commands for purpose of repetition
RP	Repeats following command until an error occurs
SI *Num* Type Head CMD	Defines menu headings and items (Release 2 or above only)
EM	Enables user-defined menus (Release 2 or above only)
SF*Keynum/Command*/	Assigns a command to immediate mode command key specified by keynumber (Release 2 or above only)
DF*Keynumber*	Displays the command string assigned to Keynumber (Release 2 or above only)
RK	Resets the immediate mode command keys to default values (Release 2 or above only)
RX/*RexxFile*/	Executes the AREXX script named RexxFile (Release 2 or above only)

Chapter 7

EDIT, the Line Editor

EDIT, the AmigaDOS line editor, can be used to inspect and change just about any kind of AmigaDOS file, including text and program files. EDIT only operates on one line at a time, instead of presenting a whole screen full of data. Unlike the newer graphic-oriented editors, EDIT is completely command-driven. It has no support whatever for the Amiga's mouse-and-menu interface.

So why use EDIT? AmigaDOS comes with a fairly powerful full-screen editor (ED), and reasonably priced word processors for the Amiga are also available. While most Amiga owners will prefer either of the latter, some users will find using EDIT comfortable. There are, after all, some users who actually prefer typing in commands to using a mouse. For those who have mastered its few commands, EDIT is probably as quick and easy to use as anything else.

EDIT does have two features which ED, its more powerful sibling, does not—it can be used to modify program files as well as text files, and it can execute a series of stored commands from a disk file (a capability which was added to the Release 2 version of ED, further reducing the incentive for using EDIT).

How EDIT Works

EDIT processes the contents of a source file (we'll call this EDIT's *From* file) sequentially—a line at a time—using editing commands specified by the user.

EDIT keeps track of its place within the material being edited. When EDIT is first invoked, the *current line* is the first line of the *From* file. As editing commands are executed, the current line changes. EDIT keeps tabs on the current line by maintaining an internal pointer called the *current line marker.*

As the current line marker is moved past a line, the line is moved into a special area called the *output buffer.* The output buffer has a fixed size for the duration of an EDIT session.

When the output buffer becomes filled, data is written to the file specified as the destination (EDIT's *To* file), on a first-in, first-out basis.

During an EDIT session, various informative messages and displays of the contents of lines are sent to EDIT's *verification device* (your Amiga's screen, unless another device is specified when EDIT is started up).

If EDIT's *To* file is different than its *From* file, the contents of the file used as input to the editor will not be altered. If the *To* file is the same as the *From* file, the original contents of the file will be moved to a temporary file called *:t/edit.backup*.

Invoking EDIT

An EDIT session is usually started from an active CLI by using AmigaDOS's EDIT command. What follows is a summary of the EDIT command's syntax. See the EDIT command section in the "AmigaDOS Command Reference" for a more detailed explanation.

EDIT [*FROM*] *fromname* [*TO*] *toname* [*WITH*] *withname* [*VER*] *vername* [*OPT option*]

EDIT's Parameters and Keywords:

FROM *fromname*—The name of the file whose contents will be edited. Throughout the rest of this chapter, this is referred to as EDIT's *From* file.

TO *toname*—The name of the file which will contain the edited text after the EDIT session is ended. Throughout the rest of this chapter, this file is referred to as EDIT's *To* file.

WITH *withname*—Lets you specify a file that may optionally be used as input to the line editor's command processor.

VER *vername*—Lets you specify where you want messages from EDIT to be displayed.

OPT P*n* or **OPT W***n* or **OPT P***n***W***n*—These options let you set the maximum line length (W*n*) and/or number of lines (P*n*) that EDIT will keep in its output buffer. In versions 1.3 and greater, the keyword PREVIOUS can be used instead of OPT P, and the keyword WIDTH can be substituted for OPT W.

While you can edit files with more lines than the value of *n*, you'll only be able to move backward n lines. If the file to be edited is not unreasonably large, it's usually a good idea to specify an n greater than the number of lines in the file to be edited. The default number of previous lines is 40, and the default line width is 120.

Starting an EDIT Session—Examples

Example 1—Edit a file called *mysource* in the current directory, using EDIT. The edited data is to be stored under the same filename. The number of lines is to be set to 40 and line width to 120 (EDIT's default values):

```
EDIT mysource
```

Example 2—Edit a file called *bigsource.* The edited data will be stored in the file called edited *bigsource*. The output buffer size is set to 1000 lines, with a maximum line width of 120:

```
EDIT bigsource "edited bigsource" OPT P1000
```

Example 3—Edit a file called universe. When EDIT starts up, execute the list of EDIT subcommands contained in the file *autocommands* located in the *myprocess/nebula* directory on drive df1:. The edited data is to be stored under the same filename. Send all messages and verification displays from the line editor to the system printer. The number of lines in EDIT's output buffer is to be set to 40 and the maximum line width to 250:

```
EDIT universe WITH df1:myprocess/nebula/auto commands
    VER PRT: OPT W25 0
```

Note: EDIT expects the *From* file to exist already. Issuing an edit for a file called *newfile* which doesn't exist, as in

```
EDIT FROM newfile
```

generates the error message *Can't open newfile.* However, you can use EDIT to type in a file by creating an empty file first and then editing the new file:

```
COPY * newfile
```

Press CTRL-\ after entering this command.
NOTE: CTRL-\ is the Amiga Equivalent of the PC's CTRL-Z.

```
EDIT newfile
```

Example 4—Let's create a sample file which you can type in, then experiment with using EDIT during the rest of this chapter. Type:

```
COPY * testfile
The door slammed and she stormed
out of the house. Meanwhile, the
toast burned and the eggs hardened.
He stared after her, wondering what
to say. Fortunately, he kept his mouth
shut. Better to say nothing than to
say something now.
```

Once you've typed this in, press CTRL-\ (end-of-file marker), which will close the file; *testfile* is now on your disk. You can access it by entering

```
EDIT testfile
```

Getting Out of EDIT

There are several ways to exit an EDIT session.

The **STOP** command exits EDIT, leaving the *From* file intact. The contents of the *To* file, if a separate one was specified, are unpredictable since STOP will not write the contents of the output buffer to the *To* file as it exits EDIT.

The **W** command (Windup) advances the current line marker to the *From* file's end-of-file (EOF) marker, moving lines into the output buffer as it goes. When the EOF is reached, EDIT saves the contents of the output buffer to EDIT's *To* file, and the editing session terminates.

The **Q** command (Quit) is used within EDIT command files to return control to the process which invoked the file's execution. If Q is issued from EDIT's primary command level, it has the same effect as W. (See the section "EDIT Command Files" later in this chapter for more information about the use of command files.)

The Current Line

EDIT keeps track of its place within the data and/or text being edited. When EDIT is first invoked, the current line is the first line of the *From* file. As EDIT subcommands are executed, this current line changes. EDIT keeps tabs on the current line by maintaining the current line marker, an internal pointer.

At the beginning of each session, EDIT associates sequential line numbers with all of the original lines of the *From* file. When EDIT begins, the current line is line number 1.

Verifying the Current Line

The **?** and **!** commands allow you to display the line number (if any) and contents of the current line.

```
?
```

displays the line number and contents of the current line.

Characters which cannot be displayed can be represented by a question mark. For instance, if issuing a ? command results in a display of

```
?
5.
Whom do you trust???
```

the question marks which appear to be a part of line 5 may not be question marks at all. In these cases, the ! command will display the hexadecimal value of the characters in question:

```
!
Whom do you trust? 11
-                  03
```

The exclamation mark (!) revealed that there's only one genuine question mark in the line, followed by characters whose ASCII values are 10 and 13. The ! also displays a dash (-) under any uppercase letters contained in the current line.

Turning Verification On and Off

EDIT often displays a verification of the line number and contents of the current line in response to many EDIT commands. If the current line has no line number, +++ will be displayed instead. Verification displays may be turned on and off with the V (Verify) command.

```
V-
```

turns off automatic line verification, while

```
V+
```

turns verification on. Verification is always set to on by EDIT when an editing session begins.

Trailing Spaces

EDIT normally suppresses all trailing spaces.

```
TR+
```

turns EDIT's trailing spaces switch on, allowing trailing spaces on both input and output lines.

```
TR-
```

reinvokes suppression of trailing spaces (EDIT's default).

Operational Windows

When a command is executed which instructs EDIT to operate on the current line, EDIT normally scans all the characters in the line from left to right, beginning with the first character.

It's possible to instruct EDIT to begin its scan at a character other than the first in the line. The current line's *operational window* consists of only that portion of the line which will be operated on. The beginning of the current line's operational window is pointed to by the *operational window pointer.*

> moves the operational window pointer of the current line one character to the right.

< moves the operational window pointer of the current line one character to the left.

PR (Pointer Reset) sets the operational window pointer line back to the start of the line.

Whenever EDIT is instructed to display verification of the current line (by ?, !, or any other command which normally ends with a verification of the current line), a greater than (>) character may be displayed under the contents of the current line. Everything to the right of the > is within the current line's operational window. For instance,

```
3.
Well this is another fine mess
           >
```

indicates that the operational window pointer of the current line has been moved so that the operational window of the line consists of the text *another fine mess.* If you told EDIT to search for the word *this,* it would not be found, since only the contents of the current line's operational window are scanned by the search operation.

Character Operations on the Current Line

EDIT supports four intraline commands which can change the case of characters, replace characters with a blank, and delete characters:

$ (dollar sign) forces the case of the first character in the current line's operational window to lowercase. After a $ command is executed, the operational window pointer is moved one character to the right.

% (percentage sign) forces the case of the first character in the current line's operational window to uppercase. After a % command is executed, the operational window pointer is moved one character to the right.

_ (underscore) forces the first character in the current line's operational window to be replaced by a blank. After an _ command is executed, the operational window pointer is moved one character to the right.

(pound sign) deletes the first character in the current line's operational window. The text remaining in the operational window is shifted one character to the left.

The intraline commands may be strung together on a single EDIT command line. Take a look at the following example.

Assume you start with the current line as *All the young dudes, carry the NEWS.* Several operations can be carried out on this line to change its appearance:

```
1.
All the young dudes, carry the NEWS
%%%####
1.
ALL young dudes, carry the NEWS
  >
>>>>>>>%%%%>>
1.
ALL young DUDES, carry the NEWS
                >
>>>>>>____>$$
1.
All young DUDES, carry  NewS
                             >
```

You could have strung all the commands in the previous example together on one command line. Several command lines were used to keep things from getting totally confusing.

Moving from One Line to Another

N (Next line). The current line marker can be moved forward by using the N command. If you attempt to move the current line marker past EDIT's end-of-file flag, the message *Input exhausted* displays, and the current line marker is set at the end-of-file flag.

```
N
```

moves the current line marker to the next line of the current *From* file. If line verification is on, the line number and text of the new current line is displayed. The current line marker may be moved ahead multiple lines by stringing multiple N commands on a single line or by preceding the command with a number:

```
N;N;N;N
```

is the same as

```
4N
```

P (Previous line). The current line marker may be moved backward with the P command. If you attempt to move the current line marker past the first line contained in EDIT's output buffer, the message *No more previous lines* displays, and the current line marker is set to the first line in the output buffer.

Remember, the default capacity of EDIT's output buffer is only 40 lines. For example, if you EDIT an 80-line file and the current line marker is pointing to line 60, EDIT's output buffer contains only lines 20 through 59 of the *From* file. Attempting to back up 40 or more lines results in the current line marker pointing to line 20.

```
P
```

moves the current line marker back one line. If verification is on, the line number and text of the new current line is displayed. The current line marker may be moved back multiple lines by stringing multiple P commands on a single line or by preceding the command with a number.

```
P;P;P;P;P;P
```

is the same as

```
6P
```

Both move the current line marker back six lines.

Ma (Move to line *a*). The current line marker may be moved backward or forward to a specific line number by using the M command. Using a period (.) in the a location, the current line marker is moved to the end-of-file flag of the *From* file.

If you attempt to move the current line marker back to a line number not in EDIT's output buffer, the message *Line number a too small* displays. If the line number specified is greater than the highest line number of the *From* file, the message *Input exhausted* displays, and the current line marker is set to the end-of-file flag.

```
M17
```

moves the current line marker to line number 17 (not the seventeenth line). If verification is on, the line number and text of the new current line is displayed.

```
M.
```

moves the current line marker to the end-of-file flag of the *From* file.

Displaying Your Text

It's often handy to examine the contents of more than one line at a time. EDIT has four commands which allow you to display multiple lines of the file being edited.

Tn (Type *n* lines). The T command displays *n* lines on the screen (or verification device, if one other than the screen has been selected), beginning with the current line. The current line marker is set to the line following the last one typed by the T command. If *n* is not specified, all lines following the current line are displayed, and the current line marker is set to the end-of-file flag.

Assume that the sixth line of a file being edited is the current line. The command

```
T5
```

displays the sixth through tenth lines of the file, and the current line marker is changed to point to the eleventh line. The line number and text of the new current line are displayed.

If the current line marker of a 100-line file is pointing to the thirty-eighth line, and the current line marker is set to the end-of-file flag,

```
T
```

displays lines 39 through 100, and the current line marker is set to the end-of-file flag.

TP (Type Previous). The TP command displays the current contents of EDIT's output buffer. If the buffer is full, the current line marker remains unchanged. If the output buffer is not full, the TP command advances the current line marker to the point which fills the buffer and then displays the contents of the buffer, followed by verification of the new current line.

For example, assume that EDIT has been invoked with the default buffer length of 40 lines and that you're editing a 70-line file. If the current line is the tenth line of the file (which means there are only nine lines currently in the output buffer) and you issue

```
TP
```

EDIT changes the current line marker to point to the forty-first line, moving lines 10 through 40 into the input buffer (along with the file's first 9 lines). All 40 lines now residing in the output buffer are displayed, followed by a verification display of the file's forty-first line. Any TP commands issued immediately thereafter will have no effect on the current line marker, since the output buffer has been filled.

TN (Type Next). The TN command acts exactly like a **T*a*** command in which the value of *a* is determined by the number of lines that the output buffer has been set to hold (OPT P*n*). The default value for *a* is 40 if no P*n* is specified in the EDIT command which started the current session.

TL*n* (Type with Line numbers). The TL command displays *n* lines preceded by their line numbers, beginning with the current line. Lines that have been inserted or that are created by splitting a numbered line in two may have no numbers. If a line has no number, EDIT displays three asterisks (***) in its place. The current line marker is set to the line following the last one typed by the T command. If *n* is not specified, all the lines following the current line are displayed, and the current line marker is set to the end-of-file flag.

Inserting New Text

EDIT allows text to be inserted before the current line or any line that may be referenced by a line number. The text to be inserted may be typed in via the keyboard or may be read directly from another AmigaDOS file.

The **I** command, when used in conjunction with a specific or relative line number, allows text to be inserted in EDIT's output stream.

- **I** or **I.** is used to insert text before the current line.
- **I*** inserts text after the last line of text in the *From* file.
- **Ia,** where *a* is the line number that EDIT associates with a given line of the file being edited. An I*a* command may search backward into EDIT's output buffer or forward, past the current line, in search of the specified line number. Once the line number is found, the line associated with it is made the current line.

Insert commands all throw EDIT into *insert mode.* Any text typed at the keyboard will be inserted before the current line (into EDIT's output buffer). Insert mode is terminated by typing a line containing *only* the letter *z* (lowercase or uppercase) in the first column and hitting the RETURN key. The inserted text will have no line number. Upon exiting insert mode, the current line will be displayed—it will be the same line as when the I command was invoked.

Let's try it out. Insert several lines before the current line:

```
I
Well this is a silly little example of how to insert a
couple of lines and then get out of Input's insert
mode.
z
```

Insert several lines before line number 17:

```
I17
Had enough folks?
EDIT can be a barrel of laughs.
z
```

Note: You may change the input mode terminator to any string of up to 16 characters by using the **Z** command. For example,

```
Z/fin/
```

changes the input mode terminator from *z* to *fin;fin* will remain the input terminator through the end of the current EDIT session or until another Z command is issued.

- **I/*filename*/** (Insert before current line from a file)

or

- **I*a/filename/*** (Insert before line a from a file)

Insert also lets you specify an AmigaDOS file as the source for lines to be inserted. Filenames used in conjunction with insert and replace commands are normally delimited by slashes (/), although the colon (:), period (.), comma (,) and asterisk (*) may also be used. Lines inserted from an AmigaDOS file into EDIT will have no line numbers associated with them. Here are some examples.

Insert the contents of the file *mytext* before the current line:

```
I/mytext/
```

Insert the contents of the file *Wow/What a Party* on the external disk drive before line 66:

```
I66 /"DF1:Wow/What a Party"/
```

Replacing Lines with Inserted Text

EDIT also allows lines of text to be replaced by inserted text typed in via the keyboard or read directly from an AmigaDOS file.

The **R** (Replace) commands' syntax is almost identical to that of the I (Insert) commands.

- **R** or **R.** is used to replace the current line with inserted text.
- **R*** inserts text after the last line of text in the *From* file.
- **R*a b*** replaces a range of lines with inserted text; *a* and *b* are line numbers which EDIT associates with specific lines of the file being edited. An I*a b* command may search backward for the specified range of lines into EDIT's output buffer, or forward, past the current line, in search of the specified line numbers. Once the lines are found, the line associated with it is made the current line. If *b* is omitted, only line *a* will be replaced by the inserted text.

Replace commands all throw EDIT into insert mode. Any text typed at the keyboard replaces the line(s) specified. Replace's insert mode is terminated by typing a line that contains only the letter *z* in the first column and hitting the RETURN key. The inserted text will have no line numbers associated with them. Upon exiting insert mode, the current line will be displayed—it will be the first line following the last replaced line.

As with the I command, you may also replace text from an AmigaDOS file.

Replace the current line with the phrase One for the Money:

```
R
One for the Money
z
```

Replace line 13 with several lines of text entered from the keyboard:

```
R13
I am Gosar, the Gosarian, keymaster
of Zuuul.
And many were those who knew what it
was to roast in the depths of the Slor
that day, I tell you
z
```

Replace lines 3-67 with the text contained in the AmigaDOS file *morestuff/edit:*

```
R3 67.morestuff/edit.
```

Renumbering Lines

As has been pointed out, EDIT normally assigns line numbers only when a *From* file is opened. Inserted text has no automatically associated line numbers. The renumber command (=) may be used to assign a line number to the current line and each line that follows it when the renumber is issued.

```
=10
```

renumbers the current line and all lines following. The current line is assigned a line number of 10.

If the file being edited contained three lines, numbered 1 through 3, and line 1 was the current line, =1 0 would change the line numbers to 10 through 12. Any line numbers associated with lines in EDIT's output buffer are lost.

Searching for Text

F/string/ (Find text). The find command searches for a specified text string beginning with the current line and proceeds forward through the lines of the *From* file until the text is found or until the end of the *From* file is reached. The search operation stops at the first occurrence of *string,* and the current line marker is updated to make the line containing the found string the current line. If

verification is on, and the line containing the match is other than the original current line, the line number and contents of the new current line are displayed. If the search string is not found, the message *Input exhausted* displays, and the current line marker is set to the end-of-file flag.

If no search string is specified in an F command, EDIT attempts to use the search argument of the last find command issued. If no previous find command has been issued, the error message *Nothing to repeat* appears.

String expressions used for search (and replace) operations within EDIT are normally delimited by slashes (/), although the colon (:), period (.), comma (,) and asterisk (*) may also be used. EDIT searches are case sensitive. The search string *AmigaDOS* does not match the text *amigados.*

Here's an example—find the string *disk.* Begin the search with the current line and move forward through the *From* file:

```
F/disk/
```

BF/string/ (Backward Find text). The BF command searches for a specified text string beginning with the current line and proceeds backward through the previous lines contained in EDIT's output buffer until the text is found or until the front end of the output buffer is reached with no match. The search operation stops at the first occurrence of *string,* and the current line marker is updated to make the line containing the found string the current line. If verification is on, and the line containing the match is other than the original current line, the line number and contents of the new current line are displayed. If the search string is not found, the message *No more previous lines* appears, and the current line marker is set to the line which was at the head of the output buffer.

If no search string is specified in a BF command, EDIT tries to use the search argument of the last find command issued. If no previous find command has been issued, the error message *Nothing to repeat* displays.

Let's try one. Find the string *disk.* Begin the search with the current line and move backward through the output buffer:

```
BF/disk/
```

Find Command Qualifiers

There are five *qualifiers,* or options, which may be used in conjunction with the find and backward find commands to further restrict the conditions that will result in a search match.

The **F** and **BF** commands normally don't care where in a line the search string is found. The **B** and **E** qualifiers let you specify whether the text must begin a line (B) or end a line (E).

The **P** qualifier allows you to restrict matches to those lines which consist of nothing but the precise (P) text specified by the search string.

EDIT's searches normally proceed rightward from the first character of each line. The L qualifier instructs EDIT to search each line leftward (L) beginning with the last character of each line.

The B, E, P, and L qualifiers are *mutually* exclusive. EDIT does not allow any of these four qualifiers to be specified together in an F or BF command.

The *U* qualifier may be used by itself or in conjunction with any of the other four. U renders the search string case insensitive—it causes EDIT to treat both the search string and searched text as if everything were in uppercase (U). A few examples follow.

Search forward, beginning with the current line, for the line which ends with the words *Natasha Fatale:*

```
F E/Natasha Fatale/
```

Search backward, beginning with the current line, for the line that begins with *WayBack*

```
BF B/WayBack/
```

Search forward, beginning with the current line, for the line that is precisely *Into the valley of death, rode the six hundred.*

```
F P/Into the valley of death, rode the six hundred./
```

Search backward for the phrase *I can play CenterField.* Each line is to be searched leftward, beginning with the last character of each line. The case of the search text is to be ignored:

```
BF LU/i can play centerfield/
```

You can also find an empty line (one containing nothing) by specifying a null string as a search argument:

```
F P//
```

Remember, the current line marker is updated to point at the line containing a found string.

Replacing Text

One of the reasons you may want to find a specific text string is so that you can make changes to it. EDIT has three commands which can be used to replace and/or insert text in the current line.

E/*string1*/*string2*/ (Exchange text). The E command lets you exchange a string of text contained in the current line with another string of text. E searches rightward for *string1* in the current line, beginning with the first character of the line. If found, *string1* is replaced by *string2,* and the entire modified line is displayed. If *string1* is not found in the current line, the message *No match displays.* In either case, the current line marker remains unchanged.

More examples—change the phrase too strange to be believed in the current line to too strange to have happened:

```
E/too stange to be believed/too strange to have
    happened/
```

B/*string1*/*string2*/ (insert Before text). The B command inserts a string of text before a specified string contained in the current line. B searches rightward for *string1* in the current line, beginning with the first character of the line. If found, *string2* is inserted immediately before *string1,* and the entire modified line is displayed. If *string1* is not found in the current line, the message *No match* appears. In either case, the current line marker remains unchanged.

A/*string1*/*string2*/ (insert After text). The A command inserts a string of text after a specified string contained in the current line. In all other respects, A functions identically to B.

The Current String Alteration Command

The previous A, B, or E command executed is known to EDIT as the *current string alteration command.* Typing a single quotation mark (') repeats the current string alteration command.

Checking on the Last-Used Search Expression

The **SHD** (SHow Data) command displays EDIT's current saved information values, including the last search expression.

Pointing Variants of Replace Commands

There's a secondary form of the E, B, and A commands which performs text replacement/insertion, and one additional one. This secondary form of each

command is referred to as the *pointing variant of each,* and they are respectively EP, BP, and AP.

If the current line is successfully modified, EDIT's character pointer is left pointing to the first character in the line which follows *string2* in the case of EP or AP, or in the case of a BP command, the first character in the line that follows *string1.*

Using Qualifiers with Replace Commands

The B, E, P, L, and U qualifiers that may be used in conjunction with find commands may also be used with the replace commands and their pointing variants. The effect of using the qualifiers and rules for their use is the same as described in the section entitled "Find Command Qualifiers."

Deleting Text

D (Delete line). The D command can be used to delete the current line, a multiple number of lines (beginning with the current line), a specific line number, or a range of lines delimited by lines having line numbers. After the requested deletion has taken place, the current line marker is advanced to the line immediately following the last line deleted by the operation, and the line number and contents of the new current line are displayed. If D does not find a specified line between the current line and the *From* file's last line, the message *Input exhausted* appears, and the current line marker moves to the end-of-file flag. The D command does not affect the contents of EDIT's output buffer. An example or two might help.

Delete only the current line. The current line marker is moved to the next line in the *From* file:

```
D
```

Delete the current line and the next three lines. The current line marker is moved to the line that was four lines after the original current line:

```
4D
```

Delete line 17. If line 17 is found, the current line marker is moved to the line which follows it after line 17 is deleted. If a line numbered 17 is not found, no deletion will take place, and the current line marker will be updated to point at the end-of-file flag:

```
D17
```

Delete the lines numbered 22, 28, and all lines between them. If line 28 is found, the current line marker is moved to the line which follows it after the requested lines are deleted. If line 22 is found, but a line numbered 28 is not, line 22 and all the lines that follow are deleted. The current line marker is updated to point at the end-of-file flag:

```
D22 28
```

Delete the current line and all the lines that follow. The current line marker is updated to point at the end-of-file flag:

```
D*
```

Delete Commands That Use Search Expressions

DTB/*string1*/ (Delete Text Before)
and
DTA/*string1*/ (Delete Text After)

The DTB and DTA commands let you delete text within the current line that occurs before or after a search expression you specify. DTA and DTB operate only upon the current line. After execution, the line number and new contents of the current line are displayed. If the search expression is not found, the message *No match* is displayed and the current line remains unchanged.

DF/*string*/ (Delete lines until Find). The DF command searches each line, beginning with the current line, for the specified search expression. If the line searched does not contain the search string, it's deleted. The search-and-delete process continues until the search string is found. The line found to contain the search expression becomes the new current line. *If the search string is not found, a DF command deletes the current line and all lines that follow until it reaches the end of the* From *file.*

Using Qualifiers with DTB, DTA, and DF

The B, E, P, L, and U qualifiers used in conjunction with find and replace commands may also be used with the DTB and DTA commands. The effect of using the qualifiers and rules for their use are the same as those described in the previous section "Find Command Qualifiers." Here are some examples to help.

Delete all text that precedes the word *gremlins* in the line *There is no reason to suspect gremlins as the cause:*

```
DTB/gremlins/
```

Delete all text to the right of the second occurrence of *ragged* in the line *Around the ragged socks the ragged rascals ran:*

```
DTA L/ragged/
```

Delete all lines encountered, beginning with the current line, until a line beginning with the phrase *enough already!* is found. Ignore the case of the search argument:

```
DF BU/enough already!/
```

Splitting and Joining Lines

EDIT provides two commands which may be used to split the current line into two lines, and a command which combines two lines into one.

SB/*string*/ (Split line Before string). The SB command searches the current line for the specified text string, and if found, splits the current line in two. The first of the two lines consists only of the text in the current line that preceded the found string. The second line begins with the found string and includes all text that followed it in the current line. After SB has executed, the second of the two new lines is made the current line.

SA/*string*/ (Split line After string). The SB command searches the current line for the specified text string, and if found, splits the current line in two. The first of the two lines consists of the text in the current line that preceded the found string and the found string itself. The second line consists solely of the text that followed the found string in the current line. After SA has executed, the second of the two new lines is made the current line.

CL/*string*/ (Combine Lines and string). The CL command combines the current line and the line which follows it into a single line; string is optional, and if specified, inserts the text string in the middle of the combined line. If the length of the combined line exceeds the current maximum line width allowed by EDIT, the rightmost characters of the line are truncated. Take a look at these examples.

Consider the line of text *I would gladly pay you Tuesday for a hamburger today.*

```
SB/ for/
```

or

```
SA/sday/
```

results in the line being split in two:

```
I would gladly pay you Tuesday
    for a hamburger today
```

If you started with a current line *Time for all good men,* followed by the line *to aid their lemon lobby*:

```
CL/ and clones /
```

the result is

```
Time for all good men and clones to aid their lemon
    lobby
```

Note: The SA, SB, and CL commands also accept string qualifiers (B, E, L, P, and U). See the previous section "Find Command Qualifiers" for further information on their uses.

Global Operations

EDIT's global operation commands let you automatically insert and replace text in lines which match specified search criteria. Global commands set up editing "phantoms" that constantly look over EDIT's shoulder as lines of the *From* file are processed. Multiple global commands may be in effect during the course of an EDIT session. The global commands are

`GA/string1/string2/` (Global insert string2 After string1)
`GB/string1/string2/` (Global insert string2 Before string1)
`GE/string1/string2/` (Global Exchange string2 with string1)

Once a global command is issued, EDIT applies the associated A, B, or E command to every line as it passes the current line marker.

Canceling Global Operations

When a global command is issued, EDIT displays an identification number associated with that particular global phantom.

An individual global phantom may be canceled by issuing the **CG** (Cancel Global) command followed by the phantom's ID number. For instance, to cancel a global command that's been issued the ID number G4, type

```
CG4
```

To stop all current global operations, simply type

```
CG
```

If you can't remember what the active global operations are, the **SHG** (SHow Globals) command will refresh your memory.

Command Groups

EDIT commands that have been strung together on a single line, separated by semicolons, may be grouped together by enclosing the commands in parentheses. The resulting expression is called an *EDIT command group.* Command groups are normally used when you wish to repeat a group of commands several times. One command group may be nested within another, such as in

```
2(25(E /red/blue/;N);5 0N)
```

This replaces the text *red* with *blue* in the current line and all lines within 24 lines of the current line. The current line marker is then moved ahead 50 lines. Occurrences of the text *red* are replaced with *blue* in the new current line and in the following 24 lines. Finally, the current line marker is again moved ahead 50 lines.

If you instruct EDIT to execute a command or command group zero times, the command continues to execute until the end-of-file is encountered or until CTRL-C is used to issue a BREAK.

EDIT Command Files

When EDIT is invoked, it accepts commands from the keyboard or from an AmigaDOS file specified by the WITH keyword in the DOS command line which started the editing session.

You can also dynamically invoke the execution of EDIT commands stored in AmigaDOS files from within EDIT by using the C command.

```
C .:my/stored/commands.
```

starts execution of the EDIT commands contained in the file *my/stored/commands* in the root directory of the current drive. Command execution continues until a **Q** (Quit) command is encountered in the command file or until the command file's end-of-file is reached. The filename must be enclosed by a valid EDIT delimiter. (Notice that in the above examples, periods were used to delimit the filename.) Command files may call other command files.

Suppose you want to set up an AmigaDOS command sequence file that will create a nicely sorted list of the contents of the current directory. The following command sequence file, when used in conjunction with a simple EDIT command file, does the trick:

```
LIST > mylist
EDIT mylist WITH df 0:unwanted
SORT mylist TO finlist
TYPE finlist TO prt:
```

The contents of the filename unwanted are

```
D M* D W
```

When the above AmigaDOS command sequence file is executed, an unsorted list of the contents of the current directory is directed to the file called *mylist.* EDIT is invoked using the WITH option to pull in the commands in the file *unwanted.* These commands remove the first and last lines of the LIST output (since they contain information about the current directory rather than the file or directory names in it). The edited file is saved and you're returned to the command sequence file. The edited file is then sorted and sent to the system's printer.

Merging Selected Parts of Files/Outputting Multiple Files

It's also possible to use EDIT to merge selected parts of different files together and to create multiple versions of the edited text. This is accomplished in a somewhat roundabout way, using facilities within EDIT that allow you to change the current *From* and *To* files on the fly from within EDIT.

FROM/*filename*/. The lines that follow the current line are replaced by the contents of the new *From* file. The original *From* file remains open and the lost lines may be accessed again by issuing a FROM command with no filename. A file opened by FROM may be closed by the CF (Close File) command, which has a format of CF/*filename*/.

The following sequence of EDIT commands merges the first 15 lines of three different files into one:

```
EDIT onefile TO myfile
14N
FROM ,twofile,
15N
FROM ,threefile,
16N
D*
CF onefile
CF twofile
W
```

TO/*filename*/. The TO command lets you dynamically switch EDIT's destination, or *To* file. TO writes EDIT's existing output buffer to the *To* file before the switch is made and then clears the buffer. TO leaves the previous *To* file open. Issuing a subsequent TO command with no filename results in the original *To* file being reselected.

The following example outputs lines 1-100 of the file *bigfile* to a file called *firsthundred,* and lines 101-200 of *bigfile* to a file called *secondhundred.*

```
EDIT bigfile TO firsthundred
100N
TO .secondhundred.
100N
CF
D*
W
```

The Rewind Command

REWIND scans the remaining lines from the current line forward, executing any global commands in effect as it proceeds, until it reaches the last line of the *From* file. The contents of the output buffer are written, and the *To* and *From* files are closed. The *To* file is then reopened as a new *From* file.

The Halt Command

H (Halt) lets you set a line number as a brick wall which the current line marker cannot be moved past.

```
H134
```

prevents EDIT from moving past line 134 of the *From* file. If a command causes line 134 to be reached, the operation is halted and the message *Ceiling reached* displays.

Point Before and After

PB (Point Before) and **PA** (Point After) move the position operational window pointer in the current line.
PA/*string*/ moves the operational window pointer immediately after string in the current line.
PB/*string*/ moves the operational window pointer immediately before string in the current line.

EDIT Command Reference

Ending an Edit Session

STOP	Quick bailout; *From* file remains intact
W	Windup; advance to EOF, save, and exit EDIT
Q	QUIT; return to previous process

Verification Commands

?	Verify current line
!	Verify current line; display codes of undisplayable characters
V+/V	Turn auto verification on/off
TR+/TR	Display/suppress trailing spaces

Operational Window Commands

>	Move operational window pointer right
<	Move operational window pointer left
PR	Reset operational window pointer
$	Change character at operational window pointer to lowercase
%	Change character at operational window pointer to uppercase
_	Change character at operational window pointer to a blank
#	Delete character at operational window pointer
PB/*string*/	Move operational window pointer before string
PA/*string*/	Move operational window pointer after string

Moving from One Line to Another

N	Next line
P	Previous line
M*a*	Move to line *a*

Display Text

T*n*	Type *n* lines
TP	Type previous lines
TN	Type next lines
TL	Type with line numbers

Inserting Text

I/I.	Insert before current line
I* or R*	Insert at end-of-file marker
I*a*	Insert before line *a*
Z/*string*/	Change input mode terminator
I/*filename*/	Insert file before current line
I*/*filename*/	Insert file at end-of-file marker
Ia/*filename*/	Insert file before line *a*
R/R.	Replace current line with inserted text
R*a b*	Replace lines *a* through *b* with inserted text

Renumbering Lines

=*n*	Renumber; assign *n* to current line
F/*string*/	Find *string*
BF/*string*/	Backward find *string*
E/*string1*/*string2*/	Exchange *string2* with *string1*
B/*string1*/*string2*/	Insert *string2* before *string1*
A/*string1*/*string2*/	Insert *string2* after *string1*
'	Repeat string alteration command

String Qualifiers

B Search string must begin line for match
E Search string must end line for match
P Entire line must match search string
L Search from right to left for string

U Ignore case of search string

Deleting Text

D*a b*	Delete lines *a* through *b*
DTB/*string*/	Delete text before string
DTA/*string*/	Delete text after string
DF/*string*/	Delete lines until string found
SB/*string*/	Split line before string
SA/*string*/	Split line after string
CL	Join line
CL/*string*/	Join lines with string

Global Operations

GA/*string1*/*string2*/	Global insert *string2* after *string1*
GB/*string1*/*string2*/	Global insert *string2* before *string1*
GE/*string1*/*string2*/	Global exchange *string2* with *string1*
CGn	Cancel global operation *n*
CG	Cancel all global operations
SHG	Show global info

External File Commands

C/*filename*/	Execute EDIT commands in filename
FROM/*filename*/	Change current *From* file
TO/*filename*/	Change current *To* file
CF/*filename*/	Close current *From* or *To* file
SHD	Show data
REWIND	Close *From* and *To* files; open previous *To* file as new *From* file
Hn	Halt movement past line *n* of the *From* file

Chapter 8

AmigaDOS Command Reference

In early versions of AmigaDOS, CLI command lines are limited to 255 characters, while later versions allow command lines of up to 512 characters. In either case, it's possible that a single command line will occupy more than one line on the screen. The console refuses to accept any keyboard input that would cause the line to exceed its limit (255 or 512 characters).

With AmigaDOS versions prior to 1.3, you cannot use editing features to recall a previously issued command, edit it, and use the revised line for a new command. Each time you issue a new command, you have to enter the entire command line from scratch. You cannot use the cursor keys to edit the line you're on. If you make a mistake at the beginning of a line, you have to erase the whole line and start over.

Version 1.3 of AmigaDOS added an optional NEWCON console device, that allows you to use the cursor keys to edit the current command line. In addition, the optional Shell handler provides command history, which allows you to recall a previous command line, edit it, and use the revised version as a new command line. To use these features with version 1.3, you must explicitly MOUNT the NEWCON handler and make the Shell-Seg handler RESIDENT, using the appropriate commands in the Startup-sequence or elsewhere. With Release 2, however, these features are built into the Kickstart ROM.

Though not really an editing character, the semicolon (;) is significant to the CLI. The CLI interprets anything in a command line which follows a semicolon as a comment, ignoring the entire rest of the line.

AmigaDOS Filename Conventions

- AmigaDOS filenames may be up to 30 characters long.
- Filenames may not contain a colon (:) or slash (/), or nonprinting CTRL characters. Versions earlier than 1.3 do not allow ALTernate characters.

CON: Editing Features

Key(s)	Function
BACKSPACE or CTRL-H	Erases character to left of cursor
CTRL-X	Erases entire current line (cancels line)
CTRL-L	Clears the screen (form-feed)
RETURN or CTRL-M	Ends the line and executes the command
CTRL-J	Moves cursor to next line, but doesn't execute the command
	Marks start of a comment
CTRL-\	End-of-file indicator
NEWCON: or Release 2 and 3 only	
CTRL-A	Moves the cursor to the beginning of the line or Shift-LeftArrow
CTRL-K	Erases everything from the cursor forward to the end of the line
CTRL-U	Erases everything from the cursor backward to the start of the line
CTRL-W	Deletes the word to the left of the cursor
CTRL-Y	Replaces the characters deleted with CTRL-K
CTRL-Z	Moves the cursor to the end of the line

- If a filename is to contain special characters, such as spaces, plus (+), equal (=), semicolon (;), or wildcard characters (see below) that have special significance to CLI, the entire filename must be enclosed in double quotation marks (").
- If a filename is to contain double quotation marks (") or an asterisk (*), each " and * must be preceded by an asterisk.
- Any combination of uppercase and lowercase can be used in naming a file. When you LIST the filenames, they'll be printed in the same combination of uppercase and lowercase used when the filename was created. The CLI, however, does not distinguish case. Since CLI ignores case, you cannot have two files with the same name in the same directory, two files named *Test* and *TEST* cannot reside in the same directory.

Pattern Matching (Wildcards)

Some AmigaDOS commands allow you to reference one or more files at a time using a technique called *pattern matching.* Pattern matching lets you do things like getting a listing of all files whose names end with the characters *.bas,* or deleting every file in a directory at one time. AmigaDOS pattern matching is similar to the concept of the *wildcard* characters used in MS-DOS, but there are important differences.

In MS-DOS, the asterisk character can be used to substitute for any string of characters in a filename. In AmigaDOS, the asterisk is used as an escape character which allows for the insertion of quotation marks (and other asterisks) in a filename. AmigaDOS also uses the asterisk to refer to the console device that's currently active. AmigaDOS Release 2 and higher provides the option of using the asterisk to substitute for any string of characters, just as in MS-DOS. This option can only be turned on with software, however.

PC wildcards can be used with more commands than AmigaDOS pattern matching, which is mostly confined to the COPY, DELETE, DIR, and LIST commands. AmigaDOS patterns, however, are much more flexible. They allow you to match names which start with the same group of characters, end with the

Other CON: Features

Key(s)	Function
TAB or CTRL-I	Moves cursor one space to the right (inserts a tab character)
CTRL-K	Moves cursor up one line (vertical tab)
CTRL-O	Switches to ALTernate character set (shifts out)
CTRL-N	Switches back to normal character set (shifts in)
ESC-[1m	Switches to bold characters
ESC-[2m	Switches character color (to black)
ESC-[3m	Italics on
ESC-[4m	Underline on
ESC-[7m	Reverse video on
ESC-[8m	Switches character color (to blue—invisible)
ESC-[0m	Switches to normal characters
ESC-C	Clears screen and switches to normal characters

same group of characters, or have the same characters in the middle (preceded by any number of characters and followed by any number of characters).

The most important pattern matching characters are the question mark (?) and the pound sign (#). The pound sign followed by a single character will match any number of repetitions of that character (including none).

For example, *#CLUTTER* matches:

CLUTTER
CCCCLUTTER
LUTTER

The question mark is used to replace any single character (but not the null string, or no character). For instance, *?LA?S* matches:

GLASS 2LABS

but not

LABS

When these two characters are paired together (#?), it creates a pattern that matches any number of any characters (or no characters at all).

For example, you could use *GLAD#?* if you wanted a pattern that matched all filenames starting with the letters *GLAD*. If you wanted to LIST all of the icon information files (whose names always end in *.info*), you could use the pattern *#?.INFO* to find them.

In addition to the pound sign and question mark, there are three other characters which have special meaning when used for pattern matching. Parentheses ( ) may be used to group a number of characters together into a single pattern element. If a pound sign is followed by a group of characters within parentheses, it will match any number of repetitions of that pattern group (including none). Thus, *#(HO)* matches these filenames:

HO
HOHO
HOHOHOHO

If you didn't use the parentheses, however, *#HO* would match:

HO
HHO
HHHHO

The *#H* can only substitute for repetitions of the letter *H*.

The vertical line (|) is used when you want either of two patterns to match the characters in the filename. For example, *Y|Z* matches:

Y
Z

While the pattern *WARM|COLD* matches:

WARM
COLD

and the pattern *MO(B|N)STER* matches:

MONSTER
MOBSTER

(Note how the parentheses were used to set off the *B|N* as a distinct pattern).

The percentage sign (%) is used to represent the null string (no character). Remember, a pattern starting with the pound sign will match any number of repetitions of the following character, including none at all.

The pattern *Z#AP* then matches:

ZAP
ZAAAP
ZP

If you want to match only a single appearance of the character or none at all, you can use the form (*A|%*), which stands for either A or the null character (no character at all). Using the same example, *Z(A|%)P* would still match:

ZAP
ZP

but would not match:

ZAAAP

which uses the A more than once.

Combining the percentage sign with the question mark in the form *(?|%)* forms an expression which matches any character or no character at all. The pattern *(?|%)A?X* matches:

LAPX
APX

but not:

MAPPX

There's one final character which addresses a problem created when using these special AmigaDOS characters. Since these characters have meaning in the language of pattern matching, it makes it difficult when you want to match a filename containing one of those characters. In order to match a filename containing a question mark, for example, you must precede the question mark with an apostrophe (') to let the pattern matching mechanism know that you want to match an actual question mark, not use the question mark as a substitute for any other character.

The pattern *?OW'?* matches filenames like:

HOW?
COW?
WOW?

Since you've used the apostrophe itself as a special character, you need to use two apostrophes to represent an apostrophe which is part of the filename. You would therefore need a pattern like *?ON"T* to match filenames like:

DON'T
WON'T.

Finally, if a pattern contains space characters, it must be enclosed by double quotation marks.

Release 2 of AmigaDOS introduces a number of very handy new pattern characters, the tilde and the dash(–). The tilde functions as a *not* operator, selecting any file that doesn't match the pattern. For example, if you wanted a directory listing of all files except icon files (those whose name end in .info), you could use the command

```
DIR ~(#?.info)
```

The other new character, the dash, is used in an expression set off by square brackets to indicate a range of characters. Thus, the expression *[W-Z]#?* would select any file beginning with W,X,Y or Z, just like the the expression *(W|X|Y|Z)#?*. Be sure that to use the square brackets (the keys just to the left of the top of the Return key), not the parentheses. To copy all files from the c: directory that begin with the letters A-D, you could use the command

```
COPY c:[a-d]#? ram:
```

Pattern Matching Summary

#c	Matches any number of repetitions of the character c (including none)

	N#O matches *N, NO, NOO*, and *NOOOOOOOOOO*
#(*group*)	Matches any number of repetitions of group (including none) *#(TOM)* matches *TOM* and *TOMTOM*
?	Matches any single character (but not the null character) *K?NG* matches *KING* and *KONG* (but not *KNG*)
#?	Matches any number of repetitions of any character (including none) *#?.BAS* matches any filename ending in .BAS
P1|P2	Matches either pattern *P1* or *P2* *B(A|O)Y* matches *BAY* and *BOY*
%	Matches the null string (no character) *(S|%)TOP* matches *STOP* or *TOP*
(?|%)	Matches any character or no character *(?|%)LOT* matches *SLOT,* CLOT and *LOT*
()	Used to set off a group of characters as its own distinct pattern *(M|P)A* matches *MA* or *PA* *M|PA* matches *M* or *PA*
'	Used in front of one of the special characters to show that you want to match it, not invoke its special meaning *?ON"T* matches *WON'T* and *DON'T*
[char1-char2]	Matches any character in the range from *char1* to *char2* (AmigaDOS Release 2 and 3 only). *C:[t-w]#?* matches *TYPE, VERSION, WHICH* and *WAIT*
~P	Matches any file that does not fall within the definition of pattern *P* (AmigaDOS Release 2 and 3 only).
~(#?.info)	Matches all files in the current directory *except* the icon files whose names end in *.info*.

AmigaDOS Templates

AmigaDOS contains a handy feature that can be used to jog your memory if you forget the command syntax of any AmigaDOS command in the C: directory. By typing the command name, followed by a question mark, the command's template is displayed on the screen. The template is a shorthand summary of the parameters and keywords associated with the command.

Using the template feature breaks the process of entering a command line into two parts. In the first part, you enter the command name and question mark, and then the command displays its template and waits for further input. The next

line that you enter is treated as if it were the arguments typed after the command name. Hitting Return without typing anything invokes the waiting command with no arguments.

Let's take a look at the template for the Release 2 and 3 COPY command.

```
COPY ?
```

displays on the screen:

```
FROM/A/M,TO/A,ALL/S,QUIET/S,BUF=BUFFER/K/N,CLONE/
    S,DATES/S,NOPRO/S,COM/S,NOREQ/S:
```

AmigaDOS command arguments are separated by commas in command templates. The first part of each argument is either the argument name or the keyword associated with the argument. Keywords are followed by *qualifiers* (/A, /K, /S, /N, /M, or /F) which tell you more information about the argument. When you invoke an AmigaDOS command, keywords, if used, must be typed exactly as presented; often you must type additional information following the keyword (depending on the command).

argument/A	The argument is *required.*
option/K	The argument is optional, and must contain the keyword (along with additional information) if that option is used.
option/S	The argument is optional, but if used, the keyword alone is sufficient.
=	The equals sign indicates that there are two equivalent versions of the keyword. The user has the choice of entering the keyword to the left of the equals sign, or the one to the right of it.
argument/M	Multiple arguments are accepted (Release 2 and 3 only). In earlier versions, multiple arguments are indicated by a series of commas. Note that there is no limit on the number of arguments, but multiple arguments must be given before any other argument or option.
value/N	The argument is a number (Release 2 and 3 only)
string/F	This string argument *must* be the final one on the command line. The rest of the command line (including spaces) will be considered part of the string (Release 2 and 3 only).

A keyword in an argument template may have more than one qualifier associated with it (such as FROM/A/M in the example above).

Some commands allow you to use different keywords to invoke the same option. For example,

```
DATE ?
```

shows

```
TIME,DATE,TO=VER/K:
```

TIME and *DATE* are the parameter names of the first and second arguments of the DATE command. Values for these arguments are set by the user. The *TO* and *VER* keywords may be used interchangeably and require additional information to be specified after them.

If the arguments you use with an AmigaDOS command don't match the template, AmigaDOS 1.3 and below displays the message *Bad arguments.* AmigaDOS Release 2 and 3 provides a more specific message, such as *Required Argument Missing* or *Wrong Number of Arguments.*

Redirected Output

The characters < and > may be used to redirect the input and output of AmigaDOS commands. AmigaDOS commands normally expect input to come from the system's keyboard and send output to the system's screen. Input and output redirection is temporary, lasting only until the invoked command completes. Here are some examples.

LIST the files and directories in *what a/silly/mess* directory on drive df1:. Send the output of the LIST command to the system's printer:

```
LIST > PRT: "df1:what a/silly/mess"
```

Send the output of the DATE command to a file called *tempdate* on the system's RAM disk:

```
DATE > RAM:tempdate
```

Use the Amiga's line editor (EDIT). Edit the file called *myfile* using the commands stored in the file *mycommands*. Store the edited data in *newfile:*

```
EDIT < mycommands FROM myfile TO newfile
```

The Shell Program on the 1.3 and Release 2 and 3 Workbench disk supports a third redirection operator. The character >> can be used to append information onto the end of an existing file. For example, if you wanted to add a listing of a directory called *MoreStuff* to an existing file called *Directories,* you could use the command

```
DIR >>Directories MoreStuff
```

Note that if a file called *Directories* does not exist, the 1.3 version of this command will not create one, but the Release 2 and 3 version will.

Format of the AmigaDOS Command Reference

The remainder of this section is a command-by-command listing of AmigaDOS. For the most part, its format is self-explanatory. However, under the "Format" heading (perhaps the most important part of each command's listing), there are several typographical devices used to show you what is required and what is optional.

- The command names (which must always appear first on the command line) and keywords that are required appear in uppercase boldface roman type. **ASSIGN** and **COPY** are examples.
- Keywords which are optional appear in uppercase boldface italics, and are enclosed in square brackets. [***LIST***] is an example.
- Arguments or command parameters are in lowercase italics. These denote where you'll enter something, such as the name of a file or directory. If required, the parameter is not enclosed in brackets. If optional, it is enclosed in brackets.

Thus,

COPY [***FROM***] *fromname* [***TO***] *toname* [***ALL***] [***QUIET***]

indicates that the keyword **COPY** is required, that the keywords **FROM, TO, ALL,** and **QUIET** are all optional, and that the two parameters *fromname* and *toname* are required.

Of course, complete explanations of each keyword and parameter are provided under the "Explanation of Parameters and Keywords" heading.

ADDBUFFERS Command

Location: C:

Function

Sets aside a portion of the system RAM to be used exclusively as disk buffer space. This disk buffer keeps information frequently accessed from the disk in the computer's memory. Thus the system accesses the physical disk less often, significantly speeding up disk operation. The default number of buffers are 5 for floppy disks and 30 for hard drives. The ADDBUFFERS command allocates additional buffer space in 512-byte increments. Although additional buffer space increases disk performance somewhat, floppy disks reach a point of diminishing returns at the level of about 30 additional buffers (15K of buffer space). Hard drives that are

formatted using the Fast File System will always benefit from the addition of more disk buffers. Usually, however, the number of hard drive buffers allocated are specified during the partitioning process, and this number is stored in the drive's Rigid Disk Blocks (RDB).

Additional buffer space means that more of the information read from disk remains in memory, where it can be accessed more quickly. Therefore, added disk buffer space only speeds up access to frequently used files. It won't speed up operations like a file copy, where the information to be copied is only used once.

Keep in mind that any memory you allocate for buffer space will be subtracted from the free memory that you have to run programs. Some programs which require the full memory of a 512K or one megabyte Amiga may not operate when a significant amount of memory has been allocated for disk buffer space.

Under AmigaDOS Release 2 and 3, this command can also be used to determine the number of buffers assigned to a drive (the Release 2 and 3 version prints the number of buffers assigned after the changes have been made). It can also be used to subtract buffers from a drive, freeing up a little memory.

1.3 Format

ADDBUFFERS *drive n*

Release 2 and 3 Format

ADDBUFFERS *drive* [*n*]

Explanation of Parameters and Keywords

drive This is the device name of a physical disk drive, either a floppy drive (df0:, df1:, and so on) or a hard drive (dh0: jh0:, and so on). Note that if you add buffers for an electronic RAM drive, you may actually slow down the operation of that drive, since all of its information is already stored in RAM.

[*n*] The number of 512-byte buffers to add. Floppy drive performance peaks first at about 30 buffers (15K of memory), and again at about 100 buffers (50K of memory).

Under AmigaDOS Release 2 and 3, you may use a negative number for *n,* and that number of buffers will be subtracted from the total currently allocated to the drive, down to a minimum of one buffer per drive. This will hurt drive performance, but may be enough to help you run some programs

that take every bit of memory in the machine. AmigaDOS Release 2 and 3 also allows you to omit the number of buffers to add completely. If you give an ADDBUFFERS command without specifying the number of buffers to add, the command will return the number of buffers assigned to *drive*.

Examples

1. Add 30 disk buffers for use by the internal disk drive:
   ```
   ADDBUFFERS df0: 30
   ```
2. Add 100 disk buffers for use by the BridgeBoard hard drive:
   ```
   ADDBUFFERS jh0: 100
   ```

ADDDATATYPES Command

Location: 3.0 C:

Function

ADDDATATYPES was added in Release 3 in support of the object-oriented DataTypes scheme which is used by the new MultiView program in the Utilities drawer. MultiView displays text files, shows pictures, and plays sound samples without "understanding" anything about these types of data. It passes all of the nitty-gritty work to system-standard libraries that are "experts" in a particular kind of data. These libraries are stored in the sys:Classes directory, and are accessed through the datatypes.library which is stored in the L: directory. In order for datatypes.library to recognize a particular data-handling library, it must first be registered with datatypes.library. The ADDDATATYPES command can be used to register such a library, by reading information about it from a file in the DEVS:DataTypes directory. Normally, all of the files in that directory are registered at boot time, but additional data types can be added later with this command.

Release 3 Format

ADDDATATYPES [*files(s)*] [***QUIET***] [***REFRESH***]

Explanation of Parameters and Keywords

[*files(s)*] The name of the file or files to add. If multiple filenames are specified, they must be separated by spaces. Normally, the description files

used to add data types are stored in the DEVS:DataTypes directory, and for each type description file, there is a corresponding library file in the SYS:Classes/DataTypes directory. For example, the DEVS:Datatypes/ILBM file describes an object that can be manipulated by means of the support found in the SYS:Classes/picture.datatype file.

[***QUIET***] This optional keyword prevents the command from printing any output, such as the names of data types it has added.

[***REFRESH***] This optional keyword causes the command to scan the DEVS:DataTypes for any new or updated entries.

Examples

1. Add the data type described in the file df0:Devs/Datatypes/JPEG:

```
ADDDATATYPES df0:Devs/Datatypes/JPEG
```

ALIAS Command

Location: Internal

Function

ALIAS is an internal function of the 1.3 or Release 2 and 3 Shell, which allows you to create short names or *aliases* for longer command strings. This concept is similar to that of the keyboard macro, where you type only a single key combination to print a longer series of letters. When the Shell encounters the alias name as the first word on the command line, it substitutes the full command string (which may include arguments as well as the command name) for that name. The ALIAS command may also be used to display the current list of alias assignments.

Aliases are only recognized by the Shell in which they are created, or Shells that are opened with the NewCLI command entered in that Shell. To create aliases that are recognized by all Shells, enter the alias definition in the *s:Shell-startup* script file. This script is automatically executed whenever a Shell window is opened.

1.3 and Release 2 and 3 Format

ALIAS [*alias* [*string*]]

Explanation of Parameters and Keywords

[*alias*] The short name you wish to substitute for the command string. If all arguments are omitted, ALIAS will display a complete list of alias

names and their corresponding command strings. If the *alias* argument is used alone, without *string*, the 1.3 version of ALIAS will remove the alias. The Release 2 and 3 version, however, will display the command string associated with this name. To remove an alias under Release 2 and 3, use the UNALIAS command.

[*string*] The command which you wish to substitute for the alias name. This string can include command arguments, but it cannot consist solely of command arguments. It must contain the command name, because the alias must be entered as the first word of text on the command line in order to have the command string substituted for it.

Additional command arguments can be incorporated into the command string by using the square brackets [] as a place holder. Any arguments typed after the alias on the command line will be inserted into the command string at the spot indicated by the square brackets.

Examples

1. Shorten the name of MAKEDIR command to MD

```
ALIAS MD MAKEDIR
```

2. Create an alias called DUPE that will copy the disk in df0: to df1:

```
ALIAS DUPE Sys:System/Diskcopy df0: to df1:
```

Whenever you enter the command DUPE, the Shell will substitute the command line starting with Sys:System/Diskcopy.

3. Create an RD command that will delete a directory even if it is not empty

```
ALIAS RD DELETE [] ALL
```

When you type the command

```
RD Directory
```

the Shell expands the command to

```
DELETE Directory ALL
```

ASK Command

Location: 1.3 C: Release 2 and 3: Internal

Function

ASK allows a script file to vary its command sequence based on input from the user. It prints a prompt string, and then waits for the user to type the response Y or YES, N or NO, or RETURN (same as NO). When the user has responded, the command exits with a return code of 5 for a YES response, or 0 for a NO response. The IF WARN command can be used to detect the results of this operation, and to direct script execution accordingly.

1.3 and Release 2 and 3 Format

ASK *promptstring*

Explanation of Parameters and Keywords

promptstring A text message that's printed to the current CLI output stream (usually requesting a yes or no answer). If this prompt message contains spaces, *promptstring* should be contained within quotation marks.

Examples

A command file that requests the user to specify a yes or no answer, and then prints a text string that indicates what his choice was:

```
ASK "What is your choice? (Yes/No) "
  IF WARN
      Echo "Your choice was YES"
  ELSE
      Echo "Your choice was NO"
  ENDIF
```

ASSIGN Command

Location: C:

Function

Builds, removes, and lists associations between logical device names and filing system directories, physical devices (DF1:, DH0:, and so on), and disk volume names.

1.3 Format

ASSIGN[*devname* [*dirname*]][***LIST***][***EXISTS***][***REMOVE***]

Release 2 and 3 and 3 Format

ASSIGN [*devname* [*dirname(s)*]] [***LIST***] [***EXISTS***] [***DISMOUNT***] [***DEFER***][***PATH***][***ADD***] [***REMOVE***][***VOLS***][***DIRS***][***DEVICES***]

Explanation of Parameters and Keywords

[*devname*] The logical device name that you wish to assign to a directory, physical device, or disk volume. After making the assignment, you can use this device name in place of specifying the entire directory, device, or volume until you change the assignments or reboot the computer.

Certain assignments are automatically made by the operating system when DOS is initialized. These are the logical devices S:, L:, C:, FONTS:, DEVS:, LIBS:, and SYS:. These correspond to the directories of the same names, which have special significance to AmigaDOS (see Chapter 4 for more information on logical devices). If corresponding directories don't exist on the boot disk, the assignment is made to the root directory of the disk.

If a specified logical device name already has a directory, physical device, or volume name associated with it, the new ASSIGNment replaces the old. If you try to assign the same logical device names as an existing volume, however, the assignment will fail (for instance you cannot assign the logical device MyDisk: when a disk whose volume name is MyDisk is already mounted). Any associations built by ASSIGN apply to all CLIs, and all are lost when the system is shut off or rebooted.

The *devname argument* is optional. If you use the ASSIGN command without any parameters, it returns a list of all of the logical device assignments, as well as the volumes and devices that are currently recognized.

[*dirname*] The directory path, physical device, or disk volume name that will be represented by references to the specified *devname*. For example, if you used the directory *df0:Daves/Wordprocessing/Documents* often, you might find it more convenient to be able to type *Docs:* instead of the entire phrase *df0:Daves/Wordprocessing/Documents*. To make the assignment, type

```
ASSIGN docs: df0:Daves/Wordprocessing/Documents
```

Notice that the full pathname of the directory to be assigned was specified. ASSIGN searches the directory path starting with the current directory, so the path should be fully spelled out if the target directory is located anywhere but in the current directory.

If you just want to remove an assignment, without replacing it, use the command form ASSIGN *devname*, with no *dirname* specified.

Under AmigaDOS Release 2 and 3, you may ASSIGN a single logical device name to multiple directories simply by listing all of the directory names after the device name. If you use a hard drive with a number of programs that created bitmap picture files, for example, and you want to see all pictures listed from any of the programs, you might use an ASSIGN command like this:

```
ASSIGN PICS: DH0:Dpaint/Pics DH0:Scanner/Pics
DH0:AmigaVision/Pics
```

[*LIST*] If you type ASSIGN without specifying a logical device name, it displays the list of current assignments. If you wish to both make or remove assignments *and* show the new assignment list, use the optional LIST keyword at the end of the command line.

[*EXISTS*] If you add the EXISTS keyword after the name of the logical device, the ASSIGN command will display assignment information for that device name only. If the device is not found, the command exits with a return code of 5 (WARN). This feature can be used in sequence files that take one action if the device name is found, and another if it is not found.

[*REMOVE*] Under AmigaDOS 1.3, this optional keyword may be used to remove a physical device (such as df0: or prt:) from the list of mounted devices. It does not free up resources (such as disk buffers) used by that device, so it cannot be used to release memory, for example, that could normally be reclaimed by physically disconnecting a second disk drive. It is useful mostly for experimental purposes, like disconnecting the internal disk drive (df0:) and MOUNTing it with a different disk format (such as the Fast File System).

Under AmigaDOS Release 2 and 3, the REMOVE keyword is used to delete a logical assignment. Most often it is used to remove one of the directories from the list of those assigned to a single logical device name, when a multiple assignment has been made. In that sense, it is the opposite of the ADD keyword (see below).

[*DISMOUNT*] Under AmigaDOS Release 2 and 3, the optional DISMOUNT keyword is used exactly as the the REMOVE option was used under 1.3 (see above), to remove a physical device from the list of mounted devices.

[DEFER] The optional DEFER keyword, new for AmigaDOS Release 2 and 3, is used to delay the search for *dirname* until a command tries to use the logical device name (normally, the ASSIGN command tries to verify that *dirname* exists before it will make the assignment). If you assign FONTS: to df0:Fonts with the DEFER option, for example, the logical device name will be assigned to the Fonts directory of whatever disk happens to be in the internal floppy drive the first time a program asks for FONTS:, and not the Fonts directory of the disk that is in the internal drive when the ASSIGN command is given. This command was designed to be used for assignments that are made automatically by one of the startup scripts. If the DEFER option is used in such an assignment, then the startup procedure won't be interrupted if the directory is not found.

[PATH] Another optional keyword added by AmigaDOS Release 2 and 3, PATH not only delays the search for *dirname* until a command tries to use the logical device name, but it causes the assignment to be reevaluated each time the logical device name is requested. In the example given above, the command ASSIGN FONTS: df0:Fonts DEFER assigns FONTS: to the Fonts directory of whatever disk is in the internal drive the first time a program asks for FONTS:. If it was a disk volume named Stuff, then FONTS: would be assigned to Stuff:Fonts, and that assignment would not change until you changed it.

With the command ASSIGN FONTS: df0:Fonts PATH, however, ASSIGN will look for the Fonts directory of whatever disk is in the internal drive each time a program requests FONTS:. For this reason, an assignment made with the PATH option is called *non-binding*.

The PATH option was designed for the convenience of floppy drive users. It can help eliminate the need to insert the original Workbench disk each time one of the system directories is required. In the example above, you would not have to replace Workbench each time that FONTS: was requested—any disk that had a Fonts directory would be acceptable. Note, however, that if you use the PATH option, you cannot assign multiple directories to the same logical device name, either by using several directory names in the original ASSIGN command, or by using the ADD option (see below).

[ADD] This Release 2 and 3-only option allows you to add new directories to an existing logical device name, without removing existing directories. Normally, when you ASSIGN a logical device name that already exists, the old assignment is lost. When you use the ADD option,

however, the specified directories are added to the existing list of directories for that device name. For example, to the Video/Fonts directory to the FONTS: device that is automatically assigned to Sys:Fonts at startup time, you could use the command:

```
ASSIGN Fonts: Video/Fonts ADD
```

[VOLS]
[DIRS]
[DEVICES] These options were added in AmigaDOS Release 2 to allow you to limit the information displayed by ASSIGN when the command is given without arguments or using the LIST option. The VOLS option shows which volumes are recognized, and whether or not they are are currently mounted. The DIRS option shows the directories to which logical device names are assigned. The DEVICES option displays the names of devices that are recognized by the system. If none of these options are used, all three kinds of information are displayed.

Examples

1. List the current logical device name/file directory associations:

```
ASSIGN or ASSIGN LIST
```

Sample Display:

```
Volumes:
RAM DISK [MOUNTED]
CLI WorkDisk [Mounted]
Directories:
S           CLI WorkDisk:s
L           CLI WorkDisk:l
C           CLI WorkDisk:c
FONTS       CLI WorkDisk:fonts
DEVS        CLI WorkDisk:devs
LIBS        CLI WorkDisk:libs
SYS         CLI WorkDisk:
Devices:
DF1   DF0   PRT   PAR   SER
RAW   CON   RAM
```

2. Associate the logical device name *Rick:* with the directory *:AmigaWord/Proposals/RickWork:*

```
ASSIGN Rick: :AmigaWord/Proposals/RickWork
```

After executing this ASSIGN statement, a file called *ACME* in the *AmigaWord/Proposals/RickWork* directory may be referenced by referring to the logical device name or the full directory specification for the file.

```
TYPE Rick:ACME
```

yields the same result as

```
TYPE :AmigaWord/Proposals/RickWork/ACME
```

If an ASSIGN or ASSIGN LIST is executed, the directory association
RICK Volume: CLI WorkDisk Dir: RickWork
shows up in the Directories section of the table.

3. Remove a logical device/directory assignment.

```
ASSIGN Rick:
```

removes the association built by the ASSIGN statement in the second example.

```
ASSIGN Rick: LIST
```

removes the association built by the ASSIGN statement in example 2 and lists the remaining logical device associations still in effect.

4. Check for the existence of a logical device name or disk volume name in a sequence file.

```
ASSIGN >nil: empty: EXISTS
IF WARN
    ECHO "The directory name doesn't exist"
ELSE
    ECHO "The directory name exists"
ENDIF
```

This script file, when executed, prints a message telling whether or not a volume named EMPTY: exists. In your own script, you might take a specified action at the points where the ECHO command appears.

5. Under AmigaDOS Release 2 and 3, assign multiple directories to the FONTS: logical device name.

```
ASSIGN FONTS: Sys:Fonts PPage:Fonts Work:PM/Fonts
```

AVAIL Command

Location: C:

Function

Prints a report of system memory resources, broken down by memory type. For each type (CHIP, FAST, and TOTAL), AVAIL reports the amount of available (free) RAM, the amount in use, the maximum (total) amount, and the largest contiguous block that is available for allocation. The Release 2 and 3 version also provides an option which causes all unused libraries and fonts to be unloaded from memory, expanding the pool of free memory.

1.3 Format

AVAIL[***FAST*** or ***CHIP*** or ***TOTAL***]

Release 2 and 3 Format

AVAIL[***FAST*** or ***CHIP*** or ***TOTAL***] [***FLUSH***]

Explanation of Parameters and Keywords

[***FAST***] or [***CHIP***] or [***TOTAL***] When one (and only one) of these keywords is used with the AVAIL command, the command returns a single number which indicates the total number of available bytes of that type of memory. This value can be used by script files for comparisons, using either the EVAL or IF GT commands.

If the AVAIL command is given without any of the optional keywords, a more complete summary of available system RAM is printed. This summary includes a breakdown by RAM type, and lists the total amount, amount used, amount free, and largest contiguous free block.

[***FLUSH***] The FLUSH option, added in version Release 2, directs AVAIL to free as much memory as possible. During the course of operation, AmigaDOS routinely loads system libraries and font information from disk as needed. These take up a small, but significant, amount of memory, and are not automatically unloaded when the system is finished with them. Using the FLUSH option causes AVAIL to unload all such modules that are not currently being used by any program, potentially freeing up some memory.

Examples

1. Print a complete summary of system RAM:

```
AVAIL
```

The summary is printed in the following format:

Type	*Available*	*In-Use*	*Maximum*	*Largest*
chip	909984	130168	1040152	909952
fast	3442976	751264	4194240	2097120
total	4352960	881432	5234392	2097120

In this summary, the first column shows free (available memory), the second column the memory in use, and the third the total (maximum) memory, so that the sum of the first two columns should equal the value in the third. The final column shows the largest contiguous available block of memory, and is a sub-set of the first column, available memory.

Contiguous memory is important because if free memory is highly fragmented (broken up into small pieces scattered here and there in the memory map), it may not be possible to load and run additional programs, even if the total amount of free memory might indicate that is was possible.

2. Show only the total available chip memory:

```
AVAIL chip
```

In the example above, the number returned would be 909984.

BINDDRIVERS Command

Location: C:

Function

Looks for *device drivers* (software instructions on how to interact with external hardware devices connected to the Amiga's expansion port) in the Expansion subdirectory of the startup disk (the disk to which the logical device name SYS: is assigned) and then integrates these drivers into the operating system so that it knows how to control the devices. As of this writing, devices that are added to the system in this way include the IBM-compatible Bridgeboard, and non-autoboot hard drives.

The BINDDRIVERS command is usually issued in the *startup-sequence* script file in the *s* directory, so that external devices are added as part of the startup process. If the Expansion directory is empty, however, as

is normally the case, the BINDDRIVERS command can safely be omitted from the startup-sequence file.

1.3 and Release 2 and 3 Format

```
BINDDRIVERS
```

Explanation of Parameters and Keywords

None

Examples

1. Load device drivers in the *Expansion* directory of SYS:

```
BINDDRIVERS
```

BREAK Command

Function

Sets attention flags which interrupt a process as if the user had pressed specified CTRL-key combinations in an active window. Since many CLI commands will exit when they receive a CTRL-C interrupt, BREAK may be used by one program to signal another task to quit.

1.3 and Release 2 and 3 Format

BREAK *tasknum* [*C*] [*D*] [*E*] [*F*] [*ALL*]

Explanation of Parameters and Keywords

tasknum The number assigned by the system to the CLI process that you wish to interrupt (for more information, see the STATUS command).

[*C*] [*D*] [*E*] [*F*] [*ALL*] The attention flag(s) associated with the interrupt type that you wish to issue. You may trigger up to four CTRL-key attention flags. If the BREAK command is issued with no flag keys specified, only the CTRL-C flag is enabled. Issuing the BREAK command simulates selecting a CLI process with the mouse and pressing the specified CTRL-key keystrokes. BREAK may be used to interrupt a background CLI task initiated by the RUN command.

Examples

1. Trigger all valid attention flags (CTRL-C, CTRL-D, CTRL-E, CTRL-F) for process number 4:

```
BREAK 4 ALL
```

2. Trigger the CTRL-C and CTRL-E attention flags for process number 1:

```
BREAK 1 C E
```

3. Trigger the CTRL-C attention flag for process number 5:

```
BREAK 5
```

4. Under AmigaDOS Release 2 and 3, signal the process that is currently running the WAIT command to quit:

```
BREAK 'STATUS Com=WAIT'
```

This command uses the *backtick* feature of the Release 2 and 3 Shell to feed the results of the STATUS command (which gives the number of the process that is currently running the WAIT command) directly to the BREAK command. The BREAK command then sends a CTRL-C interrupt telling WAIT to stop waiting. You can test this by opening two Shells, typing WAIT 5 Min in one, and then typing the above command in the other. Instead of waiting for 5 minutes, the WAIT command will quit immediately with the message ****Break.*

CD Command

Location: 1.3 C: Release 2 and 3 Internal

Function

Sets or changes the current directory or drive. Also used to display the current drive and directory.

1.3 and Release 2 and 3 Format

CD[*name*]

Explanation of Parameters and Keywords

[*name*] The name of the directory path or logical device name that you wish to make the current directory. A pathname may be fully specified or relative to the current default directory. Specifying a full pathname, such as *:major/minor/tiny* does not make any assumptions about what the current directory is. If the current directory was set to *major/minor,* the former pathname could be switched to by a relative reference, namely *CD tiny.*

You can also move the current directory back (up) one level by typing CD followed by single or multiple slashes (/). For instance, if the current directory is *major/minor/tiny,* typing ***CD //*** changes the current directory to *major.*

You can specify a logical device name in lieu of a pathname. This lets you change the default disk or change directories to a directory path associated to a logical device name (see the ASSIGN command for more information). Under Release 2 and 3's Workbench, you can substitute a wildcard pattern for the directory name, provided that only one directory matches the pattern.

CD specified by itself, with no path or device name, lists the current directory setting. The complete path is specified, starting with the volume name.

Under AmigaDOS Release 2 and 3, you can change the current directory merely by typing the *name* at the Shell prompt. The CD command is not necessary unless the desired directory has spaces in its name, and must be enclosed in quotes.

Examples

1. Change the current directory to the root directory of the volume mounted in df1:

```
CD df1:
```

Note: AmigaDOS is somewhat different from the DOS of many other microcomputers in that the way it treats a default drive is volume- rather than device-oriented. For instance, assume you had a disk volume called *Hi There* in an external Amiga drive and changed the default drive to df1: by typing `CD DF1:`. After changing the default drive, typing `DIR` would give you a directory listing for *Hi There.* If *Hi There* is ejected from the drive and another volume called *Salutations* is inserted, and you type `DIR` again, the system will ask for the *Hi There* volume to be reinserted in the drive.

How can you avoid this? Entering another `CD DF1:` causes AmigaDOS to read the volume label of the disk in the external drive again and forget any volumes that it previously defaulted to.

2. Change the current directory to *df0:particle/quark/charm,*and then back to *df0:particle:*

```
CD particle/quark/charm
CD //
```

3. List the current directory setting:

```
CD
```

4. Change the current directory to the path associated with the logical device name *Rick:*

```
CD Rick:
```

5. Change directories to the root directory of the current drive:

```
CD :
```

CHANGETASKPRI Command

Location: C:

Function

Changes the multitasking priority of a CLI task, and of subsequent tasks started from that CLI. The Amiga multitasking operating system is set up so that each task is assigned a priority number from -128 to 127. Processor time is divided among the tasks, with each task executing in turn for a few fractions of a second. Tasks with the same priority number get an equal "time-slice," but tasks with higher priority execute more often and end up getting more of the processor's time. Normally, a CLI has a priority of zero. Raising its priority makes it (and the programs which run from it) run at a higher priority than other CLI tasks, while lowering makes it run at a lower priority and get less processor time. Under most circumstances, you shouldn't raise a task's priority higher than 5, nor lower it to less than -5. This will insure that it doesn't pre-empt important system tasks, such as the input handler (which checks the keyboard and the mouse), nor will it have so low a priority as to get completely shut out.

1.3 and Release 2 and 3 Format

CHANGETASKPRI *priority* [***PROCESS*** *tasknum*]

Explanation of Parameters and Keywords

priority The priority number for the task. This number may range from -128 to 127, but generally should be kept within the range of -5 to 5. Normally, CLI tasks run at a priority of 0.

[*tasknum*] The task number whose priority is to be changed. If this optional parameter is not entered, the priority of the current CLI or Shell process is changed. If it is entered, however, it must be preceded by the keyword **PROCESS**. Normally, the prompt of an interactive CLI or Shell process shows its task number. You may also check the task number of a process with the STATUS command.

Examples

1. Increase the priority of the current CLI task to 5:

```
CHANGETASKPRI 5
```

2. Decrease the priority of CLI task number 5 to -1:

```
CHANGETASKPRI -1 PROCESS 5
```

3. Use the Release 2 and 3 *backtick* feature to change the priority of the C:CONCLIP task to 5:

```
CHANGETASKPRI 5 PROCESS 'STATUS com=c:conclip'
```

CONCLIP Command

Location: C:

Function

This command directs the Release 2 and 3 console device to use the Amiga clipboard for its copy-and-paste function. (By default, the console device uses its own internal buffer for copy-and-paste.)

The copy-and-paste feature allows you to copy a block of text from a console window by drag-selecting it with the mouse and pressing RightAmiga-C; and then paste it into another console window by positioning the cursor and pressing RightAmiga-V. CONCLIP expands the functionality of this feature, allowing the user to paste text not only into other console windows, but also into the window of any application that supports the clipboard.

CONCLIP is normally executed as part of the default Release 2 and 3 startup-sequence script. The program requires that the iffparse.library file be in the LIBS: directory, and the clipboard.device file be in the DEVS: directory.

Release 2 and 3 Format

CONCLIP [*UNIT clipnum*] [*OFF*]

Explanation of Parameters and Keywords

[*UNIT clipnum*] The UNIT option allows you to specify a clipboard device unit number from 0 to 255. This allows the exchange of data with applications that use a clipboard device number other than 0, the default device number. You do not have to enter the optional keyword UNIT when changing the unit number—entering a *clipnum* value is sufficient.

[OFF] This optional keyword can be used to turn the console device's support of the clipboard off, and return it to the use of its own internal buffer for copy-and-paste. Under normal circumstances, there is no reason to turn clipboard support off.

Examples

1. Direct the Release 2 and 3 console device to use the Amiga clipboard for its copy-and-paste function:

```
CONCLIP
```

2. Change the clipboard device unit number used by the console device to unit 1:

```
CONCLIP 1
```

COPY Command

Location: C:

Function

Copies one or more files or directories to a device, and as an option, lets you give the copy a name different from the original. If the destination device already contains a file of the same name, the new copy replaces that file. COPY can create duplicate copies of a file on the same disk as the

original, if different names are used for copies in the same directory, or if the files are copied to different directories.

1.3 Format

COPY [***FROM***] *fromname* [***TO***] *toname* [***ALL***] [***QUIET***] [***BUF*** or ***BUFFER*** =*num*] [***DATE***] [***COM***] [***NOPRO***] [***CLONE***]

Release 2 and 3 Format

COPY [***FROM***] *fromname* [***TO***] *toname* [***ALL***] [***QUIET***] [***BUF*** or ***BUFFER*** =*num*] [***DATE***] [***COM***] [***NOPRO***] [***CLONE***][***NOREQ***]

Explanation of Parameters and Keywords

[***FROM***] *fromname* Specifies the directory or file(s) you want copied. The keyword FROM is not needed as long as the source and target files are named in the correct order (*fromfile*, then *tofile*). If you change the order (COPY TO *tofile* FROM *fromfile*), the keyword FROM is required.

When a directory is specified as the FROM source, all files within the directory are copied. If no directory is specified, but the TO keyword is used (for instance COPY TO RAM:), the current directory is assumed to be the source, and all files are copied. As of version 1.3, a pair of double quotes ("") can also be used to refer to the current directory.

When you're copying individual files, and not an entire directory, you may use pattern matching to copy every file in the directory which matches the pattern. Under AmigaDOS Release 2 and 3, you may also specify multiple FROM files, each separated by spaces. Under version 1.3, you cannot use pattern matching with directory names. If you attempt to copy directories with patterns, nothing actually is done. Under Release 2 and 3, all of the directories that match the pattern will be copied.

If a physical disk drive is specified, the root directory of the drive is used as the FROM source. If a logical device name is specified, the directory path associated with it is used as the FROM source (see the ASSIGN command for more details).

[***TO***] *toname* Specifies the TO target (where you want to put the FROM files you are copying). The keyword TO is necessary only if the TO destination is listed *before* the FROM source.

When copying a single file, a device name or directory or filename can be used as the destination. A pair of double quotes ("") can be used to specify the current directory as the destination. When *toname* is a directory or device name, the name of the new file will be the same as the old name.

If the target file is in the same directory as *fromname,* you must specify a *toname* that is different from the original (since you can't copy a file to itself or have two files of the same name in the same directory). If the file is to be copied to a directory or a disk drive different from the one on which *fromname* resides, *toname* may be the same as or different from the original filename. If a file of the same name already exists in the target area, the existing file will actually be deleted and a new file with the same name is created and copied to. For this reason, a file that has been protected from deletion with the PROTECT command cannot be copied to.

If a directory is being copied to the same disk, a different directory path must be used for *toname.* In versions prior to 1.3, AmigaDOS assumes that the TO directory already exists, and the COPY will fail if it does not. Versions 1.3 and above create the new directory if it doesn't already exist, and then copies the file or files.

If a logical device name is specified, the directory path associated with it is used as *toname* (see the ASSIGN command for more details).

If *toname* is a physical disk drive, the root directory of the disk in that drive is assumed to be the target directory.

toname may be other physical devices known to the system. For instance, copying files to RAM: places a copy of the files on a RAM disk (see Chapter 4 for more details on RAM:). The contents of a file may also be copied to an attached printer by specifying PRT: as the target. Using AmigaDOS version 1.3 and above, you may have your Amiga speak the file by copying it to SPEAK:

[***ALL***] If you use this keyword, any files, subdirectories, and the files in the subdirectories located in *fromname*'s directory will be copied to the *toname* directory (normally, only files in that directory, not subdirectories, are copied). Subdirectory entries corresponding to those found in the FROM directory will automatically be created in the TO directory. (You might say that this command does the MAKEDIRty work for you.)

[***QUIET***] When copying multiple files (due to the use of pattern matching or the ALL keyword), the name of the files being copied and directories created are displayed unless this keyword is specified. Suppressing the output makes things a little quicker, and can prevent the user from becoming confused if you COPY files from within a script.

[***BUF*** or ***BUFFER*** = *num*] The BUFFERS option can be used to set the number of 512-byte buffers that are used during the copy. The default is

200 buffers or 100 Kbytes. You may want to use fewer buffers when copying a large file to the RAM, in order to avoid running out of memory. Similarly, you may want to increase the number of buffers when copying a large file from one disk to another on a single-drive system with a lot of RAM, to reduce the number of disk swaps during the copy. In version Release 2 and 3, BUF=0 causes the copy command to use a buffer that is the same size as the file.

[***DATE***] If the optional DATE keyword is used, the creation date of the original file is copied to the new file. By default, a new creation date is set when a file is copied.

[***COM***] If the optional COM keyword is used, the FILENOTE comments from the original file are copied to the new file. Normally, the new file is created without comments.

[***NOPRO***] By default, a copy of a file retains the same the protection bits of the original. The NOPRO keyword can be used to create a copy that has only the default protection bits (read, write, execute, and delete) set.

[***CLONE***] The CLONE keyword can be used to make the file copy an exact duplicate of the original, with the same creation date, comments, and protection bits set. Using this option is the same as using both DATE and COM keywords.

[**NOREQ**] This option suppresses the requester that asks you to insert a volume that COPY cannot find. Normally, for example, if you mistakenly typed *COPY DEVS:printer.device TO RAM:*, AmigaDOS would pop up a requester asking you to insert volume DEVS:. If you are using the copy command in a script, however, you may not want to present the user with a requester if the command fails. The NOREQ option will cause the command to fail without presenting a requester.

Examples

1. Copy a file called *myfile1* to *myfile2* in the same directory:

```
COPY FROM myfile1 TO myfile2
```

or

```
COPY TO myfile2 FROM myfile1
```

or

```
COPY myfile1 myfile2
```

Note that the FROM and TO keywords are optional, *unless* you reverse the order of the filenames (by putting the name of the destination file before that of the source).

2. Copy all files in the root directory of the volume in floppy disk 0 (df0:) to the volume in disk drive 1 (df1:):

```
COPY df0: df1:
```

3. Copy all files on disk drive 1 to disk drive 2, including subdirectories. Don't display the status of each copy operation:

```
COPY df1: df0: ALL QUIET
```

4. Copy a file called *burgers* in the current directory to a file of the same name in a different directory called *:fast/food,* which is in the root directory of the same disk:

```
COPY burgers :fast/food
```

5. Copy all files in the current directory to a RAM disk, retaining the original creation date of each file:

```
COPY TO RAM: DATE
```

or

```
COPY "" RAM: DATE
```

6. Copy all files ending in *.bas* from the current directory to the directory *Basicfiles* on df1:.

```
COPY #?.bas TO df1:Basicfiles
```

Note that if the root directory of the volume in df1: does not already contain the directory *Basicfiles,* the copy will fail under versions below AmigaDOS 1.3.

7. Copy selected files from the C: directory to the current directory.

```
COPY C:(DIR|DELETE|COPY|LIST|RUN) ""
```

CPU Command

Location: Release 2 and 3 C:

Function

Used to identify which processor in the Motorola 680x0 family is installed in the computer, and to set a number of options that are available with advanced processors such as the 68020, 68030, and 68040.

Release 2 and 3 Format

CPU [*CACHE*] [*NOCACHE*] [*DATACACHE*] [*NODATACACHE*] [*INSTCACHE*] [*NOINSTCACHE*] [*EXTERNALCACHE*] [*NOEXTERNALCACHE*] [*COPYBACK*] [*NOCOPYBACK*] [*BURST*] [*NOBURST*] [*INSTBURST*] [*NOINSTBURST*] [*DATABURST*][*NODATABURST*] [*TRAP*] [*NOTRAP*] [*FASTROM*] [*NOFASTROM*] [*NOMMUTEST*] [*CHECK 68010* or *68020* or *68030* or *68040* or *68881* or *68882* or *FPU* or *MMU*]

Explanation of Parameters and Keywords

[*CACHE*]
[*NOCACHE*]
[*DATACACHE*]
[*NODATACACHE*]
[*INSTCACHE*]
[*NOINSTCACHE*]
[*EXTERNALCACHE*]
[*NOEXTERNALCACHE*]
[*COPYBACK*]
[*NOCOPYBACK*] All of these commands are used to control the cache features of the various advanced processors. Processors like the 68020, 68030, and 68040 all have an instruction cache, which can store a series of instructions within the processor itself. This reduces the frequency with which the processor has to fetch instructions from memory, and allows the instructions to be executed at top speed. The INSTCACHE flag turns the instruction cache on, while the NOINSTCACHE flag turns it off. In addition to the instruction cache, processors like the 68030 and 68040 have a data cache, which allows access to data without making several fetches from memory. The DATACACHE option turns this data cache on, and the NODATACACHE option turns it off. You may use the CACHE flag to

turn on all caches, and the NOCACHE switch to turn them all off. Under AmigaDOS 1.3, caches are turned off by default; while under Release 2 and 3, they are turned on.

AmigaDOS 2.1 added some switches to help manage the 68040 processor. The data cache of the 68040 has a mode in which changes made to the cache are not automatically copied to memory. This copyback mode is enabled with the COPYBACK switch, and can be turned off with NOCOPYBACK option. Some software is not compatible with this mode, so by default it is left off. In addition to internal cache, some processors provide for external caching. The EXTERNALCACHE and NOEXTERNALCACHE options are used to turn this external caching on and off.

[***BURST***]
[***NOBURST***]
[***INSTBURST***]
[***NOINSTBURST***]
[***DATABURST***]
[***NODATABURST***] Some advanced processors include a special mode that allows them to quickly feed memory to the caches, if the memory is connected so as to support this special burst mode. The INSTBURST and DATABURST options are used to turn on burst mode for instructions and data respectively, while NOINSTBURST and NODATABURST turn them off. To control both at once, use BURST or NOBURST.

[***TRAP***]
[***NOTRAP***] These commands can be used to set or clear a memory access trap which will warn you if a program tries to access the lowest 256 bytes of memory, or any memory above the normal 16 megabyte address space of the 68000 processor. This feature is used by programmers to make sure that their programs are not accidently writing to random areas of memory. Information about illegal memory usage is sent out the serial port. You must be using an advanced processor (68020 or above) in order to use this debugging feature, as well as having a 9600 bps terminal connected to the serial port.

[***FASTROM***]
[***NOFASTROM***] On models that come with a 68000 processor, the Kickstart ROM is accessed 16-bits at a time. If you add an accelerator board that has a 32-bit processor and an MMU, you may want to move the Kickstart image to the 32-bit RAM, and use the MMU to make it appear as

if that RAM is addressed where the Kickstart ROM usually resides. The FASTROM option moves the ROM image to RAM (this will use up 256K-512K of 32-bit memory), while the NOFASTROM clears the ROM image from RAM and uses the original Kickstart.

[***NOMMUTEST***] Allows the Memory Management Unit (MMU) to be changed for the FASTROM option without checking to see if it is already in use.

[**CHECK** ***68010*** or ***68020*** or ***68030*** or ***68040*** or ***68881*** or ***68882*** or ***FPU*** or ***MMU***] The CHECK option can be used to check for the existence of a particular type of processor, floating point co-processor, or memory management unit. If you use the command *CPU CHECK 68030*, for example, the command will return a code of zero if you have a 68030 or 68040 processor installed, and a return code of five (WARN) if you have a 68020, 68010, or 68000.

Examples

1. Turn on the instruction cache, turn the data cache off, and move the Kickstart ROM image to protected 32-bit memory:

```
CPU INSTCACHE NODATACACHE FASTROM
```

2. Within a script, check to see if a 68020 (or higher) processor is installed:

```
CPU >NIL: CHECK 68020
IF WARN
   Echo "You only have a 68000 or 68010"
ELSE
   Echo "You have a 68020, 68030, or 68040"
ENDIF
```

DATE Command

Location: 1.3 and Release 2 and 3 C:

Function

Used to display, change, or store the current setting of the system date and time. Under 1.3, the SETCLOCK command is used to automatically read the day and date from the hardware clock (if present) during the startup-

sequence script. Under Release 2 and 3, this clock is read by the operating system, without using SETCLOCK. On Amiga 1000 or 512K Amiga 500 systems that don't have a hardware clock, AmigaDOS checks the boot-up disk for the date of the most recently modified or created file and sets the system date a bit in advance of that. The changes made to the system date and time using the DATE command are only temporary, and do not affect the hardware clock unless you save them with the SETCLOCK SAVE command.

Format

DATE [*date*] [*time*] [***TO*** or ***VER*** *name*]

Explanation of Parameters and Keywords

[*date*] The day of the month, the month, and the year to which the system date will be set. A specific desired date is typed in as *DD-MMM-YY*. *DD* is a two-digit number representing the day of the month to be set. Versions prior to 1.3 require a leading zero for dates before the 10th of the month, while Version 1.3 and higher do not. *MMM* is the first three letters of the month's English name, and *YY* is the last two digits of the year.

AmigaDOS also allows indirect references for setting the date. YESTERDAY, TODAY, and TOMORROW are valid values for *date*. YESTERDAY moves the present system date back by one day, TOMORROW moves the present value of the system date forward one day, and TODAY leaves the date unchanged.

The days of the week, SUNDAY through SATURDAY, can also be used as values for *date*. If the day specified is different from the current day of the week setting, the system date is advanced to match the specified day of the week. For instance, specifying WEDNESDAY when the current system day of the week is SUNDAY advances the system date by three days.

Specifying *date* does not alter the current system time.

[*time*] The time of day to which the system clock is to be set. The time should be entered in the form *HH:MM:SS*, representing hours, minutes, and seconds of the desired clock setting. Versions prior to 1.3 require two-digit numbers with leading zero, if necessary; as of version 1.3, the leading zero became optional. If seconds or minutes and seconds are omitted, they are set to zero. System time is kept in 24-hour format, also referred to as

military time. Thus, 1:00 p.m. is expressed as 13:00, and midnight as 00:00. Specifying *time* does not alter the current system date.

[***TO*** or ***VER*** *name*] The TO and VER options allow you to store the present system date and time to *name*, which may be a disk file or a physical device such as a printer. TO and VER are equivalent keywords and may be used interchangeably. If TO or VER is used when setting the time and/or date rather than just reading its current status, a blank file overwrites the specified file since the DATE command sends no output when used to change a setting. AmigaDOS does remember the date and time that the blank file was written, however.

Examples

1. Display the current system date and time:

```
DATE
```

2. Set the system date and time to September 8, 1987, 10:05 a.m.:

```
DATE 08-Sep-87 10:05:00
```

3. Change the current system date to the next day, and change the current system time to 4:00 p.m.:

```
DATE TOMORROW 16:00
```

4. If the current system day of the week is not Wednesday, change the system date to that of the next Wednesday. Leave the time alone:

```
DATE WEDNESDAY
```

Note: If the current system day of the week is Wednesday, the date remains unchanged.

5. Copy the current system date and time to a filenamed *Timestamp:*

```
DATE TO Timestamp
```

6. Change the system date and time to August 19, 2001, 2:00 a.m.:

```
DATE 2: 19-Aug-01
```

Note: Since date and time have different formats, the order in which they are specified can be reversed. Also, AmigaDOS treats year references from 78 to 99 as 1978 to 1999, and from 00 to 45 as 2000 to 2045. The DATE command does not allow years from 46 to 77. If you accidentally set the year to a number from 0 to 45, all files that you subsequently create in

that session will have a date stamp from the 21st century. The next time you set the clock to the correct date—say, 14-JUN-93—all files stamped 2043 will appear on the directory with the date-stamp *Future* since 2043 is in the future from the point of view of 1993. This renders the date-stamp useless, since you won't even be able to tell the order in which the files were created. Version 1.3 of the Workbench introduced the SETDATE command to let you change bogus date stamps.

DELETE Command

Location: 1.3 and Release 2 and 3 C:

Function

Removes files and directories from the designated drive. If no drive is designated, the current default drive is assumed. If no directory path is specified, the files and/or directories are deleted from the current directory. DELETE accepts patterns as well as specific filenames, and as of version 1.3, the 31 character limit for wildcards has been removed. See "Pattern Matching (Wildcards)" in Chapter 3 for more information. Ordinarily, you cannot retrieve files that are deleted, so care should be exercised when erasing multiple files using the wildcard feature.

1.3 Format

DELETE *name(s)* [***ALL***] [***Q*** or ***QUIET***]

Release 2 and 3 Format

DELETE *name(s)* [***ALL***] [***Q*** or ***QUIET***] [***FORCE***]

Explanation of Parameters and Keywords

name(s) The name of the file(s) or directory entry(s) to be removed. Versions prior to Release 2 and 3 only allow up to ten file or directory names to be entered within a single DELETE command, but Release 2 and 3 removed that limit. A pattern may be used in lieu of specific file or directory names. When an attempt to delete an item is unsuccessful, DELETE continues until it has attempted to process all specified items.

[***ALL***] When this keyword is used, DELETE erases all files and subdirectories contained within the directory as well as the directory itself. Attempts to DELETE directories that contain any files or subdirectories will fail unless those files and subdirectories are deleted first or the ALL keyword is used.

[*Q* or *QUIET*] Suppresses the status reports that are issued as each file's deletion is attempted during a DELETE which erases more than one file.

[*FORCE*] Normally, if the *d* protection bit of a file or directory is not set, that file or directory is protected from deletion. If you use the FORCE option, however, all indicated files and directories will be deleted, without regard to the protection bits.

Examples

1. Erase the file *unwanted*:

```
DELETE unwanted
```

2. Erase the files *oranges, kiwi, peaches,* and *herbs*:

```
DELETE oranges kiwi peaches herbs
```

3. Erase the directory *phonebook* and all files and subdirectories within it. Don't report on the status of each deletion attempt:

```
DELETE phonebook ALL QUIET
```

4. Erase the current directory and all the files and subdirectories within it:

```
DELETE #? ALL
```

5. Delete all files in the current directory which start with the letter *a*, *b,* or *c*:

```
DELETE (a|b|c)#?
```

DIR Command

Location: 1.3 and Release 2 and 3 C:

Function

Lists the files and subdirectories within the present directory or another specified directory. The list is normally grouped into a list of subdirectories, followed by a sorted list of files. Options allow you to use a special interactive mode and/or ask for an extended listing which lists the contents of subdirectories as well. The directory listing may be aborted at any time by the CTRL-C key combination (hold down CTRL and press C).

1.3 and Release 2 and 3 Format

DIR *dirname* [***OPT A*** or ***OPT D*** or ***OPT I*** or ***OPT AI***] [***ALL***] [***DIRS***] [***INTER***] [***FILES***]

Explanation of Parameters and Keywords

dirname The name of the directory or logical device whose contents you want displayed. An AmigaDOS pattern may also be used to display multiple directories. If no directory or AmigaDOS pattern is specified, the current directory is displayed.

[***OPT A*** or ***OPT D*** or ***OPT I*** or ***OPT AI***] When the OPT A keyword is used, the display includes the contents of any subdirectories residing in the directory being listed. This lets you see everything in a directory with a single command. OPT D displays only the names of subdirectories within the specified directory, and not those of the files.

OPT I invokes the special interactive mode of DIR. In interactive mode your system pauses as each subdirectory entry or file is listed, displaying a question mark to the right of the entry. When in interactive mode, you may use any of the following subcommands:

Key(s)	Function
<RETURN>	Doesn't do anything with the current item. Goes on to the next item in the DIR listing.
T <RETURN>	Types (lists) the file. To pause the display while listing, hit the space bar or any key. To resume after pausing, press the BACKSPACE key or CTRL-X. When you want to abandon the listing of the file contents before the complete file has been listed, type CTRL-C. You'll be returned to the interactive mode. T is an invalid option for subdirectories.
DEL <RETURN>	Erases the file. Subdirectories may be erased only if they're empty.
E <RETURN>	Enters a subdirectory. Displays the files and subdirectories within a subdirectory. The listing remains in interactive mode. Not a valid option for a file.
B<ENTER>	Moves back to the previous directory after you have descended into a new directory with the E command.
Q<ENTER>	Quit. Abandons the DIR listing and goes back to the CLI prompt.

As of Workbench 1.3, a new COMMAND = option was added to interactive mode, which allows almost any AmigaDOS command to be executed from interactive directory mode.

OPT AI combines both the A and I options, resulting in an interactive listing of all files and directories within the specified directory.

[*ALL*] As of Workbench 1.3, this is an acceptable synonym for OPT A.

[*DIRS*] Starting with version 1.3, this may be used instead of OPT D.

[*INTER*] As of Workbench 1.3, this is an acceptable synonym for OPT I.

[*FILES*] This option, added with Workbench 1.3, allows you to display only the files within a directory, and not the subdirectories.

Examples

1. List the current directory:

```
DIR
```

2. List all files and directories on disk drive df1: in interactive mode:

```
DIR df1: OPT AI
```

3. List all files and directories in directories beginning with the letter *Z*:

```
DIR Z#? OPT A
```

4. Display all of the directories and subdirectories that are contained within the root directory of the current disk:

```
DIR : ALL DIRS
```

DISKCHANGE Command

Location: 1.3 and Release 2 and 3 C:

Function

Lets AmigaDOS know when you've changed the disk in a 5¼-inch disk drive, or removable-media hard drive. With version 1.2, AmigaDOS added support for 5¼-inch drives as DOS devices, using the MOUNT command, and under 1.3, it became possible to use auto-mounting, removable-media hard drives. Unlike the normal 3½-inch drives, however, some of these disk drives don't have a sensing mechanism which can tell the Amiga when a disk has been removed or inserted. Therefore, if you change a disk and then

try to read it, the system will try to access it as if it was the disk that was previously mounted, confusing AmigaDOS completely. Therefore, you must enter the DISKCHANGE command to let AmigaDOS know you've changed the disk, so it can read it and adjust to the new layout. Note that some drivers for removable-media disks can sense when you change media, and will automatically do a DISKCHANGE for you.

See the MOUNT command for more information about mounting 5¼-inch drives as DOS devices.

Another use of DISKCHANGE is to notify Workbench of a change in the volume name of a disk that was made using the CLI RELABEL command.

1.3 and Release 2 and 3 and 3 Format

DISKCHANGE *drive*

Explanation of Parameters and Keyword

drive The device name of the drive. Since the internal drive is always df0:, an external 5¼-inch drive will usually be mounted as df1: (if there's no external 3½-inch drive), or df2: (if there *is* an external 3½-inch drive). Of course, both 5¼-inch floppies and removable-media hard drives can be mounted with any device name that you choose.

Examples

Inform DOS that you've changed the disk in the sole external disk drive, which happens to be a 5¼-inch drive:

```
DISKCHANGE df1:
```

DISKCOPY Command

Location 1.2 C: 1.3 and above SYS:SYSTEM

Function

Copies the entire contents of one disk volume to another. DISKCOPY can be used to make copies of your work to new disks or to used disks containing files that are no longer needed. When you use DISKCOPY, any information previously stored on the destination disk is erased. While many other computer systems require that new disks be specially prepared before use, AmigaDOS DISKCOPY automatically prepares, or *formats,* disks as

the information from the original disk is copied. In fact, copying an existing disk takes about the same amount of time as formatting a new one. Use DISKCOPY regularly to make backup copies of your work and non-copy-protected program disks.

Though DISKCOPY copies entire disks, it takes about the same amount of time to copy a disk full of data as to copy one which has only a few short files on it. If the amount of data you want to copy is relatively small, using the COPY command may be faster than DISKCOPY.

DISKCOPY is usually used to copy the contents of one 3½-inch floppy disk to another, but it can be used with other devices, as long as the source and the destination disks are of identical format and storage capacity. For example, it is possible to use DISKCOPY to copy the contents of the RAD: RAM disk to or from a floppy, if that disk has been set to 80 tracks, 11 blocks per track, the same as a floppy.

Format

DISKCOPY [***FROM***] *source drive [**TO**] destination drive* [***NAME*** *volname*][***NOVERIFY***] [***MULTI***]

Explanation of Parameters and Keywords

[***FROM***] *source drive* The name of the drive in which the disk you wish to copy will be mounted. If your system has only one drive, this will be df0:. If you have two drives, you may use df0: or df1:. While AmigaDOS supports up to four drives—df0:, df1:, df2:, and df3:—Amiga 500 owners may find, however, that the power supply furnished with the computer is not adequate for more than two or three drives. If *source drive* is the first argument of the DISKCOPY command, the FROM keyword is optional.

[***TO***] *destination drive* This is the name of the drive in which the disk to be copied to will be mounted. If your system has only one drive, this will be df0: (the same as your FROM device). Single-drive DISKCOPY operations require that both the source and destination disks be removed and reinserted multiple times. If you have 512K, each disk must be inserted three times. On an old 256K Amiga 1000 system, DISKCOPY requires eight insertions of each disk—1000 owners are strongly recommended to upgrade memory, and/or get a second disk drive. If your system has two drives, no disk swapping during the copy process is required as long as you specify different drives for the FROM and TO devices.

With versions 1.3 and below, the TO keyword *must* be used with the DISKCOPY command. AmigaDOS Release 2 eliminated that requirement, allowing for the first time the format *DISKCOPY df0: df1:*.

[***NAME** volname*] The volume name that will be given to the copy of the original disk. If the volume name contains spaces, it must be enclosed by quotation marks. If *volname* is not specified, the copy will have the same name as the original. AmigaDOS can still distinguish between volumes with the same name based upon information stored on the duplicate disk. The NAME keyword is required if a volume name is specified.

When DISKCOPY is invoked, you'll be prompted to insert the disks required to complete the copy operation. Status messages keep you advised as each track is copied. A standard AmigaDOS format 3½-inch disk requires 80 tracks of information to be read and written.

You can stop the copy process after issuing the command (when the system is waiting for the disk(s) to be inserted) by pressing CTRL-C followed by the RETURN key. You'll then be returned to the CLI prompt. If you press CTRL-C after the copy process has started, the copy is abandoned, and all information already on the destination disk is lost.

[***NOVERIFY***] Normally, the DISKCOPY command reads each sector after writing it, to verify that it was copied correctly. Using the NOVERIFY option eliminates this step, speeding up the copy process.

[***MULTI***] If you have enough memory to load the entire disk at once, you can use the MULTI option to write multiple copies without reading the disk in each time. When you use the MULTI option, the entire disk is read first, and then you are prompted to insert each destination disk in turn.

Examples

1. Make a copy of a disk with a single-drive system. The copy is to have the same volume name as the original:

```
DISKCOPY FROM df0: TO df0:
```

2. Make a copy of a disk on a dual-drive system without verifying, copying the original from the external drive to the Amiga's internal drive. The copy is to have the same volume name as the original:

```
DISKCOPY FROM df1: TO df0: NOVERIFY
```

3. Make a copy of a disk with a dual-drive system, copying the original from the internal drive to the external drive. The copy is to be named *Kings:*

```
DISKCOPY df0: TO df1: NAME "Kings"
```

Note: In this example the optional FROM keyword has been omitted.

DISKDOCTOR Command

Location: 1.3 and Release 2 C:

Function

DISKDOCTOR attempts to reconstruct the directory and file structure of corrupted disks. A disk can become corrupted because of a defect in the media, exposure to magnetic fields, or operator error (such as removing the disk while the drive activity light is still on). When this happens, AmigaDOS is unable to read the disk correctly, and it displays a system message such as *Volume Programs is a Read/Write error, Volume Programs is not validated, Disk is unreadable, Checksum error,* or even *Not a DOS Disk.*

Since each directory item contains duplicate information about the preceding and following entries, it's sometimes possible to reconstruct the corrupted disk information. DISKDOCTOR restores as much information as can be salvaged.

As of Workbench 1.3, DISKDOCTOR can be used to salvage disks that have been formatted with the Fast File System, but only if the DOSTYPE keyword in the file DEVS:Mountlist has been set to 0x444F5301.

Trying to write to damaged disks is risky. Further damage may occur while DISKDOCTOR is changing the disk, resulting in the loss of all information on the disk. For this reason, you should try to DISKCOPY the disk before using DISKDOCTOR on it. If DISKDOCTOR manages to save some data, you should always use the COPY command to copy all of the files onto another disk, since DISKDOCTOR will not have repaired the suspect media. You should not re-use the damaged disk again.

Programs that salvage files from the damaged disk to another disk are usually preferable to those that try to repair the damage. For this reason, programs like the public domain Disksalv (found on disk 251 of the Fred Fish collection) or New Horizon's Quarterback Tools are generally better alterna-

tives than DISKDOCTOR. Because of its unrealiability, the DISKDOCTOR command was removed from the Workbench as of version 2.1.

Format

DISKDOCTOR *drive*

Explanation of Parameters and Keywords

drive The device name of the drive containing the damaged disk. Note that DISKDOCTOR can be used with hard drives or hard drive partitions (dh0: or dh1:), as well as with floppy disk drives (df0: or df1:), but should be used only as a last resort on hard disks.

Examples

Attempt to restore the data on a damaged floppy disk in the external disk drive:

```
DISKDOCTOR df1:
```

Note: After DISKDOCTOR is finished with the disk, copy the files onto a new disk. Do *not* use DISKCOPY to transfer the files, since it will copy corrupted information as well. Instead, use the FORMAT command to initialize a new disk, then use the COPY command in the form:

```
COPY df0: TO df1: ALL
```

where *df0:* contains the "doctored" disk, and *df1:* contains the newly formatted disk. If the corrupted disk was bootable, you may use the INSTALL command on the new disk to also make it bootable.

ECHO Command

Location: 1.3 C: Release 2 and 3 Internal

Function

ECHO is used in command files to display a message on the system screen. This command is most often used in script files, to report the progress of operations to the user, who might otherwise be unaware of the sequence of commands which were being executed. See Chapter 5 for more information on using ECHO in scripts. AmigaDOS Workbench 2 users may also want to use the text manipulation and file output features of ECHO to create environment variables, as in example 4, below.

1.3 Format

ECHO *string* [***NOLINE***] [***FIRST*** *num*] [***LEN*** *num*]

Release 2 and 3 Format

ECHO *string* [***NOLINE***] [***FIRST*** *charnum*] [***LEN*** *numchars*][***TO*** *destination*]

Explanation of Parameters and Keywords

string The text of the message which will be printed to the currently active output stream. The output stream usually will be the system display, but text can be redirected to a file or device. If *string* contains spaces, AmigaDOS 1.3 and below requires it to be contained within quotation marks. For Release 2 and above, the quotes are not necessary if the string appears last on the command line. If you are using the FIRST or LEN options to print part of a string, however, you must always put quotes around text that includes spaces.

If you want the cursor to skip to a new line at some point during the printing, put the characters *N where you want the line break to appear. If you use this formatting character, you must enclose the string in quotes, whether in 1.3 or Release 2 and 3.

[***NOLINE***] Normally, the ECHO command appends a new line character to the end of the string. The NOLINE option, introduced in Workbench 1.3, allows you to omit the new line after printing the string, so that the next word that is output to the CLI console appears on the same line as the text that was ECHOed. This allows you to build a single line of text using the output of two or more commands.

[***FIRST*** *charnum*] This optional keyword, introduced in Workbench 1.3, allows you to print only a selected part of the text string. The FIRST keyword is used to designate the first character within the string that is to be printed. For example, if the echo string is "This is a test", FIRST 3 designates character 3, the "i" in "This", as the first letter to be printed.

[***LEN*** *numchars*] Another sub-string option introduced in version 1.3, the LEN keyword is used to designate the length of the sub-string that is to be printed. In the above example, FIRST 5 LEN 4 would print the "is a" portion of the string. If no FIRST keyword is used to indicate the starting position of the sub-string, LEN backs up *numchars* number of characters from the end of the string. In the example above, if there were no FIRST, LEN 4 would print "test".

Examples

1. A command that executes a background SORT of a file called *sortsource* on the external drive to a file called *sortdest* on the same drive and notifies you when the operation is complete:

```
RUN SORT FROM df1:sortsource TO df1:sortdest + ECHO
    "Sort Complete"
```

2. A command that executes a background COPY of all files and subdirectories in a directory called *work/mydir* on the current default drive to a directory called *storage/archive* on the same drive:

```
RUN COPY FROM :work/mydir TO :storage/archive ALL
    QUIET + ECHO "That's All Folks"
```

3. Create a one-line file called *:Joey* that contains the text string *I have the power:*

```
ECHO > :Joey "I have the power"
```

4. A Release 2 and 3 command file that creates an environment variable named *CPU*, and then performs one action if the computer has a 68000 processor, and another if it has an accelerated processor.

```
ECHO FIRST 9 LEN 5 "'CPU'" TO Env:CPU
IF $CPU EQ "68000"
ECHO "Your computer's processor is a plain old 68000.
    Too bad."
ELSE
ECHO Your computer has an advanced processor, you
    lucky devil.
ENDIF
```

The first command uses the "backtick" feature, along with text manipulation and file redirection to create the environment variable. The *"CPU"* command is in quotes, because its output includes spaces. The last ECHO command does not need quotes, however, because it comes last on the line, and does not use any of the partial string options.

ED Command

Location: 1.3 and Release 2 and 3 C:

Function

The ED command is used to edit the contents of a text file using AmigaDOS's full-screen editor. See Chapter 6 for complete information on using the full-screen editor.

1.3 Format

ED [***FROM***] *name* [***SIZE*** *numchars*]

Release 2 and 3 Format

ED [***FROM***] *name*] [***SIZE*** *numchars*] [***WITH*** *comfile*] [***TABS*** *interval*] [***WINDOW*** *device*] [***WIDTH*** *columns*] [***HEIGHT*** *rows*]

Explanation of Parameters and Keywords

[***FROM***] *name* The name of the AmigaDOS file which you wish to edit using the full-screen editor. If *name* is the first argument in an ED command statement, the FROM keyword need not be specified. If the file already exists, its contents are loaded into the editor's workspace. If the file doesn't already exist, it is dynamically created by the editor. You do not need to include the FROM keyword.

[***SIZE***] *numchars* This option is used to set the size of the editor's workspace. If *numchars* is the second argument in an ED command statement, the SIZE keyword need not be given. If no value for *numchars* is specified, the editor's default workspace is 40,000 bytes. To edit files larger than that, specify SIZE with a value for *numchars* larger than the size of the file to be edited. If the workspace size selected is not large enough, the editor will display the message *SIZE of* numchars *too small.*

[***WITH*** *comfile*] This AmigaDOS Release 2 and 3 option allows you to specify an ED command file to execute automatically. A command file is a text file containing a number of extended mode ED commands, each on its own line, which are executed as if they were entered one after another from the keyboard. A command file can be used to edit a file on a completely automated basis.

[***TABS*** *interval*] The TABS option allows you to specify the interval to skip each time that the Tab key is pressed. The default tab stop interval is three.

[***WINDOW*** *device*]
[***WIDTH*** *columns*]
[***HEIGHT*** *rows*] These three optional arguments allow you to describe your terminal type, allowing you to use ED on a remote terminal. The *WINDOW* argument specifies the device, such as AUX:, or a console window description (CON:Left/Top/Width/Height/Title). The *WIDTH* and *HEIGHT* options show many characters to print in each column and row before scrolling the display.

Examples

1. Invoke AmigaDOS's full-screen editor to edit a file called *WorkInProgress* in the *Current/Stuff* directory:

```
ED :Current/Stuff/WorkInProgress
```

2. Invoke AmigaDOS's full-screen editor to edit a 90,000-byte file called *Big* in the root directory of drive df1:.

```
ED df1:Big SIZE 100000
```

EDIT Command

Location: 1.3 and Release 2 and 3 C:

Function

The EDIT command is used to edit the contents of a file using AmigaDOS's line editor.

Unless you're a real fan of line editors, give AmigaDOS's full-screen editor (ED) a try first, particularly if your're editing a text file. The full-screen editor is both more flexible and easier to use than EDIT. In all fairness, EDIT *does* have the ability to edit binary files and can execute a prestored list of line editor commands, which may be handy features for some users (though ED also gained this capability in version Release 2). See Chapter 7 for detailed information on EDIT.

1.3 and Release 2 and 3 Format

EDIT [***FROM***] *fromname* [***TO***] *toname* [***WITH***] *withname* [***VER***] *vername* [***OPT W*** *chars* or ***WIDTH*** *chars*] [***OPT P*** *lines* or ***PREVIOUS*** *lines*]

Explanation of Parameters and Keywords

[***FROM***] *fromname* The name of the file whose contents will be edited. If *fromname* is the first argument in the EDIT command, the FROM keyword is optional. EDIT requires *fromname,* and it must already exist.

[***TO***] *toname* The name of the file to which the edited text is saved when a Q or W subcommand is executed from within the line editor. If *toname* is the second argument in an EDIT command (following *fromname*), the TO keyword is optional.

If *toname* is different from *fromname*, the contents of the file used as input to the editor will not be replaced by a save from within the line editor. If *toname* is not specified and a save is executed from within EDIT, the contents of the original file will be moved to a temporary file called *:t/edit.backup,* and EDIT will rename its work file (where it temporarily holds edited data) to *fromname.*

[***WITH***] *withname* This option lets you specify a file which will be used as input to the line editor's command processor. The contents of *withname* should be a series of valid line editor subcommands. If *withname* is the third argument in an EDIT command (following *fromname* and *toname*), the WITH keyword is optional. IF *withname* is not specified, the line editor expects manual input from the keyboard.

[***VER***] *vername* Lets you specify where you want messages and verification output produced by the line editor sent; *vername* may be a file or logical device. If *vername* is the fourth argument in an EDIT command (following *fromname, toname,* and *withname*), the VER keyword is optional.

[***OPT W*** *chars* or ***WIDTH*** *chars*]

[***OPT P*** *lines* or ***PREVIOUS*** *lines*] These options let you set the maximum line length (WIDTH *chars*) and number of lines (PREVIOUS *lines*) that EDIT will keep memory resident. The default maximum line length is 120. The default number of lines is 40. Multiplying the value for PREVIOUS *lines* by WIDTH *chars* yields the amount of memory that EDIT reserves as a temporary work area. If you use just P*lines* or W*chars*, you must use the OPT keyword before them.

Examples

1. Edit a file called *mysource* in the current directory, using AmigaDOS's line editor. The edited data, if saved, will be stored under the same filename. The number of lines is to be set to 40 and line width to 120 (EDIT's default values):

```
EDIT mysource
```

2. Edit a file called *bigsource* in the current directory, using AmigaDOS's line editor. The edited data, if saved, will be stored under the filename *edited bigsource*. The number of lines is to be set to 1000 and line width to 120:

```
EDIT FROM bigsource TO "edited bigsource" OPT P1000
```

3. Edit a file called *universe* in the current directory, using AmigaDOS's line editor. When EDIT starts up, execute the list of line editor commands contained in a file called *autocommands* in the *myprocess/nebula/* directory on drive df1:. The edited data, if saved, will be stored under the same filename. Send all messages and verification displays from the line editor to the system printer. The number of lines is to be set to 40 and line width to 250:

```
EDIT universe WITH df1:myprocess/nebula/autocommands
    VER PRT: OPT W250
```

ELSE Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The ELSE command can only be used in a script file. It is used after an IF command to specify an alternative block of commands. If the condition tested by the IF command is not true, all the commands after IF and before ELSE are skipped, and all of the commands after ELSE and before ENDIF are executed. If the condition is true, all the commands after IF and before ELSE are executed, and all of the commands after ELSE and before ENDIF are skipped. See IF command.

1.3 and Release 2 and 3 Format

ELSE

Explanation of Parameters and Keywords

None

ENDCLI Command

Location: 1.3 C: Release 2 and 3 Internal

Function

ENDCLI terminates the current Command Line Interpreter process, and closes the CLI or Shell window.

Under AmigaDOS Release 2 and 3, a Shell window may also be closed by clicking on the close gadget (if any), or sending the window an end-of-file character (CTRL-\).

1.3 and Release 2 and 3 Format

ENDCLI

Explanation of Parameters and Keywords

None

Example

Open a new CLI window and issue a directory command within the new CLI. Close the CLI window with the ENDCLI command, returning to the CLI from which the NEWCLI command was issued:

```
NEWCLI
```

Note: A new CLI window will appear on your screen. The next two commands will appear within the new window as they're typed:

```
DIR
ENDCLI
```

ENDIF Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The ENDIF command is used only within a script file. It is used to terminate a block of commands specified by IF or ELSE. If the condition tested by the IF command is not true, all the commands after IF and before ELSE or ENDIF are skipped. If the condition is true, all the commands after IF and before ELSE are executed, and all of the commands after ELSE and before ENDIF are skipped. See IF command.

1.3 and Release 2 and 3 Format

ENDIF

Explanation of Parameters and Keywords

None

ENDSHELL Command

Location: 1.3 Alias Release 2 and 3 Internal

Function

Like ENDCLI, ENDSHELL terminates the current Command Line Interpreter program, and closes the CLI or Shell window. Under AmigaDOS 1.3, ENDSHELL is not really a command, but an alias for ENDCLI which is established by the default Shell-Startup file. Under AmigaDOS Release 2 and 3, it is an internal command.

1.3 and Release 2 and 3 Format

ENDSHELL

Explanation of Parameters and Keywords

None

ENDSKIP Command

Location: 1.3 C: Release 2 and 3 Internal

Function

ENDSKIP was added to Workbench 1.3 to designate the end of a SKIP clause in a batch file. When ENDSKIP is encountered within a SKIP clause, execution continues at the line following the ENDSKIP, and the condition flag is set to WARN. See SKIP command.

Format

ENDSKIP

Explanation of Parameters and Keywords

None

EVAL Command

Location: 1.3 and Release 2 and 3 C:

Function

The EVAL command, added in Workbench 1.3, is used to evaluate simple integer expressions with one or two arguments, and to print the resulting expression in a user-specified format. Any fractional amount results are discarded. EVAL is mainly used to perform calculations in scripts, to aid in chores such as counting loops.

1.3 and Release 2 and 3 Format

EVAL [***VALUE1*** = *value*] [***OP*** = *operator*] [***VALUE2*** = *value*] [***TO*** *filename*] [***LFORMAT*** = *string*]

Explanation of Parameters and Keywords

[***VALUE1*** = *value*] The first value in the expression to be evaluated. This value may be expressed as a decimal number (the default), a hexadecimal number (indicated with a leading 0X or #X) or an octal number (indicated by a leading 0 or a leading #, followed by other digits). You can also use the ASCII value of a text character, by prefacing it with a single apostrophe (for example 'a for 97, the ASCII value of the letter "a"). The keyword VALUE1 = does not have to be included if *value* appears as the first argument after the EVAL command.

[***OP*** = *operator*] A symbol which indicates the mathematical operation to be performed on the value or values. The supported operations and their symbols are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
mod	Modulus
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
<<	Bitwise shift left
>>	Bitwise shift right
-	Negation
xor	Bitwise exclusive or
eqv	Bitwise equivalence

The keywords OP = do not have to be included if the values and operator are presented in the correct order (value1 operator value2). If you change this order, the OP = and VALUE2 = keywords must be included to indicate which is the operator and which the second value. Note that if the addition operator (+) appears at the end of the line, it must have a space after it, so that the command interpreter doesn't confuse it with the sign that joins two command lines togeter. Without the space, EVAL will not recognize the final plus sign as an operator, and will give you an error message like *Bad args* or *value after keyword missing.*

[***VALUE2*** = *value*] The second value in the expression that is to be evaluated. It may be expressed in any of the forms mentioned above. The keywords VALUE2 = do not have to be included if the values and operator are in the correct order (value1 operator value2). They only need to be included if you change the order in which they're presented.

Under AmigaDOS Release 2 and 3, you may have multiple arguments for value2, each separated by its own operator (e.g. 3 + 5 - 6 * 2 \ 4). You may also use parentheses to change operator precedence (e.g. (3+5)*4 is different from 3+(5*4)). Under AmigaDOS 1.3, you may only use multiple arguments for VALUE2 if there are no spaces in the expression (e.g. 3+5-6 is acceptable, but 3 + 5 - 6 is not).

[***TO** filename*] This optional keyword is used to send the output of this command to a file whose name is indicated by filename.

[***LFORMAT** = string*] This optional keyword is used to specify the format of the text string that this command prints. By default, the program prints the answer in decimal format, but through the use of a text string, you may specify hexadecimal (%X), octal (%O), decimal (%N), or ASCII character (%C). The hexadecimal and octal format strings require a number after the letter, indicating how many digits should be printed (for instance a string of "%X8" indicates that the answer should be printed as 8 hex digits). You may also include any additional text you want in the format string. The command option

```
LFORMAT = "The answer is %N"
```

will cause the command to print the words "The answer is" in front of the value that it has calculated. Note that by default, EVAL does not terminate its output with a new line character. If you want the cursor to skip to a new line after some output is printed, you should include the characters "*N" within the format string where you want the line break to appear, and put the format string in quotes. If there are any spaces within the format string, the entire string should be placed within quotes as well.

Examples

1. Calculate the value of 10000 divided by nine, and print the answer as 4 hexadecimal digits, followed by a new line character.

```
EVAL 10000 / 9 LFORMAT = "%X4*N"
```

2. Use EVAL in a 1.3 command sequence file to decrement a loop counter, causing the script to execute exactly 5 times:

```
SETENV Count 1
LAB Loop
ECHO "Loop #" NOLINE
TYPE ENV:Count
EVAL >NIL: <ENV:Count value2=1 op=+ to=T:Temp<$$> ?
COPY T:Temp<$$> ENV:Count
IF VAL $Count NOT GT 5
  SKIP Loop BACK
ENDIF
DELETE >nil: T:Temp<$$> ENV:Count
ECHO "*N I'm done.Five loops is my limit. *N"
```

The fifth line from the top is the tricky one. The question mark is used to get the command to take the VALUE1 argument from the ENV:Count file, and the redirection to nil: prevents the user from seeing the template printed. See Chapter 5 for more information.

3. Use EVAL in a Release 2 and 3 command sequence file to decrement a loop counter, causing the script to execute exactly 5 times:

```
SET Count 1
LAB Loop
ECHO Loop #$Count
SET Count 'EVAL $Count + 1 '
IF VAL $Count NOT GT 5
  SKIP Loop BACK
ENDIF
UNSET Count
ECHO "*N I'm done.Five loops is my limit. *N
```

The "backtick" feature of Release 2 and 3 makes it much easier to feed the results of EVAL directly into the SET command.

EXECUTE Command

Location: 1.3 and Release 2 and 3 C:

Function

The EXECUTE command is used to invoke AmigaDOS command sequence files. Command sequence files are text files which contain a series of command lines which are executed sequentially once the command file has been started by EXECUTE. EXECUTE also can pass information to the command sequence file to be used as arguments for the commands contained in the file.

Command files may be nested by issuing an EXECUTE as one of the commands in the command sequence file. See Chapter 5 for more information on sequence files.

Under AmigaDOS 1.3 and higher, it is possible to execute a script just by typing its name (and path, if necessary), if the S (or script) protection bit of the script file is set. Such script files can be executed just like normal program files, without using the EXECUTE command.

1.3 and Release 2 and 3 Format

EXECUTE *name* [*arg1 arg2*,,,,]

Explanation of Keywords and Parameters

name The name of the command sequence file to be invoked, *name* is a required parameter.

EXECUTE sometimes needs to create temporary files in the course of executing a script. Under version 1.2, these files were created in the the :T directory. The version on Workbench 1.3 uses the T: directory if one has been ASSIGNed, otherwise it uses :T. Also, as of version 1.3, an EXECUTE script will substitute the characters <$$> with the task number of the CLI from which it is run. This feature is helpful in creating unique temporary filenames.

[*arg1 arg2*,,,,] Arguments to be passed to the command sequence file. Arguments may be any valid AmigaDOS string (including filenames and logical and physical devices).

Examples

1. Invoke a command sequence file called *commikazi* on drive df1:.

```
EXECUTE df1:commikazi
```

2. Invoke a command sequence file called *games.* Pass the arguments *lacrosse, bowling, prt:* and *df1:what/the/heck* to the command sequence file:

```
EXECUTE games lacrosse bowling prt: df1:what/the/heck
```

Note: The command sequence file being called must be written so that it will receive passed arguments. Before the command file is started up, EXECUTE examines the file for special directives and characters which tell it how to insert the passed information in the command sequence file's command stream.

Command sequence lines that contain directives for EXECUTE begin with a period (.).

Directives

.K *subname1 subname2....* or **.KEY** *subname1 subname2....* Defines substitution names for passed arguments. EXECUTE scans for these names, delimited by the angle bracket (< and >) characters in subsequent lines of the command file, and substitutes passed arguments in their stead. Each substitution argument may be further qualified by /A, /K, or /S (see "AmigaDOS Templates," page 170 earlier in this reference section for information on these qualifiers).

.BRA *n* Substitutes character *n* for the < character. This comes in handy if < is to be part of a substitution name.

.KET *n* Substitutes character *n* for the > character. This comes in handy if > is to be part of a substitution name.

.DOL *n* or **.DOLLAR** *n* Substitutes character *n* for the command file's normal default delimiter ($). Substitution arguments may assume a default if no corresponding argument is given to EXECUTE by the user. For instance, *<animal$squirrel>* substitutes the string *squirrel* for the substitution argument *animal.*

.space Defines a comment line.

.DEF *subname string* Assigns the value string to all occurrences of the substitution argument *subname.*

Examples

1. When the following EXECUTE command is issued:

```
EXECUTE sortvar ingress egress
```

and the contents of the command sequence file sortvar is:

```
.KEY SFILE/A TFILE/A HEX/S
IF HEX EQ ""
SORT <SFILE$mysource> <TFILE$mysorted>
ELSE
SORT <SFILE#mysource> <TFILE#mysorted> OPT H
ENDIF
```

EXECUTE will substitute *ingress* everywhere it finds the substitution argument *SFILE* enclosed within < and >, and it will substitute *egress* everywhere it finds the substitution argument *TFILE* enclosed within < and >. Note that in this example, the dollar sign ($) is used to provide default filenames in case *SFILE* and *TFILE* are not specified. If the HEX keyword is passed with EXECUTE, the H option of SORT is used.

2. The following example illustrates how using various dot commands can affect the appearance of the same command file. The function of the command file remains unchanged:

```
.DOT !
!KEY SFILE/A TFILE/A HEX/S
!BRA (
!KET )
!DOL #
IF HEX EQ ""
SORT (SFILE#mysource) (TFILE#mysorted)
ELSE
SORT (SFILE#mysource) (TFILE#mysorted) OPT H
ENDIF
```

FAILAT Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The FAILAT command is used within command sequence files and RUN command statements to alter the failure level threshold of the system.

When AmigaDOS commands encounter an error upon execution, a numeric return code is set (usually 5, 10, or 20). Under AmigaDOS Release

2 and 3, this result code is stored in the environment variable RC, whose contents are represented by the characters $rc. The higher the return code, the greater the severity of the error. If a return code which exceeds the current failure level threshold is encountered during execution of a command sequence file or multiple command task set up by a RUN, execution stops. The default failure level threshold of AmigaDOS command sequence files and RUN background tasks is 10.

Resetting the current failure level threshold can come in handy. By setting FAILAT very high, you can test return codes with the IF command in a command sequence file (or $rc under Release 2 and 3) and react according to the return code encountered without aborting script execution when a relatively high return code is encountered.

Once the command sequence file or RUN sequence has ended, the current failure level threshold is reset to 10.

See Chapter 5, "Command Sequence Files," and the RUN command for more information.

1.3 and Release 2 and 3 Format

FAILAT *codenum*

Explanation of Parameters and Keywords

codenum The new failure level threshold. If *codenumn* is not specified, FAILAT displays the current failure level threshold.

Examples

1. Display the current failure level threshold:

```
FAILAT
```

2. Temporarily set the current failure level threshold to 55:

```
FAILAT 55
```

FAULT Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The FAULT command provides English-language explanations for many of the error codes which AmigaDOS generates. When AmigaDOS runs into a problem, it usually displays a description of the problem or a requester box

telling you what needs to be done. In some cases, nothing appears but a fault code (under AmigaDOS Release 2 and 3, this error code is also placed in the environment variable*result2*). Further questioning of the system using the WHY command might produce a message like *Last command failed with error 220.* In these cases, the FAULT command can give you more information about the nature of the problem.

1.3 and Release 2 and 3 Format

FAULT *errornum(s)*

Explanation of Parameters and Keywords

errornum(s) The error number (fault code) which you want explained. Up to ten error numbers may be specified within one FAULT command. If no information is available on the error, the system simply repeats the error number. For instance, entering `FAULT 999` results in the display:
Fault 999: Error 999

Examples

1. Display the error message corresponding to fault code 216:

```
FAULT 216
```

AmigaDOS responds with:

```
Fault 216: directory not empty.
```

2. Display the error messages associated with fault codes 220, 103, and 226:

```
FAULT 220 103 226
```

AmigaDOS responds with:

```
Fault 220: comment too big
Fault 103: insufficient free store
Fault 226: no disk in drive
```

FF Command

Location: 1.3 C: Release 2 and 3 None

Function

FF (Fast Fonts) is a program that only appears on Workbench 1.3 (faster text routines are a standard part of Release 2 and 3). This program is used to speed up the display of text to the screen when a standard 8x8 pixel or 10x9 pixel fixed width font is used. It also can be used to replace the standard system fonts with custom fonts, as long as they are of the proper size and are fixed width (not proportional).

Format

FF [***-0***] [***-N***] [*fontname*]

Explanation of Parameters and Keywords

[***-0***] This optional switch indicates that the command is being used to turn the fast text routines on. This is the same behavior as when no switches are used.

[***-N***] This optional switch is used to temporarily disable the fast text routines, which can be turned on again by running FF once more.

[*fontname*] The name of the font descriptor file of the fonts to replace the TOPAZ80 and TOPAZ60 fonts. This file has a name like Siesta.font. The directory, which corresponds to this font descriptor file (in the example given, the fonts:Siesta directory), should contain either a filenamed 8, a filenamed 9, or both. These files should contain the font information for a fixed width 8x8 pixel font and a fixed width 10x9 pixel font respectively. If fonts of the correct type are found in this directory, they will be substituted for the default system fonts. If not, the system Topaz font will be used.

Examples

1. Turn on the fast text routines:

```
FF -0
```

2. Turn on the fast text routines using the Siesta/8 and Siesta 9 fonts:

```
FF Siesta.font
```

FILENOTE Command

Location: 1.3 and Release 2 and 3 C:

Function

FILENOTE lets you attach comments to existing AmigaDOS files (there are no comments associated with files when they are first created). Any comments stored using FILENOTE remain distinct and separate from the actual contents of the file.

When a file with comment attached by FILENOTE is duplicated using the COPY command, the comment is not associated with the new file (unless you use the COM or CLONE option). When a file is RENAMEd, comments attached to the file are attached to the new filename. When the contents of a file with a comment are updated, comments remain unchanged. If a comment is already attached to a file and a FILENOTE command with a new comment is issued, the new comment replaces the old.

Comments stored using FILENOTE may be viewed by using the LIST command. Any comments about the file appear on the screen beneath the file's name and are preceded by a colon (:). If the file is accompanied by an icon (.info) file, it's possible to read and edit the comment field from the Workbench by clicking on the icon, and selecting the "Info" of "Information" item from the Workbench menu.

1.3 Format

FILENOTE [***FILE***] *filename* [***COMMENT***] *string*

Release 2 and 3 Format

FILENOTE [***FILE***] *filename* [***COMMENT***] *string* [***ALL***] [***QUIET***]

Explanation of Parameters and Keywords

[***FILE***] *filename* The name of the file that is to have a comment attached. The FILE keyword is optional if *filename* is the first argument of a FILENOTE statement. Only one filename may be specified. The 1.3 version of FILENOTE does not support AmigaDOS patterns. Releases 2 and 3 do, however, allowing you to add the same comment to several files at once.

[***COMMENT***] *string* Defines the comment assigned to the specified file. The COMMENT keyword is optional if *string* is the second argument

of a FILENOTE statement (following *filename*); *string*, the comment to be attached to the file, can be up to 80 characters in length (79 characters for version 1.3 and up), and must be enclosed in quotation marks if it contains spaces.

[***ALL***] Using this Release 2 and 3 option, you can add the same comment to every file in the specified directory, and all of its subdirectories.

[***QUIET***] This Release 2 and 3 option prevents the command from listing the names of files to which it is adding the comment, as it normally does when you add comments to multiple files either by using a pattern, or the ALL option.

Examples

1. Attach the comment *Don't delete this file until September 8, 2001* to the file *fedtax91:*

```
FILENOTE FILE fedtax91 COMMENT "Don't delete this file
    until September 8, 2001"
```

2. Attach the comment *Lattice C Object Code - Almost Works* to the filenamed *PinBallDemo* in the directory *Lattice/Code/Work* on drive df1:.

```
FILENOTE df1:Lattice/Code/Work/PinBall Demo "Lattice C
    Object Code - Almost Works"
```

FORMAT Command

Location: Pre-1.2 C: 1.2, 1.3 and Release 2 and 3 SYS:System

Function

Initializes a floppy disk or hard drive as a blank AmigaDOS disk. A volume name, which must be specified by the user, is assigned to the disk after the initialization process is complete. *Caution: If a used disk is formatted, all information on it will be erased.*

FORMAT prompts you to insert the disk to be formatted in the desired drive and hit the RETURN key. This is your last chance to change your mind about the FORMAT request. Hitting CTRL-C and then RETURN aborts the process at this point. Once the disk is inserted and the RETURN key is pressed, the FORMAT process *cannot* be interrupted under earlier versions of the command, and may not be interrupted without losing the data on the disk in any version.

A status display reports as each cylinder on the disk (0-79 for a floppy) is initialized. After initialization, another display appears as each cylinder is verified. After verification is complete, the volume name is assigned.

It's not necessary to FORMAT a disk before using the DISKCOPY command—DISKCOPY formats as it copies. Thus, if all you want to do is copy the contents of a disk, it's much faster to use DISKCOPY than to first format the destination disk, INSTALL the system information on the formatted disk, and use the COPY ALL command to copy all of the files one by one. In fact, it's even faster to use DISKCOPY to duplicate a blank formatted disk than it is to format a new one.

One situation in which you may wish to copy a disk using the FORMAT—COPY ALL approach is where the files on the source disk have been deleted and rewritten so many times that the contents of the disk have become scattered. When this occurs, the time required to access each file may increase noticeably. By copying each file to a newly formatted disk, the contents of the disk will be consolidated.

1.3 and Release 2 and 3 Format

FORMAT DRIVE *drivename* ***NAME*** *string* [***NOICONS***] [***QUICK***] [***FFS***]

Explanation of Parameters and Keywords

DRIVE *drivename* The disk drive in which you will insert the disk to be formatted. The DRIVE keyword *must* be used. The valid values for *drivename* are df0:, df1:, df2:, and df3:, if you're formatting a floppy disk. The values used most often will be df0: (your Amiga's internal disk drive) and df1: (the optional Amiga 1010 external disk drive).

Since FORMAT can be used for any valid AmigaDOS storage device, you also can use hard drive or recoverable RAM drive names here like dh0:, jh0:, rad:, and work:. Exercise extreme caution in entering the device name for the FORMAT command. Typing *dh0:* when you mean *df0:* can result in you losing the entire contents of your hard drive.

NAME *string* The volume name assigned to the formatted disk. The NAME keyword is mandatory. *string* is the text of the volume name, which is also mandatory. *string* can be up to 31 characters long and must be enclosed in quotation marks if it contains spaces.

[***NOICONS***] This option specifies that you wish the formatted disk to be completely empty. If not used, a Trashcan directory will be created on the disk, as well as a Trashcan.info file for the drawer icon.
[***QUICK***] When this option is used, FORMAT will only write over the root block, the boot block, and the bitmap block, without formatting the rest of the disk. If you're formatting a disk that has already been formatted before, this is sufficient to create an empty volume, and is much quicker than going through the entire format process.
[***FFS***] By default, floppy disks use the old file system, while hard drives use whatever file system is specified in their Mountlist file or Rigid Disk Blocks. Using the FFS keyword causes FORMAT to ignore these defaults, and use the Fast File System on the specified volume. Fast File System floppies hold slightly more than the default ones, and can be read and written somewhat more quickly. Under Workbench 1.3, however, it is difficult to get the operating system to read FFS floppies, and it cannot boot from one. Workbench Release 2 and 3 reads and writes FFS floppies like any others.

Examples

1. Format a disk in drive df0:, naming the volume *Backup9:*

```
FORMAT DRIVE df0: NAME Backup9
```

2. Format a disk in drive df1:, with the volume name *Just Another Blank Disk:*

```
FORMAT DRIVE df1: NAME "Just Another Blank Disk"
```

GET Command

Location: Release 2 and 3 only Internal

Function

GET prints the contents of a local environment variable. A local environment variable is a named text string that is stored in an environment space, and is accessible only to the Shell in which it was created (and Shells spawned from that Shell). Unlike the global variables created by SETENV, the local environment variables are stored in private system memory, rather than in the RAM disk. You can substitute the value of a local environment variable on the command line by putting a dollar sign in front of its name.

Thus, if you have an environment variable called Fred, the command ECHO $Fred does the same thing as GET Fred.

Format

GET *varname*

Explanation of Parameters and Keywords

varname The name of the environment variable to get. Under Workbench Release 2 and 3, there are a number of significant local environment variables that are automatically set for you, or which you can set yourself. These include:

Process The process number of the current Shell.

RC The return code of the last command that was executed. This allows you to examine the code without using the IF WARN or IF FAIL command.

Result2 The error number that indicates why the last command failed. You can use the FAULT command to interpret these error codes.

Echo This local environment variable controls whether or not the Shell repeats each command as it is executed. If you SET Echo on, the commands are repeated. If you set Echo to anything else (or don't set it at all), they are not repeated. Turning Echo on is a good way to debug scripts that don't work, since there is often no other way of telling at which line they failed. With Echo on, you can tell which lines are executed properly, and which fail.

Example

Print the contents of an environment variable named *Cubby*:

```
GET Cubby
```

GETENV Command

Function

GETENV prints the value of a global environment variable, a named text string that is accessible to all tasks. As of Workbench Release 2 and 3, these strings are stored in files in the ENV: directory on the RAM: disk. This means that currently, "GETENV test" does the same thing as "TYPE ENV:test". In future versions, however, global environment variables may

be stored in system RAM, and manipulated by their own device handler, like local environment variables. As with local environment variables, if you put a global environment variable name with a dollar sign in front of it in a command line, the Shell will substitute the contents of the variable.

1.3 and Release 2 and 3 Format

GETENV *varname*

Explanation of Parameters and Keywords

varname The name of the environment variable to get. Currently, this is the name of a text file that is stored in the ENV: directory. Using the GETENV command prints the contents of this text file.

Under Workbench Release 2 and 3, there are a number of significant global environment variables that are automatically set for you, or which you can set yourself. These include:

Kickstart

Workbench These variables are created by the startup-sequence script, and contain the version numbers of the Kickstart and Workbench that you are using.

Editor This environment variable is recognized by some Workbench programs like the MORE program. If you set this variable to the pathname of your text editor, MORE allows you to bring up the program to edit the current file by pressing Shift-E.

Example

Print the current Workbench version number:

```
GETENV Workbench
```

IF Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The IF command and its associates (the ELSE and ENDIF commands) are used within AmigaDOS command sequence files to carry out groups of commands within the command sequence file *if* one or more conditions are met. If an IF statement is satisfied, the commands following the statement are executed sequentially until an ELSE or ENDIF statement is encoun-

tered. If the IF conditional is not satisfied and an ELSE statement is encountered before an ENDIF, the commands between ELSE and ENDIF are executed.

For every IF command there must be an associated ENDIF.

The ELSE command, if used, must appear between IF and ENDIF commands.

1.3 and Release 2 and 3 Format

IF [***NOT***] [***WARN***] [***ERROR***] [***FAIL***] [***VAL***] [*string1**EQ** string2*] [*string1**GT** string2*] [*string1**GE** string2*] [***EXISTS*** *name*]

Explanation of Parameters and Keywords

[***NOT***] Reverses the result of the IF test. If the condition tested is true and NOT is also used, the IF statement will not be satisfied. If the condition is otherwise false and NOT is used, the IF statement will be satisfied.

[***WARN***] Is satisfied (true) if the return code of the previous command is greater than or equal to 5.

[***ERROR***] Is satisfied (true) if the return code of the previous command is greater than or equal to 10.

[***FAIL***] Is satisfied (true) if the return code of the previous command is greater than or equal to 20.

[*string1**EQ** string2*] Is satisfied (true) if *string1* is identical to *string2*. Case is ignored in the comparison. This test can be reversed to test inequality by use of the NOT keyword. It can be changed to a numeric comparison by placing the VAL keyword in front of the first string. *string1*and *string2* are normally text strings, which are enclosed in double quotes if there is a space character in the string. However, under version 1.3 and up, either or both strings may be substituted by the contents of an environment variable by using a dollar sign ($) in front of the variable name. Thus, the IF command substitutes the expression $TEST with the text string that is contained within the environment variable named TEST.

[*string1**GT** string2*] This option was introduced in Workbench 1.3. The test is satisfied (true) if the characters in *string1* come after the characters in *string2* in alphabetical order. Case is ignored in the comparison. This test can be reversed to test the LESS THAN OR EQUAL condition by use of the NOT keyword. It can be changed to a numeric comparison by placing the VAL keyword in front of the first string. As with EQ, the strings may

be text strings in quotes, or the name of an environment variable prefaced with a dollar sign.

[*string1* ***GE*** *string2*] This option was introduced in version 1.3. The test is satisfied (true) if the characters in *string1* come after the characters in *string2* in alphabetical order, or if the two strings are identical. Case is ignored in the comparision. This test can be reversed to test the LESS THAN condition by use of the NOT keyword. It can be changed to a numeric comparison by placing the VAL keyword in front of the first string. As with EQ, the strings may be text strings in quotes, or the name of an environment variable prefaced with a dollar sign.

[***VAL***] This option was introduced in Workbench 1.3. When this keyword is placed in front of a comparison expression (EQ, GT, GE), it is changed to a numeric rather than a string comparison. Normally, these tests compare each character of the string in alphabetical order. Thus, the string "91" is shown to be greater than the string "099", since the leading 9 of the first string comes after the leading 0 of the second string. If you compare VAL "91" and "099" however, "099" is correctly shown to be greater.

[***EXISTS*** *name*] Is satisfied if *name* exists; *name* may be any AmigaDOS file, directory or logical device. This test can be used to check the existance of files or directories. To check the existance of disk volumes, use the EXISTS option of the ASSIGN command.

Examples

1. Using IF-ENDIF statements, build a command sequence file which deletes any file except the file *DontDoIt:*

```
.KEY nerf/a
IF <nerf> EQ DontDoIt
ECHO "I refuse to delete that File"
QUIT
ENDIF
DELETE <nerf>
```

Note: Actually this example will delete *DontDoIt* if the value *:DontDoIt* or *DF0:DontDoIt* is passed to the command file as the value of *nerf* when the command sequence file is executed. The EQ option of IF compares the text strings, not the internal block IDs of the files. Multiple

EQ statements could have been added to the IF statement to check for filename variants.

IF-ELSE-ENDIF sequences may be nested within one another.

2. Using nested IF-ELSE-ENDIF statements, build a command sequence file that attempts to delete the file *broccoli.* If any errors are encountered, report on their severity.

Note: The commands of this example have been indented to highlight the IF–ENDIF command groupings.

```
FAILAT 100
IF EXISTS broccoli
    DELETE broccoli
    IF WARN
        IF NOT EXISTS broccoli
            ECHO "File deleted - error encountered"
            QUIT
        ELSE
            ECHO "Fatal error - file not deleted"
            QUIT
        ENDIF
    ELSE
        ECHO "File deleted"
        QUIT
    ENDIF
ELSE
    ECHO "File not found"
ENDIF
```

3. Using IF-ELSE-ENDIF statements, build a command sequence file that will copy all files in the directory *mywork/text/AmigaProject* on drive df1: to the directory *mywork/text/backup* on the same disk drive. If the *AmigaProject* subdirectory does not exist, create it. Start up the program called *Textcraft* in the root directory of drive df0:.

```
FAILAT 100
ASSIGN MYDIR: TO df1:mywork/text
IF EXISTS MYDIR:AmigaProject
ECHO "Copying Documents to Backup Area"
COPY MYDIR:AmigaProject TO MYDIR:backup ALL
SAY backup completed boss
SKIP STARTUP
ELSE
MAKEDIR MYDIR:AmigaProject
ECHO "AmigaProject Directory Created"
ENDIF
LABEL STARTUP
RUN df0:Textcraft
```

4. For an example of the VAL and GT keywords, see Example 2 under the EVAL command.

INFO Command

Location: 1.3 and Release 2 and 3 C:

Function

Displays information about disk volumes and the system RAM disk. A typical INFO display shows the following information about each disk volume currently mounted on a physical drive attached to the Amiga. INFO also will report on the status of RAM:, the Amiga's memory-based RAM disk, if it's being used. A typical INFO display might look like this:

Mounted disks:

Unit	Size	Used	Free	Full	Errs	Status	Name
DF1:	880K	1089	669	61%	0	Read Only	Graphics Demos
DF0:	880K	740	1018	42%	0	Read/Write	CLI Disk
RAM:	22K	43	0	100%	0	Read/Write	

Volumes Available:
Graphics Demos [Mounted]
CLI Disk [Mounted]

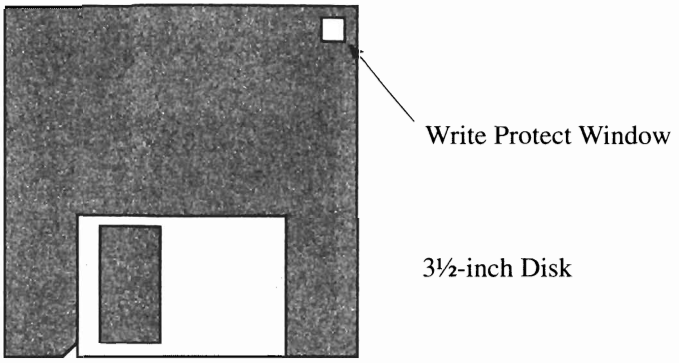
INFO

INFO tells you what disk volumes are in use and the amount of storage currently allocated to them. Amiga 3½-inch disks have a capacity of 880K (901,120 bytes) of information, of which 837K (857,088 bytes) is usable when the disk is formatted under the old file system (879K is usable under the Fast File System). Each AmigaDOS disk contains 1758 usable sectors, with each sector holding 488 bytes of information under the old file system, and 512 bytes of information under the Fast File System. INFO reports the number of sectors already used on each disk, the number of free sectors available for use, and the percentage of the disk used. The size of RAM: will vary depending upon how much information has been copied to it. Storage used for RAM: reduces the amount of real memory available for programs to run in. When RAM: is used, it will always show as being 100 percent full.

INFO also reports on the number of "soft" disk errors encountered in using the disk volume during the current session. Soft errors are those of a temporary nature. An example of a soft error is a temporary failure in reading some information from a disk. When the error is first encountered, many systems will try to read the information again for a predefined number of times. If one of the retries is successful, the original read error is considered a temporary, or soft, failure. If all retries fail, the error is considered a permanent, or hard, failure.

The status of each volume will be either *Read/Write* or *Read Only.* Read/Write indicates that the volume may be read or written to. New files may be added, and existing files on the disk may be read, updated, and deleted. A volume is made Read Only when its write-protect window has been uncovered. The write-protect window is located on the front left of a 3½-inch disk and is usually uncovered by sliding a small plastic shutter toward the front edge of the disk.

Write-Protect Window



The files on Read Only volumes can be read, but not updated or deleted. New files may not be added. Any attempt to write to a Read Only disk will result in an error. While RAM: cannot be write-protected, all files residing in RAM: can be protected from deletion by using the PROTECT command.

INFO also displays the name of the disk currently residing in each physical disk drive. This is very helpful in matching physical device names to those of the volumes mounted. A list of *Volumes Available* is also presented, indicating the status (mounted or unmounted) of disk known to the AmigaDOS filing system during the present session.

Format

INFO [*name*]

Explanation of Parameters and Keywords

[*name*] This option was added in Workbench 1.3. It allows you to obtain information on a single device or volume by typing its name after the INFO command.

Examples

1. Display information about the disk volumes known to the filing system:

```
INFO
```

2. Redirect the INFO display to an attached printer:

```
INFO > PRT:
```

3. Display information only about the disk in the internal disk:

```
INFO DF0:
```

INSTALL Command

Location:1.3 and Release 2 and 3 C:

Function

The INSTALL command makes a formatted disk capable of a minimal startup of the AmigaDOS environment (assigning SYS: to the booted disk). The key words to keep in mind here are *minimal startup.* While a blank, formatted disk, which has had an INSTALL command issued to it, will bring up the AmigaDOS window and command line prompt, none of the AmigaDOS commands will function unless invoked with their full pathnames.

If you wish to copy a bootable disk by formatting a new disk and copying each file from it one by one, you'll have to INSTALL the system information on the new disk in order for it to be accepted at the *Insert Workbench Disk* prompt.

With Workbench 1.3, additional functions were added to INSTALL to help the user identify and destroy "viruses," self-replicating computer programs that may be stored on the initial (boot) block of the disk without the user's knowlege. This also added the ability to remove the boot block from a disk, making it non-bootable (and less susceptible to viruses).

While INSTALL is meant to be used on your own AmigaDOS format disks, it also can install a boot block on non-AmigaDOS disks as well. You should be very careful only to use it on your own disks, because installing a boot block on auto-booting game disks will probably destroy the disk.

1.3 Format

INSTALL [***DRIVE***] *drive* [***NOBOOT***] [***CHECK***]

Release 2 and 3 Format

INSTALL [***DRIVE***] *drive* [***NOBOOT***] [***CHECK***] [***FFS***]

Explanation of Parameters and Keywords

[***DRIVE***] *drive* The disk drive in which the disk you wish to make bootable resides. The DRIVE keyword is optional. Valid values for *drive* are df0:, df1:, df2:, and df3:.

[***NOBOOT***] This keyword, added in version 1.3, can be used to clear the boot block, but not make the disk bootable. This can be used to get rid of unwanted information stored by viruses on the boot block, without making the disk bootable. Note that the NOBOOT option will write out a new boot block even if the disk is not set up in the standard DOS format.

[***CHECK***] This option, added in Workbench 1.3, is used to check if the disk is bootable, and if it contains the standard Commodore-Amiga boot code. When the CHECK keyword is used, the command prints the message *No bootblock installed* if the disk is not bootable, *Appears to be normal V1.2/V1.3 bootblock* or *Appears to be Release 2 and 3 FFS bootblock* if the disk is bootable and contains standard boot code, or *May not be standard V1.2/1.3 bootblock* if the disk is bootable and contains non-standard boot code. The command also sets a return code of zero if the disk is not bootable or has standard boot code, and a code of 5 (WARN) if it contains non-standard boot code.

[***FFS***] Normally, the disk will be given a boot block of the type used by the filing system for which it was formatted (old file system or Fast File System). Use of this keyword will force a Fast File System boot block to be installed. As of version 2.1, this keyword has no effect—presumably, Commodore decided that there was no valid reason to install an FFS boot block on a non-FFS disk.

Examples

1. Install boot files on the disk presently inserted in disk drive df1:.

```
INSTALL DRIVE df1:
```

2. Install boot files on the disk currently residing in the internal system drive df0:.

```
INSTALL df0:
```

Note: INSTALL does not prompt you for the disk to be inserted. For most owners of single-drive systems, this makes a direct INSTALL to drive df0: difficult. Typically, the place where AmigaDOS commands are found by the system (the C: command directory) is assigned to df0:. If you insert the disk you wish to install to ahead of time and then type `INSTALL DRIVE df0:`, you'll be prompted to insert the disk with the command library on it in any disk drive. Once you do so, INSTALL puts boot files on

the disk with the command library, which was *not* where you wanted the files installed and was bootable to begin with.

The following procedure will get single-drive users around this limitation. Put your Workbench or CLI disk into the drive and type:

```
INSTALL ?
```

When the command template (DRIVE/A etc.) appears, eject that disk from the internal drive and insert the disk you want to install or check. Type

```
DF0:
```

3. Check the boot code on the disk that is currently in drive df1:.

```
INSTALL DRIVE df1: CHECK
```

IPREFS Command

Location: Release 2 and 3 C:

Function

IPREFS manages the new Preferences scheme introduced in Release 2. It reads all of the preference setting files in the ENV:Sys directory, and puts their settings into effect. IPREFS stays in memory, and requests to be notified if any of these settings files are changed, so that it can update the settings as soon as a change occurs.

In order to correctly set the display and overscan settings, IPREFS must close the Workbench screen, if it has been opened before IPREFS runs. In order to do this, the user must close any windows (such as the Shell window) that has opened on the Workbench screen. (Intuition will print the message "Intuition is attempting to reset the Workbench screen. Please close all windows except drawers.") If you get this message every time you boot up your system, you probably have added a new command line somewhere in the startup-sequence which creates some output, causing the Workbench screen to open before IPREFS has run. The simple solution is to redirect the output of any new command line to NIL:. If the program is called "MYCOMMAND", for example, and it takes the command arguments "arguments", change the command line from "MYCOMMAND arguments" to "MYCOMMAND >NIL: arguments".

Release 2 Format

IPREFS [*QUIT*]

Release 3 Format

IPREFS

Explanation of Parameters and Keywords

[*QUIT*] Under Release 2, you may run IPREFS a second time using the optional QUIT keyword to unload the program, and prevent it from changing the preferences as they are updated. Under Release 3, this option was removed.

Examples

None. Under normal circumstances, this program is run automatically by the startup-sequence script, and any attempt to run it again results in a requested box that says "ERROR: Attempt to run multiple instances of IPrefs! New instance cannot start."

JOIN Command

Location:1.3 and Release 2 and 3 C:

Function

JOIN lets you merge the contents of several files into a new file. Files are merged in the order given to JOIN, and none of the original files are changed or deleted.

Format

JOIN *name1 name2* ,,,,,,,,,,,, **AS** or **TO** *destname*

Explanation of Parameters and Keywords

name1 name2 ,,,,,,,,,,,, The names of the files you want merged together. Under versions prior to Release 2, a minimum of two filenames must be given, with a space between each name, and a maxium of fifteen can be specified. These limits were removed in version Release 2. Version Releases 2 and 3 also allow you to use pattern matching to specify the names of the files to be merged.

AS or **TO** *destfile* The name of the file into which the contents of all files preceding the AS or TO keyword (one of which is required) will be merged; *destname* can be a new or old file, but it cannot be any of the files

which precede the AS or TO keyword. If *destname* already exists, its previous contents will be replaced. The TO keyword was added in Workbench 1.3.

Examples

1. Merge two files (*Dick* and *Jane*) in the current directory into a file (*HusbandAndWife*) in the same directory:

```
JOIN Dick Jane AS HusbandAndWife
```

2. Merge four files, from various drives, devices, and directories into one:

```
JOIN myparty :spritzers/white/chablis df1:softdrinks/
    cola/moxie RAM:mydate AS ":party/animal/March 25
    1986"
```

LAB Command

Location:1.3 C: Release 2 and 3 Internal

Function

LAB is used within command sequence files to define a location in the command file to skip to using the SKIP command. See Chapter 5 for complete information on command sequence files.

1.3 and Release 2 and 3 Format

***LAB** string*

Explanation of Parameters and Keywords

string A "signpost" or label that can be used by a SKIP command to jump to the spot in the command file where a specific LAB statement is located. Once jumped to, command file execution continues with the commands following the LAB statement.

Example

Define a location called *DontDo* that may be jumped to by a SKIP instruction:

```
IF EXISTS work.backup
SKIP DontDo
COPY work work.backup
LAB DontDo
RENAME work work.old ...
```

LIST Command

Location: 1.3 and Release 2 and 3 C:

Function

Displays the name, size, protection status, time and date of creation, and the Amiga filing system block numbers of *(a)* a directory, *(b)* a selected portion of a directory, or *(c)* a single file. LIST also displays any comments attached to a file by a FILENOTE command.

Here's an example of a typical LIST output:

```
Directory ":" on Wednesday 12-Dec-85
bagel                20      rwed    Today          00:57:23
c                    Dir     rwed    Yesterday      23:49:01
fonts                Dir     rwed    Yesterday      23:49:01
lox                  1921    rwe-    10-Dec-85      14:29:54
: A history of Nova Scotia's Finest
libs                 Dir     rwed    Yesterday      23:49:01
t                    Dir     rwed    Yesterday      23:49:01
2 files – 4 directories – 7 blocks used
```

The file and directory names are listed at the left. To the right of each name is additional information about the file. The first number indicates each file's size in bytes (directories are shown by the letters *Dir*).

The protection flags currently turned on for each item (see the PROTECT command for further information) are listed next, then finally the date and time the item was created or last updated. Any comment attached to a file or directory by the FILENOTE command appears directly beneath

the file's information line in the LIST display and is preceded by a colon (:).

With 1.3, a formatting option was added which allows you to output the directory listing in a customized text display.

1.3 Format

LIST *listname* [***P*** or ***PAT*** *pattern*] [***KEYS***] [***DATES***] [***NODATES***] [***TO****device* or *filename*] [***SUB*** *string*] [***SINCE*** *date*] [***UPTO*** *date*] [***QUICK***] [***BLOCK***] [***NOHEAD***] [***FILES***] [***DIR***] [***LFORMAT*** =*string*]

Release 2 and 3 Format

LIST *listname* [***P*** or ***PAT*** *pattern*] [***KEYS***] [***DATES***] [***NODATES***] [***TO*** *device or filename*] [***SUB*** *string*] [***SINCE*** *date*] [***UPTO*** *date*] [***QUICK***] [***BLOCK***] [***NOHEAD***] [***FILES***] [***DIR***] [***LFORMAT*** =*string*] [***ALL***]

Explanation of Parameters and Keywords

listname This can be the device name or volume name of a disk, a directory, or the name of a specific file. As of Workbench 1.3, this name may include a pattern. Thus, LIST *#?.info* gives a list of all the files ending in the five characters ".info". This eliminates the need for the PAT option, below.

[***P*** or ***PAT*** *pattern*] When you use this option, the P or PAT keyword must precede the pattern. A pattern allows you to specify a number of files, each of which has some common characteristic (see Chapter 3 for more information on creating AmigaDOS patterns). Since recent versions of LIST allows you to use patterns in *listname*, this option is no longer needed.

[***KEYS***] Specifying this option includes the block number associated with each file and directory displayed. The AmigaDOS filing system automatically assigns and uses block numbers to keep track of things. Each file and directory has a single, unique block number. The block number on the display appears to the left of the file length (or *Dir).*

[***DATES***] Includes file and directory creation date and time information in the LIST display. DATES is usually optional since LIST defaults to displaying creation dates and times unless either QUICK or NODATES is used.

[***NODATES***] Instructs LIST to suppress the display of file and directory creation date and time information. NODATES is optional.

[***TO*** *device* or *filename*] Selects where the output of LIST is to be sent; *device* or *filename* may be any valid AmigaDOS filename or a logical device known to the system. If a file of the same name already exists, the existing file will be deleted and a new file with the same name is created. For this reason, if TO *device* or *filename* is a file that has been protected from deletion with the PROTECT command, LIST will fail. If TO *device* or *filename* is not specified, LIST's output is displayed on the system screen.

[***SUB*** *string*] To use this option, the SUB keyword must precede *string*, which can be any character string. LIST then displays only those filenames or directories which include *string*. If spaces are included in *string*, quotation marks must enclose it. Since recent versions allow patterns in the *listname*, this option has been rendered unnecessary.

[***SINCE*** *date*] Displays information only for those files and directories created or modified on or after *date; date* may be specified in the format *DD-MMM-YY,* or as an indirect reference of YESTERDAY, TODAY, or TOMORROW. The days of the past week, SUNDAY through SATURDAY, can also be used as *date*. See the DATE command for more information.

[***UPTO*** *date*] Instructs LIST to display information only for those files and directories created or modified on or before *date*, which is subject to the same restrictions as the SINCE keyword.

[***QUICK***] Instructs LIST to display only file and directory names. However, if the DATES and/or KEYS keywords are specified as well, LIST displays file and directory names along with the information associated with DATES and/or KEYS. As of version Release 2, KEYS no longer has any effect if QUICK is specified. With either version, you may use the FILES or DIR option to limit the display to files or directories.

[***BLOCK***] This option, added in version 1.3, specifies that file sizes be displayed in terms of 512-byte blocks, rather than bytes.

[***NOHEAD***] This option, new in Workbench 1.3, allows you to suppress the "Directory..." heading on the first line, and the "x files -x directories -x blocks used" footnote on the last line of the listing.

[***FILES***] This option, added in version 1.3, allows you to list only files, and not directories.

[***DIR***] This option, also added in version 1.3, allows you to list only the directories, and not the files.

[***LFORMAT*** = *string*] The LFORMAT option was added in Workbench 1.3 to allow you to customize the text output of the LIST command. This feature lets you use LIST to quickly generate script files. In order to send the output to a script file, you must either use the redirection operator > or the TO option. If you use the TO option, it is best to put the TO keyword at the end of the line.

When you use LFORMAT, the output of the LIST command is totally controlled by the format string that appears after the keyword. This format string can contain any text that you want to appear as output. At the place in the text string that you want the names of files and directories to appear, you use the substitution string %S. This substitution string may be used more than once in the text, allowing you to use the listing name more than once per line. If you use the string twice, the first time the relative path will be substituted for %S, and the second time the file or subdirectory name (the relative path is simply the list of directories you must traverse in order to get to a specified directory from the current one). If you use it three times, the first occurrence will be replaced by the relative path, and the second and third by the file or subdirectory name. If you use it four times, the first and third will be replaced by the relative path, and the second and fourth by the file or subdirectory name.

As of version Release 2, several new substitution strings were added to extend the flexibility of this listing. These are:

String	**Information Substituted for String in Output**
%A	Attributes (protection flags)
%B	Size of file in blocks
%C	Comments attached to the file
%D	Date of file creation or last update
%F	The complete absolute path (starting with the volume name)
%K	Key block number
%L	Length of the file in bytes
%N	Name of the file
%P	The relative path (starting at the current directory)
%T	Time of creation or last update

Version 2.1 adds two more substitution strings. These are:

String	Information Substituted for String in Output
%E	Extension only (the part of the filename after the last period)
%M	Filename minus the extension (everything up to the last period)
ALL	This option, added in version Release 2, lists the contents of all of the subdirectories of the specified directory, as well as the directory itself.

Examples

1. Display standard LIST information about the contents of the current directory on the screen:

```
LIST
```

2. Output all standard LIST information and the block number of each item in the current directory to the system printer:

```
LIST KEYS TO PRT:
```

or

```
LIST > PRT: KEYS
```

3. Display standard LIST information about each file in directory *water/sports* whose name contains the character string *skin*:

```
LIST water/sports SUB skin
```

or

```
LIST water/sports/#?skin#?
```

Note: Information for both *Snorkel & Skin Diving* and *SkinnyDipping* would be displayed by this example.

4. Output just the names and date information for items in the current directory beginning with the letters *compute* that were created or last updated on or before November 4, 1991. Send the output to a file called *MySelections:*

```
LIST compute#? QUICK DATES UPTO 04-Nov-91 TO
    MySelections
```

5. Create a script file which, when executed, will set the pure bit of every file in the C: directory:

```
LIST >RAM:TEMP C:#? LFORMAT "protect %S +p"
```

This creates a script file called RAM:TEMP which contains a PROTECT command for each file in the C: directory. To set the pure bit in each file, you need only EXECUTE RAM:TEMP.

LOADWB Command

Location:1.3 and Release 2 and 3 C:

Function

The LOADWB command is used to start the Workbench program. This command is usually issued automatically at boot time, as part of the *s:startup-sequence* batch file. When the Workbench is started, it notes the search paths that are currently in effect, and sets these same paths for each CLI or Shell that is started from a Workbench icon.

Versions prior to Release 2 will cause the Workench to reinitialize if you issue a LOADWB command after the Workbench is already loaded. Under Release 2 and 3, it will merely give you a message saying that the Workbench is already loaded.

1.3 Format

LOADWB [***DELAY*** *or* ***-DEBUG***]

Release 2 and 3 Format

LOADWB [***DELAY***] [***-DEBUG***] [***CLEANUP***] [***NEWPATH***]

Explanation of Parameters and Keywords

[DELAY] The DELAY option was added in version 1.3 to create a three-second pause before the program returned. This pause allows the floppy disk activity that LOADWB initiates to stop before the next command in the script is executed. If there is no pause and the next command starts before LOADWB finishes using the disk, both commands have to share the floppy disk, which causes disk access to slow down and causes the disk heads to make a loud "thrashing" noise.

[-DEBUG] The -DEBUG option, also added in version 1.3, enables a new Workbench menu which is normally not displayed. This menu contains two items that are provided for the convenience of programmers. The

first item, Debug (called ROMWack on the Release 2 and 3 menu), executes the ROMWACK debugger, a program which communicates through a 9600-baud terminal connected to the serial port. If no such terminal is connected, your computer will appear to lock up, since not even the mouse will move until it receives its commands from the remote terminal. The second item, Flushlibs, causes Workbench to expunge any libraries, devices, fonts or other resources that are resident in memory, but are not currently being used (just like the FLUSH option of the AVAI command). This frees up the memory used by these resources, allowing programmers to more easily check if there programs have freed all of the memory that they allocated.

In version 1.3, either DELAY or -DEBUG may be used, but not both at the same time.

[CLEANUP] This option, added in Release 2, automatically performs a "Cleanup" operation on the Workbench window, assuring that the disk icons are neatly lined up.

[NEWPATH] This option was added in Release 2 to allow you to change the path associated with the Workbench. Normally, Workbench remembers the path of the CLI or Shell window that issued the LOADWB command, and assigns that same path to each Shell that is started from an icon. You can change that path at a later time by issuing a LOADWB NEWPATH command. This causes the Workbench to use the path associated with the Shell window from which that command was issued.

Examples

1. Load the Workbench environment and close the CLI window:

```
LOADWB DELAY
ENDCLI >NIL:
```

2. Load the Workbench environment with the DEBUG menu activated:

```
LOADWB -DEBUG
```

3. Add the SYS:REXX directory to the path that Workbench assigns to new Shell windows that are opened from an icon:

```
PATH SYS:AREXX ADD
LOADWB NEWPATH
```

LOCK Command

Location: 1.3 and Release 2 and 3 C:

Function

LOCK was added to Workbench 1.3 to enable or disable write protection of a hard drive partition that uses the Fast File System. Under Release 2 and 3, its scope was broadened to include all kinds of disks, including floppies. Once write protection is set, it remains in force until reset with another LOCK command, or until the system is rebooted. If an optional password is used to lock the drive, the same password must be used to unlock the drive.

1.3 and Release 2 and 3 Format

LOCK *drive* [***ON*** *or* ***OFF***] [*password*]

Explanation of Parameters and Keywords

drive The device name of a disk or disk partition. Under 1.3, it must be a hard drive that is formatted with the Fast File System, but under Release 2 and 3, it can even be a floppy drive. This command works differently with floppy drives than any other command. Normally, commands that concern a disk that is mounted in the floppy drive apply only to that particular disk. When a floppy drive (such as df1:) is locked, however, you cannot write to *any* disk in that drive until the lock is removed.

[***ON*** *or* ***OFF***] These optional keywords can be used to write protect the disk (LOCK ON) or write enable the disk (LOCK OFF). If neither is specified, the disk will be write enabled.

[*password*] An optional password of the user's choosing (in early versions, this had to be four characters long, but now can be any length). This password is a normal text string, and as such, if it contains spaces it must be enclosed in quotes. If a password is used as part of the LOCK ON command, the same password must be used in the LOCK OFF command in order for the disk to be write enabled.

Example

1. Write protect hard drive DH0:, with a password of *fish*.

```
LOCK dh0: ON fish
```

[***REW*** or ***REWIND***] This option rewinds the tape to the beginning.

[***SKIP*** *numfiles*] This option moves forward past a given number of files. The number *numfiles* indicates how many files to skip.

Example

Rewind the tape on a A3070 tape unit whose SCSI unit number is set to zero, and which is attached to a Commodore SCSI controller:

```
MAGTAPE DEVICE scsi.device UNIT 0 REWIND
```

MAKEDIR Command

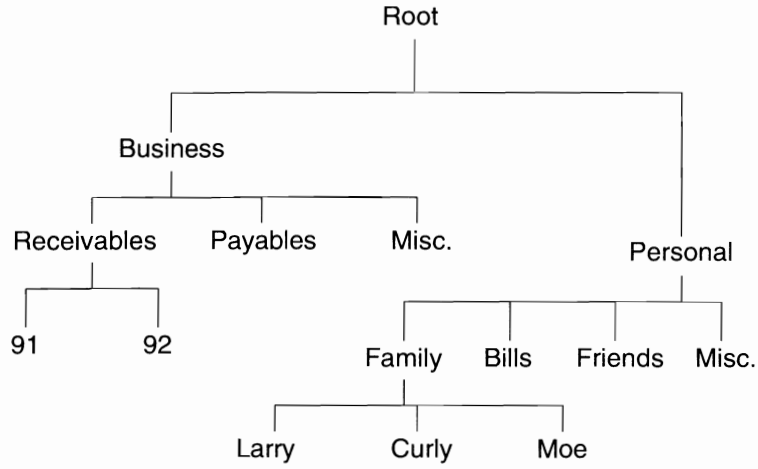
Location: 1.3 and Release 2 and 3 C:

Function

MAKEDIR creates a new directory, where lists of file and directory names may be stored. This allows you to create a multi-tiered filing system within a disk volume.

Suppose you wanted to separate your written correspondence by category and recipients. Your business correspondence usually deals with accounts payable and receivable, with some occasional miscellaneous letters. Your personal correspondence is mostly letters to your family and friends, letters concerning your personal bills, and some other occasional things. You might decide that you want things organized like this:

Planned Directory Form



Assuming that you begin with the root directory of an AmigaDOS disk, this is one of the possible sequences of AmigaDOS commands that will set up such a directory structure:

```
MAKEDIR Business
MAKEDIR Business/Receivables
MAKEDIR Business/Receivables/86
MAKEDIR Business/Receivables/87
MAKEDIR Business/Payables
MAKEDIR Business/Misc
MAKEDIR Personal
CD Personal
MAKEDIR Family
MAKEDIR Friends
MAKEDIR Bills
MAKEDIR Misc
CD Family
MAKEDIR Larry
MAKEDIR Moe
MAKEDIR Curly
CD //
```

Let's examine how this was created, starting with the business correspondence first. Note the top-down order in which the directories were created. MAKEDIR builds only one subdirectory at a time. When you type `MAKEDIR Business/Receivables/86`, the only directory entry created is 86, the rightmost portion of the specified directory path. For the command to execute successfully, both the *Business* directory and a subdirectory within it called *Receivables* must have already been created.

As the business correspondence MAKEDIR commands illustrate, you can expend a lot of keystrokes typing pathnames. Just look at all the times you had to type `Business`. You can use the CD command to significantly reduce the number of keystrokes required. Look at the sequence of commands again, paying particular attention to the last half, in which the personal correspondence directories were created. After the MAKEDIR Personal used to create the directory for personal letters, a CD Personal changed the current directory so that the pathname *Personal* could be omitted from all subsequent MAKEDIRs. CD was used again to "drop down" into the *Family* subdirectory and keep unnecessary keystrokes to a minimum. Note that once again, care has been taken to insure that the

directories are built from the top down. A final CD // at the end backs you up two levels to your starting point (see the CD command for more information on its use).

For information on removing directory entries, see the DELETE command; for more information on directory structures, see Chapter 3, "The Filing System."

1.3 and Release 2 and 3 Format

MAKEDIR *name*

Explanation of Parameters and Keywords

name The name of the directory to be created; *name* must be specified. MAKEDIR fails if *name* is the name of a file or subdirectory which already exists in the "parent" directory (the next highest directory in the hierarchy). MAKEDIR also fails if a nonexistent pathname is specified.

Examples

1. Create a subdirectory called *YellowPages* in the current directory:

```
MAKEDIR YellowPages
```

2. Create a subdirectory called *Dictionary* in the root directory of the disk inserted in drive df1:.

```
MAKEDIR df1:Dictionary
```

3. Create a subdirectory called *Encyclopedias* in the root directory of the current drive, create five subdirectories within *Encyclopedias,* and then change the default directory to the root of the current drive:

```
MAKEDIR :Encyclopedias
CD :Encyclopedias
MAKEDIR "World Book"
MAKEDIR Grolier
MAKEDIR Britannica
MAKEDIR "World Book"
MAKEDIR "Funk & Wagnals"
CD :
```

4. Create a subdirectory called *Lightning* on the Amiga's RAM disk:

```
MAKEDIR RAM:Lightning
```

5. ASSIGN a logical device name *QWIK:* to the directory created in example 4 and create a subdirectory called *WarpSpeed* in it:

```
ASSIGN QWIK: RAM:Lightning
MAKEDIR QWIK:WarpSpeed
```

Note: This results in creating the same subdirectory as

```
MAKEDIR RAM:Lightning/WarpSpeed
```

MAKELINK Command

Location: Release 2 and 3 C:

Function

This command creates links, which are files that point to other files. Whenever programs ask for the link file, they actually get the file to which the link points. Links are handy when different programs want to reference the same file in several directories. For instance, the icons for document files often try to load the document into a reader program like More, but they may specify this program as C:More or Sys:Utilities/More, or C:MuchMore. To satisy all of these types of document icons, you could make three copies of the reader program, but that would take up unnecessary disk space. With links, you can have one copy of the reader program, and two links that direct other programs to that file.

Release 2 and 3 Format

MAKELINK [***FROM***] *fromname* [***TO***] *toname* [***HARD***] [***FORCE***]

Explanation of Parameters and Keywords

[***FROM***] *fromname* The name of the link file to create. The *FROM* keyword is needed only if the order of *fromname* and *toname* are reversed.

[***TO***] *toname* The name of the target file to which the link file will refer. This file must be in the same volume as *fromname*, and cannot be a directory name unless the FORCE option is used.

[***HARD***] This optional keyword is used to indicate that the link is a hard link—one in which the link and its target are on the same volume. Currently, the command does not support soft links, in which the link file and its target are on different volumes.

[*FORCE*] This optional keyword is used to create directory links, in which the link file refers to a directory rather than a file. These types of links can create confusion within the file system, particularly if the target directory contains link files, or the link refers to a target which is its own parent directory. MAKELINK will not let you create certain types of links which are obviously confusing, but you should still exercise care.

Example

Create link files C:More and C:MuchMore that refer to the file SYS:Utilities/More:

```
MAKELINK C:More SYS:Utilities/More
MAKELINK C:MuchMore SYS:Utilities/More
```

Whenever a file tries to run the program C:More or C:MuchMore, it will run SYS:Utilities/More instead.

MOUNT Command

Location: 1.3 and Release 2 and 3 C:

Function

This command allows AmigaDOS to recognize an external hardware device (such as a 5¼-inch disk drive or a second serial port) as a DOS device. It can also be used to add a virtual device like a recoverable RAM disk, or a speech output device. MOUNT is most often issued as part of the *startup-sequence* script file in the *s* directory of the Workbench disk, so that the device is automatically configured each time that the Amiga boots up.

MOUNT looks for a description of the device requested in the file *DEVS:Mountlist,* or an optional FROM file. The *Mountlist* file is a text file which describes various attributes of the device. This file shares some of the traits of C language source code files. If more than one description appears on a line, they must be separated by semicolons, hexadecimal numbers must start with the characters 0x, and comments must start with the characters /* and end with */. Each entry in the *Mountlist* ends with the pound sign character (#).

The standard Workbench 1.3 disk comes with a sample *Mountlist* file that can be used to mount an external 5¼-inch drive as device DF2:. This device description looks like:

```
DF2:Device = trackdisk.device
    Unit = 2
    Flags = 1
    Surface = 2
    BlocksPerTrack = 11
    Reserved = 2
    Interleave = 0
    LowCyl = 0;  HighCyl = 39
    Buffers = 20
    BufMemType = 3
#
```

The most important line is the first, which tells AmigaDOS that the device driver for this device is *trackdisk.device*. Since this device driver is an intrinsic part of the operating system, it need not be loaded in from disk. External device drivers can be loaded from disk if necessary, however. AmigaDOS will look for such drivers in the path specified, or in the DEVS: directory if no path is included in the device driver name.

The rest of the information tells AmigaDOS what kind of disk drive this is—it's double-sided (*Surface = 2*), has 40 tracks per side (*LowCyl = 0; HighCyl = 39*), with 11 sectors of 512 bytes per sector (*BlocksPerTrack = 11*). This means that the total available storage on the drive is 440K (.5K $\times 2 \times 11 \times 40$).

As configured, this list assumes that you have an external 3½-inch drive, so it mounts the external drive as DF2:. If you don't have a 3½-inch drive as DF1:, you'll want to mount the 5¼-inch drive as DF1: (it should always be last in the chain). To do this, make two changes to the *Mountlist* file.

First, change the device name from DF2: to DF1:. Second, change the unit number from 2 to 1. Now the command `MOUNT DF2:` mounts the 5¼-inch drive as DF2:.

The MOUNT command also can be used to mount devices other than file storage devices. The *Mountlist* file in the DEVS: directory contains a sample mountlist for a non-buffered serial device named AUX:, which shares the standard serial port hardware with the buffered device SER:

```
AUX: Handler = L:aux-handler
     Stacksize = 1000
     Priority = 5
```

Instead of describing the storage capacity of the device, this list merely shows where to find the handler (a program whose purpose is similar to that of the device driver), the size of the stack used by that program, and the priority at which it runs.

Versions 1.2 and higher of CLI commands such as FORMAT and DISKCOPY have been altered so that they work with devices which have been mounted. Note, however, that you still can't use DISKCOPY between devices that aren't identical in storage size and layout. Thus, while you can use DISKCOPY between two 5¼-inch drives, two hard disk partitions of equal size, or two 3½-inch drives, you can't use DISKCOPY from a 5¼-inch drive to a 3½-inch drive.

AmigaDOS Release 2 and 3 users should note that as of version 2.1, a DOSDrivers drawer has been added to the DEVS directory. This directory was designed to hold new Mountlist-style files with icons. These files differ from a normal mountlist in that (a) they contain only one device description per file (b) the name of the device is the name of the file and is not specified in the text of the mountlist and (c) they do not end with a pound sign. Any device whose mountlist file is dragged into the DOSDrivers drawer will automatically be MOUNTed when the Workbench is loaded. Mountlist files which you do not want to be mounted automatically are stored in the DOSDrivers drawer of the Storage directory.

1.3 and Release 2 and 3 Format

MOUNT *device* [***FROM*** *filename*]

2.1 Format

MOUNT *device(s)* [***FROM*** *filename*]

Explanation of Parameters and Keywords

device This is a AmigaDOS device name, such as DF2:, DH0:, or AUX:, which refers to either a hardware device like a disk drive or serial port, or a logical device such as a RAM disk. The device name should be the same as the label given an entry in the *Mountlist* file, and the device driver or handler file indicated by that *Mountlist* should be available to the

2. To turn off the write protection, you'd use the command:

```
LOCK dh0: OFF fish
```

or just

```
LOCK dh0: fish
```

MAGTAPE Command

Location: Release 2 and 3 only C:

Function

MAGTAPE was added to Workbench Release 2 to provide a means of controlling the Commodore A3070 tape backup unit. This tape drive is a sequential device, like a video or audio tape unit, so you must manually "fast forward" or "rewind" to get to a particular file, rather than just skipping directly to it, as you would on a random-access device like an audio CD player or a disk drive. The MAGTAPE command allows you to rewind, retension, or skip a given number of files on the tape.

Release 2 and 3 Format

MAGTAPE [***DEVICE*** *devicename*] [***UNIT*** *unitnum*] [***RET*** or ***RETENSION***] [***REW*** or ***REWIND***] [***SKIP*** *numfiles*]

Explanation of Parameters and Keywords

[***DEVICE*** *devicename*]

[***UNIT*** *unitnum*] By default, MAGTAPE assumes that the SCSI unit number of the A3070 tape drive is set to four (the factory setting), and the drive is connected to a Commodore SCSI controller, whose device drive name is scsi.device. If you wish to change the SCSI unit number of the tape drive, or use it with a controller other than Commodore's, you must specify *both* the device name of the SCSI controller and the unit number of the tape drive, using the DEVICE and UNIT keywords. Normally, you can change the unit number of the tape drive by setting switches on the drive itself. If you do not know the device name of your SCSI driver, you can obtain it from the controller's manufacturer. This name will end in ".device", as in Commodore's "scsi.device".

[***RET*** or ***RETENSION***] This command is used to re-tension the tape, by running it to the end, and then rewinding.

system. Some of the standard MOUNTable devices are described in Chapter 4.

device(s) Under Workbench 2.1, you can list multiple device names, and thus mount several devices with a single MOUNT command. Also, the MOUNT command supports mounting devices from the new icon-based mountlist files, as well as conventional entries in the DEVS:Mountlist file. If the device name ends in colon (such as SPEAK:), MOUNT will look for an entry in DEVS:Mountlist, or whatever other Mountlist file you specify. If the device name doesn't end in a colon (for instance SPEAK), MOUNT will look for the file of that name in the DEVS:DOSDrivers or SYS:Storage/DOSDrivers drawers (wildcards can even be used to MOUNT several device files). If it doesn't find the file in either place, it will look for a corresponding entry in the Mountlist file.

[***FROM** filename*] This option, added in version 1.3, allows you to specify a file other than DEVS:Mountlist as the place to look for the description of the device to be MOUNTed.

There are a number of keyword options that can used in the *Mountlist* file to describe a device. Not all of them are required for all devices—in fact, most are optional.

Keywords include:

Handler =	The name of the device handler file.
EHandler=	The name of the environment handler file (Release 2 and 3).
FileSystem =	The name of the file system file.
Device =	The name of the device driver file.
Priority =	The task priority of the process; 5 is customary for handlers, 10 for file systems.
Unit =	The unit number of the device.
Flags =	Flags setting for OpenDevice call (usually 0).
Surfaces =	Number of write surfaces.
BlocksPerTrack =	The number of disk blocks (sectors) per track (cylinder).
Reserved =	The number of blocks used for boot block; usually 2.
PreAlloc =	The number of blocks reserved at the end of a partition; used with a few IBM-style hard drives. Usually set to 0.

Interleave =	Interleave value (controls DOS interleave, not physical hard drive interleave).
LowCyl =	Starting cylinder to use for this device.
HighCyl =	Ending cylinder to use for this device. Total number of cylinders = HighCyl-LowCyl+1.
Stacksize =	The amount of working memory to allocate to the process.
Buffers =	Number of cache buffers to use with the device.
BufMemType =	Type of memory to use for cache buffers. 0 or 1 = Any 2 or 3 = CHIP 4 or 5 = FAST
Mount =	If this value is positive, MOUNT loads the handler or driver software as soon as the device is MOUNTed, rather than the first time the device is accessed. Workbench 2.1 adds ACTIVATE as a synonym for this keyword.
MaxTransfer =	The maximum number of blocks transfered at one time; used with Fast File System devices.
Mask =	Address mask that specifies the memory range that can be used for DMA transfers; used with Fast File System.
GlobVec =	If the handler is written in BCPL, it needs a global vector. A value of 0 sets up a private global vector; anything else indicates that the handler is written in C or assembly language, and no global vector is needed. If this keyword isn't used, the shared AmigaDOS global vector is used.
Startup =	A string passed to the handler, device, or file system on startup. This string is passed as a BPTR to a BSTR.
BootPri =	The boot priority of a bootable device, expressed as a number between -129 and 127. A value of -129 indicates that the device isn't bootable, as is appropriate for use with the recoverable RAM disk if you don't want to boot from that device on reset.

DosType =	Indicates the format of the file system used. If the Fast File System is used, this value should be set to 0x444F5301 (DOS/1). Other types introduced in 2.1 include 0x444F5302 (DOS/2), an international version of the old file system that allows mixed-case accented characters in filenames, and 0x444F5303 (DOS/3), an international version of the Fast File System.
Baud=	Serial device speed (in bits per second).
Control=	Serial device control parameters—word length, parity, and stop bits (e.g. 8N1, 7E1).
ForceLoad=	A new 2.1 option. When this value is zero (the default), the system will check the resource list to see if the file system named in the entry has already been loaded. If it has, the system will use that one, instead of loading a new copy. When ForceLoad is set to one, however, a new version will always be loaded from disk.

Examples

Mount an external 5¼-inch disk drive as device DF1:

```
MOUNT DF1:
```

NEWCLI Command

Location: 1.3 C: Release 2 and 3 Internal

Function

NEWCLI opens a new CLI window on the system display. The new window sports the same gadgets (drag gadget, back/front gadget(s), sizing gadget, and under Release 2 and 3, zoom and close gadget) as a CLI process that's started either by double-clicking the CLI icon from the Amiga Workbench or by booting up a specially prepared CLI disk. A window created by NEWCLI becomes the current, active window immediately after NEWCLI is executed. It inherits the current directory from the CLI from which NEWCLI was executed.

Every CLI window represents an independent CLI environment. You may change the active CLI window by moving the mouse pointer within any CLI window and clicking.

The default window title of CLI windows opened by NEWCLI with no title specified is *New CLI* (*AmigaShell* under Release 2 and 3). The new CLI's prompt line will be preceded by the message *New CLI task n,* where *n* is the task number assigned to the new CLI window.

The task number associated with the new CLI window is different from all other CLI windows currently open on the screen. For instance, if two CLI windows are created by issuing one NEWCLI command, the command line prompt of the first CLI is 1> and the command line prompt of the second is 2>. The CLI prompt of a window created by issuing another NEWCLI is 3>. A new CLI (task 3) can be created by issuing a NEWCLI from *either* of the two original CLI windows.

The resolution of the AmigaDOS screen display is 640 pixels (picture elements) wide and 200 pixels high. Think of an invisible 640 × 200 grid superimposed over your Amiga's display. AmigaDOS creates new CLI windows in a location that starts at the top, left edge of the screen, and extends 640 pixels wide by 100 pixels high. All new CLI windows are created in the same place—in the same size—unless you specify otherwise. This means that the third CLI window appears on top of the second, and you'll have to drag one out of the way if you want to use both.

The obvious question is, aside from impressing your friends and running a computerized version of a three-ring circus, what good is NEWCLI? One obvious use is preventing a helpful display of information from scrolling off the screen. If you're attempting to clean up or reorganize a directory full of files, having to issue repetitive DIR commands to refresh your memory can be tedious, especially considering AmigaDOS's less than speedy directory searches. Opening a new window with NEWCLI and issuing a DIR command brings up a directory display which may be sent out of sight and recalled at will by using the front and back gadgets of the two active windows. Your file maintenance commands may be issued from the original CLI, whose scrolling display will not affect the directory display in the new CLI window.

You can even start a process in one CLI window and, while it's executing, make another existing CLI the active environment and start up another process in it. Multiple AmigaDOS functions can be set churning away in separate windows. While this multitasking is somewhat similar to

the facilities offered by the RUN command, opened CLI windows remain available until closed by the ENDCLI command.

Versions prior to Release 2 support a maximum of 20 open CLI windows. This limit is not present in Release 2 and 3.

1.3 Format

NEWCLI [***AUX:*** *or* ***CON:****hpos/vpos/width/height/windowtitle*] [***FROM*** *filename*]

Release 2 and 3 Format

NEWCLI [***AUX:*** *or* ***CON:****hpos/vpos/width/height/windowtitle/options*] [***FROM*** *filename*]

Explanation of Parameters and Keywords

[***AUX:*** *or* ***CON:****hpos/vpos/width/height/windowtitle*] CON: lets you specify the size, position, and title of the new CLI window. (If the NEWCON: device has been MOUNTed under 1.3, NEWCON: can be substituted for CON:.) CON: is required if *any* of the following parameters are specified.

- *hpos* is the horizontal position of the top left corner of the window (expressed as the number of pixels in from the left edge of the screen). If a value for *hpos* is omitted, it's assumed to be zero.
- *vpos* is the vertical position of the top left corner of the window (expressed as the number of pixels down from the top edge of the screen). If a value for *vpos* is omitted, it's assumed to be zero.
- *width* and *height*, which must be specified, give the size of the window in pixels. The maximum size for a CLI window is the screen size, 640 ´ 200 pixels. The minimum is 90 ´ 25 pixels. Unless a window of exact size is required, it's usually easier to resize and drag a default size NEWCLI window (200 ´ 100 pixels) to a desired size and screen location rather than typing the required size parameters.
- *windowtitle*,which is optional, allows you to enter the text of a title to appear in the title bar. If you want to set *windowtitle*,all preceding parameters must also be set. If you don't enter any text for *windowtitle,* the title bar is left blank. Even if you want the title bar to be blank, the last slash (/) following *height* is required. Titles with spaces can be entered, but quotation marks must enclose the entire list of NEWCLI parameters—see example 3 below. (The default title, if you do not specify *any* parameters, is *New CLI.*)

- *options* A number of options were added to the console window in AmigaDOS Release 2 and 3 to expand its usefulness. The CON: window description can have one or more *options* entries appearing after the title, each separated by a slash. These options include:

AUTO	The window opens automatically when the program that opened it requires input or produces output.
CLOSE	A close gadget is included in the window border. This is the default case if no options are selected.
BACKDROP	The window type is changed to backdrop, which means that it appears behind all other windows on the Workbench screen, and you cannot depth-arrange, move, or resize it (except by using the zoom gadget).
NOBORDER	No visible line is drawn around the window, although the zoom and close gadgets will still appear above it. If you zoom this window to full size, those gadgets will disappear, and you will have a full-screen window that can't be sized or moved.
NODRAG	The window can't be dragged. It will have a zoom, depth, and size gadget, but no close gadget.
NOSIZE	The window will not have zoom, size, or close gadgets. Only a depth gadget will appear.
SCREENname	The window will open on the public screen whose name is *name*. For example, to open the CLI window on the public screen named Fred, you would add / SCREENFred to the end of the window description. This option only works if a public screen of the specified name is already open.
SIMPLE	Chooses the simple window refresh scheme. If you enlarge such a window, the text expands to fill the available space, allowing you to see more information, including information that had scrolled off the screen. This is the default refresh type for Release 2 and 3 Shell windows if no option is specified.
SMART	Chooses the smart window refresh scheme. If you enlarge this kind of window, existing text is not redrawn. This is the kind of CLI window used by AmigaDOS 1.3 and earlier.

WAIT — The window does not close automatically when the program that created it terminates. Rather, it waits until its close gadget is selected (if it has one), or the user enters the CTRL-\ key combination.

Although CON: and NEWCON: windows are the most likely targets of a NEWCLI window, you also may direct the input and output of a CLI window out the serial port, by mounting the AUX: device, and then issuing a NEWCLI AUX: command. Such a CLI will only be able to effectively run text-based programs that direct their output to the CLI window, and cannot be used to cancel requesters like *Insert volume Workbench in any drive*.

[***FROM** filename*] This option allows you to specify a batch file that is to be executed automatically when the CLI opens, just like the *s:startup-sequence* file executes when the initial CLI window opens. Starting with version 1.3, if you do not specify a startup file, NEWCLI attempts to execute a default startup file, *s:CLI-Startup*, if such a file is present in the s: directory. (A Shell windows executes *s:shell-startup* if present.) The default startup file allows you to retain settings, such as the CLI prompt, which do not carry over from one CLI to the next.

Examples

1. Create a new CLI window using AmigaDOS's defaults. The upper left corner of the new 200 × 100 window will be located at the top left corner of the screen. The new window will be titled *New CLI*:

```
NEWCLI
```

2. Create a 250 × 125 pixel CLI window in the upper left corner of the screen. The new window is to have no title:

```
NEWCLI CON://250/125/
```

3. Create a new CLI window 450 × 40 pixels, located 25 pixels to the right and 30 pixels below the upper left corner of the screen. The new window is to have the title *Flying High with CLI:*

```
NEWCLI "CON:25/30/450/40/Flying High with CLI"
```

4. Create a new CLI that uses the serial port for input and output:

```
NEWCLI AUX:
```

NEWSHELL Command

Location: 1.3 Alias created by the Shell-startup script Release 2 and 3 Internal

Function

In Version 1.3, NEWSHELL opens an interactive Shell using a NEWCON: console window, only if the Shell-Seg has been made resident and the NEWCON: device MOUNTed. This Shell is an enhanced CLI which supports features such as command aliases, resident commands, and a prompt that reflects the current directory.

In order to open a Shell window, the Shell-Seg program must have been made resident with the command

```
RESIDENT CLI L:Shell-Seg SYSTEM pure
```

If Shell-Seg has not been made resident, the NEWSHELL command will open a normal CLI window instead of a Shell window, but will still try to use a NEWCON: window. If the NEWCON: device hasn't been MOUNTed, NEWSHELL will use a CON: window instead.

As of Release 2, both the enhanced CLI and enhanced console device are built into AmigaDOS, and using either the NEWCLI or NEWSHELL command will open a new Shell window.

1.3 Format

NEWSHELL [***AUX:*** *or* ***NEWCON****:hpos/vpos/width/height/windowtitle*] [***FROM*** *filename*]

Release 2 and 3 Format

NEWSHELL [***AUX:*** *or* ***CON****:hpos/vpos/width/height/windowtitle/options*] [***FROM*** *filename*]

Explanation of Parameters and Keywords

[***AUX:*** *or* ***NEWCON****:hpos/vpos/width/height/ windowtitle/options*]
NEWCON: lets you specify the size, position, title, and options for the new Shell window. (If the NEWCON: device has not been MOUNTed, CON: can be substituted for NEWCON:.) See NEWCLI, above, for details.

As with NEWCLI, NEWSHELL can be used to open an interactive Shell through the serial port, with the command NEWSHELL AUX:.

[***FROM*** *filename*] This option allows you to specify a batch file that is to be executed automatically when the Shell opens, just like the *s:startup-sequence* file executes when the initial CLI window opens. If you do not

specify a startup file, NEWSHELL attempts to execute a default startup file, *s:Shell-Startup*, if such a file is present in the s: directory. Use of the default startup file allows you to retain settings, such as command aliases, which do not carry over from one Shell to the next.

Example

Create a new Shell window 450 × 100 pixels, located 25 pixels to the right and 30 pixels below the upper left corner of the screen. If using 1.3, assume that the Shell-Seg has already been made RESIDENT, and NEWCON: has been MOUNTed. The new window is to have the title *My Shell:*

```
NEWSHELL "NEWCON:25/30/450/100/My Shell"
```

PATH Command

Location: 1.3 C: Release 2 and 3 Internal

Function

Changes or displays the search path used by AmigaDOS to locate a command file. When you type a command at the CLI prompt, AmigaDOS first looks for the command file in the current directory—if the file is not there, DOS looks for it in whatever directory was assigned as the C: directory. (See Chapter 2 for more information on search paths.) The PATH command allows the user to specify additional directories to be searched after the current directory but before the C: directory.

This command also may be used to display the current search path.

1.3 Format

PATH [***SHOW***] [***ADD*** *or* ***RESET***] *dir* [*,dir, dir...*] [***QUIET***]

Release 2 and 3 Format

PATH [***SHOW***] [***ADD*** *or* ***RESET*** or ***REMOVE***] *dir(s)* [***QUIET***]

Explanation of Parameters and Keywords

[***SHOW***] This optional key word displays the search path that AmigaDOS is currently using. Typing the command `PATH` without specifiying any directories accomplishes the same thing. The current search path is displayed in the following format:

```
Current directory
Workbench 1.3:System
C:
```

[***ADD***] *dir(s)* The optional ADD keyword allows you to add from one to ten additional directories to the current search path. The same effect may be achieved by typing PATH, followed by one or more directory names. The *dir(s)* parameter is the name of the directory or directories to add. This directory may be specified relative to the current directory, or the entire path name may be used to specify it. If more than one directory is added, each directory name is separated by a space. Each new directory that's added gets searched after the other user-specified directories, but before the C: directory.

[***RESET*** *dir(s)*] The optional RESET key word is used to delete the current search path and optionally to replace it with one or more directories. The optional *dir(s)* parameter is the name of the directory to be added to the search path. This directory may be specified relative to the current directory, or the entire path name may be used to specify it. If more than one directory is added, each directory name is separated by a space. If no directory names are specified, the default search path is reset to the current directory and the C: directory.

[***REMOVE*** *dir(s)*] The optional REMOVE keyword, added in Release 2, is used to delete one or more directories from the current search path. The *dir(s)* parameter is the name of the directory or directories to be removed from the search path. If more than one directory is added, each directory name is separated by a space.

[***QUIET***] This optional keyword was added in version 1.3. It can be used in combination with the SHOW option to suppress the *Please insert volume xxx* requester for disks that aren't currently mounted. When the QUIET option is used, only the volume name will be diplayed for paths in unmounted volumes.

Examples

1. Add the *System* and *Utilities* directories on the root directory to the current search path:

```
PATH ADD :System :Utilities
```

or

```
PATH :System :Utilities
```

2. Display the current search path, suppressing the *Please insert volume* requester for unmounted disks:

```
PATH SHOW QUIET
```

or

```
PATH QUIET
```

3. Reset the search path to the current directory and the C: directory:

```
PATH RESET
```

4. Replace the current search path with the *Demos* and *Utilities* directories in the root directory of the disk in the external 3½-inch disk drive:

```
PATH RESET df1:Demos df1:Utilities
```

PROMPT Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The PROMPT command changes the prompt for the currently active CLI or Shell. The default prompt for any given CLI or Shell is *n*>, where *n* is the task number associated with that CLI. For instance, if only one CLI has been started, its prompt is 1>. If two more CLI windows are then started with the NEWCLI command, their prompts will be 2> and 3>.

When used with a Shell window, PROMPT can automatically display the current directory as part of the command prompt. The Shell-startup script file automatically changes the default prompt string of a Shell window to the task number of the Shell, followed by a period, the current directory path, a right angle-bracket, and a space.

1.3 and Release 2 and 3 Format

PROMPT *prompt*

Explanation of Parameters and Keywords

prompt The string you want to substitute for the active CLI's prompt. If no value for *prompt* is specified, the CLI prompt will be changed to >. The string specified in *prompt* may be a maximum of 59 characters. If it contains spaces, the entire prompt must be enclosed by double quotation

marks. Note that the ANSI escape sequences shown in the table in Chapter 2 can be used in the prompt string to change the prompt to a different color, or italics.

There's a special substitution string allowed with the value specified for *prompt*. If *prompt* contains the two-character combination %N, the task number associated with the current CLI is substituted for those two characters. The Workbench 1.3 Shell adds another substitution string. When the characters %S are used in a Shell prompt, they are replaced with the current directory path. Release 2 adds yet another substitution string, %R, which displays the return code for the last program that was executed from the Shell.

Examples

1. Change the current CLI prompt to *Ready]:*

```
PROMPT Ready]
```

2. Change the current CLI prompt to *Really Ready* (with a trailing space):

```
PROMPT "Really Ready "
```

3. Change the current CLI prompt to *CLI* n *Ready* (with a trailing space, and where *n* is the current CLI's task number):

```
PROMPT "CLI %N Ready "
```

4. Change the current Shell prompt to show the Shell task number and current directory in reverse text, separated by angle brackets (>):

```
PROMPT "<esc>[7m%N>>%S><esc>[0m "
```

where <esc> represents a single press of the Escape key.

5. Under Release 2 and 3, change the CLI prompt to reflect the current date and time each time it is printed:

```
prompt "*'DATE*' > "
```

This command uses the back apostrophe (back tick) character to output the results of the DATE command to the prompt string. The asterisk character is used as an escape to indicate that the DATE command is to be executed each time that the prompt string is printed. If this escape character is not used, the prompt string will always print the same time and date.

PROTECT Command

Location:1.3 and Release 2 and 3 C:

Function

PROTECT allows you to alter the attributes of AmigaDOS files and directory entries. Originally, there were protection flags associated with each of four attributes. These flags are *r, w, e,* and *d;* they tell the system if the file or directory entry may be read (r), written over (w), executed (e), or deleted (d). In Workbench 1.3, three more flags were added. These flags are *s, p* and *a*. They indicate whether the file is a script that may be directly executed from a Shell (s), a pure command file that may be made resident (p), or if the file has not remained unchanged since the last archival back up (a).

The LIST command is used to examine the status of a file or directory entry. In the display provided by the LIST command, there's room for eight characters to the left of the date information. Seven of these characters, *sparwed,* correspond to the seven protection status flags. When a file or directory entry is first created, the last four flags are turned on, and they may be modified thereafter using PROTECT. The three new bits must be set either by the user or by a backup program.

If a flag character is present in the LIST display, it is said to be on, and the operation may be carried out. The Read flag lets you read from a file or directory entry, the Write flag lets you update the file or directory with new information, the Delete flag allows the file or directory entry to be removed altogether, and the Execute flag is meaningful only for files which are actual programs for the Amiga. The Execute flag allows DOS to execute (run) the program. If you set the Execute flag on a nonprogram file (like a text file, for instance), you cannot expect DOS to load and run the file. The Script bit will allow you to execute a script file just by typing the name of the file at the Shell prompt. Remember, this only works with a Shell window, not a CLI, and the filename must be of a valid script file. The Pure bit is used in connection with the Resident Shell command, and indicates that a command is suitable for being made resident. The Archive bit is used mostly by backup programs, to indicate which files have not changed since the last back up.

If a flag is off, the LIST display shows a dash (–) in place of the flag character.

Of the initial four bits (rwed) only the Delete and Execute flags actually do anything. You can set the other flags, but DOS does not act on those settings.

1.3 Format

PROTECT [***FILE***] *name* [***FLAGS***] [***S***] [***P***] [***A***] [***R***] [***W***] [***E***] [***D***] [***ADD*** *or* **+**] [***SUB*** *or* **–**]

Release 2 and 3 Format

PROTECT [***FILE***] *name* [***FLAGS***] [***S***] [***P***] [***A***] [***R***] [***W***] [***E***] [***D***] [***ADD*** *or* **+**] [***SUB*** *or* **–**] [***ALL***] [***QUIET***]

Explanation of Parameters and Keywords

[***FILE***] *name* The name of the file whose protection flags are to be modified; *name*, which is mandatory, may be any valid AmigaDOS filename or directory name. The FILE keyword is optional. Under Release 2 and 3, a pattern may be used to change the protection flags on a number of files at once.

[***FLAGS***] [***S***][***P***][***A***][***R***][***W***][***E***][***D***] The protection flags which will be turned on by PROTECT. The FLAGS keyword does not have to be entered—it's optional. The protection flags to be turned on must be specified as a single string in any desired order. Remember that if a flag is set to on, the operation associated with the flag may be carried out. If no flags are specified, all flags are turned off. These are the operations associated with each flag:

S—Script	This script file can be run without the Execute command.
P—Pure	This program file can be made Resident.
A—Archive	This file has not changed since the last backup was made.
R—Read	This file may be read.
W—Write	This file may be written to.
E—Execute	This file is an executable program.
D—Delete	This file may be deleted.

Note: AmigaDOS commands that overwrite existing files (such as COPY) actually delete the old file and then create a new one with the same name. For this reason, COPY and other commands which behave in this manner will fail when they try to overwrite files that are protected from deletion.

[***ADD*** *or* **+**]

[***SUB** or –*] Version 1.3 of the Protect command introduced the capability to add or subtract individual bits, rather than requiring you to set all of a file's flags at once. To set specific flags, you can name the flags and follow them with the keyword ADD, or just put a plus sign (+) in front of the initials. To turn off specific flags, follow the list with the keyword SUB, or precede it with a minus sign (–).

[***ALL***] This optional keyword, added in Release 2, allows you to set or reset the specified protection bits in all of the files in directory *name* , as well as all of the its subdirectories.

[***QUIET***] This option can be used to suppress the progress report that is printed when changing the protection status of multiple files using the Release 2 and 3 ALL option or pattern matching.

Examples

1. Make the file *S:Script* executable directly from a Shell window:

```
PROTECT FILE Script S ADD
```

or

```
PROTECT Script +S
```

2. Protect the file *Public Knowledge* in subdirectory *:info/expose* from being deleted. Allow it to be executed as a program, and flag it as capable of being read, but not written to:

```
PROTECT ":info/expose/Public Knowledge" RE
```

3. Protect a file called *transitory* on the system's RAM disk from being read, written to, deleted, or executed:

```
PROTECT RAM:transitory
```

4. Reset a file called *Enough Already* on drive df1: to the protection attributes it had upon creation:

```
PROTECT "df1:Enough Already" WRED
```

5. Protect a directory entry called *shuttle/columbia* from being deleted:

```
PROTECT shuttle/columbia -D
```

QUIT Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The QUIT command is used within command sequence files (see Chapter 5 for complete information on command files). The QUIT command allows you to exit a command sequence file and, optionally, to set the return code.

1.3 and Release 2 and 3 Format

QUIT [*returncode*]

Explanation of Parameters and Keywords

returncode The return code which is reported when the command sequence file is terminated by a QUIT. If *returncode* is above the FAILAT threshold, the message

`Quit failed returncode returncode`

is displayed on the screen, with the number specified substituted for the second *returncode*. If *returncode* is set below the failure threshold, or is not specified, no message is displayed on termination of the command sequence file by QUIT.

Examples

1. Exit a command sequence file using the QUIT command. The QUIT in the following example is executed only if the file *wolfbane* is found on drive df1:. No return code is to be set:

```
IF df1:wolfbane EXISTS
ECHO "Get the silver bullets"
QUIT
ENDIF
TYPE :Transylvania/here/I/come
```

2. Exit a command sequence file using the QUIT command. A return code of 88 is to be set:

```
ECHO "This is just a silly example"
QUIT 88
LIST
```

The LIST command in the above example will never be executed. The message *quit failed returncode 88* will be sent to the system display when the QUIT 88 is executed.

RELABEL Command

Location: 1.3 and Release 2 and 3 C:

Function

RELABEL lets you change the name associated with a disk volume. Volume names are initially assigned when a disk is formatted by the FORMAT command or created by a DISKCOPY operation. The volume name is the name that appears under a disk icon on the Workbench screen and should not be confused with the device name of a disk, such as df0: or dh0:.

Format

RELABEL [***DRIVE***] *drive* [***NAME***] *name*

Explanation of Parameters and Keywords

[***DRIVE***] *drive* The device name or the volume name of the disk you wish to relabel. The DRIVE keyword is optional if *drive* precedes the volume name in the RELABEL statement. RELABEL does *not* prompt you for the disk to be inserted. If you have a single-drive system and issue the RELABEL command using the device name (df0:), you'll be prompted to insert the disk with the command directory (C:) library on it in any disk drive. Once you do so, RELABEL promptly renames the volume with the command library on it, not the disk you originally wanted to RELABEL. Therefore, single disk owners should always specify the volume name of the disk they want to RELABEL, not the device name. If you have single-drive system and don't know the volume name, the following procedure will allow you to use the device name:

```
RELABEL ?
```

At the prompt, swap the disk you want to relabel into the internal drive, and type:

```
df0: NewName
```

[***NAME***] *name* The volume name which will replace whatever name is currently associated with the target disk; *name* may be up to 30 characters

long. If the volume name contains spaces, quotation marks must enclose it. The NAME keyword is optional if *name* follows *drive.*

Examples

1. Relabel the disk in drive df1: as *Various Programs:*

```
RELABEL DF1: "Various Programs"
```

2. Relabel the disk in drive df0: as *Home on the Range:*

```
RELABEL NAME "Home on the Range" DRIVE DF0:
```

Notice that in this example, both NAME and DRIVE were specified, since their order was switched in the RELABEL statement.

REMRAD Command

Location: 1.3 and Release 2 and 3 C:

Function

REMRAD allows you to remove the recoverable RAM disk device, RAD:, from the system without turning off the power. Although this device's ability to survive a warm boot can be handy, it also presents a problem when you decide that you wish to de-allocate the memory set aside for the RAM disk. REMRAD commands the device to delete all of its files and release most of its memory. The next time the system reboots, the recoverable RAM disk is removed entirely.

1.3 Format

REMRAD

Release 2 and 3 Format

REMRAD *unitnum* [***FORCE***]

Explanation of Parameters and Keywords

unitnum Under Release 2 and 3, you can create multiple recoverable RAM disk volumes by specifying different unit numbers in the Mountlist. REMRAD allows you to selectively remove one of these volumes by specifying a unit number (the default unit number is zero).

[***FORCE***] If a RAM drive unit is in use (it is your current directory, or it has a logical device name assigned to it, for example) the REMRAD

command will ordinarily fail. By using the optional FORCE keyword, however, you can force removal of the drive even if it is currently in use.

Example

Remove recoverable RAM drive unit 1 that is currently in use:

```
REMRAD 1 FORCE
```

RENAME Command

Location: 1.3 and Release 2 and 3 C:

Function

RENAME allows you to change the name of AmigaDOS files and directories. This command also lets you move files from one directory to another on the same disk and reorganize entire directory structures.

1.3 Format

RENAME [***FROM***] *fromname* [***TO*** or ***AS***] *toname*

Release 2 and 3 Format

RENAME [***FROM***] *fromname* [***TO*** or ***AS***] *toname* [***QUIET***]

Explanation of Parameters and Keywords

[***FROM***] *fromname* The file or directory that's to be renamed. The FROM keyword is not required if *fromname* is the first argument of a RENAME statement. Under Release 2 and 3, *fromname* may be a wildcard pattern, when *toname* is a directory into which the files that match the pattern will be moved.

[***TO*** or ***AS***] *toname* The new name to be given to the file or directory specified by *fromname*. The TO and AS keywords may be used interchangeably and are optional if *toname* is the second argument of a RENAME statement. If *fromname* already exists, RENAME will fail.

Note: fromname and *toname* must reside on the same disk volume.

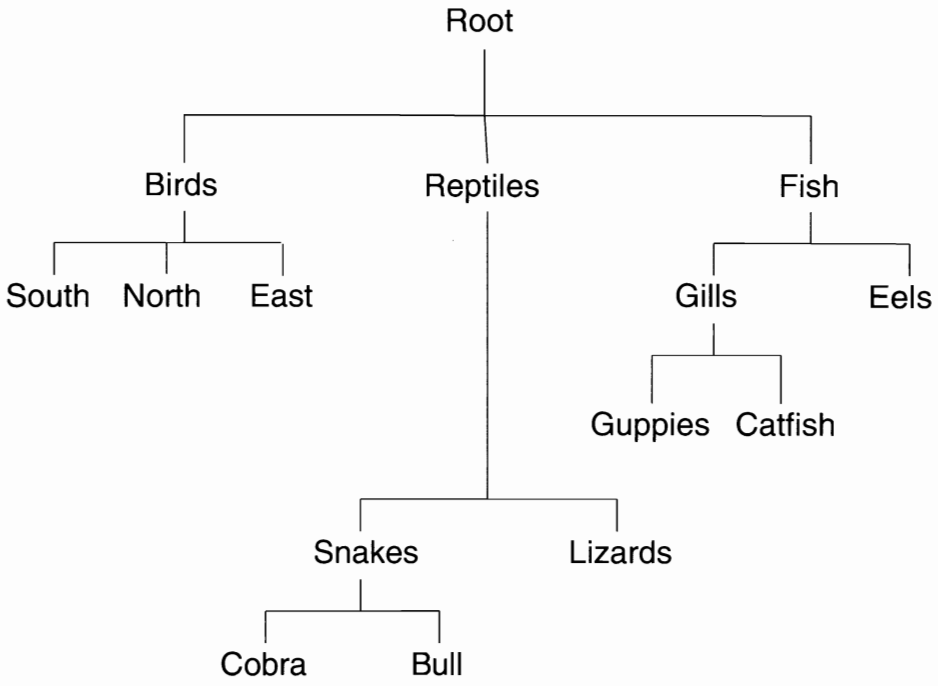
[***QUIET***] This option, introduced in Release 2, suppresses the progress report that is normally printed when multiple files or directories are being renamed using pattern matching.

RENAME's ability to manipulate AmigaDOS directory structures makes this one of the most powerful AmigaDOS commands and, consequently, a command that should be used with great care. An entire directo-

ry, including all files, subdirectories, and files within its subdirectories may be moved to another location in the volume's directory tree structure with a single RENAME.

For instance, suppose the directory structure of a disk volume looks like this:

Volume Animals, Before RENAME

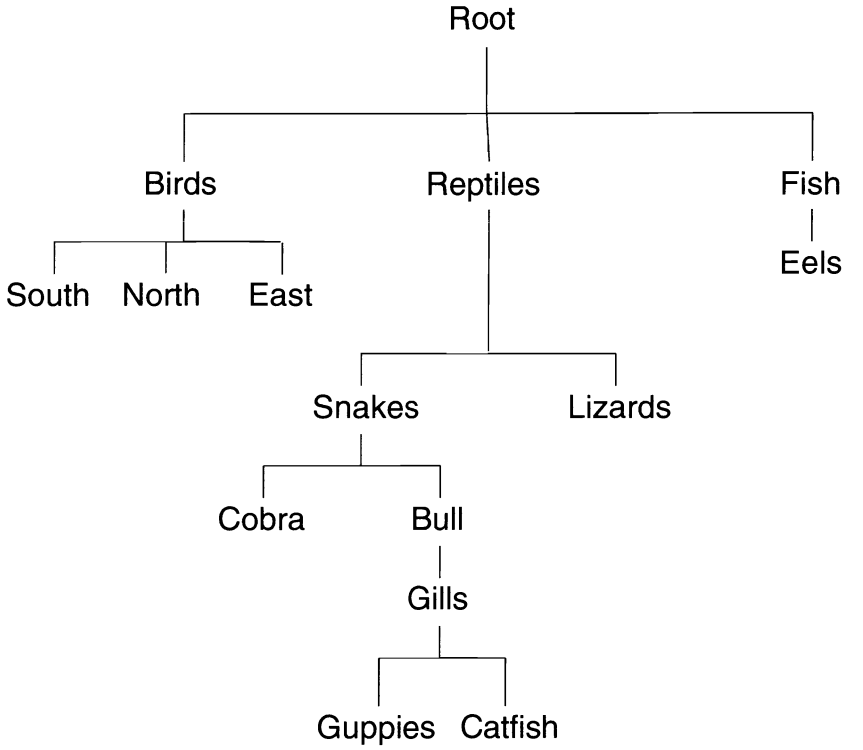


Issuing the following RENAME command:

```
RENAME :FISH/GILLS :REPTILES/SNAKES/BULL/PETS
```

results in a new directory structure.

Volume Animals, After RENAME



Examples

1. Rename a file called *birddog* to *hounddog:*

```
RENAME birddog hounddog
```

2. Move a file called *Lights Out* to a directory called *HeavyMetal/JGeils* without changing the filename:

```
RENAME "Lights Out" ":HeavyMetal/JGeils/Lights Out"
```

3. Move a directory called *LaserDiscs* and all the files and subdirectories within it to a directory called *Phils/Video* without changing the directory name:

```
RENAME LaserDiscs Phils/Video/LaserDiscs
```

4. Move a file called *Apple* in the *fresh/fruits* directory to a directory called *Desserts/Light*, changing the name of the file to *RomeApple*:

```
RENAME fresh/fruits/Apple Desserts/Light/RomeApple
```

5. Move a directory called *Ancient Computers* and all the files and subdirectories within it to a directory called *8-Bit Processors*, changing the directory name to *Ancient History:*

```
RENAME "Ancient Computers" "8-Bit Processors/Ancient
    History"
```

REQUESTCHOICE Command

Location: Release 3 C:

Function

This command was added in Release 3 to allow the user to easily add a decision requester to a script file or AREXX program. A requester opens a window that presents the user with some text, and one or more buttons to choose. This window remains in place until the user clicks on one of the buttons, at which point, the program prints out the number of the button that was chosen.

Release 3 Format

REQUESTCHOICE *titletext bodytext buttontext(s)* [***PUBSCREEN*** *screenname*]

Explanation of Parameters and Keywords

titletext The string of text that you wish to appear in the title bar of the requester window. This string should identify the program that opened the window (e.g. "Setup Program Window"). Some text string is required, even if it is only the blank text string (""). As always, if the text string contains spaces, the entire string must be surrounded by quotes.

bodytext The text string that prompts the user's choice (e.g. "Do you wish to continue?"). This string is also required.

buttontext(s) Text strings for one or more button choices. A minimum of one button is required, but you can add additional buttons just by tacking on extra text screens. Each button is assigned a number, and when the user makes a choice, the command prints the number of that choice to the

standard output (usually the Shell console). If you provide text for multiple buttons, the buttons appear in the window from left to right. The leftmost button is assigned the number 1, the next the number 2, etc. The only exception is that the last button is always assigned the number zero. That button is usually reserved for the "Cancel", "No" or "Quit" option.

[***PUBSCREEN*** *screenname*] This optional keyword allows you to place the requester on a public screen, by using the keyword followed by the name of the screen.

Examples

1. Have the user select a number for 0 to 4, and print that number in the Shell window:

```
SET Number 'REQUESTCHOICE " " "Pick a number" 1 2 3 4 0'
ECHO "You chose the number $Number"
```

2. Write a script that asks the user if he wishes to continue, and quits if he does not:

```
SET Continue 'REQUESTCHOICE Script "Shall we contin-
  ue?" Yes No'
  IF VAL $Continue EQ 0
  ECHO "Time to quit"
  QUIT
  ENDIF
ECHO "Let's go on"
```

REQUESTFILE Command

Location: Release 3 C:

Function

The REQUESTFILE command was added in Release 3 to allow the use of the standard ASL file requester (the interactive window that displays a list of filenames and asks you to select one) in a script or AREXX program. The command returns the name of the file that was selected, surrounded by double quotes. If more than one file is selected, each name is printed to the standard output device (usually the console), separated by spaces. If the user selects the window close gadget, or the "CANCEL" button instead of a

filename, the program ends with a return code of 5, which can be detected by the "IF WARN" statement (see chapter 5 for more information about the use of IF in scripts). If a file is selected, the program ends with a return code of zero.

REQUESTFILE takes a bewildering array of command parameters, but all of them are optional. If you just issue the REQUESTFILE command by itself, it presents the standard file requester with which almost every user will be familiar.

Release 3 Format

REQUESTFILE [***DRAWER*** *dir*] [***FILE*** *filename*] [***PATTERN*** *pat*] [***TITLE*** *titletext*] [***POSITIVE*** *postext*] [***NEGATIVE*** *negtext*] [***ACCEPTPATTERN*** *pat*] [*REJECTPATTERN pat*] [***SAVEMODE***] [***DRAWERSONLY***] [***MULTISELECT***] [***NOICONS***] [***PUBSCREEN*** *screenname*]

Explanation of Parameters and Keywords

[***DRAWER*** *dir*] If this option is used, the initial directory listing will be that of the drawer *dir,* and this directory path will appear in the drawer gadget of the file requester.

[***FILE*** *filename*] If this option is used, the requester comes up with *filename* selected as the default file. If this file is not in the current directory, use the DRAWER option to specify its directory.

[***PATTERN*** *pat*] When this option is used, the file requester comes up with a pattern gadget, and the file list only shows the files that match the AmigaDOS pattern specified in *pat.* This can be used in the case where the name of the file to be selected usually follows a standard naming convention. The user is free, however, to change this pattern, and thus may still select any file.

[***TITLE*** *titletext*] This option permits you to designate a text string (*texttitle*) that will appear in the title bar of the window. This title should indicate the purpose behind the file selection (e.g. "Load a picture file"). There is limited space on the title bar, so a message that is more than about 30 characters or so may be truncated.

[***POSITIVE*** *postext*] This option allows you to change the message on the leftmost button on the file requester, which the user presses to select a file. If this option is not used, the button is labeled "Ok". For example, if

the file is to be deleted, you might change the button to read "DELETE", or if the file is to be loaded, to "LOAD".

[***NEGATIVE*** *negtext*] This option allows you to change the message on the rightmost selection button, which the user presses to exit without selecting a file, from its default text of "CANCEL." You may wish to change the button to something more meaningful, such as "Skip" or "Abort."

[***ACCEPTPATTERN*** *pat*] This option allows you to limit the file display to those files whose names match the pattern *pat.* Unlike the PATTERN keyword, this option does not give the user a chance to change the pattern, so there is no way to select a file that doesn't match the pattern. If *pat* is "#?.c", for example, only C language source code files whose names end in ".c" will be displayed.

[***REJECTPATTERN*** *pat*] This option allows you to exclude from the display files whose names match the pattern *pat.* This option does not give the user a chance to change the pattern, so there is no way to select a file that doesn't match the pattern. If *pat* is "#?.h", for example, C language header files whose names end in ".h" will not be displayed.

[***SAVEMODE***] This option causes the requester to operate in "save mode," in which the file display window is black with gray letters instead of gray with black letters, and in which the multiselect option is disabled.

[***DRAWERSONLY***] This option eliminates all filenames from the display, and lists only directory names.

[***MULTISELECT***] The MULTISELECT option allows the user to select more than one file by holding down the Shift key while clicking on each additional filename. All selected filenames are printed to the standard output device (usually the console), separated by spaces, when the "Ok" button is pressed.

[***NOICONS***] This option eliminates icon files (those whose names end in ".info") from the display. It is the same as specifying "REJECTPATTERN #?.info".

[***PUBSCREEN*** *screenname*] This option allows you to display the file requester on the public screen whose name is *screenname.*

Examples

1. Have the user select a directory within a script, and store the directory name in the local environment variable dirname:

```
UNSET dirname
SET dirname 'REQUESTFILE DRAWERSONLY'
IF "$dirname" EQ "*$dirname"
ECHO "You didn't choose a directory name"
ELSE
ECHO "The directory you chose was $dirname"
ENDIF
UNSET dirname
```

Note that we had to test for the Cancel button by seeing if "$dirname" (which is replaced by the contents of the environment variable) is equal to the literal string "$dirname" (the asterisk means that the IF command will interpret the dollar sign as a dollar sign, not the contents of an environment variable). We could not use the IF WARN test because we only get the return code from the SET command, not the REQUESTFILE command that it contains.

RESIDENT Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The RESIDENT command is used to load programs into memory and to keep them resident there, where they may be executed without having to load them from disk each time. This not only saves the time required for loading the command, but also can save memory in a multitasking environment, since several Shell windows can execute the same program code simultaneously, without having to load a separate copy of the program for each Shell.

Commands may be made resident only from a Shell window, and not from a pre-1.3 CLI. Moreover, only program files that meet certain specifications can be made resident, as explained below.

1.3 Format

RESIDENT *name filename* [***REMOVE***] [***ADD***] [***REPLACE***] [***PURE***] [***SYSTEM***]

Release 2 and 3 Format

RESIDENT *name filename* [***REMOVE***] [***ADD***] [***REPLACE***] [***PURE*** or ***FORCE***] [***SYSTEM***]

Explanation of Parameters and Keywords

name An optional resident name for the program. For example, you may choose to call your resident version of the DIR program by the letter D. If no resident name is specified, the filename is used as the resident name.

filename The name of the program file to be made resident. The full path name should be used.

Not all commands may be made resident. Resident commands must be reexecutable, which means that they must be able to be run a number of times in a row without being reloaded or reinitialized. They must also be reentrant, which means that the same copy of the program must be able to be executed from different Shells simultaneously, without destroying internal data that other copies of the program may generate. Many of the programs in the C directory of Workbench 1.3 can be made resident, and their files have the pure protection bit set to indicate this fact. How can you tell if a program can be made resident? If it can, the documentation usually will say so, and the pure bit will be set on the program file.

[***REMOVE***] This keyword is used to remove the indicated resident name from the resident list. This operation will succeed only if the resident command is not currently in use. Under Release 2 and 3, REMOVE may also be used to disable Internal commands (they may be re-enabled with the REPLACE option).

[***ADD***]

[***REPLACE***] Use of these keywords is strictly optional, since the behavior of RESIDENT is about the same whether or not they are used. That is, if these keywords are used without a filename (RESIDENT ADD or RESIDENT REPLACE), the command lists the programs on the resident list, just as it would if you issued the command RESIDENT. If either is used with a filename, the RESIDENT command tries to place that file on the resident list. If there is another file with the same filename or resident

name already on the list, the new command will replace the old one, unless the old one is already in use and cannot be deleted. Under Release 2 and 3, however, use of the REPLACE option will fail if the resident name is not already on the list. Also, under Release 2 and 3, REPLACE may be used to re-enable an Internal command that had been disabled with the REMOVE option.

[***PURE*** or ***FORCE***] Normally, a command cannot be made resident unless its file has the pure bit set (see PROTECT). When the PURE keyword is used, however, RESIDENT is forced to load the program whether or not the pure bit is set, and to print the warning message *Pure bit not set.* Of course, the file still must be an executable program—RESIDENT can't make a data file resident. Under Release 2 and 3, FORCE was added as a synonym for PURE.

Using the PURE option to make possibly unsuitable command resident is a hazardous proposition which can lead to a system crash. Therefore, if you wish to experiment with making commands resident, try to do so under conditions that won't lead to catastrophic data loss. Don't experiment with RESIDENT, and then go to work on your most important project.

[***SYSTEM***] This option is used to add a command to the system portion of the resident list. Once added, this command cannot be deleted by the user. The most common use of this keyword is to add the Shell-Seg program that enables Shell windows in place of ordinary CLI windows under Workbench 1.3. The command you use to make the Shell-Seg resident is:

```
RESIDENT CLI L:Shell-Seg SYSTEM
```

This command is normally issued automatically by the 1.3 Startup-sequence script. It is not needed under Release 2 and 3, because the Shell is built into the operating system.

If used without a filename, the SYSTEM keyword can be used to list the resident system commands along with the resident user commands.

Examples

1. Make the DIR command resident, using the resident name D:

```
RESIDENT D C:DIR
```

2. Remove the EXECUTE command from the resident list:

```
RESIDENT execute REMOVE
```

3. List all of the commands (including the system commands) on the resident list:

```
RESIDENT SYSTEM
```

4. Temporarily disable the Internal RUN command:

```
RESIDENT RUN REMOVE
```

RUN Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The RUN command may be used to create a system CLI task which executes in the Amiga's background (in other words, the task doesn't present you with an interactive CLI window). RUN allows multiple AmigaDOS commands (each separated by a plus sign) to be executed in sequence. Once all commands given to a RUN statement are executed, the background task disappears.

When RUN is initiated the system prints the message

```
[CLI n]
```

where *n* is the task number assigned to the background task. Immediately after the message is issued, control is returned to the CLI from which RUN was issued. The background task keeps running until all commands are completed or until the task is interrupted by the BREAK command. The commands are executed sequentially. If any command fails with an error code, the background task terminates and removes itself.

With version 1.3 or higher, RUN checks the resident list before looking in the current directory for a command, so that it uses the resident copy if available. Also, if the input and output of the RUN command is redirected from and to the NIL: device, the existence of a background task usually will not prevent the closing of the CLI window from which that task was launched.

1.3 and Release 2 and 3 Format

RUN *command+command...*

Explanation of Parameters and Keywords

command+command... The AmigaDOS command you want executed in the background. More than one command may be executed by a single RUN command. To build a RUN sequence with multiple commands, end each command line with a plus sign (+) and press RETURN. RUN treats the plus sign as a command delimiter. The cursor will jump to the beginning of the next line, at which point you may enter another command. Keep ending each command line with a + until you've entered the last one for this RUN sequence. End the last command line with a RETURN (no + preceding it). RUN then begins processing the commands—one by one—in the background. You may receive messages and requester boxes from background tasks.

Examples

1. Print a complete directory and file listing of the current drive to the printer. The print operation is to be executed in the background:

```
RUN DIR > PRT: OPT A
```

2. Format a blank disk in drive df1: and then install boot files on the newly formatted volume. Print a message on the screen when the format and install are done. The operations are to be executed in the background by a single task:

```
RUN FORMAT DRIVE DF1: NAME EMPTY +
INSTALL DF1: +
ECHO "Format and Install Finished"
```

3. Execute the command sequence file *My Command File* located on the system RAM disk. The command file is to be executed in the background by a single task:

```
RUN EXECUTE "RAM:My Command File"
```

4. Start the *Clock* program as a background task, and redirect the output of the RUN command to NIL: so that you can close the CLI while *Clock* is still running:

```
RUN >NIL: Clock
```

SEARCH Command

Location: C:

Function

SEARCH lets you scan AmigaDOS files for a specified string of characters. You may SEARCH a single file, all files matching an AmigaDOS pattern, all files within a directory, and, optionally, all files within a directory's subdirectories.

SEARCH displays the name of the AmigaDOS file currently being searched and, if the search text is found, all lines containing the search text. Each displayed line is preceded by a line number. In version 1.3 and higher, the command returns a code of zero if the object is found, and 5 (WARN) if it isn't, which makes the command more useful in scripts. The CTRL-C key combination may be used to stop the search. The CTRL-D combination may be used to stop searching the current file, and proceed with the next file.

AmigaDOS treats the carriage return character as an end-of-line character. SEARCH examines only the first 205 characters of each line. If SEARCH comes across a line longer than 205 characters, the message *LINE n truncated* displays and SEARCH continues.

1.3 Format

SEARCH [***FROM***] *name* [***SEARCH***] *string* [***ALL***] [***NONUM***] [***QUIET***][***QUICK***] [***FILE***]

Release 2 and 3 Format

SEARCH [***FROM***] *name* [***SEARCH*** or ***NAME***] *string* [***ALL***] [***NONUM***] [***QUIET***][***QUICK***] [***FILE***] [***PATTERN***]

Explanation of Parameters and Keywords

[***FROM***] *name* The file or directory that you want searched; *name* also may be an AmigaDOS pattern. (See Chapter 3, "The Filing System," for detailed information on patterns and their uses.) If *name* is the first argument in the SEARCH command, the FROM keyword is optional.

[***SEARCH*** or ***NAME***] *string* The text string that will be searched for. If *string* is the second argument in the SEARCH command, this second SEARCH keyword is optional. If *string* contains any spaces, it must be enclosed in quotation marks. Case (uppercase, lowercase) within *string* is

ignored by SEARCH. A search string of *rubber ducky,* for instance,will match the text found in a file which contains the phrase *Ernie bought his rubber ducky an Amiga.* Under Release 2 and 3, NAME can be used as a synonym for the SEARCH keyword.

[***ALL***] If the ALL keyword is specified and name is an AmigaDOS directory, all files within the directory and its subdirectories are searched.

[***NONUM***] This option, added in version 1.3, suppresses the output of line numbers along with the strings.

[***QUIET***] This option, added in version 1.3, suppresses all output, to facilitate the use of the command in scripts.

[***QUICK***] The QUICK option, introduced in version 1.3, uses a more compact format for the output.

[***FILE***] The FILE option, introduced in version 1.3, searches for a file whose name is exactly the same as string, rather than searching for the string within the contents of the file. If the file is found, its name is printed. In version 1.3 only the filename is printed, but in Release 2 and 3, the full path name is printed. Thus, the Release 2 and 3 version can be used to find a particular file on a large disk volume.

[**PATTERN**] This optional keyword, introduced in Release 2, indicates that a pattern is used in the search.

Examples

1. Search all files within the directory called *Mayan/Civilization* and all files within its subdirectories for the phrase *ancient astronauts*:

```
SEARCH Mayan/Civilization "ancient astronauts" ALL
```

2. Search a file called *MyLetters* for the word *gorilla*:

```
SEARCH MyLetters gorilla
```

3. Search all files which end with *.bills* in the current directory for the phrase *blank disks.* Redirect the QUICK output to the system printer:

```
SEARCH > PRT: #?.bills "blank disks" QUICK
```

4. Under Release 2 and 3, search volume Kalamazoo and its subdirectories for all files whose names contain the word *ELVIS* in them:

```
SEARCH Kalamazoo: #?ELVIS#? PATTERN FILE ALL
```

; (Semicolon) Command

Location: Internal

Function

The semicolon (;) command allows the insertion of comments in command sequence files and command lines. The comments may be on the same line as other AmigaDOS commands, or they may stand by themselves on a separate line. Anything to the right of a semicolon in an AmigaDOS command line is considered a *comment.*

1.3 and Release 2 and 3 Format

; [*comment*]

Explanation of Parameters and Keywords

[*comment*] May be any text string, up to 254 characters in length (if the ; is the first character of a line).

Example

Here is a simple example of a command sequence file with comments, using the ; command. Remember, everything to the right of a semicolon is considered a comment.

```
; Niagra Falls Routine
IF Curly EXISTS ; Test for a stooge
SAY Slowly I turned ; Set em up for the gag
WAIT 5 SECS ; Dramatic pause
SAY Inch by Inch
ELSE
;Sign off without gag
SAY th..th..th..thats all folks
ENDIF
```

SET Command

Location: Release 2 and 3 only Internal

Function

SETENV is used to assign a text string to a local environment variable, or to delete a text string already assigned to such a variable. A local environment variable is a named text string that is stored in an environment space and is accessible only to the Shell in which it was created (and Shells spawned from that Shell). Unlike the global variables created by SETENV, the local environment variables are stored in private system memory, rather than in the RAM disk.

Release 2 and 3 Format

SET *varname* [*string*]

Explanation of Parameters and Keywords

varname The name of the local environment variable to set.

[*string*] The text string to assign to the environment variable. Like all such strings, if there are spaces in the text, the entire string must be enclosed in quotes. If no string is specified, an empty string will be copied to the variable, effectively removing the text string currently associated with it. To remove the variable completely, use the UNSET command.

Under Workbench Release 2 and 3, there are a number of significant local environment variables that are automatically set for you, or which you can set yourself. These include:

Process	The process number of the current Shell.
RC	The return code of the last command that was executed. This allows you to examine the code without using the IF WARN or IF FAIL command.
Result2	The error number that indicates why the last command failed. You can use the FAULT command to interpret these error codes.

Echo — This local environment variable controls whether or not the Shell repeats each command as it is executed. If you SET Echo on, the commands are repeated. If you set Echo to anything else (or don't set it at all), they are not repeated. Turning Echo on is a good way to debug scripts that don't work, since there is often no other way of telling at which line they failed. With Echo on, you can tell which lines are executed properly, and which fail.

Example

Copy the word *Swordfish* to a local environment variable named *Password*:

```
SET Password Swordfish
```

SETCLOCK Command

Location: 1.3 and Release 2 and 3 C:

Function

SETCLOCK is used to copy the time and date from the hardware clock on the 2000 and 3000 (optional on the 500 and 600) to the AmigaDOS software clock, or vice versa. It only works with Commodore's own clock/calendar, or compatible units.

Format

SETCLOCK *LOAD* or ***SAVE*** or ***RESET***

Explanation of Parameters and Keywords

LOAD or ***SAVE*** or ***RESET*** — One of these three keywords must be used with SETCLOCK. If the SAVE option is used, the current AmigaDOS system time and date (the one set with the DATE command) is copied to the hardware clock. If the LOAD option is used, the stored time and date is copied from the hardware clock to the system clock (this is performed automatically in the 1.3 startup sequence, and by the Release 2 and 3 Kickstart ROM). The RESET option is used to start the clock up again if it has been turned off by some runaway program that accidentally wrote to its hardware registers. (The system may show the clock as unset, or report "battery backed clock not found.")

Example

Set the system time and date from the hardware clock:

```
SETCLOCK LOAD
```

SETDATE Command

Location:1.3 and Release 2 and 3 C:

Function

Changes the date or time associated with a file or directory. The time and date of the file creation can be displayed with the LIST command.

SETDATE is useful when the date and time associated with a directory or file doesn't reflect its true creation date, either because the date and time wasn't set correctly when the file was created, or because the file was copied from another disk, in which case the date-stamp reflects the date of the *copy,* not the date of creation. It can also be used to manipulate a "make" type program, which directs a C compiler to compile files based on the date-stamp.

1.3 Format

SETDATE *name* [*date*] [*time*]

Release 2 and 3 Format

SETDATE *name* [*date*] [*time*] [***ALL***]

Explanation of Parameters and Keywords

name The name of the directory or file whose date stamp you wish to change. Under Release 2 and 3, you may use pattern matching to change several files at once.

[*date*] The day, month, and the year associated with the file. The date is usually specified as *DD-MMM-YY,* where DD is a two-digit number representing the day, MMM is a three-letter abbreviation of the month (such as FEB or JUN), and YY is the last two digits of the year. The SETDATE command, like the DATE command, also allows indirect references for setting the date, such as YESTERDAY, or WEDNESDAY. For more complete details, see the DATE command. If no date or time is included with the filename, the file is set to the current system time and date.

[*time*] The optional time-stamp for the file, expressed in the format *HH:MM:SS,* where each is a two-digit number representing the hours, minutes, and seconds. Hours are set in 24-hour format, where 1:00 p.m. is referred to as 1300 hours. If minutes or seconds are omitted, they're set to zero. If this optional parameter is omitted when the day, month, and year are specified, the time-stamp on the file is set to 00:00:00.

Examples

1. Change the date-stamp of the *Printers* subdirectory of the *Devs* directory on the Workbench disk to show a creation date of January 5, 1987 at 1:56 p.m.:

```
SETDATE Workbench:Devs/Printers 05-jan-87 13:56
```

2. Change the date-stamp of the file *Mydata* in the current directory to midnight yesterday:

```
SETDATE Mydata Yesterday
```

3. Change the date-stamp of the file *"Au Courrant"* in the current directory to the current time and date:

```
SETDATE "Au Courrant"
```

SETENV Command

Location: 1.3 C: Release 2 and 3 Internal

Function

SETENV is used to assign a text string to an environment variable, or to delete a text string already assigned to such a variable. An environment variable is a named text string that is stored in an environment space that is accessible to all tasks. Currently, an ENV: directory is created on the RAM: disk to hold global environment variables. This means that "SETENV test TestString" is really the same as "ECHO >ENV:test TestString". In future versions, the global environment variables may be stored in system RAM, and manipulated by their own device handler, as the local environment variables are.

1.3 and Release 2 and 3 Format

SETENV *varname* [*string*]

Explanation of Parameters and Keywords

varname The name of the environment variable to set. Currently, this is the name of a text file that is stored in the ENV: directory. Using the SETENV command copies the text string to this file.

Under Workbench Release 2 and 3, there are a number of significant global environment variables that are automatically set for you, or which you can set yourself. These include:

Kickstart

Workbench—These variables are created by the startup-sequence script, and contain the version numbers of the Kickstart and Workbench that you are using.

Editor—This environment variable is recognized by some Workbench programs like the MORE program. If you set this variable to the path name of your text editor, MORE allows you to bring up the program to edit the current file by pressing Shift-E.

[*string*] The text string to assign to the environment variable. Like all such strings, if there are spaces in the text, the entire string must be enclosed in quotes. If no string is specified, an empty string will be copied to the variable, effectively removing the text string currently associated with it. To totally remove the environment variable under Release 2 and 3, use the UNSETENV command.

Example

1. Copy the name *Darlene* to an environment variable named *Cubby*:

```
SETENV Cubby Darlene
```

2. Remove the text string associated with the environment variable named *Annette*:

```
SETENV Annette
```

SETFONT Command

Location: Release 2 and 3 C:

Function

The SETFONT command was added in Release 2 to allow the user to specify the text font, size, and style used in a particular Shell window.

Normally, each Shell window uses the System Default Text font that was set from the Font preferences program.

Release 2 and 3 Format

SETFONT*fontname fontsize* [***SCALE***] [***PROP***] [***ITALIC***] [***BOLD***] [***UNDERLINE***]

Explanation of Parameters and Keywords

fontname The name of the font to install. This must either be a system bitmap font that is installed in the FONTS: directory, or a scalable outline font that was installed with the Fountain program.

fontsize The size of the font, expressed as a given number of lines in height. If no bitmap of that size is installed, SETFONT will give you the next closest size, unless you set the SCALE option.

[***SCALE***] This optional keyword enables bitmap scaling. This means that if no bitmap font of size *fontsize* is installed, the SETFONT program will create one by scaling the characters of the next closest size.

[***PROP***] This optional keyword permits the use of proportional fonts. If you try to use a proportional font without using this keyword, the command will fail with the message *Object not of required type.* Some proportional fonts will not print correctly in a console window.

[***ITALIC***] This keyword gives you the italic version of the font.

[***BOLD***] This keyword gives you the bold version of the font.

[***UNDERLINE***] This keyword gives you the underlined version of the font.

Examples

1. Change the font in the current Shell to a scaled version of the Topaz font, sixteen lines high:

```
SETFONT Topaz 16 SCALE
```

2. Change the font in the current Shell to a bold italic version of the proportional Diamond font, twelve lines high:

```
SETFONT Diamond 12 BOLD ITALIC PROP
```

SETKEYBOARD Command

Location: 2.1 C:

Function

Changes the default key map of the keyboard for this particular Shell window, allowing the use of different keyboard layouts for foreign countries. For more information on key maps, see the SETMAP command.

The SETKEYBOARD command looks for its key map data files in the *Keymaps* subdirectory of the directory designated as the logical device DEVS: (usually the *Devs* directory of the startup disk). You can, however, designate another directory by typing the entire path name of the key map file (for instance SYS:Storage/Keymaps/po). The standard key map files on the 2.1 Workbench disk include:

cdn	French Canadian
ch1	Swiss French
ch2	Swiss German
d	German
dk	Danish
e	Spanish
f	French
gb	Great Britain
i	Italian
n	Norwegian
p	Portugese
s	Swedish
usa0	Emulates the standard key mapping of the 1.1 Workbench.
usa1	Includes maps for additional numeric pad keys on 500/2000.
usa2	U.S. keyboard with Dvorak layout

2.1 Format

SETMAPmapfile

Explanation of Parameters and Keywords

mapfile The name of the key map data file describing the keyboard layout to be used. If the file is located in the *Keymaps* subdirectory of the directory designated as *DEVS:*, you only need include the filename. Otherwise, you must specify the entire path name.

Example

Install the British key map in this Shell window:

```
SETKEYBOARD gb
```

SETMAP Command

Location: 1.3 and 2.0 SYS:SYSTEM

Function

Changes the default key map of the keyboard, allowing the use of different keyboard layouts for foreign countries.

The key map is a data table to which the Amiga refers when a key is pressed. By changing this key map, you can change which character is printed when you press a particular key. For example, the French keyboard has the letter *Q* reversed with the letter *A*, so when you use the SETMAP command to install the French key map, every time you press the *A* key a *q* appears, and vice versa. Keymaps can also be used to assign strings to the function keys.

The SETMAP command looks for its key map data files in the *Keymaps* subdirectory of the directory designated as the logical device DEVS: (usually the *Devs* directory of the startup disk). The standard key map files on the Workbench disk include:

cdn	French Canadian
ch1	Swiss French
ch2	Swiss German
d	German
dk	Danish
e	Spanish
f	French
gb	Great Britain
i	Italian
is	Icelandic
n	Norwegian
s	Swedish
usa0	Emulates the standard key mapping of the 1.1 Workbench.
usa1	Includes maps for additional numeric pad keys on 500/2000.

usa2 U.S. keyboard with Dvorak layout

Some of the new key maps implement a new feature known as *dead keys.* A dead key is one which prints out a character only when struck preceding another character. It's often used for accented vowels. In order to produce an *a* with an accent mark over it, for example, you would hit the accent key (the dead key) first. Nothing prints on the screen when you hit the accent key, but if the next key you press is an *a,* an accented *a* appears.

Complete information about the layout of the various foreign keyboards is included in *Introduction to the Amiga.* The *Keytoy* program (named *KeyShow* in Release 2 and 3), included in the *Tools* directory of the *Extras* disk, displays a graphics representation of the current key map as well.

Version 2.1 and higher of the operating system incorporates language localization. In those versions, the key map is set by the Locale preference program. You can change the mapping for a particular Shell window using the SETKEYBOARD command.

1.3 and Release 2 Format

SETMAP mapfile

Explanation of Parameters and Keywords

mapfile The name of the key map data file describing the keyboard layout to be used. This file should be located in the *Keymaps* subdirectory of the directory designated as *DEVS:*. In order to restore the default key map, use the name *usa* for the mapfile. This key map is part of the operating system and need not be read from a disk file.

Examples

1. Install the French language key map:

```
SETMAP f
```

2. Restore the default system key map:

```
SETMAP usa
```

SETPATCH Command

Location: 1.3 and Release 2 and 3 C:

Function

The SETPATCH command is supplied by Commoodore to fix any known problems with the system software in the Kickstart ROMs. It should be run as the first command in the *startup-sequence* file. The command prints a list of the patches (program corrections) made.

1.3 Format

SETPATCH [*r*]

Release 2 and 3 Format

SETPATCH [***NOCACHE***] [***QUIET***]

Explanation of Parameters and Keywords

[*r*] This optional switch is used to allow the recoverable RAM disk (RAD:) to recover sucessfully on machines that are running Workbench 1.3 with one megabyte of CHIP RAM. It need only be used with systems that include the one-Meg Agnus chip (on which AVAIL shows over 512K of CHIP RAM). Note that the letter *r* should be in lower case, as some versions of this command are case sensitive.

[***NOCACHE***] This Release 2 and 3 option can be used to disable the data cache on accelerator cards. Normally, Kickstart turns the cache on, since this increases the speed of operations, but some older programs do not run correctly with the data cache turned on.

[***QUIET***] This option prevents the command from printing out a status report of the patches that it has made.

Example

Patch the 1.3 Kickstart routines without printing a list of the patches:

```
SETPATCH >nil:
```

SKIP Command

Location: 1.3 C: Release 2 and 3 Internal

Function

The SKIP command is used within command sequence files to jump to a specified label. When a SKIP is executed, command execution continues at the line following the label.

If SKIP is executed with no label specified, command execution continues with the commands following the next LAB command in the command file.

If a SKIP is executed and the label specified is not found, or if a SKIP with no label searches to the end of the command file without encountering a LAB command, command file execution is terminated and the message *label* label *not found by SKIP* is displayed.

In versions 1.3 and higher, an option allows skipping backwards in the file as well as forwards.

1.3 and Release 2 and 3 Format

SKIP [*string*] [***BACK***]

Explanation of Parameters and Keywords

[*string*] The string attached to a LAB command which SKIP searches for in the currently executing command file. The search starts at the command following SKIP and continues downward toward the end of the command file. If the matching LAB *string* command precedes the SKIP command, SKIP will not find it, and the command file terminates with an error (unless the BACK option is used).

If *string* is not specified, the first LAB command following SKIP will be skipped to.

[***BACK***] This option, added in Workbench 1.3, starts the search for the specified label at the beginning of the file, not at the line following the SKIP command. You still may not SKIP backwards past an EXECUTE statement, however.

Examples

1. Transfer control to the commands immediately following the next *LAB filecontrol* command in the current command sequence file:

```
SKIP filecontrol
```

2. Transfer control to the commands immediately following the next LAB command in the current command sequence file:

```
SKIP
```

3. Transfer control to the commands immediately following the preceding *LAB Jumpback* command in the current command sequence file:

```
SKIP Jumpback BACK
```

SORT Command

Location:1.3 and Release 2 and 3 C:

Function

SORT performs an alphabetic sort of the lines within an AmigaDOS text file. Lines are sorted according to the ASCII value of their beginning characters.

Within a file, AmigaDOS treats any string of characters that ends with a linefeed character as a single line. SORT compares lines, beginning with the first character, unless a different sort start position is specified via the COLSTART keyword. Lines that begin with numbers will precede those that begin with alphabetic characters in the sorted version of the original file. Lines with numbers will be in ascending order. Case is ignored by SORT (unless an available version 2 option is used). For instance, a SORT of a file containing these lines:

1,234,576
a sunny spring day
AOK
rags to riches
.Hiya.
1
R2D2 and C3PO
3.14159
.0000000001

results in this output:

.0000000001
.Hiya.

1
1,234,576
3.14159
a sunny spring day
AOK
R2D2 and C3PO
rags to riches

Older versions of SORT are not particularly fast, especially when the size of the file to be sorted is longer than 50 lines. They also have a problem sorting files that are over 200 lines long without increasing the stack size (see the STACK command description for details). SORT fails if the file to be sorted is larger than the system's available free memory.

1.3 Format

SORT [***FROM***] *fromname* [***TO***] *toname* [***COLSTART*** *n*]

Release 2 and 3 Format

SORT [***FROM***] *fromname* [***TO***] *toname* [***COLSTART*** *n*] [***CASE***] [***NUMERIC***]

Explanation of Parameters and Keywords

[***FROM***] *fromname* The name of the AmigaDOS file whose contents are to be sorted. If *fromname* is the first argument of a SORT command, the FROM keyword is optional.

[***TO***] *toname* The name of the AmigaDOS file or logical device to which the sorted lines from *fromname* will be sent. If *toname* is the second argument of a SORT command, the TO keyword is optional; *toname* must be different from *fromname* or the SORT will fail.

[***COLSTART*** *n*] Causes SORT to compare lines beginning with the *nth* character in each line. If *n* is given, the COLSTART keyword *must* be used. If COLSTART *n* has been specified and lines are found to be equal, SORT attempts a secondary sort of the equal lines, starting with the first character of each line.

[***CASE***] If the CASE option is used, capital letters are given precedence over lower case letters. Normally, case is ignored.

[***NUMERIC***] When the NUMERIC option is used, all lines are interpreted as numbers, stopping at the first non-numeric character. Lines that begin with letters are assigned a value of zero. If both the CASE and NUMERIC options are used, CASE is ignored.

Examples

1. Sort the contents of a file called *Mixed Up* to a file called *InOrder:*

```
SORT "Mixed Up" InOrder
```

2. Sort the contents of a file called *Inventory* in the *majorappliance/washers* directory. Print the sorted output on the system printer:

```
SORT :majorappliance/washers/Inventory PRT:
```

3. Sort the contents of a file called *widgets* located on the system RAM disk, comparing lines beginning with the fifth character of each. Display the sorted output on the current system console screen:

```
SORT RAM:widgets * COLSTART 5
```

STACK Command

Location: 1.3 C: Release 2 and 3 Interial

Function

The STACK command may be used to display or set aside the amount of stack space for the currently active CLI. The stack space is used by AmigaDOS commands and all other programs as a sort of intermediate work area. The default stack size for a CLI environment is 4000 bytes, which is large enough to execute the vast majority of AmigaDOS commands successfully.

Older versions of two AmigaDOS commands may require a stack size greater than 4000 bytes. If an older SORT command is executed on a file with more than 200 lines, or if an older DIR is issued against a file structure with more than six levels of directories, the stack size should be increased. The exact size is open to question. According to the developers of AmigaDOS, optimum stack sizes for a specific heavy SORT or DIR are a matter of trial and error.

You can check the stack size of all active system tasks with the STATUS command.

Format

STACK [*newsize*]

Explanation of Parameters and Keywords

[*newsize*] The amount of space, in bytes of memory, that you wish to assign as stack space for the currently active CLI. If *newsize* is omitted, the current stack size is displayed.

Examples

1. Display the stack size of the currently active CLI:

```
STACK
```

2. Change the stack size of the currently active CLI to 12,000 bytes:

```
STACK 12000
```

STATUS Command

Location: Internal

Function

The STATUS command displays system information about active tasks. STATUS displays the stack size, global vector size, priority, and program name associated with active tasks. This kind of detailed technical information is of interest mostly to advanced programmers. STATUS can come in handy when using the RUN and BREAK commands, however. The CLI STATUS keyword may be used to check what command is currently active in both foreground and background CLI environments, something you may forget once you execute a RUN. STATUS also may be used to find the task number of a particular program, so that you may send it a BREAK command.

1.3 and Release 2 and 3 Format

STATUS [*tasknum*][***FULL***] [***TCB***] [***CLI*** or ***ALL***] [***COMMAND*** or ***COM***]

Explanation of Parameters and Keywords

[*tasknum*] The number of the task which STATUS is to report on. If *tasknum* is not specified, all active tasks are reported.

[***FULL***] FULL displays all the information normally reported by STATUS if both the TCB and ALL keywords were all specified. The FULL keyword is optional.

[***TCB***] Causes STATUS to display information dealing with the stack size, global vector size, and priority of each active task known to the system. The TCB keyword is optional.

[***CLI*** or ***ALL***] Specifying either CLI or ALL causes STATUS to report on all currently active CLI tasks and display the names of all commands currently loaded within the CLIs (this is the same result as when you issue the STATUS command by itself). The CLI and ALL keywords are interchangeable and optional.

[***COMMAND*** or ***COM***] This option, added in version 1.3, prints the task number of the CLI from which the program named in *filename* was run. This allows you to send that program a BREAK, for example, using a script file. If the command is not found, a return code of 5 (WARN) is set.

Examples

1. Display an abbreviated status report on all active tasks:

```
STATUS
```

2. Print the stack size, global vector size, priority, and segment list section names of each active task known to the system on the system printer:

```
STATUS > PRT: FULL
```

3. Send a BREAK to the task running the WAIT command:

```
STATUS >ram:temp COMMAND WAIT
BREAK <ram:temp >NIL: ?
DELETE ram:temp
```

Under Release 2 and 3, you can accomplish the same task with one line:

```
BREAK 'STATUS COMMAND WAIT'
```

TYPE Command

Location:1.3 and Release 2 and 3 C:

Function

The TYPE command lets you output the contents of any AmigaDOS file to the screen, a disk file, or any AmigaDOS physical device.

TYPE is most often used to examine the contents of a file, although it may actually be used to copy a file. TYPE can format its output as hexadecimal numbers, or include line numbers at the beginning of each output line.

TYPE's output may be paused by hitting the space bar (or any other key) and resumed by hitting the RETURN key, BACKSPACE key, or holding down the CTRL-X key combination. Its output may be canceled by breaking the command with CTRL-C.

1.3 and Release 2 and 3 Format

TYPE [***FROM***] *fromname* [[***TO***] *toname*][***OPT N*** or ***NUMBER*** or ***OPT H*** or ***HEX***]

Explanation of Parameters and Keywords

[***FROM***] *fromname* The name of the file you want TYPEd; *fromname* is required and may be any valid AmigaDOS filename. The FROM keyword is optional and need not be specified if *fromname* immediately follows TYPE. Under Release 2 and 3, you may specify multiple filenames, each separated by a space, or use pattern matching to specify multiple files.

[[***TO***] *toname*] The name of the file or device you want the output of the TYPE operation sent to. In versions 1.3 and below, the TO keyword is optional if the first argument of TYPE is *fromname* and the second argument is *toname*. In version 2, a second filename will be presumed to be another *fromname* unless the TO keyword is used.

If no destination for TYPE's output is specified, the output is displayed on the screen. *toname* may be an AmigaDOS file or an AmigaDOS device, such as the printer (PRT:). If *toname* is an existing file, its contents are overwritten; if *toname* is a file which does not exist, it will be created by the TYPE operation. If *toname* is a directory with files in it, TYPE fails; if *toname* is an empty directory, the directory will be deleted and a file called *toname* created.

[***OPT N*** or ***NUMBER*** or ***OPT H*** or ***HEX***] Adding *OPT N* or *NUMBER* to a TYPE command instructs the system to precede each line output by TYPE with a line number. AmigaDOS treats any number of characters within a file ending with a linefeed as one line.

Specifying *OPT H* or *HEX* instructs TYPE to produce a formatted hexadecimal dump of the *fromname* file's contents. The *N* and *H* options are mutually exclusive—only one may be specified. If either option is

specified using the initial instead of the full word, the OPT keyword *must* be used.

Examples

1. Output the contents of a file in the current directory called *textwiz* on the screen:

```
TYPE textwiz
```

2. Copy a file called *copyclone* in a directory called *qwikbuck* to a file of the same name in the directory called *copies* on df1:.

```
TYPE :qwikbuck/copyclone to df1:/copies/copyclone
```

3. Produce a formatted hexadecimal dump of a file called *objectcode* on the printer:

```
TYPE objectcode to PRT: OPT H
```

or

```
TYPE > PRT: objectcode HEX
```

4. List the contents of the file*namelist* with line numbers before each line to a file on the system RAM disk called *tempname*:

```
TYPE namelist to RAM:tempname OPT N
```

UNALIAS Command

Location: Release 2 and 3 Internal

Function

UNALIAS is used to removed an alias from the system list (see ALIAS command for definition of an alias). It also can be used to list the current aliases.

Release 2 and 3 Format

UNALIAS [*name*]

Explanation of Parameters and Keywords

[*name*] The name of the alias you wish to remove. If no *name* is specified, the command will list all current aliases.

Example

Remove the XCOPY alias :

```
UNALIAS xcopy
```

UNSET Command

Location: Release 2 and 3 Internal

Function

UNSET removes the specified local environment variable (see SET command for definition of a local environment variable). If no variable is specified, UNSET merely lists the current local environment variables and their contents.

Release 2 and 3 Format

UNSET [*name*]

Explanation of Parameters and Keywords

[*name*] The name of the local environment variable to remove. If no *name* is specified, the command lists the current local environment variables and their contents.

UNSETENV Command

Location: Release 2 and 3 Internal

Function

UNSETENV removes the specified global environment variable (see SETENV command for definition of a local environment variable). This in effect deletes the named file from the ENV: directory. If no variable is specified, UNSETENV merely lists the current global environment variables.

Release 2 and 3 Format

UNSETENV [*name*]

Explanation of Parameters and Keywords

[*name*] The name of the global environment variable to remove. If no *name* is specified, the command lists all of the global environment variables.

VERSION Command

Location: 1.3 and Release 2 and 3 C:

Function

Displays the internal version number of the *Kickstart* and *Workbench* disks used to start the computer. This provides a CLI equivalent of the Version selection on the Workbench's Special menu. The internal version numbers give much more precise information about which release is in use than a mere 1.3 or 2.0 identifier. Programs that open one or more operating system libraries may specify the release required, so that the program doesn't try to use certain features not available with earlier versions of the system. With the first 1.3 release, the VERSION command returns the following information:

```
Kickstart version 34.5 Workbench version 34.20
```

VERSION returns the following information about the first Release 2 release:

```
Kickstart version 37.175 Workbench version 37.64
```

As of Workbench 1.3, VERSION can be used to find the version and revision number of any library or device. In Release 2 and 3, this capability is expanded to include any command that contains a version string. These capabilities also may be used to test for a particular version in a script file.

1.3 Format

VERSION *libraryname* or *devicename versionnum revisionnum unitnum*

Release 2 and 3 Format

VERSION [*libraryname* or *devicename or filename*] *version_num revision_num unit_num* [***FILE***] [***INTERNAL***] [***RES***] [***FULL***]

Explanation of Parameters and Keywords

libraryname or *devicename* or *filename* This optional parameter can be used to specify the library or device whose version you wish to check. A

library name ends in the letters ".library" (such as "graphics.library"), while a device name ends in the letters ".device" (like "trackdisk.device"). With Release 2 and 3, you also may specify the name of a program file, since many programs contain a version string. If you do not specify a library or device name, VERSION returns the version number of Kickstart and Workbench.

unit_num When checking the version number of a device, an optional unit number may be specified if the device has more than one unit.

version_num
revision_num These optional parameters may be used to verify that the version number (or revision number) is greater than the number specified. If the version number (and, optionally, revision number) is greater than or equal to the one specified, the command returns a code of zero. Otherwise, it returns a code of 5 (WARN).

[***FILE***] When this option is used, VERSION will treat the library or device name specified as a file.

[***INTERNAL***] This option allows you to find the version number of an Internal command.

[***RES***] This option allows you to find the version number of a resident command.

[***FULL***] The FULL option causes VERSION to print out the entire version string, including the date.

Examples

1. Display information about the Kickstart and Workbench disks used to start the computer:

```
VERSION
```

2. In a script, check to see if the version of the graphics library is greater than or equal to 37.35. Print a warning message if it isn't.

```
VERSION >NIL: graphics.library VERSION=37 REVISION=35
IF WARN
ECHO "Graphics library version below 37.35"
ENDIF
```

WAIT Command

Location: 1.3 and Release 2 and 3: C:

Function

WAIT can be used to put a task in a state of suspended animation for a user-definable period of time or until a specified time of day. WAIT can be used in command sequence files or in conjunction with a RUN command.

When WAIT is encountered by the system, the task sits in a seemingly idle state for the specified period of time and then continues with the next command. Sending a BREAK to a WAITing task causes the WAIT to conclude.

Format

WAIT [*n*][***SEC*** or ***SECS***] [***MIN*** or ***MINS***] [***UNTIL*** *time*]

Explanation of Parameters and Keywords

[*n*][***SEC*** or ***SECS***] [***MIN*** or ***MINS***] The amount of time, in minutes or seconds, that the system will wait. If *n* is omitted and the SEC or MIN keyword is specified, *n* defaults to one (1). Using the SEC or SECS keyword tells AmigaDOS to wait *n* seconds, while using MIN or MINS causes the CLI task to wait *n* minutes before continuing. SEC/SECS and MIN/MINS keywords are optional. If they're omitted, the default unit of time is seconds.

[***UNTIL*** *time*] The time of day until which the current process will wait before continuing. If *time* is specified, the UNTIL keyword is required; *time* must be stated in the format *HH:MM,* where *HH* and *MM* are the hour and minute of the day in military (24-hour) time. If UNTIL *time* is used, the system will "wake up" sometime between *HH:MM:*00 and *HH:MM:*59.

Examples

1. Wait for one second:

```
WAIT
```

2. Wait for one minute:

```
WAIT MIN
```

3. Wait for three minutes:

```
WAIT 3 MIN
```

or

```
WAIT 180
```

4. Wait until 10:15 a.m.:

```
WAIT UNTIL 10:15
```

5. Set up a background process using the RUN command that will wait until 11:00 p.m. and then copy all the files in a directory called *documents* to a directory on df1: called *backupdir*:

```
RUN WAIT UNTIL 23:00 +
COPY :documents/#? to df1:backupdir
```

WHICH Command

Location: 1.3 and Release 2 and 3 C:

Function

The WHICH command, introduced in Workbench 1.3, can be used to display the path for a given command. The command in question must be on the resident list, or in the current directory or search path. If there is more than one version of the command available to AmigaDOS, this command will tell you WHICH version will be found first.

1.3 Format

WHICH *filename* [***NORES*** *or* ***RES***]

Release 2 and 3 Format

WHICH *filename* [***NORES*** *or* ***RES***] [***ALL***]

Explanation of Parameters and Keywords

filename The name of the file or directory or logical device to find. WHICH searches in the resident list, the current directory, and the search path. If a directory or logical device is specified, the ASSIGN list is also checked.

[***NORES*** *or* ***RES***] If the NORES option is selected, the resident list is not checked for the command. If the RES option is selected, only the resident list is checked.

[*ALL*] The ALL option causes the command to search all of the paths, as well as internal and resident lists. This should reveal all of the copies of a particular command.

Example

1. Display the directory in which the Setmap program is stored. Don't check the resident list:

```
WHICH SETMAP NORES
```

2. Display the directory to which the logical device WP: is ASSIGNed:

```
WHICH WP:
```

WHY Command

Location: c:

Function

WHY can be used to obtain additional information about failing commands. AmigaDOS is relatively friendly compared with most other computers' disk operating systems. Most DOSs will give no error messages or, at best, minimal messages when a command fails. Even when an error message is displayed, it's often a cryptic numeric which sends you scurrying for the appendix of a DOS manual. When AmigaDOS runs into a problem, it will usually display a message telling you that the command failed, an English language description of the problem or a requester box telling you what needs to be done. Issuing a WHY immediately after a command failure can provide more detailed information on the reason for the failure.

In some instances, WHY will indicate a numeric return code as the reason for the failure. When this happens, the FAULT command can be used to investigate the error code.

WHY can provide meaningful information only if the previous command fails with a nonzero return code. A WHY issued after a successful command, or after a failed command which has already given you all information available, results in the message *The last command did not set a return code.*

Format

WHY

Explanation of Parameters and Keywords

None

Example

A WHY command is issued after an EXECUTE command fails to get more information about the failure:

```
1>EXECUTE nowherefile
EXECUTE: Can't open nowherefile
1>WHY
Last command failed because object file not found
```

Appendix A

Files On the Workbench Disk

The unassuming Workbench disk you received with your Amiga holds a score of directories which contain well over a hundred different files. This appendix will help you get a handle on what files these directories contain, and the purpose of these files.

As you look at a directory listings of your Workbench disk, you'll notice that a number have names ending in the characters *.info.* These files are used by the Workbench to store the information needed to display icons for the program files (tools), data files (projects), directories (drawers), trashcan, and disks. Thus, *Disk.info* contains information needed to display the disk icon on the Workbench itself, *System.info* contains icon information for the System drawer, and so on.

You may also notice that prior to Release 2, each directory for which there is an icon also contains a file named *.icon.* These files contain information about the other icon files in that directory. In general, all .icon files are used only by the Workbench and may be removed if you don't plan to use the Workbench environment.

The files on the Workbench disk are grouped together into a number of directories.

Workbench Directories

Trashcan

The Trashcan directory is used as a temporary holding area for files whose icons are moved to the trashcan icon on the Workbench. When a file named *Programfile* is moved to the Trashcan, it's actually renamed *Trashcan/Programfile*, as is its associated icon file. When you select Empty Trash from the Workbench, the contents of this directory are deleted. The Trashcan directory starts out empty, except for the .info file which is used to hold information about the icon files moved to this directory. Although you cannot delete this directory from the Workbench, you can delete it by using the CLI command:

```
DELETE SYS:Trash#? ALL
```

c

Holds all CLI command programs. Whenever you issue a command to the CLI, AmigaDOS first looks in the current directory for a filename matching the first word of the command line. If it doesn't find the command in the current directory, it then searches the C: device directory. If you primarily use the Workbench interface, and wish to free up some space on your Workbench disk, you can delete all the commands here, except for the ones named in the s:startup-sequence file. These include ADDBUFFERS, BINDDRIVERS, COPY, MAKEDIR, ASSIGN, EXECUTE, RESIDENT, MOUNT, ECHO and a few more. Since these vary depending on the Workbench version, check your startup-sequence file to see which of these commands are used.

Demos

For AmigaDOS 1.2 and earlier. Contains three graphics demonstration programs, which merely open windows onto which dots, boxes, and lines are drawn. This whole drawer can be discarded to make more room on the disk.

System

Contains several system utilities programs, some of which are Workbench equivalents of CLI command programs like NEWCLI, DISKCOPY, and FORMAT. Although most of these programs are useful enough to keep around, there are only a few which you absolutely must keep. DISKCOPY, FASTMEMFIRST, and SETMAP are used by the system, and must not be moved or deleted.

l

AmigaDOS looks for its own library functions in this file. These are extensions to AmigaDOS itself, such as the Ram-Handler file which controls the RAM: disk. Most should not be deleted, and a few, like the Disk-Validator and Port-Handler are absolutely essential.

devs

Contains device drivers for the various devices which the Amiga uses. When a program wants to open a device, it calls the system routine *OpenDevice*, which looks in this directory for the device driver if it's not already been loaded.

Some of the Amiga devices are discussed in Chapter 3—the serial device, the parallel device, and the printer device. The printer device uses printer drivers contained in the subdirectory *printers*. These files provide specific information about the command codes that the named printers use. You select the printer

driver that you wish to use in the Change Printer section of the Preferences program. In addition to the printers named, there is a generic printer driver which performs only minimal translations. You can select this driver from Preferences by choosing Custom and typing in the name generic in the space marked Custom Printer Name.

The devs directory also contains drivers for devices which the CLI commands do not use directly, like the *narrator.device* (speech synthesizer) and the *clipboard.device.* Finally, it contains the system-configuration file that holds the preference settings you save from the Preferences program. About the only thing you may want to delete from this drawer are drivers for printers that you don't use (later Workbench versions store the printer drivers on the Extras disk, and you must install your own printer drivers).

In versions 2.1 and higher, this directory also contains a DOSDrivers drawer, which is used to MOUNT additional devices automatically. Release 3 adds a "datatypes" drawer to facilitate the object-oriented scheme for dealing with different types of sound, picture, and text data used by the MultiView program.

s

Used to hold command sequence files (batch files). When the EXECUTE command is told to execute a sequence file, it first looks in the current directory. If it doesn't find it, it tries the directory to which the logical device name S: has been assigned. The most important file in this directory on the Workbench disk is called *startup-sequence*. This batch file is automatically loaded and run when the Workbench disk is inserted, and the script it contains causes AmigaDOS to load and run the Workbench program. By modifying this file, you can perform additional tasks at boot time or skip loading the Workbench entirely when you boot up. Other files such as DPAT, SPAT, and PCD may be deleted if you don't use these scripts, but they are small enough so that you may as well keep them.

t

Starts out empty, but is used by some programs (such as the system editors) to hold temporary work files. If you get tight on disk space, you may want to delete the contents of this directory.

fonts

Contains the files for the various text fonts that the Amiga uses. When a program wants to open a new font, it makes a call to the operating system routine *OpenFonts*. *OpenFonts* checks to see if the font is already loaded into memory.

If not, the routine tries to find a disk file containing the new font in this directory. This directory contains a file for each font, the name of which ends in *.font,* for example, *ruby.font.* This contains information about the font, such as the type sizes available. In addition, the fonts directory contains a subdirectory for each font, which contains an image data file for each size of the typeface, such as 12 (dots high) or 8 (dots high). All of the files in the fonts directory may be deleted if you do not use any of the disk-loaded fonts in your graphics programs. Note also that the Release 2 Workbench disk may not have any fonts loaded, as it uses a separate FONTS disk.

libs

Holds the system library files. These are used for operating system extensions implemented as a library of functions. Libraries are used to implement such functions as text-to-speech conversion (the *translator.library* file), the loading and unloading of disk-based text fonts (*diskfont.library*), certain aspects of the Workbench (*icon.library* and *info.library*), single- and double-precision floating-point math functions (*mathieeedoubbas.library* and *mathieeesingbas.library*), and transcendental math functions (*mathtrans.library*). Whenever a call is made to the *OpenLibrary* routine, the operating system looks in this directory for the library file if the library is not already memory resident. Generally, all of the math files may be safely deleted, as can the translator library if none of your programs use synthesized speech.

Empty

Just what the name implies. The only file it contains is the .icon file needed to keep track of other icon files which may be added later. Empty is needed because the pre-2.0 Workbench does not have the equivalent of a MAKEDIR command. The only way to create a new directory from the Workbench is to copy the Empty directory. When you do this, the new directory is named Copy of Empty, and you must then use the Rename menu option to give the directory its chosen name. The Release 2 Workbench has a "New Drawer" menu option that makes the Empty drawer unnecessary.

Utilities

Contains small programs that perform tasks like displaying and printing text files, and displaying and printing graphics files. This whole drawer may be safely deleted if you don't use the programs in it.

Expansion

For AmigaDos 1.2 and higher. Used to store optional device drivers like those required for the Bridgeboard, or the very first Amiga hard drive controller. If this drawer is empty, you can safely delete it.

Prefs

For AmigaDOS 1.3 and higher. Holds the 1.3 Preferences program or the Release 2 preference editor programs. Once you have set you preferences, you can delete the preference program(s), but don't delete the directory itself.

Wbstartup

For AmigaDOS Release 2 and higher. A new feature that added to Release 2 enables the user to start some program automatically at boot-up time, just by dragging their icons into this drawer.

Monitors

For AmigaDOS Release 2.0 only. This drawer is used in the initial 2.0 scheme to hold the monitor description files. To add a monitor description (which lets the system know whether you have a PAL, NTSC, or MULTISYNC monitor), you drag the proper icon from the MonitorStore drawer on the Extras disk to the Monitors drawer on the Workbench. In Release 2.1 and higher, the Monitors drawer is moved into the DEVS drawer, and the MonitorStore drawer becomes the Monitors drawer in the Storage drawer.

Storage

For AmigaDOS 2.1 and higher only. This drawer contains the same sub-drawers as the DEVS drawer (DOSDrivers, KeyMaps, Monitors and Printers), and is used to store device drivers, keyboard mapping files, monitor descriptions, and printer drivers that aren't currently in use. Any of these "extra" files may be safely deleted.

Classes

Workbench 3 includes a Classes drawer which contains the operating system programs that provide a standard system of access to various types of picture, sound, and text data files, as well as some new system gadgets.

Appendix B

AmigaDOS Error Messages

Eventually, you'll see an AmigaDOS error message. Just how intelligible that message is depends in part upon the version of AmigaDOS you are using. Generally, the more recent the version, the more helpful the error message.

The code numbers and short messages listed below show what appears when you use the FAULT or WHY command. A brief explanation of the cause (and cure) or the situation complained of in the message follows. Where there are significant variations of the message between DOS versions, both versions are given. Where an extra word or two appears in some versions and not in others, those words are enclosed in parentheses.

103: insufficient free store (1.3) not enough memory available (2.0 and higher)

There's not enough contiguous free memory available to run the program. You may have too many other programs already running, in which case you may be able to reclaim enough memory to run the program by closing any program that is running, but that you aren't actively using. Another possible cause may be memory fragmentation. This condition may arise after you have run a few program—you have enough memory, but it has been broken down into hunks that are too small to use by programs that have taken a bit of memory here, and a bit there. The only way to un-fragment is to reboot the system and try again. If you get this message frequently, it may be a sign that you don't have enough memory installed for the the type of operation you want, in which case you should consider buying some expansion memory.

114: bad template (2.0 and higher)

The arguments (the additional items of information after the command name) that you've specified are incorrect or do not apply to that command. For a quick check of the argument template for any AmigaDOS command, type a space and question mark (?) immediately following the command and hit RETURN. See

the "AmigaDOS Command Reference" for complete information on the proper format for all of the commands.

115: bad number (2.0)

The command expects you to supply a number, but the group of characters that occupies the space where the number should be cannot be interpreted as a number. Check the command template and make sure to put the number where it should be.

116: required argument missing (2.0)

This command requires an additional item of information which you have left out. Check to make sure what arguments are required.

117: value after keyword missing (2.0)

Some of the newer commands include optional keywords that require a value in addition to the keyword, sometimes appearing after an "equals" sign. For example, the LFORMAT option of the list command requires a text string (such as LIST LFORMAT = "%S" or just LIST LFORMAT "%S"). If you include the keyword, you must also include the value, whether or not you use the equals sign.

118: wrong number of arguments (2.0)

You have specified too many or too few items of information. For example, "FORMAT df0: df1:" fails because you can only specify one drive at a time to format.

119: unmatched quotes (2.0)

Commands that include quotes may require that you include a beginning quote mark for every end quote, and vice versa. Check to make sure that your quote marks come in pairs.

120: argument line invalid or too long (1.3 and 2.0)

The arguments (the additional items of information after the command name) that you've specified are too long, or include invalid characters. For example, the command LIST "RAM:TEST is invalid, because the directory name starts with a quote mark, but doesn't end with one. For a quick check of the argument template for any AmigaDOS command type a space and question mark (?) immediately following the command and hit RETURN. See the "AmigaDOS Command Reference" for complete information on the proper format for all of the commands.

121: file is not an object module (1.3) file is not executable (2.0)

The file you've attempted to run is not a valid program file. If a file named *Picture* contains graphic information, for example, you cannot type the command *Picture* or *RUN Picture* and expect to see the picture, because the file contains only the graphic information itself, not the program required to display that information. To show the picture, you would need a command like *DISPLAY Picture,* where *DISPLAY* is a program that displays picture files. Another example is AmigaDOS command sequence files (scripts), which are text files, not binary program files. Use the EXECUTE command to start up a command sequence file.

202: object (is) in use (1.3 and 2.0)

The directory or file specified as an argument in the invoked AmigaDOS command is being used by another active task. For example, if you CD to a certain directory, making it the current directory of a Shell window, and then try to delete that directory, AmigaDOS won't let you do it, because the Shell is using that directory. You must wait until the task using the file or directory has freed the object (in the example above, you could just CD to a different directory), then try again.

203: object already exists (1.3 and 2.0)

You've attempted to create a directory or file that already exists. Since AmigaDOS is case-insensitive, you would get this message is you tried to create a directory named STUFF, for example, if you already had a directory named stuff in the same place. The solution is to rename or delete the existing object if you wish to use the specified name for a new file or directory.

204: directory not found (1.3 and 2.0)

You've referred to a directory which does not exist. Check the complete pathname and spelling of the specified directory. LIST and DIR may be used to get a complete listing of all the directories on a disk. See the "AmigaDOS Command Reference" sections on the LIST and DIR commands for complete information.

205: object not found (1.3 and 2.0)

You've referred to a file or device which does not exist. Check the spelling of the specified object—you may have typed LIST Fed instead of LIST Fred. LIST and DIR may be used to get a complete listing of all the files on a disk. The ASSIGN

command can be used to check on the name of all logical and physical devices known to the system. See the "AmigaDOS Command Reference" sections on the LIST, DIR, and ASSIGN commands for complete information.

206: invalid window description (1.3 and 2.0)

You've attempted to open a new window on the screen with invalid width, height, or position, or you've specified a physical device which does not support display windows (for instance, SER: or PAR:).

210: stream name component invalid (1.3) object name invalid (2.0)

The filename you've specified contains one or more invalid characters (such control characters) or is longer than 30 characters.

212: object (is) not of required type (1.3 and 2.0)

The type of the object you've specified is incompatible with the AmigaDOS command. An example of this is attempting an operation on a file that can only be performed on a directory, or vice versa. See the "AmigaDOS Command Reference" for complete information on the command and its options.

213: disk not validated (1.3 and 2.0)

An error has occurred during the validation of a disk. The disk may be bad or the validation process was interrupted before it was completed. If the disk was in use, try copying all of the existing information on it to another disk. You may also be able to salvage the information using a program like the shareware *DiskSalv* or the commercial program *Quarterback Tools.* You cannot write to an unvalidated disk.

214: disk (is) write protected (1.3 and 2.0)

You've attempted to write to a disk whose write-protection tab is in the write-protected position. If you're sure you want to write to the disk, slide the black plastic write-protect tab so that it completely covers the small, square cut-out.

215: rename across devices attempted (1.3 and 2.0)

You've specified different devices in the FROM and TO (or AS) arguments of the RENAME command. Both arguments must reside on the same device. See the "AmigaDOS Command Reference" section on the RENAME command for further information.

216: directory not empty (1.3 and 2.0)

You've attempted to DELETE a directory that's not empty. You can force the deletion to occur by adding the ALL keyword to the DELETE command. See the "AmigaDOS Command Reference" section on the DELETE command for further information.

218: device (or volume) not mounted (1.3 and 2.0)

You've referenced a disk volume that isn't currently in any of the disk drives. Check the name specified, or locate the desired volume and insert it in one of the system's drives, then try again.

220: comment too big (1.3) comment is too long (2.0)

You've specified a comment which exceeds 80 characters in conjunction with the FILENOTE command. Try again with a shortened version of the comment.

221: disk (is) full (1.3 and 2.0)

The disk that you've attempted to write to does not have enough free space to complete the specified command. Free up enough space by deleting any unneeded files and/or directories, or use another disk.

222: file (object) is protected from deletion (1.3 and 2.0)

You've attempted to delete a file which has been protected from being deleted by the PROTECT command. The status of a file's protection flags may be examined using the LIST command. See the "AmigaDOS Command Reference" sections on the PROTECT and LIST commands for complete information.

223: file is write protected (1.3 and 2.0)

You've attempted to write to a file which has been protected from being written to by the PROTECT command. The status of a file's protection flags may be examined using the LIST command. See the "AmigaDOS Command Reference" sections on the PROTECT and LIST commands for complete information.

224: file is protected from reading (1.3 and 2.0)

You've attempted to read a file which has been protected from being read by the PROTECT command. The status of a file's protection flags may be examined

using the LIST command. See the "AmigaDOS Command Reference" sections on the PROTECT and LIST commands for complete information.

225: not a (valid) DOS disk (1.3 and 2.0)

You've inserted a disk that is not an AmigaDOS format disk into one of the system's drives. If you are using CrossDOS to read an IBM disk, you will get this message whether you've inserted either an IBM or Amiga format disk, since an IBM disk will be invalid to the Amiga device (df0:) while an Amiga disk will be invalid to the IBM device (pc0:)

226: no disk in drive (1.3 and 2.0)

You've referenced a disk drive which does not contain a disk. Insert an AmigaDOS format disk in the specified drive to proceed.

Index

THE BUDDY SYSTEM

SOFTWARE TUTORIAL SERIES

The Buddy System for AmigaDOS™ is your personal guide through the fundamentals and features of the Amiga® and its operating system. Increase your productivity with an online help system that gives you the information that you want and need to know!

- *Multitasking software gives help at the touch of a key*
- *Point & Click Hypertext interface interactively follows your thinking process*
- *Topic index & search functions get you to important information quickly*
- *Visual Demonstrations show you how to use various functions and features **on your** computer*
- *Amiga® Speech narration & captioning*
- *A **Complete** source of reference:*
  - *Getting started with your Amiga system*
  - *Beginner to advanced level tutorials*
  - *Professional tips and techniques*
  - *Workbench™ AND Shell techniques*
  - *Online AmigaDOS™ & Arexx™ command reference*
  - *Scripting, Mountlists & Startup-Sequences*
  - *The Preference Editors and System Utilities*
  - *Compatible with AmigaDOS™ releases 1.3 and 2.x (upgrades will be made available for future releases)*
  - *And much more!*

"If you are a new Amiga User... you can't afford to be without the Buddy System... If you are an experienced user, the Buddy System is still a valuable asset..."
- Chuck Raudonis, Amazing Computing, Jan 1992

Also Available for:
PageStream™
DELUXEPAINT™ IV*
imagine™

$49.95 ea.
Suggested Retail
****expanded DPaint version available ($69.95)***

for AmigaDOS™

Products mentioned are trademarks or registered trademarks of their respective companies.

HelpDisk, Inc., 13860-12 Wellington Trace, #200
Wellington, FL 33414 (407) 798-8865

AMI-BACK version 2.0

The Ultimate Amiga Hard Disk Backup Utility

Ami-Back is the solution of choice for all your data backup needs. Designed to be both powerful and flexible, Ami-Back gives you complete control over the backup process. There is no other backup program for the Amiga that can even come close to offering the features, ease of use, and reliability of Ami-Back.

- *The fastest hard disk backup program for the Amiga*
- *"911-Recovery"™ mode recovers lost data without the need for disk utility programs*
- *Backs up to floppies, hd floppies, hard drives, and SCSI tape drives and DAT drives.*
- *Backs up to a single AmigaDOS file or device*
- *Appends multiple backups to a single tape*
- *Compresses data to backup with no speed loss*
- *Includes a graphical scheduler for unattended backups*
- *Has Online Help just a mouse button click away*
- *Allows password protection of valuable data*
- *Supports AREXX*
- *Includes many more features*

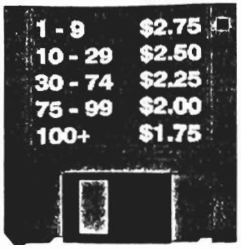
ABT-250

Tape Drive Offer!

This is an internal SCSI interface streaming cartridge tape drive with 250 megabytes of storage capacity. Data transfers at rates over 5 mb per minute. The ABT-250 kit is bundled with the *Ami-Back* backup utility and one tape for $539.95 (without *Ami-Back* for only $489.95. An external drive kit is available for an additional $120.00)

Moonlighter Software Development, Inc.
3208-C E. Colonial Drive, Suite 204
Orlando, Florida 32803
Phone: (407) 384-9484 FAX: (407) 384-9391

Programs & Programmers wanted — Write Us.



DISKS O'PLENTY INC
Public Domain/Shareware For AMIGA

Order One Of The Many Programs Listed Below Or Request A Free Catalog Of Our Complete AMIGA Program Selections
Catalogs Also Available For C64/C128 And IBM & Compatible

AMIGA PROGRAMS

GAMES

ARCADE GAMES #1 (GA-002)
Features the games CYCLES; COSMO-ROIDS, MAZEMAN, STONE-AGE; TRICLOPS; and 3-D BREAKOUT
WHEEL (GA-003)
Talking Wheel of Fortune game
HACK! (GA-006)
Adventure game features both text and graphic versions
MORIA (GA-011)
Role playing adventure, requires 1 Meg
GAME PACKAGE (GA-013)
Includes SLOT CAR arcade racing game; ZERG fantasy role-playing game; MO-NOPOLY board game
THE SIMPSONS GAME (GA-016)
1 or 2 player arcade, Help Bart and Lisa make the daring & exciting escape from Elementary School (joystick)
GAME ASSORTMENT (GA-018)
CLUE board game; AMIGA CRIBBAGE card/board game; ROLL-ON 9 arcade game; CHECKERS board game; OB-SESS-O-MATIC clever tetris-type game
KINGDOM AT WAR (GA-019)
Strategy game requires 1 Meg
ARCADES & TRIVIA (GA-020)
DOWNHILL CHALLENGE skiing arcade; HEAD GAMES arcade game; HOLLY-WOOD TRIVIA choose topics (joystick)
SHOOTEM'UP ARCADES (GA-024)
Features these arcade games WOLFBANE; 109; and DEATH (joystick)
THE HOLY GRAIL (GA-030)
Self-Booting Text adventure game

FRED FISH & TBAG DISKS

Fred Fish and TBag Disks Are Available

BUSINESS

THE CLERK (BU-001)
Complete Accounting Package for Small Businesses, includes account receivables, account payables, general ledger and file cabinet
LEGAL FORMS (BU-002)
Ready-to-use legal forms to use with your word processor

EDUCATION

EDUCATION PACKAGE (ED-002)
Includes SPELL.TEST enter spelling words and take test; STATES QUIZ study or test on states by spelling, capitals, nicknames & more; MATH.TEST study or test on addition, subtraction, multiplication, metrics, weights, measures & more; YOUR BOOKS features book-maker and reader; ORBIT3D survive in orbit game (joystick); PUSHOVER strategy game; BLACKBOX logic game

UTILITIES

NIB (UT-002)
Precision disk copier/nippler requiring 2 drives
CLI TUTORIAL DISK (UT-005)
Learn the ins & outs of CLI
UTILITY DISK (UT-008)
AMCAT disk cataloger; FORMATTER; DCOPY disk copier; MACGAG; SLIDESHOW CONSTRUCTION KIT; BOOTBLOCK GENERATOR; POWER-PACKER pack files; and ENVELOPE PRINTER
MASTER VIRUS KILLER (UT-015)
Detect & Destroy Viruses

HOME & HOBBIES

TAROT READER (HH-003)
Fortune Teller of Past, Present & Future
BIO-RHYTHMS (HH-006)
Computes & charts biorhythms

MUSIC

PERFECT SOUND EDITOR (MU-002)
Creates, Edits, Plays and Saves
GALLERY OF CHRISTMAS (MU-010)
Images, Sounds, Scenes and Music
CLASSICAL MUSIC (MU-021)
J.S.Bach's Prelude #2,3,4,5,6,9,10; E Minor; and Peer Gynt Suite Selections

GRAPHICS

FONTS #1 (GR-003)
BLITZFONTS faster output; ALTICON separate icon images; SETFONT changes font; and 40+ NEW FONTS
ACTION MOVIE MAKER (GR-012)
Present pictures, sounds & speech
BOOT SCROLL MAKER (GR-029)
Make your own scrolling text
PAINT PROGRAMS (GR-034)
ULTRA-PAINT; SIMPLE PAINT; PAT-TERN EDITOR; and MOVIE

ORDERING INFO

Shipping & Handling Charges are $3.50 per order, Foreign Orders are $3.50 plus $1.00 per disk. Florida Residents add 6% Sales Tax. Remit in U.S. Funds, Order By Disk #

Prices are per disk ordered:
1 to 9 $2.75 10 to 29 $2.50 30 to 74 $2.25
75 to 99 $2.00 100+ $1.75

Send check or money order to:
DISKS O'PLENTY INC.
8362 PINES BLVD., SUITE 270-E
PEMBROKE PINES FL 33024

AmigaDOS Revealed

This best-selling reference to AmigaDOS has been completely updated to cover all current versions, including 1.3, and Releases 2 and 3. Written by well-known Amiga authority Sheldon Leemon, the *AmigaDOS Reference Guide* is the only book written for both beginners and advanced users because it includes both easily understood tutorials and a comprehensive reference.

The CLI, an MS-DOS-style command-driven environment, allows the user to customize almost any disk-operating function of the Amiga. With it you can:

- Create command-sequence files to automate almost any task
- Set up customized file directories and subdirectories
- Directly control the printer screen, hard disk, and console devices
- Use a RAM disk to set aside portions of memory as an electronic disk drive
- Access two text editors

Author Sheldon Leemon takes the reader, step by step, through the intricacies of AmigaDOS, from creating a CLI disk to building a personalized command sequence file. Thoroughly illustrated with practical examples, this book covers every AmigaDOS command including the new Release 2 and Release 3 commands. This is the perfect guide for both novice and experienced Amiga users.

Sheldon Leemon is a noted Amiga authority and regular columnist and contributor to *COMPUTE* magazine's Amiga Resource edition. His byline has also appeared in *Amiga World* and *Amazing Computing*. Leemon is the author of more than half a dozen computer books, including the best-selling *Mapping the Commodore 64* and *Inside Amiga Graphics*.

$22.95 US
£20.95 UK

ISBN 0-87455-268-0

90000

9 780874 552683